UNIVERSITÀ DEGLI STUDI DI PARMA
FACOLTÀ DI INGEGNERIA
CORSO DI LAUREA IN INGEGNERIA ELETTRONICA

METODI PROIETTIVI PRECONDIZIONATI PER LA RISOLUZIONE SU ARCHITETTURE PARALLELE DI RETI DI PETRI

Relatore:
Chiar.mo Prof. Stefano Caselli

Correlatori:
Chiar.mo Prof. Mauro Diligenti
Chiar.mo Prof. Gianni Conte

Tesi di laurea di:
Stefano Bottazzi

Anno Accademico 1998-99

Ringraziamenti

Ringrazio il relatore Prof. Stefano Caselli ed i correlatori Prof. Mauro Diligenti e Prof. Gianni Conte per la grande disponibilitá dimostrata, l'aiuto fornitomi e per l'attenzione prestata allo sviluppo del lavoro di tesi

Un ringraziamento speciale ai miei genitori ed a mia sorella per avermi sopportato e sostenuto in questi anni di studio.

Indice

Introduzione

I continui e rapidi progressi in diversi settori tecnologici e scientifici hanno portato ad un significativo aumento della complessitá dei sistemi da descrivere nelle varie aree di applicazione. L'efficace sfruttamento di tutte le possibilitá offerte da questi sistemi complessi richiede l'adozione di strumenti di modellizzazione e simulazione in grado di confrontare in modo semplice ed economico le prestazioni e caratteristiche delle diverse soluzioni possibili. Fra i vari metodi proposti per descrivere sistemi complessi, le reti di Petri sono fra i pochi che permettono di integrare capacitá descrittiva e di valutazione delle prestazioni [13]. Come esempi di sistemi che possono essere efficacemente descritti ed analizzati mediante l'uso di reti di Petri si possono citare i sistemi di elaborazione parallela e distribuita, sistemi di telecomunicazioni, sistemi di produzione complessi e sistemi organizzativi.

Le reti di Petri (PN) furono introdotte originariamente da Carl Adam Petri nella sua tesi di dottorato. Sono uno strumento grafico adatto a descrivere formalmente un sistema caratterizzato da *concorrenza*, *sincronizzazione*, *mutua esclusione* e *conflitto*; sono basate sul concetto di stato e su di una regola per le transizioni da uno stato all'altro. Originariamente non includevano un concetto di tempo, che fu introdotto in seguito in termini di transizioni temporizzate. Le reti di Petri con transizioni temporizzate ed in cui il ritardo di sparo é una variabile casuale con funzione di distribuzione di probabilitá esponenziale negativa sono dette stocastiche (SPN). I modelli SPN in cui transizioni temporizzate del tipo sopracitato coesistono con transizioni immediate sono definite SPN generalizzate (GSPN).

Una PN comprende quattro tipi di elementi: *posti*, *transizioni*, *archi* e *gettoni*.

Lo stato del sistema in un certo istante é determinato dalla posizione dei gettoni nei posti, rappresentati dai cerchi. Esso é anche detto *marcatura* della rete. Se un posto rappresenta il verificarsi di una condizione, allora conterrá un gettone se la condizione é verificata, oppure nessuno in caso contrario. Se il posto descrive una situazione, allora il numero di gettoni che contiene specificherá la situazione stessa.

Le transizioni rappresentano eventi che modificano lo stato del sistema. Gli archi uniscono i posti alle transizioni e viceversa. Essi descrivono le relazioni fra stati locali ed eventi: quelli *entranti* indicano lo stato locale in cui l'evento puó accadere, mentre quelli *uscenti* mostrano le modifiche allo stato dovute all'evento stesso.

L'evoluzione dinamica della rete é governata dalla regola di *sparo*: una transizione puó sparare (e quindi l'evento puó accadere) se tutti i posti in ingresso contengono almeno un gettone. In questo caso la transizione é *abilitata*. Lo sparo della transizione rimuove un gettone da tutti i posti in ingresso e ne pone uno su tutti quelli in uscita.

Le transizioni possono essere *immediate* o *temporizzate*. Le transizioni temporizzate, rappresentate da un rettangolo vuoto, hanno associato un contatore locale detto timer: quando la transizione viene abilitata il timer é fatto partire e quando raggiunge lo zero la transizione spara. Se l'abilitazione viene persa prima dello sparo, il timer si ferma, e viene fatto ripartire solo quando la transizione viene nuovamente abilitata. Lo sparo di transizioni temporizzate in una SPN avviene con un ritardo casuale avente distribuzione esponenziale negativa; ogni transizione di questo tipo ha quindi associato un *rate*, sufficiente a caratterizzare la funzione di distribuzione di probabilitá esponenziale negativa. Tali transizioni sono adatte a descrivere lo svolgimento di un'attivitá.

Le transizioni immediate, al contrario di quelle temporizzate, sparano non appena sono abilitate. Risultano pertanto piú adatte a descrivere il verificarsi di una condizione logica. Se piú transizioni immediate sono abilitate contemporaneamente é necessario stabilire un criterio per sceglire quale sparerá per prima: é possibile assegnare una prioritá oppure introdurre una distribuzione discreta di probabilitá assegnata alle transizioni.

Il conflitto fra transizioni temporizzate ed immediate é usato per descrivere eventi come la preemption o l'arrivo di interrupt; normalmente le transizioni immediate hanno prioritá nello sparo rispetto a quelle temporizzate.

L'esistenza delle transizioni immediate introduce una distinzione fra le marcature: le marcature nelle quali non é abilitata nessuna transizione immediata sono dette *tangible*, mentre se é abilitata almeno una transizione immediata la marcatura é detta *vanishing*. In quest'ultimo tipo di marcatura il sistema non consuma tempo.

L'analisi dei modelli GSPN si articola in piú fasi:

1. Calcolo dell'insieme di raggiungibilitá : tale insieme contiene l'elenco di tutte le marcature raggiungibili a partire da una certa marcatura iniziale M_0 ed é indicato con $RS(M_0)$.

2. Determinazione del grafo di raggiungibilitá : contiene le informazioni sulle transizioni che hanno sparato per raggiungere una determinata marcatura. Il grafo é formato da nodi, uno per ogni marcatura presente nell'insieme di raggiungibilitá, ed archi, che uniscono i nodi e rappresentano la transizione che collega direttamente due marcature. É indicato con $GR(M_0)$.

3. Soluzione della catena di Markov isomorfa alla rete e calcolo degli indici di prestazione.

Riguardo al terzo punto, é possibile dimostrare che i sistemi SPN sono isomorfi alle catene di Markov continue nel tempo(CTMC). In particolare la CTMC associata ad un modello SPN puó essere ottenuta applicando le seguenti regole:

- Lo spazio degli stati $S = s_i$ di una CTMC corrisponde all'insieme di raggiungibilitá $RS(M_0)$ di una SPN.

- La velocitá di transizione da uno stato s_i(corrispondente alla marcatura M_i) allo stato s_j (relativo ad M_j) é ottenuta sommando le velocitá di sparo delle transizioni che sono abilitate in M_i ed il cui sparo porta ad M_j.

Queste regole sono alla base di algoritmi per la costruzione automatica del generatore infinitesimale della CTMC isomorfa alla SPN. Detta Q la matrice del generatore infinitesimale, w_k le velocitá di sparo delle transizioni T_k e detti $E_j(M_i)$l'insieme delle transizioni il cui sparo porta la rete dalla marcatura M_i alla marcatura M_j allora gli elementi q_{ij} del generatore infinitesimale sono:

$$q_{ij} = \begin{cases} \sum_{k:T_k \in E(M_i)} w_k & i \neq j \\ -q_i & i = j \qquad \text{dove } q_i = \sum_{k:T_k \in E(m_i)} w_k \end{cases} \tag{1}$$

Definito η il vettore riga che rappresenta la distribuzione di probabilitá sulle marcature della SPN, é possibile calcolare η risolvendo il seguente sistema lineare:

$$\begin{cases} \eta Q = 0 \\ \eta 1^T = 1 \end{cases} \tag{2}$$

dove 0 é un vettore riga con tutte le componenti nulle, mentre 1^T é un vettore colonna di componenti unitarie; entrambi i vettori hanno le stesse dimensioni di η. Ogni elemento di η rappresenta la probabilitá che la rete si trovi nella marcatura corrispondente. Il vettore η é la base per l'analisi quantitativa delle prestazioni e del comportamento del sistema studiato. Si possono infatti definire degli indici di prestazione mediante la seguente formula generale, in cui la generica funzione r viene pesata mediante la distribuzione di probabilitá η:

$$R = \sum_{i:M_i \in RS(M_0)} r(M_i)\eta_i$$

dove $r(M_i)$ sono i valori assunti da r in corrispondenza della marcatura M_i ed R é l'indice di prestazione cercato. Fra i possibili indici di prestazioni é opportuno ricordare il *throughput*, numero medio degli spari per unitá di tempo di una transizione T_j (ed in questo caso $r = w_j$), ed il *numero medio dei gettoni* in un dato posto p_j (per il quale $r = M_i(p_j)$).

Un problema che si presenta utilizzando modelli GSPN di sistemi reali é costituito dall'esplosione dello spazio degli stati generato, che pone dei limiti al campo di applicabilitá delle soluzioni al calcolatore a causa dei tempi di calcolo e dell'occupazione di memoria.

Il problema della generazione su architetture parallele dello spazio degli stati é giá stato affrontato in [8], mentre lo studio di algoritmi proiettivi di soluzione di sistemi lineari adatti per questa classe di problemi é stato compiuto in [11, 12]. Scopo di questa tesi é invece lo studio di alcune tecniche di precondizionamento, da applicare al sistema (2), e lo sviluppo di algoritmi paralleli da impiegare nella risoluzione di modelli GSPN. L'approccio parallelo al problema é finalizzato non solo al miglioramento delle prestazioni, ma anche all'aumento delle dimensioni dei sistemi trattabili.

Per la realizzazione dei programmi paralleli si é scelto di utilizzare la libreria MPI, che costituisce uno standard affermato, e la recentissima API OpenMP, sviluppata in collaborazione fra varie industrie e dedicata alla programmazione parallela su macchine di tipo shared memory. Le sperimentazioni sono state condotte principalmente sul sistema *Onyx2 Infinite Reality* del Centro di Calcolo dell'Universitá di Parma. Le scelte adottate consentono tuttavia un'agevole portabilitá dei programmi di soluzione sia su piattaforma distribuita (con l'uso di MPI), sia su altre macchine a memoria condivisa (mediante OpenMP).

Nel primo capitolo della tesi vengono esposte alcune nozioni generali sul calcolo parallelo quali i principali paradigmi di programmazione parallela utilizzabili, gli indici di prestazione necessari per valutare l'efficienza di un programma parallelo ed una metodologia generale applicabile per la parallelizzazione di un'algoritmo. Sono inoltre descritti gli strumenti software ed hardware utilizzati.

Nel secondo capitolo l'attenzione viene fissata sul problema specifico trattato. Vengono descritte le operazioni fondamentali necessarie per la soluzione parallela di sistemi lineari, con particolare riferimento agli algoritmi proiettivi GMRES e BiCGStab. Viene successivamente presentata la realizzazione di versioni parallele di tali algoritmi, mettendo in evidenza le differenze fra il modello di programmazione *message passing* e quello *shared memory.*

Il terzo capitolo é invece dedicato alla descrizione dei precondizionatori. Vengono studiati in dettaglio due metodi di precondizionamento e ne viene descritta l'implementazione in un algoritmo di tipo proiettivo. A conclusione del capitolo vengono esposte alcune considerazioni sull'effetto numerico del precondizionatore.

Il quarto capitolo riporta una descrizione dei modelli parametrici delle reti di Petri utilizzate per studiare il comportamento dei programmi realizzati. Viene inoltre riportata una sintesi dei risultati ottenuti. Per il GMRES sequenziale e precondizionato sono analizzati i tempi di esecuzione, il numero di iterazioni e la precisione ottenibile oltre alla scelta della dimensione ottimale del sottospazio di proiezione. Questi risultati sono posti a confronto con la versione non precondizionata e con altri algoritmi in modo da poter valutare l'utilitá dell'impiego del precondizionatore. Per i programmi paralleli viene invece posta l'attenzione sui tempi di calcolo e sullo speedup ottenuto. Sono inoltre formulate osservazioni sul livello di parallelismo ottenuto con i vari programmi e sull'effetto dell'ambiente multiutente sulle prestazioni riscontrate.

Il capitolo conclusivo espone le considerazioni riassuntive sul lavoro svolto e sui risultati raggiunti e propone spunti per lo sviluppo degli argomenti trattati.

L'appendice A é relativa alla descrizione ed implementazione di un ulteriore algoritmo proiettivo, il TFQMR, che é stato inizialmente preso in considerazione come alternativa al GMRES e BiCGStab, ma che non ha mostrato risultati particolarmente validi per questa classe di problemi.

Capitolo 1

Aspetti generali del calcolo parallelo

1.1 Programmazione delle architetture parallele

Lo scopo originale che sta alla base di un sistema parallelo é sicuramente legato alla ricerca di alte prestazioni ma non si esaurisce in questo. Lo sviluppo di calcolatori paralleli fornisce ed ha fornito in passato la possibilitá di sperimentare architetture e concetti molto innovativi. La libertá dal classico modello di Von Neummann, propria degli elaboratori paralleli consente di esplorare soluzioni originali in tutti i campi di un sistema: dall'architettura, ai vari componenti (processori, gerarchie di memorie, bus), alle reti di interconnessione, fino ai vari strumenti software (sistema operativo, compilatori, applicativi).

Per sfruttare le capacitá di un'architettura parallela é indispensabile avere a disposizione un linguaggio di programmazione in grado di fornire strumenti per l'uso combinato di piú processori.

Sono stati sviluppati veri e propri linguaggi di programmazione parallela, a volte ricorrendo ad estensioni di linguaggi sequenziali come nel caso di C* e del Fortran D. Un'alternativa all'impiego di linguaggi dedicati é l'uso di librerie (quali ad esempio MPI e PVM) o di API (come OpenMP) destinate alla gestione del parallelismo. In tempi recenti sono anche stati compiuti considerevoli sforzi nello sviluppo di tecnologie di parallelizzazione automatica: l'obbiettivo é lasciare liberi i programmatori di utilizzare il linguaggio di alto livello preferito, delegando al compilatore il compito di ottenere un efficiente programma parallelo (*implicit parallel programming*). Un tale risultato non é ancora stato raggiunto ed al programmatore spetta sempre la parte piú cospicua della costruzione di un programma parallelo.

In dipendenza dagli strumenti software utilizzati si realizzano diversi modelli di programmazione parallela. Se si considera la modalitá di accesso ai dati da parte dei processori sono possibili due tipi di paradigmi di programmazione: a memoria condivisa oppure a scambio di messaggi.

Un programma di tipo **a memoria condivisa** é costituito da un insieme di processi che possono accedere a variabili condivise. In questo caso il linguaggio di programmazione parallelo (o la libreria di parallelizzazione) deve fornire le primitive per risolvere i problemi di mutua esclusione che si generano quando piú processi accedono agli stessi dati. Questo modello di programmazione si adatta in maniera naturale a eleboratori paralleli di tipo shared memory.

Un programma di tipo **a scambio di messaggi** é costituito da un insieme di processi totalmente indipendenti, dotati esclusivamente di variabili locali private e della capacitá di inviare e ricevere messaggi dagli altri processi. Questo modello di programmazione é particolarmente adatto a sistemi distribuiti, ma anche sistemi shared memory, soprattuto se di tipo NUMA (Non Uniform Memory Access), possono trarre benefici da un tale approccio che migliora la localitá dei dati.

Se invece dell'accesso ai dati si considera l'aspetto funzionale del programma, é possibile introdurre la distinzione fra data-parallelism e control-carallelism.

In un programma **data parallel** i vari processi eseguono le stesse operazioni su dati diversi. Un tale paradigma di programmazione risulta naturale su macchine di tipo SIMD o vettoriali, ma in alcuni casi puó essere necessario l'uso di calcolatori MIMD. L'utilizzo su macchine di tipo MIMD é sempre possibile ma é necessario prestare attenzione alle esigenze di frequenti sincronizzazioni tipiche di questi programmi.

In un programma basato sul **control parallelism** i vari processi eseguono diverse sequenze di istruzioni utilizzando lo stesso insieme di dati oppure no. Un caso specifico é rappresentato dal parallelismo di tipo pipelined: ogni processore elabora un pacchetto di dati e spedisce i risultati parziali al successivo. Un modello di programmazione di tipo control parallelism puó essere adottato efficacemente solo su macchine di tipo MIMD.

1.2 Prestazioni di un programma parallelo

La valutazione delle prestazioni di un algoritmo sequenziale é generalmente attuabile misurando i tempi di esecuzione in funzione delle dimensioni dei dati in ingresso. I tempi di esecuzione di un programma parallelo dipendono, oltre che dall'algoritmo scelto, anche dal numero di processori e dall'architettura su cui il programma viene eseguito. La valutazione di un'algoritmo parallelo non puó pertanto avvenire senza considerare l'architettura su cui é stato implementato. Per misurare le performance di un sistema parallelo, costituito da un algoritmo parallelo eseguito su una certa architettura, sono comunemente utilizzate le seguenti metriche (definite in [4]):

- **Run time**

  Per un programma seriale é l'intervallo trascorso tra l'inizio e la fine dell'esecuzione. Per un programma parallelo é il tempo trascorso fra l'inizio dell'esecuzione di un programma e l'istante in cui l'ultimo processore termina l'esecuzione. Nel resto del capitolo il *parallel run time* sará indicato con T_p, mentre con T_s si indicherá il *serial run time*.

- **Speedup**

  Lo speedup é definito come il rapporto tra il tempo impiegato per risolvere un problema su un singolo processore e quello impiegato per risolvere il problema utilizzando p processori identici.

$$S = \frac{Time(1)}{Time(p)} \tag{1.1}$$

Si tratta di uno degli indici di prestazioni piú significativi per programmi paralleli in quanto misura il miglioramento di prestazioni rispetto alla versione seriale in funzione del numero di processori utilizzati. In teoria lo speedup non puó mai superare il numero di processori p. Nella pratica si sono osservati speedup superiori a p e tale fenomeno viene definito *superlinear speedup*: normalmente esso é dovuto ad un algoritmo sequenziale non ottimo oppure a caratteristiche dell'hardware utilizzato, come la maggiore disponibilità di memoria per l'esecuzione parallela.

- **Efficienza**

  Questa grandezza é definita come

  $$E = \frac{S}{p} \tag{1.2}$$

  dove S é lo speedup e p il numero dei processori. La misura di questo valore mostra la frazione di tempo in cui un processore é impiegato utilmente. Un sistema parallelo ideale ha uno speedup uguale a p e quindi $E = 1$.

- **Costo** É anche chiamato *processor-time product* ed é definito come il prodotto del parallel run time per il numero di processori impiegati: il suo valore rappresenta la somma dei tempi che ogni processore impiega nella risoluzione del programma in esame. Un sistema parallelo é considerato ottimale dal punto di vista del costo se il costo é proporzionale a T_s.

- **Scalabilitá**

  Si é giá osservato che generalmente lo speedup non puó superare il numero p di processori. Esiste peró un limite superiore dello speedup ancora piú restrittivo dovuto alla legge di Amdahl:

legge di Amdahl: Se un problema di dimensione W ha una parte sequenziale W_s allora lo speedup é sempre inferiore a W/W_s.

La conseguenza della legge di Amdahl é che nella quasi totalitá delle applicazioni reali in cui W_s é diverso da zero si osserva una diminuzione dell'efficienza all'aumentare del numero dei processori. La scalabilitá di un sistema parallelo é la capacitá che esso ha di aumentare lo speedup in proporzione al numero di processori impiegati: un sistema é scalabile se mantiene costante l'efficienza all'aumentare di p, eventualmente aumentando le dimensioni del problema trattato.

1.2.1 Fattori che determinano le prestazioni

Non sono molti i programmi paralleli che riescono ad ottenere, su di una certa architettura, uno speedup prossimo a p ed un efficienza vicina ad 1. Oltre ai limiti dovuti all'algoritmo utilizzato, che in genere presenta una porzione di calcolo intrinsecamente non parallelizzabile, esistono non idealitá dovute principalmente a tre cause [10, 4]:

Comunicazioni interprocessore. Il tempo speso nelle comunicazioni interprocessore é spesso l'origine del principale overhead di un processo di calcolo parallelo. Si definisce *latenza di comunicazione* [1] il tempo impiegato per completare la comunicazione di un certo messaggio tra due processori di un'architettura parallela. I principali parametri che determinano la latenza sono:

- startup time: questo termine comprende il tempo necessario a preparare il messaggio da inviare, il tempo per eseguire l'algoritmo di routing ed il tempo di interfacciamento fra processore e la rete di comunicazione. É quindi il tempo richiesto per maneggiare il messaggio dal lato del processore che invia

[1] Alcuni autori indicano con il termine latenza il tempo impiegato dalla fase di startup della comunicazione. Detta t_s la latenza e $t_w = 1/bandwidth$ il per-word transfer time allora il costo complessivo della comunicazione per un pacchetto di k dati é dato da $t_s + kt_w$

il messaggio e viene speso una sola volta per una singola comunicazione.Nel seguito sará indicato con t_s.

- latenza di nodo o per-hop time: é il tempo impiegato dall'header del messaggio per transitare fra due processori direttamente interconnessi; viene indicato con t_h.

- larghezza di banda: é la velocitá con la quale i dati attraversano la rete di interconnessione una volta che la comunicazione é iniziata. Se la banda é r si puó definire $t_w = 1/r$ il *per-word transfer time*, che é il tempo impiegato da una word per attraversare un link della rete di interconnessione.

Ulteriori parametri possono esser necessari per descrivere la latenza di comunicazione su una specifica architettura, ma quelli indicati sono comunque in grado di fornire un modello di prima approssimazione delle comunicazioni all'interno di un'architettura parallela.

Sbilanciamento del carico di lavoro. In numerose applicazioni é impossibile, o comunque molto dispendioso, determinare a priori il carico di lavoro assegnato ad ogni processore (ad esempio con algoritmi di ricerca ed ottimizzazione, [4]). In questi casi i carichi di lavoro assegnati ai vari processori non sono uguali e se ad un certo istante i processori si devono sincronizzare (ad esempio per condividere un risultato parziale) allora alcuni di essi termineranno i propri compiti prima degli altri e dovranno attendere in una condizione *idle* finché anche l'ultimo processore non abbia raggiunto il punto di sincronizzazione. Questo comporta uno sfruttamento non ottimale delle risorse disponibili.

Necessitá di calcolo extra. A volte puó succedere che il migliore algoritmo seriale non sia facilmente parallelizzabile, rendendo cosí obbligatorio l'uso di un algoritmo di caratteristiche peggiori (e che quindi richiede un numero superiore di calcoli) per rendere piú agevole la fase di parallelizzazione. In altri casi la parallelizzazione stessa puó richiedere dei calcoli aggiuntivi ad esempio per determinare i sottoinsiemi dei dati in ingresso da assegnare ai vari processori, o per riordinare tali dati in modo da rendere possibile l'impiego di algoritmi paralleli piú efficienti.

A conclusione del paragrafo é importante fare alcune considerazioni riguardo alle misure di prestazioni ed alle operazioni di I/O. Come viene fatto notare in [10] spesso, soprattutto su sistemi molto veloci, le fasi di I/O sono molto dispendiose se confrontate ad operazioni di calcolo. Non é inoltre vero in generale che elaboratori paralleli dispongano di "sistemi di I/O paralleli" in grado di velocizzare anche le operazioni di lettura e scrittura di dati. In tali circostanze la misura di T_s e T_p, cosí come sono stati definiti, porterebbe a risultati validi sperimentalmente ma non in grado di descrivere dettagliatamente le caratteristiche del programma parallelo realizzato. A titolo di esempio si puó osservare che se un dato programma ha un tempo di esecuzione T(1) ed una porzione non parallelizzabile dovuta ad operazioni di I/O del 20 % allora, in conseguenza della legge di Amdahl, lo speedup ottenibile é al massimo 5, indipendentemente dal numero dei processori e da come sia stata parallelizzata la parte di calcolo del programma. É opportuno per questo motivo distinguere nelle misure di prestazioni i due contributi:

$$T(n) = T_{calc}(n) + T_{I/O}(n) \tag{1.3}$$

Se si vogliono fare considerazioni riguardo alle prestazioni nell'uso applicativo del programma é necessario considerare la somma dei termini; se invece si vuole studiare la qualitá del programma parallelo realizzato e la sua efficacia su di una certa architettura é ragionevole centrare l'attenzione sul termine $T_{calc}(n)$.

1.3 Costruzione di un programma parallelo

Nella maggioranza dei casi la scrittura di un programma parallelo si basa sulla parallelizzazione di un algoritmo o di un programma seriale preesistente. Supponiamo perciò che l'algoritmo sequenziale che dobbiamo rendere parallelo sia noto. In alcuni casi puó accadere che il migliore algoritmo seriale che risolve un certo problema non fornisca un livello di parallelismo sufficiente ed allora diviene necessario scegliere un altro algoritmo.

Il primo compito che spetta al programmatore é individuare le parti dell'elaborazione che possono essere svolte in parallelo. A seconda del paradigma di programmazione adottato, data parallel o control parallel, la suddivisione del lavoro sará basata su sottoinsiemi dei dati da elaborare oppure su sottoinsiemi delle funzioni da svolgere. L'algoritmo che si vuole parallelizzare viene in questo modo suddiviso in task elementari, definiti in modo arbitrario dal programmatore e che costituiscono la piú piccola frazione del lavoro complessivo che puó esser svolta in modo concorrente. In questa fase, che puó essere definita di decomposizione, é fondamentale evidenziare la concorrenza presente nell'algoritmo in modo da ridurre la porzione di codice sequenziale presente nel programma parallelo finale; la presenza di parti sequenziali nel programma parallelo imporrebbe infatti un limite allo speedup massimo ottenibile, a causa della legge di Amdahl citata nel paragrafo 1.2.

Dopo avere suddiviso il lavoro in task, é necessario assegnare i task ai processi. Risulta importante sottolineare la distinzione fra **processo** e **processore**: i processi sono un modello di astrazione utile per rappresentare un calcolatore parallelo svincolando la scrittura del programma dalla nozione di processore, che é invece una risorsa fisica di calcolo disponibile in un dato momento su di una certa macchina. Il numero dei processi é indipendente dal numero dei processori presenti, mentre il numero dei task in cui si é suddiviso il lavoro complessivo, costituisce un limite superiore al numero dei processi che possono essere in esecuzione contemporaneamente. In questa fase di assegnazione dei task ai processi é importante evitare che il carico di lavoro fra i processi sia sbilanciato e che quindi alcuni di essi rimangano inoperanti. L'assegnazione puó essere fatta una sola volta all'inizio del programma oppure variare dinamicamente durante l'esecuzione. Politiche di assegnazione troppo complesse, che richiedono molti calcoli o numerose comunicazioni e sincronizzazioni, possono portare ad un calo di prestazioni significativo. Viceversa metodi di assegnazione inadeguati al problema compromettono lo sfruttamento delle risorse in quanto non garantiscono che ogni processo sia sempre impegnato in una attivitá.

Per completare la scrittura del programma parallelo é necessaria una fase di orchestrazione in cui, sfruttando i meccanismi messi a disposizione dal linguaggio di programmazione, vengono stabiliti i modi con cui i vari processi accedono ai dati, si sincronizzano e comunicano tra loro. In questa fase diventa di fondamentale importanza la distinzione fra modello di programmazione *message passing* e quello *shared memory* in quanto, come descritto nel paragrafo 1.1, stabiliscono metodi fondamentalmente diversi per le interazioni fra i processi. Gli scopi di questa fase consistono nel ridurre il costo delle comunicazioni e delle sincronizzazioni, nel preservare per quanto possibile la localitá dei dati, e nel mantenere basso l'overhead dovuto alla gestione del parallelismo.

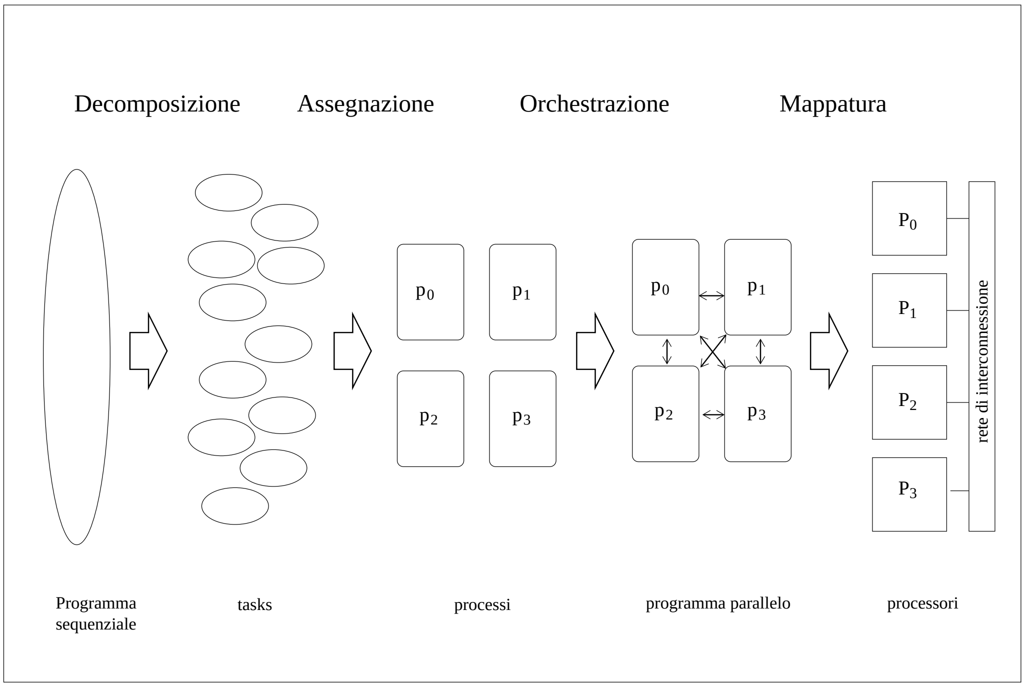


Figura 1.1: Fasi del procedimento di parallelizzazione e relazioni fra task, processi e processori.

Con l'orchestrazione si é sostanzialmente completata la scrittura del programma. Rimane da scegliere se far effettuare la mappatura dei processi sui processori al programma o lasciare tale compito al sistema operativo durante l'esecuzione del programma. Questa scelta é fortemente vincolata dal sistema operativo e dagli strumenti di programmazione utilizzati.

Una prima possibilitá é che il programma abbia la capacitá di scegliere quale processo far eseguire su di un certo processore e garantire questa assegnazione durante l'esecuzione, con il vantaggio di mantenere alta la localitá dell'accesso ai dati.

Una seconda possibilitá é che il sistema operativo controlli dinamicamente la mappatura dei processi sui processori, applicando l'usuale criterio di scheduling attuato in ambiente multitasking oppure estendendo le capacitá dello scheduler in modo da tenere conto dell'architettura multiprocessore. In quest'ultimo caso é utile, ad esempio, che lo scheduler agisca contemporaneamente su tutti i processi di una stessa applicazione (*gang scheduling*) e che ciascun processo sia assegnato in prevalenza ad uno stesso processore(*processor affinity*), in modo da sfruttare meglio la cache.

La maggior parte dei sistemi attuali si collocano fra questi due estremi ma in generale, anche se l'utente od il programmatore possono avere alcune prerogative di controllo sulla mappatura, il sistema operativo puó sempre intervenire per cambiare la mappatura dinamicamente, in base a criteri di efficienza della gestione delle risorse.

In figura 1.1 é riportata una descrizione schematica delle fasi di **decomposizione**, **assegnazione**, **orchestrazione** e **mappatura** appena descritte.

1.4 Supporti alla programmazione parallela

Per la realizzazione dei programmi sviluppati in questa tesi sono stati utilizzati tre distinti strumenti software:

- la libreria MPI
- l'API OpenMP
- l'API mp

Nei paragrafi succesivi verranno descritte le caratteristiche principali di questi strumenti e le differenze che li contraddistingono.

1.4.1 MPI (Message Passing Interface)

MPI é una libreria basata sullo standard definito in [15, 16] e sviluppata da un apposito comitato. É disponibile per i principali linguaggi di programmazione (C, C++, Fortran) e fornisce un insieme di funzioni che ben si adattano alla programmazione parallela in ambiente distribuito e su sistemi di tipo MIMD. La libreria MPI fornisce un insieme piuttosto esteso di primitive di comunicazione che includono comunicazioni uno-ad-uno, tutti-ad-uno, tutti-a-tutti, di tipo bloccante e non, sincrone o bufferizzate (a seconda dell'implementazione). Fornisce anche dei mezzi per definire contesti di comunicazione separati fra piú gruppi di processi e la possibilità di definire topologie "virtuali", per facilitare la denominazione dei processi di un gruppo o per agevolare il sistema run-time a mappare i processi sull'architettura hardware.

Esempi delle principali funzioni di comunicazione di tipo uno ad uno sono:

- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)` : questa funzione serve per inviare un messaggio di dimensioni *count* e memorizzato a partire dall'indirizzo *buf* al processo *dest*.

- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status )` È la funzione duale della precedente e serve a ricevere il messaggio.

Altre funzioni importanti, destinate a comunicazioni collettive sono invece :

- `int MPI_Bcast (void *buffer, int count,MPI_Datatype datatype, int root, MPI_Comm comm )` : questa funzione consente al processo *root* di inviare un messaggio a tutti i processi del gruppo (identificato da *comm*).

- `int MPI_Barrier (MPI_comm comm)` : é la primitiva utilizzata a sincronizzare mediante una barriera tutti i processi del gruppo *comm*.

- `int MPI_Gather (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm )` : questa funzione consente di raccogliere sul processo *root* porzioni di un array che si trovano su piú processi.

- `int MPI_Reduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm )` : questa funzione consente di effettuare un'operazione di *riduzione*. Tale operazione consiste nel porre in ingresso alla funzione *op* i dati presenti nel *sendbuf* di tutti i processi. Il risultato prodotto viene inviato al *recvbuf* del processo *root*.

Utilizzando questa libreria il programmatore ha il compito di stabilire fino nei minimi dettagli le azioni che i vari processi devono compiere e le sincronizzazioni necessarie.

Una della principali caratteristiche positive della libreria MPI é quella di essere portabile e scalabile in modo agevole: programmi realizzati con essa possono esser eseguiti su sistemi distribuiti formati da calcolatori eterogenei collegati in rete, cosí come su elaboratori multiprocessore di tipo shared memory.

1.4.2 OpenMP ed mp

OpenMP è un'API (Application Program Interface) comprendente una libreria di funzioni, un insieme di direttive di compilazione, e variabili d'ambiente. É stata sviluppata in tempi molto recenti (la prima versione risale al 1997) ed attualmente lo sviluppo é seguito da numerosi produttori fra i quali IBM, Silicon Graphics, Hewlett-Packard e Sun. É disponibile per i linguaggi C,C++ e Fortran, sia per sistemi operativi Unix che per WindowsNT.

Rispetto ad MPI ha lo svantaggio di esser utilizzabile solo su sistemi multiprocessori a memoria condivisa ed inoltre richiede un compilatore compatibile con tale standard. Ha peró il pregio di fornire al programmatore un'interfaccia di piú alto livello costituita dalle direttive di compilazione, le quali delegano al compilatore una parte significativa del lavoro di parallelizzazione. Inoltre, almeno teoricamente, l'uso di tali direttive, unitamente a quello della compilazione condizionale, permette di sviluppare un' unica versione del codice sorgente di un programma sia per la versione seriale sia per quella parallela.

Si puó sottolineare che l'uso di OpenMP non inibisce la possibilitá di usare MPI: é possibile utilizzare MPI per eseguire un programma su piú nodi di un sistema distribuito ed OpenMP per eseguire, con un ulteriore livello di parallelismo sugli eventuali nodi con architettura di tipo *shared memory multiprocessors* presenti, i singoli processi MPI.

É opportuno a questo punto, anche in considerazione della novitá dello standard, descrivere alcuni aspetti basilari della programmazione con OpenMP. Gli elementi base di un programma parallelo che fâ uso di OpenMP sono le *regioni parallele*: esse sono definite nel seguente modo (relativo al linguaggio C) :

```
#pragma omp parallel [<opzioni>]
 {
 <codice eseguito in parallelo da tutti i processi>
 }
 <codice seriale eseguito dal solo processo padre>
```

La direttiva *#pragma omp parallel* fa si che venga generato un certo numero di processi figli, i quali saranno terminati alla fine della regione. Tutti i dettagli relativi alla creazione ed alla gestione di basso livello dei processi sono lasciati al compilatore. Il numero di tali processi puó essere scelto a tempo di esecuzione dall'ambiente run-time (in modo statico per la singola regione e dinamico fra regioni distinte), dall'utente mediante il settaggio dell'opportuna variabile d'ambiente, dal programmatore mediante l'uso della funzione *omp_set_num_threads(int n)*. In ogni caso l'assegnazione dei processi ai processori é gestita dal sistema operativo in base al criterio di sfruttare tutti i processori liberi. Per questo motivo l'esecuzione in ambiente multiutente puó influenzare profondamente i risultati delle misure di speedup in quanto non é noto il numero di processori che hanno effettivamente eseguito il programma parallelo durante tutta l'esecuzione.

Fra le opzioni possibili, le piú utili sono quelle che permettono un utilizzo locale o condiviso delle variabili, mediante la sintassi:

```
#pragma omp parallel private(<variabili locali>) shared(
<variabili condivise>)
```

Per le variabili specificate come *private*, viene creata una copia della variabile accessibile dal solo processo locale, copia che viene cancellata quando il processo termina. Per le variabili condivise l'accesso é possibile da tutti i processi, ed i problemi di mutua esclusione vengono gestiti in modo trasparente al programmatore.

Al fine di ridurre l'overhead associato alla creazione e distruzione dei processi é opportuno ridurre al minimo il numero di regioni parallele distinte: idealmente si vorrebbe un'unica regione parallela per tutto il programma.

All'interno della regione parallela si possono poi utilizzare specifici costrutti, uno dei quali é il ciclo for parallelo:

```
#pragma omp for [<opzioni>]
 for(.....)
 <corpo del ciclo for>
```

Senza nessuna opzione il compilatore e l'ambiente run-time si occupano di partizionare i cicli fra i vari processi, altrimenti é possibile scegliere la dimensione dei blocchi, il tipo di scheduling, variabili *private* o *shared* specifiche per il costrutto for, ed altre opzioni. Fra queste opzioni, particolarmente importante é quella che permette di effettuare la riduzione:

```
#pragma omp for reduction(<operazione>:  <variabile risultato>)
```

Altre direttive molto utili sono:

```
#pragma omp for barrier
```

`#pragma omp critical` { ... }

`#pragma omp master` { ... }

Queste ultime permettono di fissare *barriere* per sincronizzare tutti i processi, di definire regioni "critiche" che vengono eseguite da un solo processo alla volta, oppure regioni eseguite solo dal processo padre.

Identica struttura generale (basata su regioni parallele) ed identiche funzionalità di base rispetto ad OpenMP sono fornite da **mp**, un'API proprietaria sviluppata da Silicon Graphics per i suoi sistemi multiprocessore. Essa puó essere considerata un sottoinsieme di OpenMP in quanto alcune capacitá di quest'ultima non trovano un corrispondente in mp. Anche la sintassi risulta estremamente simile fra le due come si può vedere dalla tabella 1.1, dove sono mostrati a titolo di confronto alcuni esempi di funzioni definite con **mp** e con **OpenMP**.

OpenMP	**mp**
`#pragma omp parallel`	`#pragma parallel`
`#pragma omp for`	`#pragma pfor`
`#pragma omp barrier`	`#pragma synchronize`
`#pragma omp critical`	`#pragma critical`
`shared( <var list> )`	`shared( <var list> )`
`private( <var list> )`	`local( <var list> )`
`omp_set_num_threads(int n)`	`mp_set_numthreads(int n)`
`omp_get_num_threads(void )`	`mp_numthreads(void )`

Tabella 1.1: Confronto delle sintassi di OpenMP e di mp

1.5 Il sistema Onyx2 Infinite Reality

La Silicon Graphics Onyx2 Infinite Reality, su cui sono state svolte le attivitá di sperimentazione di questa tesi, é un versione della famiglia di elaboratori Origin 2000 con accresciute capacitá di elaborazione grafica 2D-3D.

La Onyx2 é una macchina multiprocessore basata su architettura ccNUMA (cache coherence Non Uniforme Memory Access) ed é stata progettata per garantire la massima scalabilitá e modularitá. Particolari sforzi sono stati fatti per ottenere una gestione della memoria efficiente: a tal fine é implementato il sistema di tipo DMS (Distributed Shared Memory) con consistenza della cache mantenuta mediante un protocollo *directory based*.

Date le notevoli caratteristiche di scalabilitá e la condivisione della memoria, la Onyx2 puó esser definita come sistema S^2MP (Scalable Shared-memory multiprocessing).

1.5.1 Architettura di sistema

Due degli aspetti principali di questo sistema sono la scalabilitá e modularitá: l'elemento base é costituito da un nodo di calcolo formato da 2 processori, una memoria locale ampia fino a 4 GByte, ed i necessari dispositivi di interfaccia con la rete di interconnessione e con i dispositivi di I/O. Una configurazione di partenza puó essere espansa aggiungendo dei nodi fino ad un massimo di 512 nodi complessivi.

Sul nodo elementare oltre ai due processori, la memoria locale con la corrispondente directory memory utilizzata per mantenere la consistenza della cache, é posto un chip chiamato Hub che gestisce le comunicazioni fra processori, memoria locale, ed i router della rete di interconnessione.

La rete di interconnessione realizza una topologia ad ipercubo. Piú precisamente é definita come "bristled hypercube": "bristled" poiché due nodi condividono uno stesso router; per le macchine con piú di 32 nodi la topologia diventa "bristled fat hypercube". In figura 1.2 sono mostrati esempi relativi a 32, 64 e 128 processori.

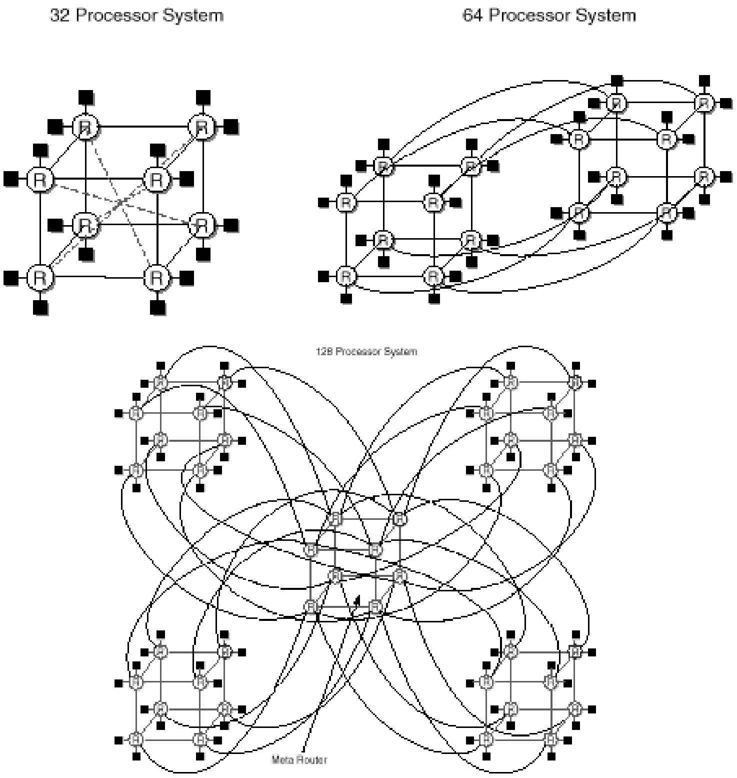


Figura 1.2: Topologia di interconnessione nei casi di 32, 64 e 128 processori.

Le potenzialitá di quest'architettura sono elevate, in quanto sono previsti fino a 512 nodi di calcolo per un totale di 1024 processori ed 1TB di memoria. I nodi elementari sono realizzati su una singola *printed circuit board* di $16''x11''$, descritta mediante un diagramma a blocchi in figura 1.3. I due processori del nodo operano in modo indipendente condividendo lo stesso bus fisico connesso ad un unico hub. Questa caratteristica distingue il sitema Onyx da altre architetture ccNUMA in cui i nodi sono costituiti da gruppi SMP (Symmetric Multiprocessor): tale scelta riduce la latenza della memoria locale (e di conseguenza anche di quella remota) in quanto un bus che supporta al massimo due processori puó funzionare a frequenze superiori di quelle possibili nel caso di un numero superiore di processori.

Il sistema DSM gestisce l'indirizzamento globale della memoria ed anche del sottosistema di I/O: le physical I/O operations (PIOs) possono esser reindirizzate da ogni processore ad ogni dispositivo di I/O, e viceversa i dispositivi di I/O possono

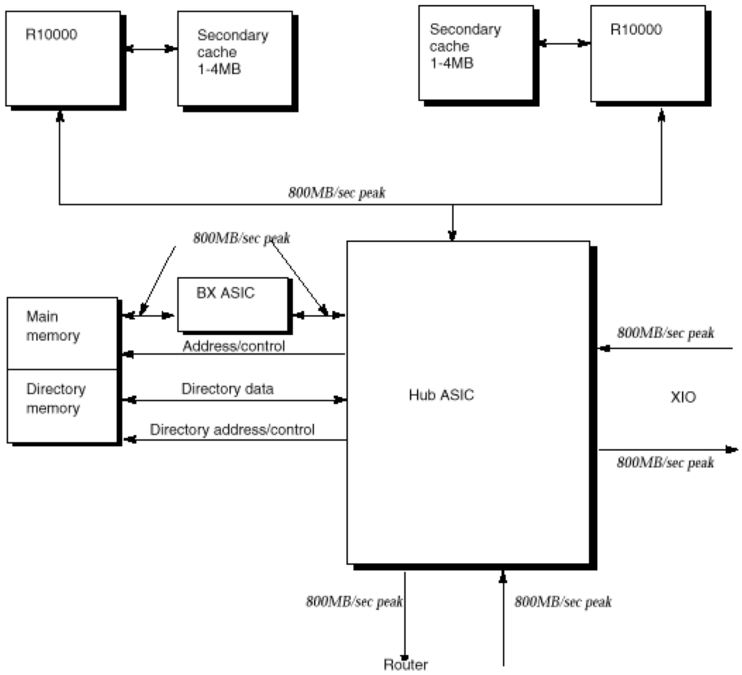


Figura 1.3: Schema del nodo di base dell'architettura.

accedere a tutta la memoria (per compiere operazioni di tipo DMA, *Direct Memory Access*) e non solo a quella locale del nodo cui sono connessi.

Un' altra caratteristica volta a migliorare gli accessi alla memoria remota é la realizzazione mediante strumenti hardware e software di un'efficiente *page migration*. La page migration e la page replication hanno lo scopo di rendere locali la maggior parte di accessi alla memoria: la Onyx sfrutta dei contatori hardware di riferimento per ogni pagina di memoria fisica ed un engine di copia di blocchi di dati in grado di copiare dati quasi alla velocitá di picco consentita dalla memoria stessa.

A completamento di questa sintetica descrizione dell'architettura vanno citate la rete di interconnessione ad alte prestazioni ed un completo set di primitive di sincronizzazione.

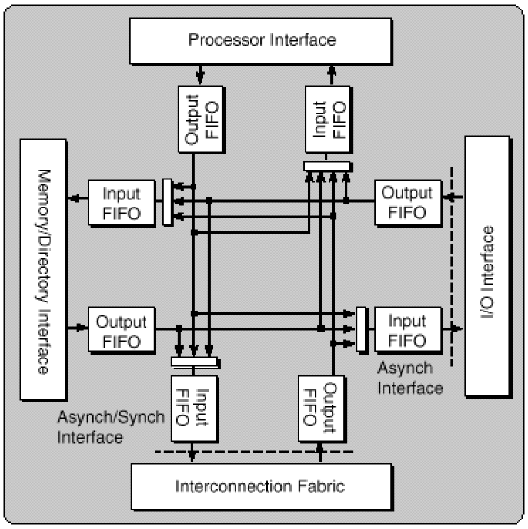


Figura 1.4: Schema dell' Hub.

1.5.2 La rete di interconnessione

Gli elementi base della rete sono dei router realizzati ciascuno con un chip realizzato da SGI chiamato SPIDER. Tali chip sono caratterizzati da 6 coppie di link unidirezionali, tabelle di routing programmabili via software, gestione di 256 livelli di prioritá, rivelazione dell'errore mediate CRC e ritrasmissione dei pacchetti in caso di errore, gestione di 4 canali virtuali per ogni link fisico, ognuno avente un quarto della banda del link. Due dei canali virtuali sono dedicati alle *network transaction* (uno per le richiesta e l'altro per la risposta), mentre gli altri due possono essere usati per l' I/O, per eseguire transaction ad elevata prioritá o per superare periodi di congestione della rete.

La connessione fra la rete ed il nodo di calcolo avviene grazie all'Hub schematizzato in figura 1.4: l'Hub é sicuramente uno degli elementi piú critici di tutta l'architettura in quanto attraverso di esso transitano tutti gli accessi alla memoria, tutte le comunicazioni internodo ed anche le operazioni di I/O; l'hub gestisce anche il protocollo per la consistenza della memoria cache. L'Hub é realizzato con un circuito integrato da 500k porte logiche in tecnologia CMOS da $0.5\mu m$: tale circuito realizza un crossbar switch a quattro porte bidirezionali, ciascuna caratterizzata da

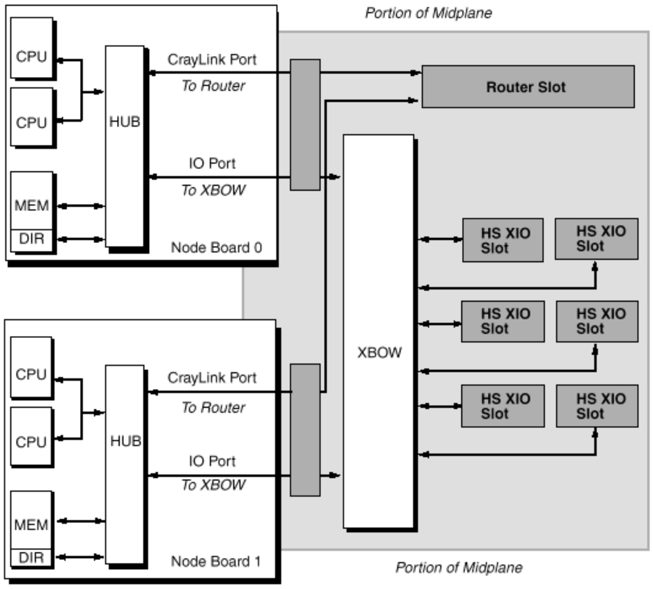


Figura 1.5: Collegamento di un Xbow al sistema.

una banda di picco complessiva di 1.6 GB/s (800MB/s in entrambe le direzioni).

Due nodi condividono un'interfaccia di I/O chiamata Xbow che connette gli Hub dei due nodi alle periferiche di I/O. L'Xbow é costituito da un chip ASIC che realizza una crossbar ad 8 porte, simile al router ma con una bufferizzazione semplificata e con 8 porte anziche le 6 del router. Delle otto porte, due sono collegate agli Hub di due nodi e le rimanenti sei sono usate per l' I/O: per queste ultime sono disponibili numerose interfacce in tecnologia ASIC per collegarle a dispositivi di tipo PCI, SCASI, Ethrnet, ATM ed altri ancora. Un'altra caratterisctica del chip Xbow é la possibilitá di riservare della banda per dispositivi ad altre prestazioni o con esigenze di real-time come le schede grafiche, alcune delle quali (come la Infinite Reality Graphics) possono essere connesse direttamente alle porte dell'Xbow senza dover prima passare per un'interfaccia PCI. In figura 1.5 é descritta la connessione tra un Xbow ed il resto del sistema.

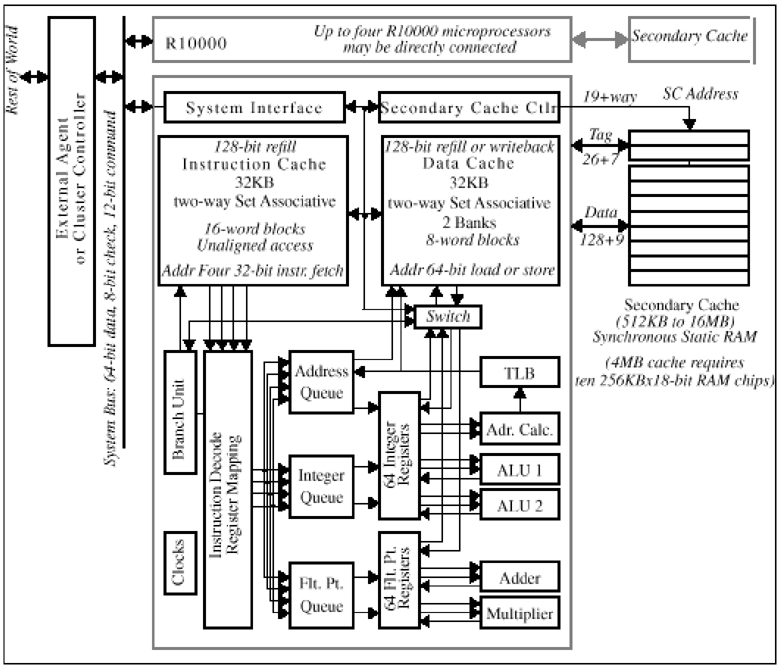


Figura 1.6: Schema a blocchi del microprocessore R10000.

1.5.3 Il microprocessore

Il microprocessore impiegato sulla Onyx2 é lo R10000 prodotto da MIPS Technologies. É un microprocessore superscalare in grado di eseguire fino a 4 istruzioni per ciclo; é realizzato in tecnologia CMOS da $0.35\mu m$, supporta l'esecuzione fuori ordine delle istruzioni, un meccanismo di predizione dei salti, cinque unitá funzionali indipendenti, bus dedicato alla cache di secondo livello a 128 bit, tre code separate per operazioni intere, floating point e di indirizzamento. La cache di primo livello é formata da 32 KB di memoria dati e 32 KB di memoria per le istruzioni, mentre quella di secondo livello, che non richiede circuiteria aggiuntiva ha una dimensione compresa tra 512 Kb e 16 MB. In figura 1.6 é schematizzato il microprocessore. Si possono notare in basso a destra le cinque unitá di calcolo indipendenti: due per operazioni intere, due per operazioni in virgola mobile (una per la moltiplicazione ed una per la somma), ed una per il calcolo degli indirizzi.

1.5.4 Caratteristiche dell' Onyx2 Infinite Reality

Le caratteristiche harware specifiche del sistema su cui sono stati testati i programmi realizzati in questa tesi sono le seguenti:

- **Microprocessori**: 8 MIPS R10000 a 195 MHz
- **Memoria RAM**: 4096 Mbyte
- **Spazio su disco**: 60Gbyte
- **Scheda grafica**: Infinite Reality 2 con 16Mbyte
- **Scheda di rete**: Integral fast Ethernet (100Mbit/s)
- **processore audio**: Iris version RAD 7.0,number 1
- **Costo**: circa 600 milioni di lire

I principali software di sistema istallati sono:

- **Sistema operativo**: Irix 6.4
- **Compilatori C/C++**: MIPSpro Compilers Version 7.30, gcc version 2.8.1
- **Scheduler per l' esecuzione batch**: LSF 3.2.2 di Platform Computing Corporation
- **MPI**: MPI 3.1.1.0 (che implementa lo standard MPI 1.2)

Capitolo 2

Parallelizzazione di metodi proiettivi

2.1 Soluzione parallela di sistemi lineari

Nel capitolo precedente si sono presi in considerazioni gli obbiettivi di un programma parallelo, alcune metriche utili per valutarne le prestazioni, e si é descritta una metodologia del tutto generale per affrontare la parallelizzazione di un algoritmo. É ora possibile concentrare l'attenzione sul problema specifico della risoluzione parallela di sistemi di equazioni lineari mediante metodi iterativi.

La risoluzione di sistemi di equazioni lineari non é limitata all'argomento specifico trattato in questa tesi, relativo alla soluzione di reti di Petri, ma é un problema basilare in numerose applicazioni tecniche e scientifiche. Molti programmi di simulazione dinamica e circuitale, di modellizzazione di fenomeni fisici, di studio matematico, richiedono la risoluzione di sistemi di equazioni lineari.

In questa tesi si é studiato il caso specifico di risoluzione mediante metodi proiettivi di sistemi sparsi di grandi dimensioni. Entrambe queste caratteristiche sono normali quando il sistema da risolvere trae origine da un rete di Petri: la dimensione dello spazio degli stati di una rete di Petri puó facilmente portare a sistemi con centinaia di migliaia di equazioni, e le relative matrici risultano fortemente sparse.

Rispetto al caso generale in cui la matrice che descrive il sistema é densa la differenza fondamentale consiste nella necessitá di usare strutture dati piú complesse al fine di risparmiare spazio in memoria ed evitare il calcolo di operazioni inutili. Fra i vari formati (descritti in [4] ed in [1]) utilizzati normalmente per memorizzare matrici sparse ricordiamo il *Coordinate format*, il *Compressed sparse row format*, il *Diagonal storage format*, l' *Ellpack-Itpack format*. La struttura dati scelta é descritta in dettaglio in §3.3 ed é caratterizzata dal mantenere un'informazione completa sugli indici di riga e di colonna degli elementi della matrice (come nel *Coordinate format*) ma anche sui punti di inizio delle singole righe (come nel *Compressed sparse row format*). La scelta di tale struttura é stata fatta inizialmente per consentire una costruzione efficiente del precondizionatore, in seguito si é rivelata in grado di assolvere alle necessitá specifiche dei programmi paralleli sviluppati.

Indipendentemente dall'algoritmo iterativo scelto per la soluzione di un sistema di equazioni lineari, si possono individuare alcune operazioni fondamentali di algebra vettoriale la cui composizione in un dato ordine da origine ad uno specifico algoritmo. Queste operazioni fondamentali possono essere parallelizzate in maniera indipendente adottando un modello di tipo *data parallel*, giá descritto nel paragrafo 1.1.

Questo approccio é suggerito dalla caratteristiche del problema: l'applicazione di un qualsiasi algoritmo iterativo comporta l'esecuzione di un insieme ridotto di operazioni floating-point (in genere una moltiplicazione seguita da una somma o sottrazione) su di un numero elevatissimo di dati ed inoltre esiste un rapporto di causalitá fra quasi tutti i passi elementari di un'iterazione. Ad esempio, con il BiCGStab, il calcolo del vettore $t = As$ è seguito da $\omega_i = (t,s)/(t,t)$ che a sua volta è seguito da $x_i = x_{i-1} + \alpha p_i + \omega_i s$: le tre operazioni (come sostanzialmente anche tutte le altre) non possono esser eseguite in modo concorrente da tre processi perché il risultato dell'una determina quello della successiva. Al contrario ciascuna é suddivisibile in piú operazioni dello stesso tipo che operano su un sottoinsieme dei dati globali.

Approfondendo l'analisi del problema si possono descrivere le operazioni fondamentali di algebra vettoriale che possono essere parallelizzate agevolmente con un approccio di tipo questo tipo:

1. **Prodotto scalare**. Il calcolo del prodotto scalare di due vettori densi é comunemente usato all'interno di metodi iterativi di risoluzione di sistemi lineari. Spesso esso contribuisce a determinare la complessitá delle comunicazioni e la scalabilitá dell'intero algoritmo. Si ricorda che dati i due vettori $X = (x_1, x_2, ...x_N)$ e $Y = (y_1, y_2, ...y_N)$ il prodotto scalare $s = X \cdot Y$ è dato da $s = \sum_{i=1}^{n} x_i \; y_i$.

   Se i vettori da moltiplicare hanno dimensione N e vengono ripartiti uniformemente su P processori, ogni processore svolge N/P moltiplicazioni e $(N/P)-1$ addizioni. I P risultati prodotti dai processori devono essere accumulati per ottenere il risultato finale. Tale operazione viene normalmente chiamata *riduzione*. La fase finale di composizione dei risultati parziali puó trarre un notevole beneficio dall'eventuale presenza di hardware dedicato ad operazioni di riduzione, o perlomeno da primitive software ottimizzate per comunicazioni collettive di questo tipo.

   In figura 2.1 viene schematizzato la modalitá di esecuzione parallela del prodotto scalare $X \cdot Y$: ogni processore calcola un risultato parziale relativo ad una porzione del vettore; i veri risultati parziali vengono raccolti e combinati su di un nodo (il nodo P_0 nella figura). Alcune librerie di calcolo parallelo forniscono oltre alla primitiva *reduce* mostrata nell'esempio una primitiva di tipo *all-reduce* in cui il risultato finale viene ridistribuito a tutti i processori.

   Operazioni del tutto analoghe, costituite da un certo numero di calcoli locali e successiva riduzione, sono realizzabili per il calcolo della **norma** euclidea di un vettore e della **somma delle componenti** di un vettore, operazione quest'ultima necessaria in caso di normalizzazione del vettore soluzione.

   Si fa notare che all'operazione di riduzione é implicitamente associata una *barriera* che garantisce la sincronizzazione fra i vari processi.

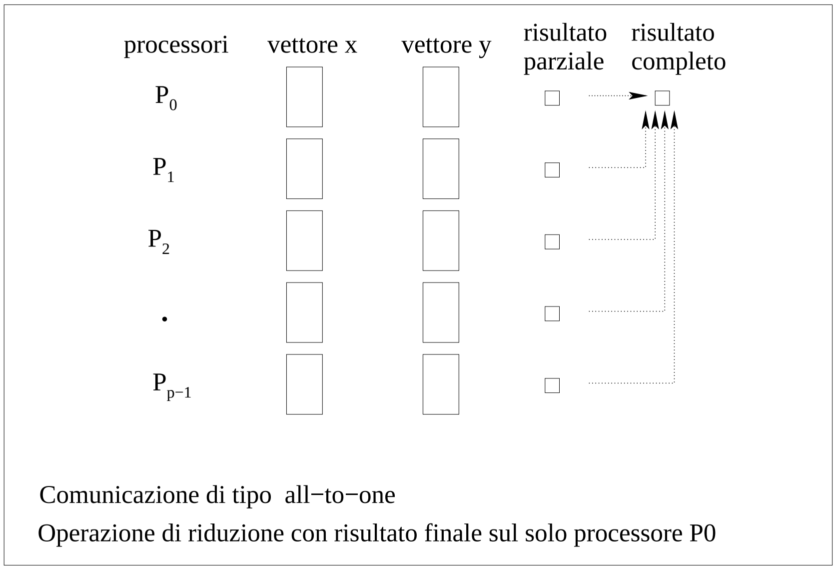


Figura 2.1: Descrizione di una tipica operazione di riduzione applicata al calcolo vettoriale

2. **SAXPY**. Tale acronimo stá per *Simple ax plus y* ed indica la somma o sottrazione del vettore y con il vettore x moltiplicato per la costante scalare a. Se la costante a é disponibile su tutti i processori e se i due vettori vengono ripartiti uniformemente su P processori, ogni processore svolge N/P moltiplicazioni e (N/P) addizioni, e non sono necessarie comunicazioni e sincronizzazione fra i vari processori.

3. **Moltiplicazione matrice sparsa per vettore**. É probabilmente l'operazione piú critica per la parallelizzazione di un metodo iterativo di soluzione di sistemi lineari, sia perché richiede il numero maggiore di operazioni in virgola mobile, sia perché puó richiedere una quantitá non trascurabile di comunicazioni interprocessore.

   Il caso peggiore si ha quando la matrice sparsa é di tipo non strutturato, ovvero la posizione degli elementi non nulli non rispecchia nessuna struttura nota, quali ad esempio quelle delle matrici n-diagonali, a bande, simmetriche. Questo caso é proprio quello che si presenta per le matrici trattate in questa tesi. Ipotizziamo che la matrice di N righe ed NZ elementi non nulli sia distribuita

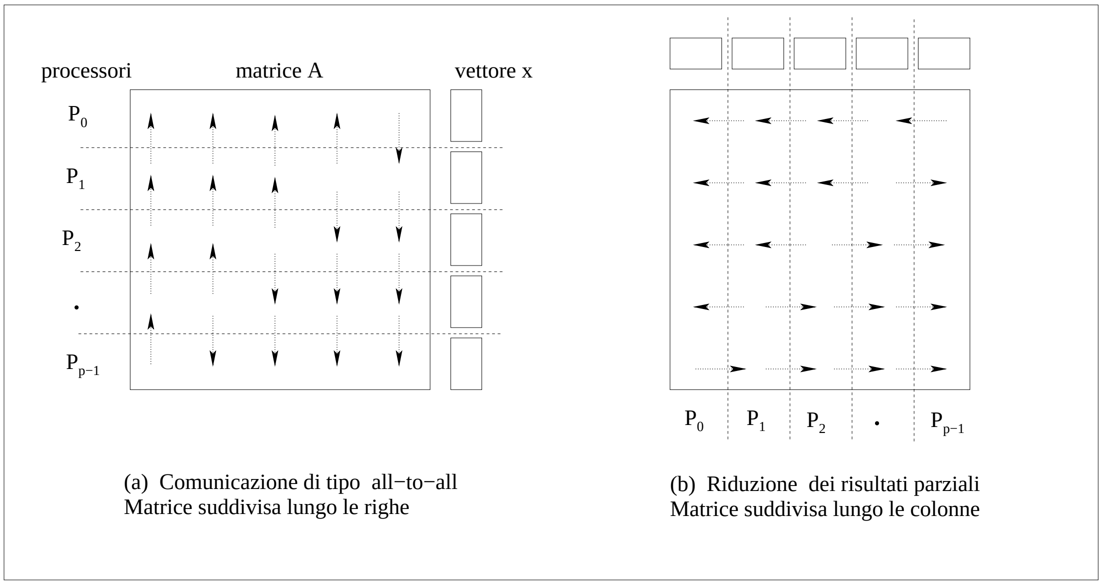


Figura 2.2: Tipi di comunicazione nel calcolo del prodotto matrice sparsa per vettore

sui processori secondo gruppi di N/P righe, come mostrato in figura 2.2 (a), e che lo stesso valga per il vettore di dimensione N. Una tale ripartizione dei dati é vantaggiosa solo se gli elementi non nulli della matrice sono uniformemente distribuiti fra le righe, caratteristica che risulta verificata con buona approssimazione per le matrici considerate in questa sede.

Dalla definizione di prodotto fra matrice e vettore si deduce che l'intero vettore deve essere accessibile ad ogni riga, perché non sono noti a priori gli elementi nulli della riga: é perció necessario far precedere alla fase di calcolo una fase di comunicazione da tutti a tutti(*all-to-all-broadcast*) in cui le porzioni locali del vettore vengono rese disponibili a tutti i processori. Supponendo che gli elementi non nulli abbiano una distribuzione uniforme e quindi ogni riga abbia NZ/N elementi, si puó concludere che ogni processore debba calcolare (NZ/P) somme e moltiplicazioni.

Se si definisce $m = NZ/N$ il numero medio di elementi per ogni riga, ricordando le definizioni di t_s e t_w date in §1.2 ed utilizzando il risultato ottenuto in [4] che mostra come su una topologia ad ipercubo la comunicazione di tipo all-to-all-broadcast descritta richieda un tempo $t_s logP + t_w N$, é possibile modellizzare il tempo totale necessario per il calcolo del prodotto matrice per vettore con l'equazione:

$$T_p = t_c m \frac{N}{P} + t_s logP + t_w N \quad (t_c\text{=tempo per una moltiplicazione e somma}) \tag{2.1}$$

da confrontarsi con il tempo di esecuzione seriale $T_s = t_c\ m\ N$. L'equazione appena citata non ha tanto lo scopo di offrire una stima sui tempi reali, ma quello di prestarsi alla seguente osservazione: il costo computazionale del prodotto matrice vettore parallelo é $\Theta\ (N)$ ([1]) cosí come quello seriale. Una tale realizzazione non porta ad una riduzione asintotica del tempo di esecuzione ed in base alle definizioni del paragrafo 1.2 é da considerarsi di costo non ottimale e non scalabile.

La situazione non migliora se si partiziona la matrice lungo le colonne come mostrato in 2.2 (b). In questo caso non sono necessarie comunicazioni preliminari al calcolo dei singoli prodotti e somme; peró se si vuole che il vettore risultante sia memorizzato nello stesso formato del vettore iniziale, tutti i risultati parziali relativi alla i-esima riga devono essere accumulati sul processore relativo all'i-esima colonna (che corrisponde all'i-esima componente del vettore iniziale). Sono perció necessarie N operazioni di riduzione ciascuna coinvolgente N/P dati: il costo complessivo delle comunicazioni risulta identico al caso precedente.

4. **Soluzione del sistema** $Mz = r$. Questo tipo di operazione risulta necessaria solo se il metodo iterativo é precondizionato. Nel capitolo 3 si vedrá in dettaglio il caso di precondizionatore basato sulla fattorizzazione incompleta di tipo ILU in cui $M = LU$ dove L ed U sono matrici triangolare inferiore e superiore rispettivamente. Il sistema $LUz = r$ puó venire risolto mediante

[1] La funzione $\Theta()$ é usata per confrontare l'andamento asintotico di due funzioni. Formalmente é definita come segue: f(x)=$\Theta(g(x))$ se e solo se, data una coppia di costanti c_1 e c_2 con $c_2 > c_1 > 0$, esiste $x_0 \geq 0$ tale che: $c_1 g(x) \leq f(x) \leq c_2 g(x)$ per ogni $x > x_0$.

la soluzione successiva dei sistemi triangolari $Ly = r$ ed $Uz = y$. Nei casi trattati L ed U non godono di proprietá particolari tali da consentire un'efficiente parallelizazzione del calcolo, e le singole equazioni dei sistemi triangolari devono essere risolte una dopo l'altra in modo rigidamente sequenziale.

2.2 Realizzazione: GMRES precondizionato e parallelo

Il primo algoritmo di cui si é sviluppata una versione parallela é il GMRES(m) con precondizionatore di tipo ILU0. La descrizione dell'algoritmo seriale é riportata nella sezione successiva, mentre la costruzione del precondizionatore e la sua integrazione nell'algoritmo sono descritti nel capitolo 3. Sono state sviluppate due versioni parallele, una basata su MPI ed una su OpenMP.

2.2.1 L'algoritmo seriale

Il GMRES (Generalized Minimum RESidual) é un metodo proiettivo molto efficace per la risoluzione di sistemi lineari non simmetrici di grandi dimensioni. É stato sviluppato negli anni '80 e risulta essere una delle numerose generalizzazioni del metodo del gradiente coniugato (CG). Si é utilizzata la variante iterativa del metodo, chiamata GMRES(m) [1, 14, 11], in cui il parametro m é fissato (ad esempio in base alla disponibilitá di memoria sul calcolatore) e l'algoritmo viene fatto ripartire ogni m passi utilizzando come approssimazione iniziale $x_0 = x_m$. La descrizione di questo algoritmo, comprendente l'uso di un generico precondizionatore M é la seguente:

1. Inizio: si scelga x_0 e si calcoli:
   $r_0 = b - M^{-1}Ax_0$ e
   $v_1 = r_0/\|r_0\|$.

2. Ciclo: for $j = 1, 2, \ldots, m$ do:

- $h_{i,j} = (M^{-1}Av_j, v_i), i = 1, 2, \ldots, j,$
- $\hat{v}_{j+1} = M^{-1}Av_j - \sum_{i=1}^{j} h_{i,j}v_i$,
- $h_{j+1,j} = \|\hat{v}_{j+1}\|$, e
- $v_{j+1} = \hat{v}_{j+1}/h_{j+1,j}$.

3. Calcolo della soluzione approssimata: $x_m = x_0 + V_m y_m$
dove y_m minimizza: $\|\beta e_1 - \bar{H}_k y\|$, dove $y \in R_m$ e $\beta = \|r_0\|$.

4. Riparti:

- si calcoli $r_m = b - M^{-1}Ax_m$;
- se la condizione di convergenza è soddisfatta stop
altrimenti poni $x_0 = x_m, v_1 = r_m/\|r_m\|$ e ritorna al passo 2.

2.2.2 GMRES parallelo con MPI

Il programma *message passing* realizzato é formato da due soli tipi di processi, descritti in modo semplificato nella tabella 2.1:

- un processo master che gestisce le operazioni di I/O, stabilisce inizialmente ed una volta per tutte la ripartizione del carico di lavoro fra i vari processi, esegue le parti di codice non parallelizzate (costruzione del precondizionatore, risoluzione dei sistemi triangolari piú alcune funzioni di costo trascurabile)
- N-1 processi slave (con N che puó variare da 1 ad un valore convenzionale N_MAX_PROC) che eseguono esclusivamente il codice parallelizzato. Il codice é identico a quello del master tranne che per l'assenza delle parti specifiche di esso. Il vantaggio é che i vari slave sanno cosa fare (il tipo di calcolo e di comunicazione) senza dover attendere ordini dal master: l'overhead dovuto al controllo del parallelismo è molto ridotto in quanto risulta sufficiente un unico messaggio di controllo per ogni iterazione, necessario per comunicare l'eventuale fine dell'esecuzione.

pseudocodice processo Master	pseudocodice processi slave
Initialize_MPI();	Initialize_MPI();
f_parse_args();	
A = f_read_matrix();	
M = f_build_precond();	
f_setup_parall_work();	f_setup_parall_work();
/* inizio algoritmo*/	/* inizio algoritmo*/
v=f_matXvet_parall(A,x);	v=f_matXvet_parall(A,x);
r=f_solve_triang(M,v);	
beta=f_norm_parall(r);	beta=f_norm_parall(r);
v=f_vetXcost_parall(r,1/beta);	v=f_vetXcost_parall(r,1/beta);
do	**do**
forj:=1 **to** m	**for**j:=1 **to** m
f_projection_step(H,V,A,v_1,j);	f_projection_step(H,V,A,v_1,j);
g=f_QR_decomp(Q,R,H);	
y=f_solve_triang(R,g);	
v=f_matXvet_parall(V,y);	v=f_matXvet_parall(V,y);
x=f_vet_sub_vet_parall(x_0,v);	x=f_vet_sub_vet_parall(x_0,v);
v=f_matXvet_parall(A,x);	v=f_matXvet_parall(A,x);
r=f_solve_triang(M,v);	
beta=f_norm_parall(r);	f_norm_parall(r);
if(beta<precision) **then** end=TRUE	
else end=FALSE;	
f_Broadcast(end);	f_Broadcast(end);
while(end=TRUE);	**while**(end=TRUE);
f_Output_result(x);	
return;	

Tabella 2.1: Struttura esemplificativa del programma basato su MPI con stesse notazioni di matrici e vettori usate per la descrizione dell'algoritmo seriale.

Inizialmente il master deve distribuire ai vari slave informazioni riguardo ai compiti loro assegnati, che corrispondono ad una porzione dell'insieme dei dati. Piú precisamente ogni processo riceve una struttura dati contenente le seguenti informazioni:

- due dati globali: dimensione vettori e valore del parametro m.
- due dati locali per le operazioni sui vettori: prima riga e numero di righe della porzione di vettore assegnata al processo.
- due dati locali per il prodotto matrice vettore: indice del primo elemento e numero di elementi (riferiti alla struttura dati descritta in dettaglio in §3.3) della porzione di matrice assegnata.

Le varie funzioni parallele provvedono poi a gestire le comunicazioni e sincronizzazioni richieste dall'algoritmo.

2.2.3 GMRES parallelo con OpenMP

Anche se il modello di parallelizzazione utilizzato è ancora di tipo *data parallel*, la struttura del programma é sostanzialmente diversa da quella basata su MPI. Nel codice sorgente non vi é alcuna distinzione esplicita fra un processo master e processi slave: il codice seriale viene parallelizzato introducendo le varie direttive di compilazione laddove sono necessarie.

Anzichè riportare una descrizione mediante pseudocodice della struttura del programma, è parso piú utile descrivere la realizzazione, basata su OpenMP, di alcune funzioni specifiche, riportate nelle tabelle 2.2 e 2.3.

La struttura dati necessaria per il controllo del parallelismo (chiamata *Block_type block* nello pseudocodice), è piú semplice di quella utilizzata con MPI, in quanto servono solo due dati per il prodotto matrice vettore: indice del primo elemento (indicato con *block.first_element*) e dell'ultimo elemento (indicato con *block.last_element*) di ogni porzione in cui é stata suddivisa la matrice.

Pseudocodice della funzione per il calcolo del prodotto matrice sparsa per vettore

```
procedure matXvet(double M[], integer M_row[], integer M_col[],
double v[], double result[],Block_type block[])
begin
   integer i,process, first, last;

#pragma omp parallel for shared(result) private(i)
   for i:=1 to block[0].non_zero_elements
      result[ i ] :=0.0;
#pragma omp parallel shared(v, result, M, M_row, M_col, bloc) private(last, first, i)
   begin
      processes = omp_get_thread_num();
      first = block[process].first_element;
      last = block[process].last_element;
      for i:=first to last
         result[ M_row[i] ] := result[ M_row[i] ] + M[i] * v[ M_col[i] ];
   end
end
```

Tabella 2.2: Pseudocodice della funzione matrice per vettore basata su OpenMP

Naturalmente, dato il modello di programmazione a memoria condivisa, tutti i dati del problema sono immediatamente disponibili a tutti i processi, e la fase inizializzazione del parallelismo serve solo a determinare i dati accenati.

2.3 Realizzazione: BiCGStab parallelo

Il motivo della scelta del BiCGStab come secondo algoritmo da parallelizzare é il seguente: dai risultati ottenuti con il GMRES precondizionato e parallelo, riportati nel capitolo 4 si é osservato come l'uso del precondizionatore, seppure in grado di migliorare notevolmente le prestazioni della versione seriale, compromette in una

Pseudocodice della funzione per il calcolo della norma euclidea di un vettore

procedure norma(double *result,double v[],Block_type block[])
begin
  integer i;
  double sum:=0.0;
 	
#pragma omp parallel for shared(v) private(i) reduction(+:sum)
  for i:=1 **to** block[0].non_zero_elements
    sum := sum + v[i]*v[i];
  (*result):= sqrt(sum) ;
end

Tabella 2.3: Pseudocodice della funzione di calcolo della norma euclidea basata su OpenMP

certa misura le possibilitá di ottenere un programma parallelo efficiente e con un buon speedup. Si é allora pensato di parallelizzare un algoritmo non precondizionato, al fine di verificare l'effettiva validitá del modello di parallelizzazione utilizzato, e per poter valutare se l'assenza del precondizionatore é in grado di migliorare in modo significativo il livello di concorrenza sfruttabile per la parallelizzazione. Inoltre, come si é osservato nel capitolo 1, in alcuni casi non é vantaggioso parallelizzare il piú veloce programma seriale disponibile per compiere un certo compito, ma é meglio parallelizzare quello che evidenzia il maggior livello di concorrenza e puó sfruttare maggiormente un'architettura parallela.

La scelta é caduta sul BiCGStab anche perché é stato possibile sfruttare i risultati relativi alla versione seriale ottenuti in [11, 12], in base ai quali tale algoritmo si é dimostrato piuttosto efficiente, pur essendo piú lento del GMRES con precondizionatore ILU0.

All'inizio di questo lavoro di tesi é stato implementato un ulteriore algoritmo, il TFQMR che, almeno in certi ambiti specifici, sembra unire la buona velocitá degli algoritmi basati sul metodo del gradiente coniugato come il CG-S ed il BiCGStab, all'ottima affidabilitá mostrata dal GMRES. Nell'appendice A é descritta piú in dettaglio l'implementazione di tale algoritmo. I risultati ottenuti (descritti nel capitolo 4) hanno evidenziato come, nei casi trattati in questa sede, tale algoritmo non sia risultato molto efficiente sia in termini di velocitá che di affidabilitá nel raggiungimento della soluzione. Anche per questo si é preferito utilizzare il BiCGStab piuttosto che il TFQMR come secondo algoritmo da parallelizzare .

2.3.1 L'algoritmo seriale

Il BiCGStab come GMRES, é un metodo proiettivo di soluzione di sistemi lineari non simmetrici, ed é stato sviluppato come evoluzione del metodo del gradiente coniugato (CG). L'algoritmo BiCGstab è il seguente:

1. Inizio:
   - si sceglie x_0 vettore iniziale e si calcola $r_0 = b - Ax_0$;
   - si sceglie $\hat{r}_0$ vettore arbitrario tale che $(r_0, \hat{r}_0) \neq 0$,
     ad esempio: $\hat{r}_0 = r_0$
   - si pone: $\rho_0 = \alpha = \omega_0 = 1$ e $v_0 = p_0 = 0$;
2. Ciclo: for $i = 1, 2, 3, \ldots$ finchè x_i non è abbastanza accurata
   - $\rho_i = (\hat{r}_0, r_{i-1})$;
   - $\beta = \rho_i/\rho_{i-1}(\alpha/\omega_{i-1})$;
   - $p_i = r_{i-1} + \beta(p_{i-1} - \omega_{i-1}v_{i-1})$;
   - $v_i = Ap_i$;
   - $\alpha = \rho_i/(\hat{r}_0, v_i)$;
   - $s = r_{i-1} - \alpha v_i$;
   - $t = As$;
   - $\omega_i = (t, s)/(t, t)$;

- $x_i = x_{i-1} + \alpha p_i + \omega_i s$;
- $r_i = s - \omega_i t$;

2.3.2 BiCGStab parallelo con MPI

Poiché si é adottato un modello di parallelismo a grana fine con il quale vengono parallelizzate le singole funzioni di algebra vettoriale, come descritto nei primi paragrafi di questo capitolo, si sono potute riutilizzare le funzioni parallele prodotto matrice vettore, prodotto scalare, SAXPY e simili giá realizzate per il GMRES parallelo.

Con il BiCGStab si é peró dovuta integrare la struttura dati, che nel programma seriale originale prevedeva un semplice formato COO (Coordinate format). Per poter supportare le funzioni parallele sviluppate per il GMRES, e che sfruttano la struttura dati piú complessa descritta in §3.3, si é dovuta scrivere una semplice procedura di inizializzazione per tale struttura dati.

Anche la forma complessiva del programma message passing é identica a quella giá descritta in precedenza e quindi si rimanda per maggiori dettagli al paragrafo 2.2.2.

2.3.3 BiCGStab parallelo con OpenMP e con mp

Anche in questo caso vale quanto appena scritto. Il codice sorgente basato su OpenMP sviluppato per il GMRES é stato riutilizzato in maniera estesa per ottenere la corrispondente versione parallela del BiCGStab,

Per tale algoritmo si é anche utilizzata l'API **mp** per la realizzazione di un terzo programma parallelo. Grazie alle notevoli somiglianze fra *mp* ed OpenMP, descritte nel paragrafo 1.4.2 , la realizzazione di questa terza implementazione parallela del BiCGStab ha richiesto una limitata quantitá di lavoro.

Capitolo 3

Precondizionamento di sistemi lineari

3.1 Precondizionamento di un sistema lineare

Nell'ambito della risoluzione dei sistemi lineari una tecnica di precondizionamento consiste nel rimpiazzare il sistema originale con un nuovo sistema di equazioni, in modo tale da lasciare inalterata la soluzione cambiando peró la distribuzione degli autovalori. Scopo di tale modifica é rendere piú veloce la convergenza di un metodo di risoluzione iterativo.

In generale, per precondizionare il sistema non omogeneo descritto dall'equazione matriciale

$$Ax = b$$

é sufficiente moltiplicare entrambi i membri dell'equazione per la matrice P^{-1} ottenendo cosí il nuovo sistema:

$$P^{-1}Ax = P^{-1}b$$

Tale sistema é detto sistema precondizionato, e P viene chiamata matrice di precondizionamento o precondizionatore sinistro. Oltre a precondizionatori sinistri sono anche definibili precondizionatori destri e precondizionatori centrati. Con i primi il sistema originale viene trasformato nel seguente sistema:

$$AP^{-1}y = b, \qquad \text{dove y é definito da } y = Px$$

mentre con i precondizionatori centrati esso diventa:

$$P_L^{-1}AP_R^{-1}y = P_L^{-1}b, \qquad \text{dove y é definito da } y = P_Rx$$

Si possono anche definire precondizionatori che operano su singoli elementi di A o su blocchi di A: in questi casi si parla di precondizionatori per punti o precondizionatori a blocchi.

Assegnato un certo sistema lineare sarebbe utile poter determinare un precondizionatore ottimale, in grado di minimizzare il numero di iterazioni necessarie per il raggiungimento della soluzione. Idealmente si potrebbe scegliere P=A e garantire quindi la risoluzione del sistema in un'unica iterazione, ma chiaramente tale scelta non é vantaggiosa perché rende necessario il calcolo di $P^{-1} = A^{-1}$ con un costo computazionale uguale a quello di un metodo diretto. Come riportato in [3] non esistono risultati generali per la determinazione del precondizionatore ottimale, ma per numerosi metodi iterativi si é osservato che P é un buon precondizionatore se $P^{-1}A$ risulta essere una buona approssimazione della matrice identitá ed inoltre possiede autovalori confinati in un regione piccola di $\mathbf{C}$. Nella scelta del precondizionatore da applicare ad un certo problema, oltre alle considerazioni sulla capacitá del precondizionatore di ridurre il numero di iterazioni necessario per ottenere la soluzione del sistema, bisogna considerare anche il costo computazionale necessario a calcolarlo e l'occupazione di memoria richiesta.

I precondizionatori possono essere divisi in due categorie principali: precondizionatori algebrici e precondizionatori funzionali. I precondizionatori algebrici sono indipendenti del problema che ha generato il sistema, mentre quelli funzionali sono definiti in funzione di tale problema e quindi possono sfruttare informazioni aggiuntive su di esso, a scapito peró della generalitá. É opportuno a questo punto elencare i piú comuni precondizionatori algebrici, descrivendone in modo sintetico le caratteristiche.

1. *Precondizionatori diagonali.* Tali precondizionatori sono caratterizzati dal fatto che la matrice di precondizionamento P é diagonale. Una scelta classica, nel caso in cui A non sia simmetrica, consiste nel definire gli elementi della matrice diagonale P nel modo seguente:
$$p_{ii} = \left(\sum_{j=1}^{n} a_{ij}^2 \right)^{1/2}$$
Una scelta alternativa é invece :
$$p_{ii} = a_{ii}^{-1}$$
Quest'ultimo precondizionatore ha il pregio di essere realizzabile con un costo computazionale trascurabile.

2. *Precondizionatori basati sulla fattorizzazione incompleta LU.* In questa categoria rientra un'ampia famiglia di precondizionatori basati sulla fattorizzazione incompleta LU (ILU) o di Cholesky (IC): anziché determinare una fattorizzazione completa, si sceglie un criterio, anche basato su considerazioni computazionali come l'occupazione di memoria, che consenta di scartare alcuni elementi della fattorizzazione: questo approccio é utile se il criterio scelto consente di conservare almeno in parte la sparsitá della matrice A. Utilizzando diversi criteri sono state sviluppate numerose varianti di tali precondizionatori. L'esistenza della fattorizzazione ILU non é dimostrata per ogni tipo di matrice infatti esistono casi di matrici in cui il processo di fattorizzazione si arresta per la presenza di elementi pivotali nulli. Teoremi di esistenza sono invece disponibili (come riportato in [5]) nel caso di M-matrici e di matrici a dominanza diagonale. Una variante della fattorizzazione ILU é quella MILU (Modified Incomplete LU): con questa tecnica gli elementi pivotali vengono scelti in modo che la matrice $P - A$ abbia somma degli elementi per riga uguale a zero: per molte classi di matrici questa caratteristica risulta essere un vantaggio.

3. *Precondizionatore polinomiale di Neumann.* Questa tecnica mira a determinare un precondizionatore P che approssimi A^{-1} utilizzando una serie tron-

cata di Neumann. Posto $A = D - C$ (dive D é non singolare), segue che $A = (I - CD^{-1})D$. Si puó allora scrivere:

$$A^{-1} = D^{-1}(I - CD^{-1})^{-1} = D^{-1}(I + CD^{-1} + (CD^{-1})^2 + \dots).$$

Il precondizionatore si ottiene troncando la serie ad una certa potenza p. L' efficacia di tale metodo é limitata ai casi in cui la serie di Neumann converge, quindi solo se $\rho(CD^{-1}) < 1$.

4. *Precondizionatori ai minimi quadrati.* Questi precondizionatori, efficaci solo se applicati a matrici simmetriche e definite positive, operano nel seguente modo: si approssima A^{-1} utilizzando un'approssimazione polinomiale $p_s(A)$ ai minimi quadrati. La forma polinomiale $p_s(A)$ viene scelta in modo tale da minimizzare la funzione $\phi(x) = 1 - p_s(x)x$ rendendo cosí la matrice $I - P^{-1}A$ il piú "simile" possibile alla matrice nulla.

Si puó infine osservare che se il sistema lineare in esame é relativo alla risoluzione numerica di catene di Markov isomorfe ad una rete di Petri allora la matrice di sistema $A = Q^T$ é singolare ed il sistema omogeneo si può scrivere:

$$Q^T x = 0$$

dove Q é il generatore infinitesimale della catena di Markov ed il vettore x rappresenta la distribuzione di probabilitá di stato stazionario; il sistema precondizionato diventa allora

$$P^{-1}Q^T x = 0.$$

nel caso in cui si usi un precondizionatore sinistro. É opportuno ricordare che per il generatore infinitesimale Q (e per Q^T) é possibile dimostrare l'esistenza della fattorizzazioe ILU [1].

3.2 Precondizionatori di tipo ILU

Alcune fra le piú diffuse tecniche di precondizionamento sono basate sulla fattorizzazione incompleta LU e sono anche note come metodi combinati diretti-iterativi, in quanto composte da due fasi. La prima fase prevede il calcolo di una fattorizzazione incompleta LU della matrice Q^T; si inizia una normale fattorizzazione di tipo LU della matrice Q^T e ad un certo punto durante il calcolo si eliminano, applicando una qualche regola, alcuni dei termini non nulli della fattorizzazione. In questo modo, anziché arrivare alla fattorizzazione completa $Q^T = LU$, dove L ed U sono matrici rispettivamente triangolare inferiore e triangolare superiore, si ottiene un'espressione del tipo: $Q^T = LU - E$ dove E é detto residuo e prevedibilmente é "piccolo" secondo una qualche definizione. Questa prima fase costituisce la parte diretta del metodo. La seconda fase consiste nell'inserire la fattorizzazione incompleta ottenuta nella fase precedente in una procedura iterativa basata sulle seguenti relazioni:

(i) $Q^T x = (LU - E)x = 0$

(ii) $LUx_{(k+1)} = Ex_{(k)}$ o equivalentemente $x_{(k+1)} = x_{(k)} - (LU)^{-1}Q^T x_{(k)}$

La seconda relazione, in entrambe le forme riportate, puó essere utilizzata come regola di iterazione all'interno di un procedimento di risoluzione iterativo del sistema; l'uso di un tale schema di iterazione risulta equivalente alla risoluzione mediante il metodo delle potenze del sistema precondizionato

$$U^{-1}L^{-1}Q^T x = 0.$$

Fra le numerose varianti di precondizionatori basati sulla fattorizzazione ILU descritte in letteratura, si possono individuare tre tipi fondamentali di tali precondizionatori: la differenza fra questi tre metodi consiste nella diversa regola utilizzata per determinare quali termini non nulli scartare durante la fase di fattorizzazione. Il primo metodo, noto come ILU(0) (incomplete LU factorization, level 0 fill-in), realizza una fattorizzazione del tipo $Q^T = LU - E$ in modo tale che la matrice $L + U$

abbia la stessa struttura di termini non nulli della matrice Q^T. Tale proprietá risulta particolarmente utile quando la matrice Q^T è sparsa e di notevoli dimensioni: se si adotta uno dei formati di memorizzazione usuali per matrici sparse, risulta allora che la fattorizzazione ILU(0) richiede lo stesso spazio necessario per memorizzare Q^T. L'algoritmo (descritto in [5]) che realizza tale tipo di fattorizzazione relativa ad una matrice A di dimensione N x N è il seguente:

1. Inizio fattorizzazione **ILU(0)** :

2. Definito $NZ(A)$ l'insieme degli indici (i, j) per i quali $a_{i,j} \neq 0$, dove $a_{i,j} \in A$; L ed U sono due matrici triangolari inferiore e superiore rispettivamente, inizialmente nulle.

   Ciclo: for $i = 1, 2, 3, \ldots n$

      Ciclo: for $j = 1, 2, 3, \ldots n$

         if $(i, j) \in NZ(A)$ then

            $kmax$ = min(i, j)-1

            $r = 0$

            Ciclo: for $k = 1, 2, \ldots, kmax$

               $r = r + l_{ik}\ u_{kj}$

            $s = a_{ij} - r$

            if $(i \leq j)$ then $u_{i,j} = s$

            else $l_{i,j} = s/u_{i,j}$

Il risultato della fattorizzazione incompleta é memorizzato nelle matrici U ed L; per quest'ultima non vengono memorizzati gli elementi della diagonale noti implicitamente ed uguali ad 1. Riguardo a quest'ultima considerazione é opportuno ricordare che la fattorizzazione completa LU, da cui l'algoritmo appena descritto trae origine, presenta N gradi di libertá se N é la dimensione della matrice: infatti l'equazione matriciale

$$LU = A$$

ha $N^2 + N$ incognite (i coefficienti di L ed U) e solo N^2 equazioni(una per ogni coefficienti di A), perció rimangono da fissare altre N condizioni. Normalmente si adotta una delle due possibilitá seguenti :

$l_{ii} = 1, \qquad \text{con } i = 1, 2, \ldots, N$

oppure

$u_{ii} = 1, \qquad \text{con } i = 1, 2, \ldots, N$

Quando si attua una fattorizzazione incompleta di tipo LU si parla di fattorizzazione ILU nel primo caso oppure $IC(Incomplete\ Choleski)$ nel secondo. L'algoritmo appena descritto (di tipo ILU) puó essere facilmente modificato per ottenere un fattorizzazione di tipo IC: é sufficiente sostituire l'ultimo costrutto if ... *then* con il seguente:

if $(i \geq j)$ then $l_{i,j} = s$

else $u_{i,j} = s/l_{i,j}$

Studiando la successione delle operazioni dell'algoritmo e dato che $L + U$ ha la stessa struttura di elementi non nulli di Q, si puó notare che é possibile impiegare la stessa area di memoria usata per memorizzare gli elementi non nulli di A, per memorizzare i termini di L e U calcolati ad ogni ciclo: le locazioni di memoria contenenti inizialmente i termini $a_{i,j}$ (con $i \leq j$) conterranno alla fine del calcolo gli elementi di U, quelle contenti i termini $a_{i,j}$ $(i > j\ )$ conterranno alla fine del calcolo gli elementi di L, tranne quelli della diagonale principale uguali ad uno e perció noti implicitamente.

Il secondo precondizionatore di tipo Incomplete LU esaminato viene normalmente chiamato ILUK (descritto in [1]). La fattorizzazione avviene riga per riga, calcolando ogni riga come se si effettuasse una fattorizzazione completa di tipo LU; al termine di ogni riga si esaminano gli elementi non nulli e si conservano solo i K-1 elementi maggiori in modulo piú l'elemento posto sulla diagonale principale. Alla fine, la matrice contenente la fattorizzazione incompleta é formata da righe composte al massimo da K elementi non nulli.

Come per il caso ILU(0), anche tale metodo definisce a priori lo spazio di memoria necessario per contenere la fattorizzazione.

Di seguito é riportato l'algoritmo che realizza tale tipo di fattorizzazione per una matrice A di dimensione N x N:

1. Inizio fattorizzazione **ILUK** :

2. Definito V un vettore di dimensione N; a_{ij} gli elementi della matrice A; M una matrice di dimensione N x N e di elementi m_{ij}

3. $M = A$

   Ciclo: for $i = 2, 3, \ldots N$

      Ciclo: for $k = 1, 2, 3, \ldots N$

         $v_k = a_{ik}$

      Ciclo: for $j = 1, 2, 3, \ldots N$

         $kmax = \min(i, j)$-1

         $r = 0$

         Ciclo: for $k = 1, 2, \ldots, kmax$

            $r = r + l_{ik}\ u_{kj}$

         $v_j = a_{ij} - r$

         if $(i \leq j)$ then $m_{ij} = v_j$

         else $m_{ij} = v_j / u_{ij}$

      calcola $MAXK(M_i)$ = insieme dei K elementi maggiori in modulo della riga i di M

      Ciclo: for $j = 1, 3, \ldots N$

if ($m_{ij} \in MAXK(M_i)$) then $m_{ij} = m_{ij}$

else $m_{ij} = 0$

Il risultato della fattorizzazione incompleta é memorizzato nella matrice M, la cui parte triangolare inferiore rappresenta la matrice L della fattorizzazione (con i termini della diagonale principale omessi essendo unitari per definizione), mentre la parte triangolare superiore rappresenta la matrice U.

Un ulteriore metodo di precondizionamento basato su fattorizzazione di tipo ILU é noto come ILUTH, *Incomplete LU with THreshold*, ed é basato su un' operazione di soglia. La fattorizzazione avviene riga per riga, come per il precedente ILUK. Al termine di ogni riga gli elementi non nulli vengono confrontati con una soglia, scartando quelli che risultino inferiori ad essa in modulo (ad eccezione del termine della diagonale principale).

L'algoritmo, descritto in [1], é leggermente meno complesso di quello necessario nel caso ILUK dato che non richiede un'operazione di ordinamento, ma ha lo svantaggio di non determinare a priori lo spazio di memoria necessario per contenere il risultato della fattorizzazione incompleta. La scelta di una soglia troppo bassa potrebbe portare al calcolo della fattorizzazione completa, con conseguenti tempi di esecuzione e occupazione di memoria inaccettabili.

3.3 Realizzazione di un precondizionatore ILU0 per il GMRES

L'algoritmo utilizzato per la fattorizzazione incompleta ILU0 é quello descritto in 3.2. L'implementazione di tale algoritmo deve tener conto delle strutture dati utilizzate: per minimizzare l'occupazione di memoria é indispensabile utilizzare strutture dati che sfruttino l'elevata sparsitá della matrice che rappresenta il sistema.

L'uso di strutture dati efficienti é obbligatorio se si considera la dimensione dei problemi trattati. L'esplosione dello spazio degli stati di una rete di Petri genera facilmente sistemi con piú di centomila equazioni: per memorizzare in forma completa una matrice di 100000 equazioni, utilizzando dati di tipo double del C (supposto ad 8 byte), servono quasi 80 Gbyte di memoria.

Fra le varie strutture dati utilizzate normalmente per la memorizzazione di matrici sparse, quella scelta originariamente per la sua efficienza se applicata al problema trattato e considerando anche la struttura dati delle matrici prodotte dal programma GreatSPN (descritto in [8]), viene normalmente definita $COO\ format$ (COOrdinate format) ed é costituita da tre vettori di lunghezza pari al numero degli elementi non nulli della matrice in cui:

- il primo vettore (di double) memorizza il valore dell'elemento non nullo; l'ordine dei dati nel vettore rispecchia la suddivisione per righe della matrice, ma non necessariamente l'ordinamento all'interno della singola riga.
- il secondo vettore (di interi) memorizza la riga dell' elemento non nullo di indice corrispondente nel primo vettore
- il terzo vettore (di interi) memorizza la colonna dell' elemento non nullo di indice corrispondente nel primo vettore

Si puó immediatamente notare che un tale formato é molto efficiente se impiegato in operazioni di tipo matrice per vettore, perché in tal caso é sufficiente scandire tutta la matrice con un unico ciclo iterativo di tipo *for...do* ed utilizzare gli indici di riga e di colonna memorizzati per ogni elemento per eseguire la corretta moltiplicazione. Le cose si complicano se si considera invece l'algoritmo di fattorizzazione incompleta descritto in 3.2. Come si puó osservare nel ciclo $for\ k$ piú interno

Ciclo: for $k = 1, 2, \ldots, max(i, j)$

$r = l_{ik}\ u_{kj}$

$s = q_{ij} - r$

la lettura degli elementi della matrice contenente la fattorizzazione avviene sia operando lungo le righe che lungo le colonne: mentre la lettura dei dati in successione lungo le righe (l_{ik} al variare di k) é realizzabile facilmente con la struttura dati descritta che rispecchia la suddivisione per righe della matrice, la lettura dei dati in successione lungo le colonne (u_{kj} al variare di k) rende necessario integrare la struttura dati in modo tale da fornire informazioni sulla topologia delle colonne della matrice. Poiché inoltre la fattorizzazione procede per righe successive sarebbe utile, al fine di ottenere un'implementazione efficiente, disporre di una informazione sul punto di inizio di ogni riga. Si é pensato di integrare il formato COO utilizzato inizialmente in modo da soddisfare queste esigenze, ottenendo la struttura dati a sei vettori descritta di seguito (un esempio di come tale struttura viene utilizzata é riportato in figura 3.1).

Indicati con N la dimensione della matrice e con NZ il numero di elementi non nulli, la struttura dati risulta composta da sei vettori di cui:

- il primo vettore, composto da NZ dati double (indicato con *Vet_valore_aij* in figura 3.1), memorizza il valore dell'elemento non nullo della matrice; l'ordine dei dati nel vettore rispecchia la suddivisione per righe della matrice ma non necessariamente l'ordinamento all'interno della singola riga;
- il secondo vettore, composto da NZ dati interi (indicato con *Vet_Indice_righe* in figura 3.1), memorizza la riga dell'elemento non nullo; di indice corrispondente nel primo vettore;
- il terzo vettore, composto da NZ dati interi (indicato con *Vet_Indice_colonne* in figura 3.1), memorizza la colonna dell'elemento non nullo di indice corrispondente nel primo vettore;
- il quarto vettore, composto da N dati interi (indicato con *Vet_Inizio_righe* in figura 3.1), memorizza l'indice (relativo ai tre vettori precedenti) di inizio riga; per ogni riga della matrice;

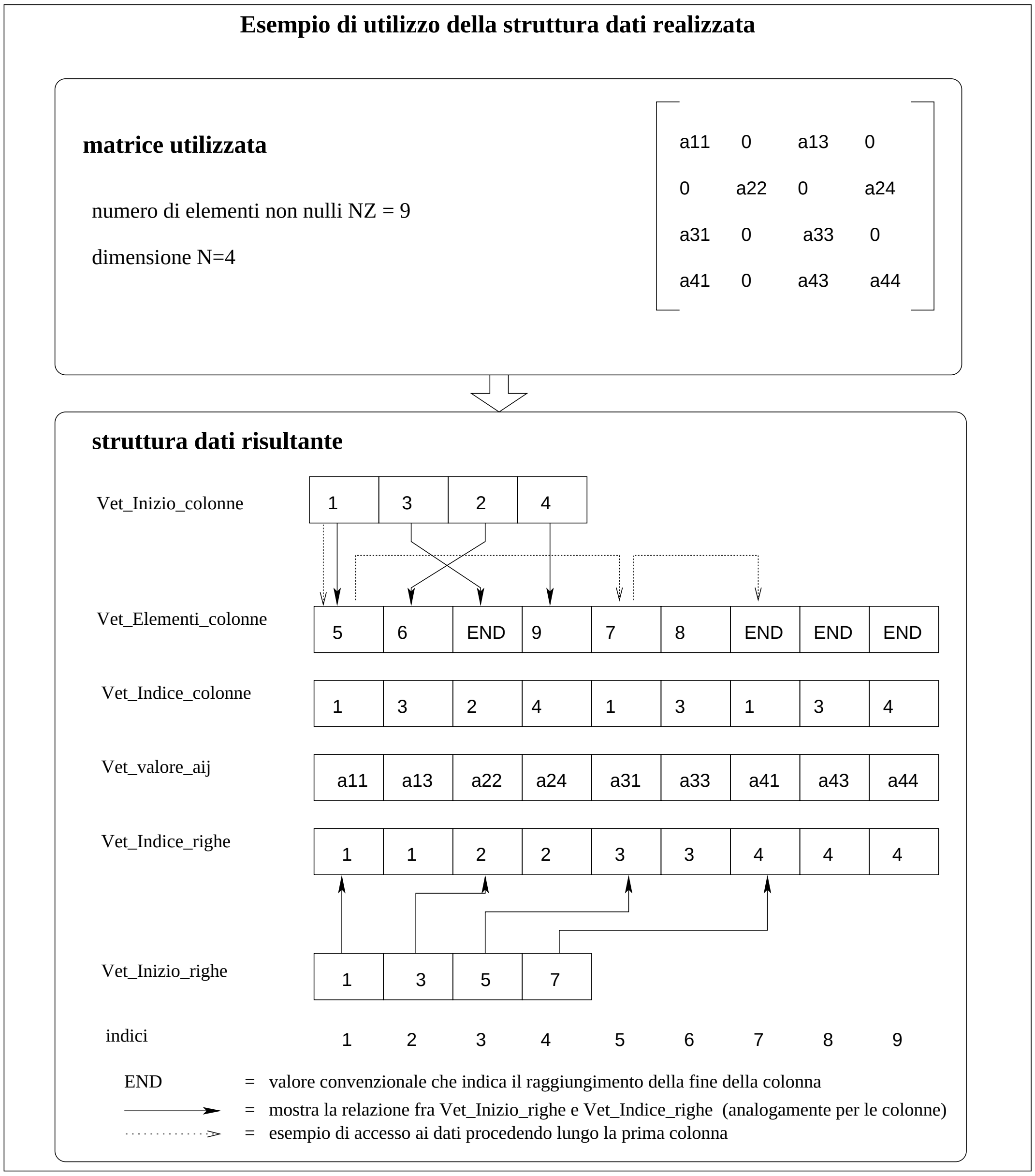


Figura 3.1: Esempio di utilizzo della struttura dati realizzata per memorizzare una semplice matrice 4x4

- il quinto vettore, composto da N dati interi (indicato con *Vet_Inizio_colonne* in figura 3.1), memorizza l'indice (relativo ai primi tre vettori citati) di inizio colonna per ogni colonna della matrice;

- il sesto vettore, composto da NZ dati interi (indicato con *Vet_Elementi_colonne* in figura 3.1), memorizza l'indice degli elementi successivi lungo le colonne.

Una fase di inizializzazione in cui vengono generati il quarto, quinto e sesto vettore, che non sono immediatamente disponibili, deve quindi precedere la costruzione della matrice di precondizionamento P. Nella tabella 3.1 é riportato lo pseudocodice che descrive la procedura di costruzione del precondizionatore ILU0.

É opportuno infine spiegare come integrare tale precondizionatore nell'algoritmo non precondizionato. I precondizionatori di tipo ILU possono essere applicati ad algoritmi come il GMRES in modo abbastanza semplice (come descritto in [5]). É peró importante osservare che sono necessarie modifiche inerenti ad alcuni aspetti dell'algoritmo:

1. *Normalizzazione della soluzione.* Come si é detto in §4.6 il sistema $Q^T x = 0$ é omogeneo e Q singolare in quanto è il generatore infinitesimale di una catena di Markov; la soluzione possiede quindi un grado di libertá e per determinarla univocamente é necessaria un'ulteriore equazione linearmente indipendente dalle altre. Sapendo che il vettore x è un vettore di probabilitá si puó allora imporre la condizione di normalizzazione $\sum x = 1$ che garantisce che la somma della componenti del vettore x sia unitaria. Se peró la matrice Q é generata da una catena di Markov isomorfa ad una rete di Petri si puó affermare che se il GMRES converge, la soluzione ottenuta ha somma delle componenti uguale ad 1, e questo grazie alle proprietá di questa particolare categoria di matrici. Applicando un precondizionatore tale proprietà viene persa e risulta quindi necessario imporre la condizione di normalizzazione della soluzione.

2. *Calcolo del prodotto matrice vettore.* Nell'algoritmo GMRES(m) la matrice dei coefficienti del sistema lineare A é impiegata esclusivamente in operazioni di moltiplicazione matrice per vettore del tipo:

$$w = Av, \qquad \text{con } A \text{ e } v \text{ noti ed il vettore } w \text{ incognito}$$

Definiti i vettori della struttura dati in accordo con l'esempio di figura 3.1, la costante END e le funzioni aggiorna_pos_colonna() e aggiorna_pos_riga(), lo pseudocodice della procedura per la costruzione del precondizionatore di tipo ILU0 é il seguente:

```
for( c:=1 to NZ )
  begin
    i := Vet_indice_righe[c];
    j := Vet_indice_colonne[c];
    fine_riga := false;
    if( i<j ) then
        Vet_valore_aij[c] := Vet_valore_aij[c] / Vet_valore_aij[Vet_diagonale[i]];
    pos_riga := Vet_Inizio_righe[i];
    pos_colonna := Vet_Inizio_colonne[j];
    k_max := min( j-1 , i-1);
    do
        if( Vet_indice_colonne[pos_riga] = Vet_indice_colonne[pos_colonna] OR
        Vet_indice_colonne[pos_riga] <= k_max ) then
          begin
            if( i>=j ) then
                Vet_valore_aij[c] := Vet_valore_aij[c]-Vet_valore_aij[c]*Vet_valore_aij[pos_colonna];
            else
                Vet_valore_aij[c] := Vet_valore_aij[c]-Vet_valore_aij[c]*Vet_valore_aij[pos_colonna]
                /Vet_valore_aij[Vet_diagonale[i]];
            aggiorna_pos_riga(pos_riga);
            if( pos_riga <> i OR Vet_indice_colonne[pos_riga] > kmax ) then
                fine_riga := true;
            if(aggiorna_pos_colonna(pos_colonna,Vet_Elementi_colonne)=END) then
                fine_riga := true;
            else if( Vet_indice_righe[pos_colonna] > k_max) then
                fine_riga := true;
          end
            else if( Vet_indice_colonne[pos_riga] > Vet_indice_righe[pos_colonna ]) then
              begin
                if(aggiorna_pos_colonna(pos_colonna,Vet_Elementi_colonne)=END) then
                    fine_riga := true;
                else if( Vet_indice_righe[pos_colonna] > k_max) then
                    fine_riga := true;
              end
            else
              begin
                aggiorna_pos_riga(pos_riga);
                if( pos_riga <> i OR Vet_indice_colonne[pos_riga] > k_max ) then
                    fine_riga := true;
              end
        while ( fine_riga = true );
  end
```


Tabella 3.1: Pseudocodice della procedura di calcolo del precondizionatore ILU0.

Se si considera invece il sistema precondizionato, la matrice A deve essere sostituita dalla matrice precondizionata:

$$w = (LU)^{-1}Av \tag{3.1}$$

Naturalmente, dato che L ed U sono matrici triangolari, sarebbe possibile calcolare esplicitamente $(LU)^{-1} = U^{-1}L^{-1}$. Dette $Y = L^{-1}$ e $Z = U^{-1}$ i coefficienti di Y si calcolano con il seguente procedimento [2]:

- $y_{jj} = 1/l_{jj}, j = 1, 2, \dots, n$
- Ciclo: for $j = 1, 2, \dots, n-1$

  $y_{ij} = -\left(\sum_{k=j}^{i-1} l_{ik}ykj\right)/l_{ii}$

In modo simile si opera il calcolo degli elementi di Z:

- $z_{jj} = 1/u_{jj}, j = 1, 2, \dots, n$
- Ciclo: for $j = 1, 2, \dots, n-1$

  $z_{ij} = -\left(\sum_{k=i+1}^{j} u_{ik}zkj\right)/u_{ii}$

L'inversione di una matrice triangolare richiede $\frac{n^3}{6}$ operazioni aritmetiche e quindi ha complessitá confrontabile a quella di un metodo diretto di soluzione di un sistema lineare, annullando cosí le potenzialitá di un metodo iterativo. Si puó allora considerare che l'espressione (3.1) é equivalente a

$$LUw = Av$$

ed allora il vettore w puó esser determinato risolvendo, nell'ordine, i due sistemi lineari triangolari

$$Ly = Av \quad \text{(vettore y incognito)}$$

$$Uw = y \quad \text{(vettore w incognito)}$$

La soluzione dei due sistemi triangolari richiede n^2 operazioni, ed ha quindi una complessitá paragonabile al calcolo del prodotto matrice per vettore. Complessivamente il calcolo del prodotto matrice precondizionata per vettore ha un costo computazionale doppio rispetto al calcolo del prodotto matrice non precondizionata per vettore.

3. *Soluzione con sostituzione di una riga.* Per quanto visto al punto precedente, se si sceglie di calcolare il prodotto matrice per vettore risolvendo i due sistemi triangolari bisogna notare che se il sistema non fosse omogeneo la determinazione del vettore residuo r renderebbe necessario il calcolo di

$$r = b - U^{-1}L^{-1}Ax$$

da cui deriva

$$LUr = LUb - Ax$$

Nell'applicazione trattata in questa tesi il sistema é non omogeneo solo se si sceglie di effettuare la sostituzione di una riga di A per superare la singolaritá della matrice: il vettore b acquista un componente diverso da zero e, di conseguenza, il calcolo di r richiede due ulteriori prodotti matrice vettore, $y = Ub$ e successivamente $w = Ly$. Questo costo computazionale aggiuntivo, unitamente alle ottime caratteristiche del GMRES(m) il quale a differenza del TFQMR non rende necessario adottare tale scelta per ottenere la convergenza, suggerisce di evitare la sostituzione di una riga ed effettuare invece la normalizzazione ad ogni iterazione.

4. *Uso di un'espressione ottimizzata per l'ultima iterazione del processo di Arnoldi.* Una delle ottimizzazioni possibili dell'algoritmo GMRES(m) consiste nell'utilizzare le seguenti uguaglianze 3.2

$$h_{m+1,m} = \|Av_m - \sum_{i=1}^{m} h_{i,m}v_i\|^2 = \|Av_m\|^2 - \sum_{i=1}^{m} h_{i,m}^2 \tag{3.2}$$

per rendere piú semplice il calcolo del coefficiente $h_{m+1,m}$ durante l'ultima iterazione del processo di Arnoldi: anziché m prodotti vettore per scalare basta il calcolo della somma di m scalari elevati al quadrato, oltre naturalmente al calcolo del prodotto matrice-vettore in entrambi i casi.

L'uguaglianza di destra nell'espressione 3.2 non é peró valida se al posto di A si considera la matrice precondizionata $(LU)^{-1}A$, ed in tal caso non si puó utilizzare la semplificazione accennata.

5. *Scelta fra una fattorizzazione di tipo ILU(0) ed una di tipo IC(0).* Le differenze nell'implementazione dei due casi sono minime, dunque sono state realizzate entrambe fornendo la possibilitá all'utente di scegliere quale impiegare mediante un'opzione a linea di comando.

3.4 Realizzazione di un precondizionatore ILUK per il GMRES

La realizzazione di tale precondizionatore sfrutta in parte lo stesso approccio usato per il precondizionatore ILU0. Identici sono la struttura dati utilizzata ed i vari metodi con cui il precondizionatore viene integrato all'interno dell'algoritmo.

Diverso é invece il modo in cui viene gestito l'accesso ai dati. Mentre nel caso ILU0, durante la costruzione della matrice di precondizionamento, la riga corrente viene trattata in modo omogeneo alle precedenti e non richiede un vettore temporaneo, in questo caso la riga corrente viene gestita in modo particolare. Come si é giá detto durante la descrizione degli algoritmi di precondizionamento di tipo ILU il processo di fattorizzazione opera riga dopo riga. La riga corrente inizialmente viene copiata in un vettore temporaneo di dimensione N uguale alla dimensione della matrice, successivamente viene calcolata la fattorizzazione in base alle righe precedenti, memorizzate nella struttura dati descritta in figura 3.1, ed in base agli elementi della riga corrente memorizzati nel vettore temporaneo. Durante questa fase si hanno dei "riempimenti" dovuti al calcolo della fattorizzazione stessa, come descritto nell'algoritmo **ILUK** nel paragrafo 3.2. Infine, il vettore temporaneo viene esaminato e vengono trovati i k elementi maggiori in modulo i quali vengono copiati nella struttura dati usuale. In figura 3.2 viene illustrato il modo di operare appena descritto.

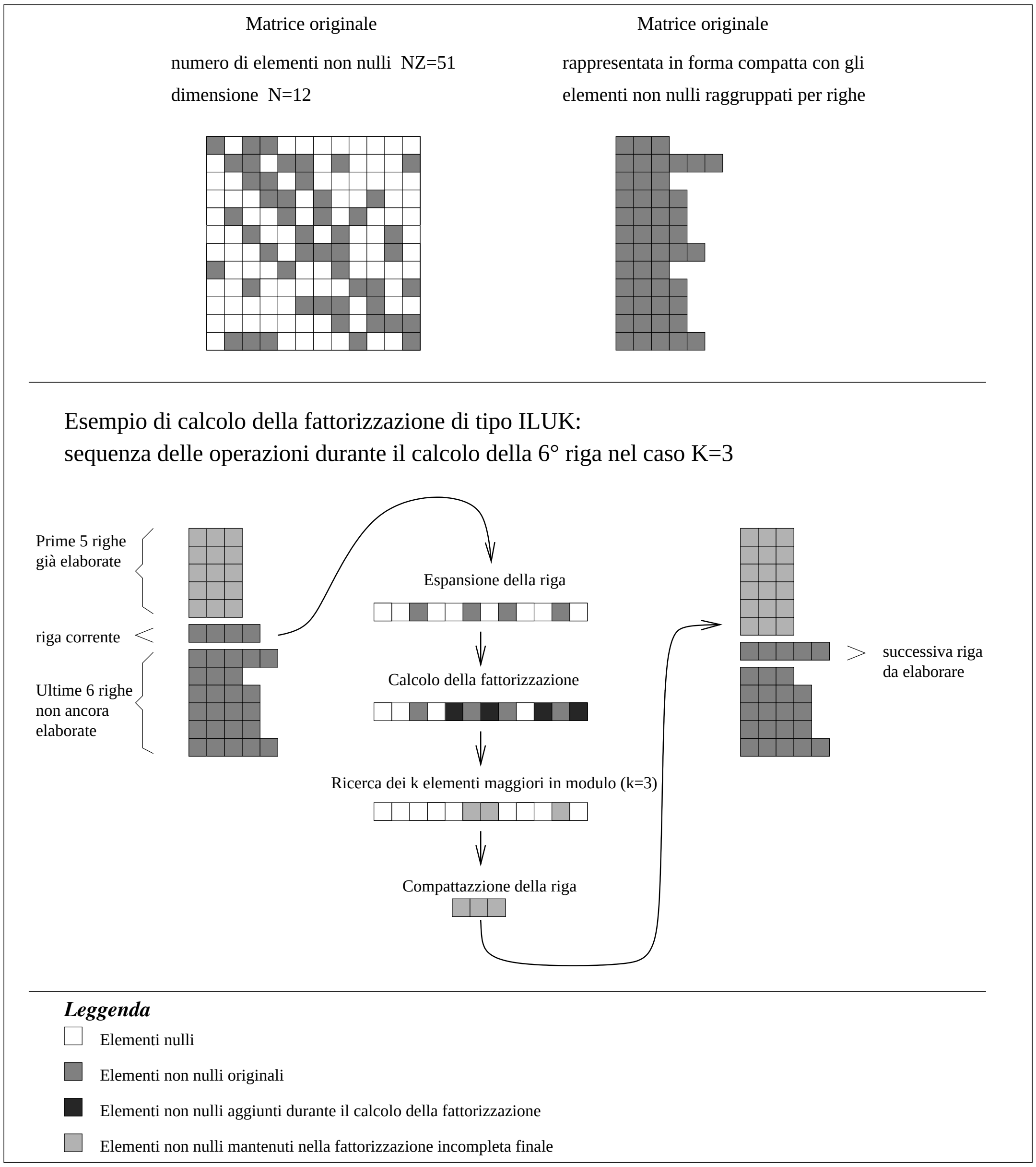


Figura 3.2: Descrizione schematica delle varie fasi di calcolo della matrice di precondizionamento di tipo ILUK.

3.5 Effetto del precondizionatore ILU0

Nel paragrafo 4.2 sono riportati i risultati ottenuti con l'uso del precondizionatore ILU0 e sono messi in evidenza i benefici effetti dell'uso di tale precondizionatore applicato al GMRES. In particolare si é visto come in tutti i casi trattati si sia ottenuto un miglioramento dei tempi di esecuzione, sia rispetto alla versione non precondizionata sia rispetto alla versione con precondizionatore diagonale. Il miglioramento é stato in alcuni casi rilevante, in particolare con la rete benchprod per la quale i tempi di esecuzione si sono ridotti fino ad un trentesimo di quelli ottenuti in origine. A questo punto sarebbe interessante valutare dal punto di vista numerico l'effetto di tale precondizionatore sulla matrice di sistema A. Da quanto riassunto nella descrizione delle tecniche di precondizionamento e dalla teoria generale relativa alla risoluzione iterativa di sistemi lineari si possono adottare due approcci per valutare l'effetto del precondizionatore sul condizionamento del problema:

1. studiare il numero di condizionamento della matrice
2. studiare la distribuzione degli autovalori della matrice di iterazione

3.5.1 Calcolo del numero di condizionamento

Ricordiamo innanzitutto la definizione di numero di condizionamento per una matrice A non singolare:

Definizione 1 $cond(A) = ||A||\ ||A^{-1}||$.

Quando si risolve un sistema di equazioni lineari, non omogeneo e con matrice dei coefficienti A non singolare, il numero di condizionamento stabilisce una relazione fra l'errore relativo sulla soluzione ed il residuo, infatti vale la maggiorazione:

$$\frac{||\tilde{x} - x||}{||x||} \leq cond(A)\frac{||r||}{||b||}$$

Da questa relazione si vede come $cond(A)$ possa fornire informazioni sull'accuratezza della soluzione trovata: se $cond(A)$ é grande allora il residuo non risulta essere una buona stima della precisione raggiunta, viceversa nel caso contrario. Piú in generale, se si lavora in aritmetica non esatta, relazioni simili alla precedente valgono anche per gli errori relativi su A e b, ed in tal caso $cond(A)$ permette di valutare la propagazione di tali errori sulla soluzione $\tilde{x}$. Nell'analisi che si vuole compiere si presenta subito un ostacolo, infatti le matrici trattate sono singolari, come é stato giá osservato in precedenza, e l'applicazione della definizione non risulta possibile in modo automatico. Si puó pensare allora di ricorrere alla definizione di *inversa generalizzata*, detta anche *inversa di gruppo*, che nei problemi relativi alla risoluzione di catene di Markov puó svolgere un ruolo analogo a quello della matrice inversa, come osservato in [1].

Definizione 2 *Data la matrice Q, si definisce definisce inversa generalizzata $Q^{\#}$ l'unica matrice che soddisfa le relazioni seguenti:*
$QQ^{\#}Q = Q,\ Q^{\#}QQ^{\#} = Q^{\#},\ QQ^{\#} = Q^{\#}Q$

Poiché Q é un generatore infinitesimale é possibile dimostrare che vale anche l'equazione: $I - QQ^{\#} = e\pi$ [1], dove π é il vettore probabilitá di stato stazionario del sistema.
Per il calcolo dell'inversa generalizzata si puó utilizzare l'algoritmo proposto in [1] e riportato di seguito:

1. calcolo del vettore soluzione di stato stazionario π

2. Ciclo: for $i = 1, 2, \ldots, n$

   - calcolo di una qualunque soluzione del sistema: $Qz_i = e_i - \pi_i e$
   - modifica di z_i in modo da trovare la i^a colonna di $Q^{\#}$: $q_i^{\#} = z_i - (\pi z_i)e$

3. costruzione di $Q^{\#}$: $Q^{\#} = [q_1^{\#}, q_2^{\#}, \ldots, q_n^{\#}]$

L'applicazione dell'algoritmo appena descritto consente in teoria di calcolare il numero di condizionamento della matrice Q, soprattutto se si utilizza una norma opportuna, come la $\| \, \|_1$ che non richiede la memorizzazione di tutta la matrice ma solo di una colonna per volta. Anche in questo caso risulta peró necessario risolvere N sistemi lineari, se N é la dimensione di Q. Le cose peggiorano ulteriormente se si considera la matrice precondizionata: in questo caso si dovrebbe calcolare la quantitá:

$\|P^{-1}Q^T\| \, \|(P^{-1}Q^T)^{-1}\|$.

L'algoritmo precedente non é piú applicabile perché la matrice precondizionata $(P^{-1}Q^T)$ non gode delle proprietá $I - QQ^{\#} = e\pi$ che é specifica della matrice Q, generatore infinitesimale di una catena di Markov; in particolare non é piú verificato che la somma degli elementi di una riga sia uguale a zero. Al contrario, l'inversa di P é facilmente calcolabile nel caso in cui $P = LU$, come si verifica per il precondizionatore basato su fattorizzazione incompleta realizzato. É possibile allora, mantenendo separati i termini della moltiplicazione, calcolare l'espressione:

$$cond(Q^T) = \|(LU)^{-1}Q^T\| \, \|((LU)^{-1}Q^T)^{-1}\| = \|U^{-1}L^{-1}Q^T\| \|(Q^T)^{-1}LU\|$$

da cui

$$cond(Q^T) \simeq \|U^{-1}L^{-1}Q^T\| \|(Q^T)^{\#}LU\|$$

L'approssimazione é dovuta al fatto che per la matrice Q^T si utilizza l'inversa generalizzata, mentre l'inversa di L ed U puó essere calcolata normalmente. Resta ora da valutare il costo computazionale dell'espressione:

$$\|U^{-1}L^{-1}Q^T\| \, \|(Q^T)^{\#}LU\|$$

In questo caso non é piú possibile evitare di memorizzare la matrice $(Q^T)^{\#}$ la quale deve venire moltiplicata per LU. É necessario quindi memorizzare sei matrici e tre di esse, le inverse, non sono sparse. Complessivamente si puó valutare l'occupazione di memoria in $6N^2s$ byte, dove s é la dimensione in byte del tipo di dato utilizzato per memorizzare gli elementi delle matrici; il numero di operazioni aritmetiche richieste é invece $4N^3$ moltiplicazioni e somme, per il calcolo dei prodotti fra matrici, a cui si devono aggiungere le operazioni richieste per la costruzione di $(Q^T)^{\#}$,

che dipendono dal metodo usato per la risoluzione dei sistemi lineari (comunque dell'ordine o superiore ad N^2) moltiplicato per N sistemi da risolvere. Entrambi gli aspetti del problema risultano fortemente limitanti e rendono praticamente impossibile calcolare $cond(Q^T)$ se non per matrici di piccole dimensioni. Nella misure effettuate utilizzando il metodo appena descritto, anche per una matrice con solo 5327 stati sono stati necessari circa 3 giorni di calcolo su un elaboratore Onyx2, e l'occupazione di memoria ha raggiunto il livello di circa 1.5 GByte.
Nella tabella 3.2 sono riportati i valori di numero di condizionamento delle matrici non precondizionate e precondizionate mediante precondizionatore ILU. Come si

rete	k	numero di stati	matrice precond.	matrice non prec.
mesh 2x2	1	57	$2.57 \cdot 10^2$	$6.24 \cdot 10^2$
mesh 2x2	2	461	$3.38 \cdot 10^3$	$7.56 \cdot 10^3$
mesh 2x2	3	1837	$1.89 \cdot 10^4$	$4.08 \cdot 10^4$
mesh 2x2 not p.	1	189	$1.74 \cdot 10^3$	$1.04 \cdot 10^3$
mesh 2x2 not p.	2	5327	$3.21 \cdot 10^4$	$1.35 \cdot 10^3$
multopoll	1	80	$4.72 \cdot 10^4$	$7.10 \cdot 10^3$
multopoll	2	933	$9.37 \cdot 10^4$	$1.11 \cdot 10^3$
benchprod	1	172	$4.92 \cdot 10^3$	$1.04 \cdot 10^3$

Tabella 3.2: GMRES precondizionato con $\epsilon = 10^{-7}$;caso k=4.

puó vedere i risultati ottenuti sono discordanti: in tre casi su otto si é addirittura calcolato per la matrice precondizionata un numero di condizionamento superiore a quello della matrice non precondizionata. Nei casi restanti si é avuta una diminuzione, ma di entitá modesta,sempre inferiore ad un ordine di grandezza. É da ricordare comunque che l'effetto del precondizionatore, sia per quel che riguarda il tempo di esecuzione sia per il numero di iterazioni, diventava particolarmente evidente per le matrici piú grandi di quelle che é stato possibile utilizzate per queste misure. Finché non sará possibile calcolare il numero di condizionamento anche per matrici di grandi dimensioni l'analisi appena fatta risulta perció di utilitá molto parziale.

3.5.2 Studio degli autovalori della matrice di iterazione

Una seconda possibilitá per valutare l'effetto del precondizionatore é studiare la distribuzione degli autovalori per la matrice di iterazione. Sono perció stati calcolati gli autovalori delle matrici:

- $P^T = I + \delta t\ Q^T$, dove $\delta t \leq \frac{1}{max|q_{ij}|}$ in modo tale che P sia stocastica.
- $H = I - U^{-1}L^{-1}Q^T$

La matrice P^T é la matrice delle probabilitá di transizione ed é anche la matrice di iterazione per il metodo delle potenze, mentre la H é la matrice di iterazione per il metodo delle potenze precondizionato. Di seguito sono riportate tabelle in cui sono elencati i dieci autovalori maggiori in modulo per ogni caso considerato. Come si puó vedere dai valori riportati nelle tabelle e contrariamente a quanto avvenuto per il numero di condizionamento, stavolta l'effetto del precondizionatore é chiaramente visibile: in tutti i casi si ha una netta diminuzione del modulo degli autovalori a partire dal secondo in poi. É opportuno ricordare che, detti $\lambda_i, (i = 1, 2, \ldots, n)$ gli autovalori della matrice di iterazione, tali che $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$ allora la velocitá di convergenza per un metodo iterativo dipende dal rapporto $|\lambda_2|/|\lambda_1|$: piú tale rapporto é piccolo, piú la convergenza é veloce.
Nelle seguenti tabelle gli autovalori sono presentati nella forma usuale per i numeri complessi $x + iy$ (si puó notare come la parte immaginaria sia risultata nulla in tutti i casi).

rete mesh 2x2 pre.	
k=1	
P^T	H
1.0000	1.0000
0.6125	0.1393
0.5896	-0.1220
0.5666	-0.1220
0.5666	-0.1220
0.5666	-0.1220
0.5666	-0.1216
0.5000	-0.1159
0.5000	-0.1159
0.5000	-0.0488

rete mesh 2x2 pre.	
k=2	
P^T	H
1.0000	1.0000
0.7935	0.2014
0.7903	0.1739
0.7903	0.1515
0.7844	0.1515
0.7373	0.1160
0.7373	-0.1145
0.7321	-0.1137
0.7321	-0.1137
0.7321	-0.1134

rete mesh 2x2 pre.	
k=3	
P^T	H
1.0000	1.0000
0.8186	0.2310
0.8147	0.2210
0.8147	0.2030
0.8114	0.2030
0.7637	0.1607
0.7566	0.1452
0.7566	0.1319
0.7539	0.1319
0.7539	-0.1144

Tabella 3.3: I dieci autovalori maggiori in modulo; rete mesh 2x2 con preemption.

rete multipoll	
k=1	
P^T	H
1.0000	1.0000
0.9783	0.3842
0.9783	0.3842
0.9777	0.3801
0.9748	0.3613
0.9710	-0.2941
0.9636	-0.2941
0.9597	-0.2940
0.9589	-0.2940
0.9589	0.2128

rete mesh 2x2	
k=1	
P^T	H
1.0000	1.0000
0.9769	0.3655
0.9596	0.1523
0.9575	0.0836
0.9575	0.0836
0.9551	0.0719
0.9507	-0.0673
0.9507	-0.0673
0.9330	0.0292
0.9320	0.0292

rete mesh 2x2	
k=2	
P^T	H
1.0000	1.0000
0.9920	0.6512
0.9920	0.6512
0.9903	0.4885
0.9853	0.4885
0.9824	0.4278
0.9824	0.4278
0.9822	0.4172
0.9822	0.4172
0.9808	0.2824

rete benchprod	
k=2	
P^T	H
1.0000	1.0000
0.9773	-0.3239
0.9773	0.2336
0.9762	0.0812
0.9753	0.0532
0.9753	0.0255
0.9711	0.0199
0.9591	-0.0132
0.9574	-0.0036
0.9574	0.0020

Tabella 3.4: I dieci autovalori maggiori in modulo; rete mesh 2x2 senza preemption, multipoll e benchprod.

Capitolo 4

Risultati ottenuti

In qusto capitolo sono mostrati i risultati ottenuti dai vari algoritmi applicati alla soluzione di alcune reti di Petri. I modelli utilizzati sono stati tracciati con il programma GreatSPN, un'applicazione grafica dedicata al disegno e all'analisi delle GSPN. Semplici utility convertono i file delle matrici dal formato di GreatSPN a quello utilizzato dai programmi solutori sviluppati, ed i file con i risultati ottenuti dal formato utilizzato a quello di GreatSPN in modo da renderli fruibili direttamente dall' interfaccia grafica.

Per i risultati ottenuti con i precondizionatori e con l'algoritmo TFQMR si é concentrata l'attenzione sull'aspetto numerico (numero di iterazioni, residuo di uscita, parametri di esecuzione) e si sono ripetute le misure per tutte le matrici disponibili. I programmi paralleli sono invece stati sperimentati solo su alcune matrici, in quanto gli aspetti numerici sono equivalenti alle versioni sequenziali.

4.1 Modelli di reti utilizzati

I modelli di reti analizzati sono quattro:

- Mesh 2x2 preemptive
- Mesh 2x2 non preemptive

- Multipoll
- Benchprod

Sono tutti modelli parametrici di tipo GSPN, tranne l'ultimo che é una SPN. Variando il parametro k specifico della rete si sono ottenute piú matrici di diverse dimensioni per ognuna delle reti.

4.1.1 Mesh 2x2

Questa rete, tratta da [13], costituisce il modello di un'architettura parallela di tipo MIMD. I vari nodi di calcolo vengono realizzati da una sottorete e la struttura complessiva é ottenuta componendo i nodi elementari. Il modello GSPN prevede scambio di dati solo fra processori adiacenti e fisicamente interconnessi ed inoltre un carico di lavoro equidistribuito fra i vari nodi. L'interazione fra processori vicini puó essere gestita mediante due distinte politiche:

- **preemptive**. Le richieste dei processori vicini hanno prioritá maggiore di quelle locali e provocano la loro interruzione.
- **non preemptive**. Le richieste dei vicini vengono soddisfatte dopo il completamento dell'operazione locale in corso.

Il parametro k é stato fatto variare per simulare diversi livelli di multitasking ottenendo matrici di dimensioni diverse, elencate nella tabella 4.1 per il caso non preemptive, ed in tabella 4.2 per il caso preemptive. Per il modello non preemptive si sono considerati un numero inferiore di casi poiché la dimensione delle matrici ottenute é aumentata molto piú rapidamente rispetto al caso preemptive.

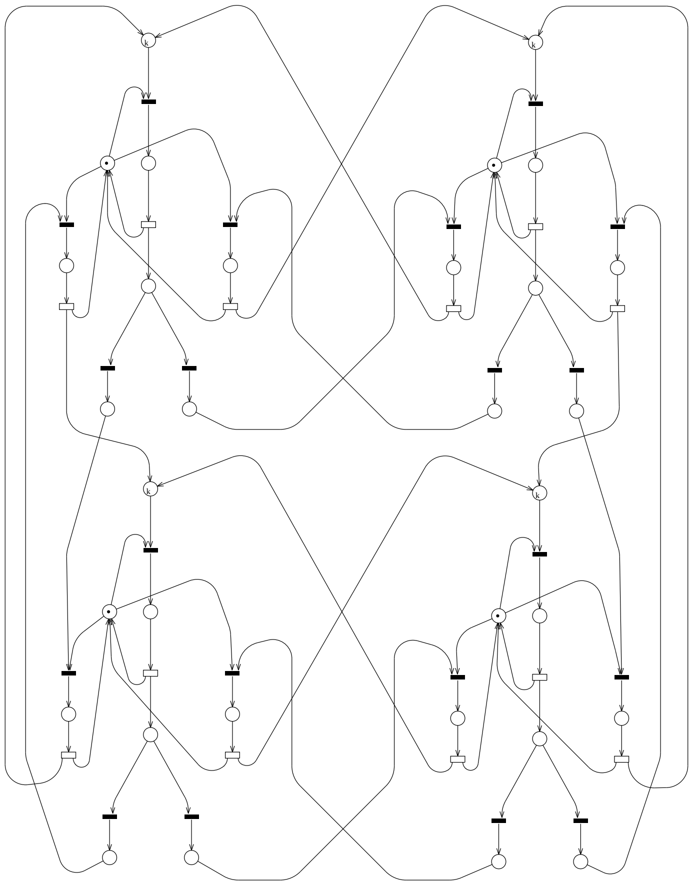


Figura 4.1: Modello di un sistema multiprocessore Mesh 2x2 senza preemption

k	numero di stati	elementi non nulli
1	189	893
2	5327	32167
3	37527	250071
4	141537	990145
5	384917	2770645

Tabella 4.1: Matrici generate dalla rete Mesh 2x2 senza preemption.

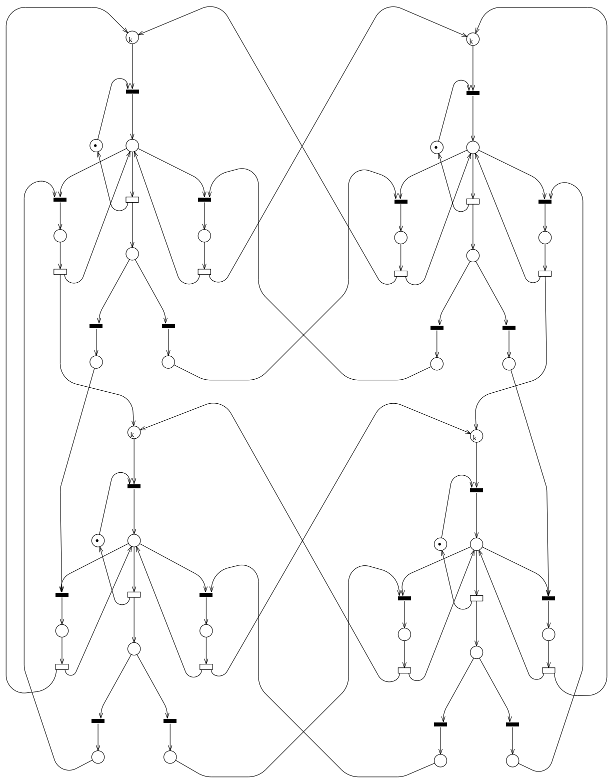


Figura 4.2: Modello di un sistema multiprocessore Mesh 2x2 con preemption

k	numero di stati	elementi non nulli
1	57	193
2	461	2117
3	1837	9637
4	5145	29065
5	11681	69161
6	23077	141133
7	41301	258637
8	68657	437777
9	107785	697102
10	161661	1057621
11	233597	1542773

Tabella 4.2: Matrici generate dalla rete Mesh 2x2 con preemption.

4.1.2 Random polling system

Il sistema di polling piú semplice ma anche piú comune é costituito da un insieme di linee dati che ricevono messaggi dall'esterno e da un'unico server. Tale server periodicamente verifica se sono arrivate nuove informazioni.

Il modello GSPN utilizzato, definito in [13], presuppone una situazione piú complessa in cui i server sono 3 e sono rappresentati dalla marcatura iniziale in figura 4.3. Tali sistemi vengono definiti *multiserver polling* oppure *MSMQ (multiserver multiqueue systems)*. Il sistema modellizzato é anche di tipo random polling, perché l'ordine con cui il server visita le code é casuale. Un'altra differenza del modello utilizzato rispetto a quello classico é che al posto della transizione temporizzata diversa per ogni coda, che rappresenta il tempo impiegato dal server per spostarsi, viene utilizzata un'unica transizione temporizzata che rappresenta il tempo di spostamento e che viene conteggiato prima della scelta della successiva coda da servire. Questo cambiamento é possibile perché il tempo di spostamento é indipendente dalla coda scelta. Il modello é perció di tipo *walking before choosing* ed ha il vantaggio di ridurre notevolmente il numero delle marcature tangibili. In tabella 4.3 sono descritte le matrici ottenute al variare di k, che in questo caso rappresenta la dimensione delle code. Nel resto del capitolo tale tipo di rete verrà chiamata semplicemente *multipoll*.

k	numero di stati	elementi non nulli
1	80	396
2	933	5965
3	4240	29668
4	12405	91089
5	28704	216748
6	57253	440353
7	103008	802692

Tabella 4.3: Caratteristiche delle matrici generate dalla rete Multipoll.

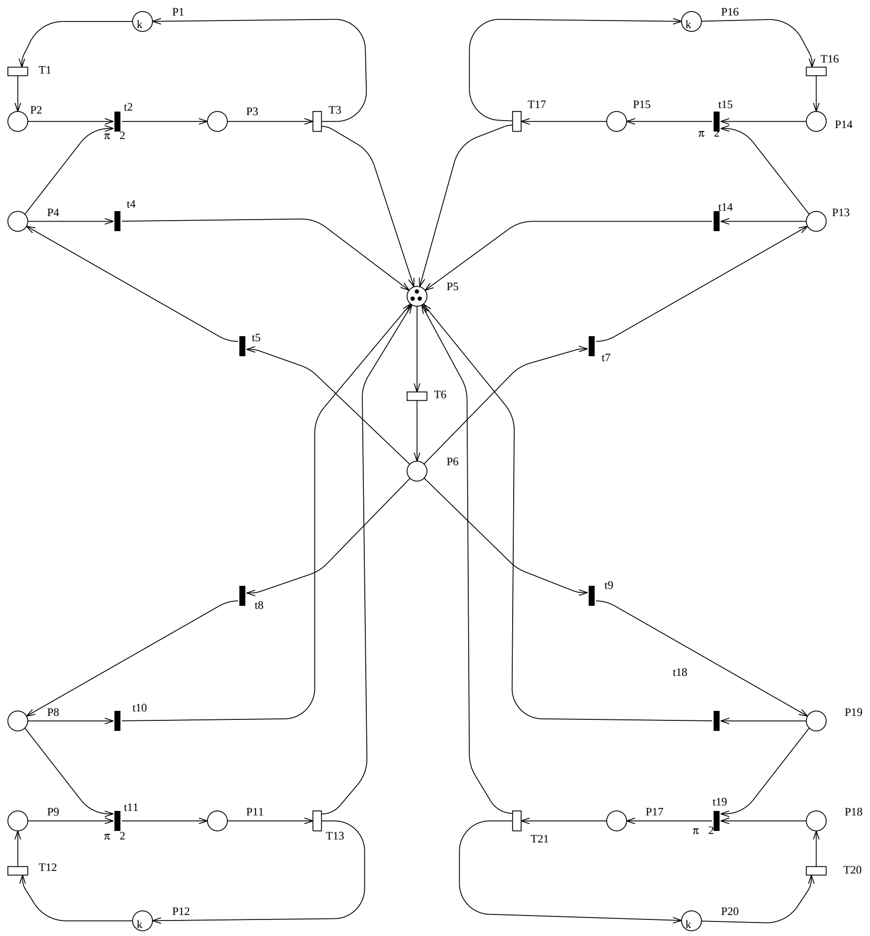


Figura 4.3: Modello di un sistema random polling multiserver

4.1.3 Rete benchprod

Il modello descritto da questa rete rappresenta la linea di trasporto di un sistema di produzione ed é stato sviluppato in [17]. Nel sistema descritto i posti sono le risorse del sistema (macchine, dati e linee di comunicazione), un gettone in un posto indica che la risorsa é disponibile mentre le transizioni rappresentano inizio e fine di un evento. Questa rete presenta una differenza sostanziale da quelle precedenti in quanto é di tipo SPN, e quindi priva di transizioni immediate.

Il parametro k rappresenta il numero massimo di prodotti sulla linea e le caratteristiche delle matrici al variare di k sono riportate nella tabella 4.4. La figura 4.4

descrive in dettaglio il modello utilizzato.

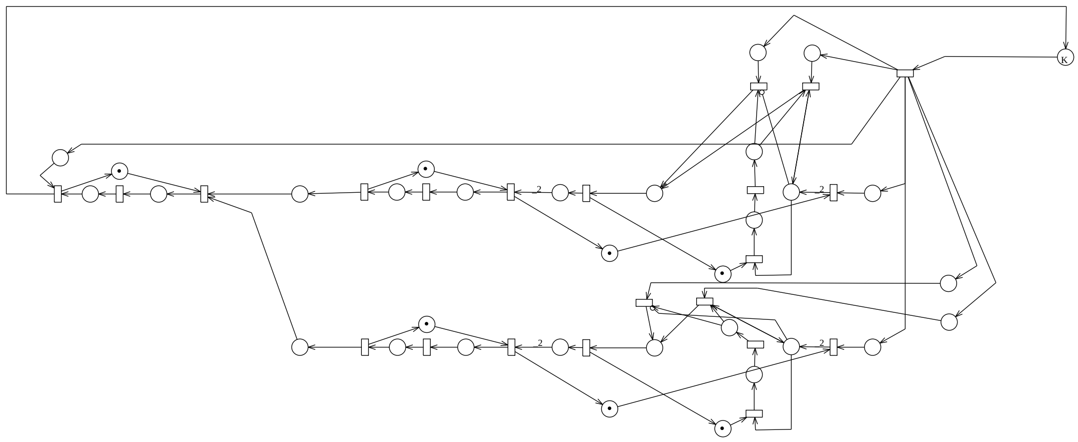


Figura 4.4: Modello di un sistema di produzione

k	numero di stati	elementi non nulli
1	172	488
2	2359	9137
3	11386	51986
4	32653	161939
5	71560	372476
6	133507	717007
7	223894	1229222

Tabella 4.4: Caratteristiche delle matrici generate dalla rete Benchprod.

4.2 GMRES con precondizionatore ILU0

In questa sezione verranno esaminati i risultati della versione precondizionata mediante precondizionatore ILU0 dell' algoritmo GMRES, ed in particolare essi saranno confontati con quelli ottenuti dalla versione non precondizionata e dalla versione con precondizionatore diagonale ottenuti in [11]. I risultati in [11] hanno posto in evidenza come la versione non precondizionata del GMRES mostri ottime caratteristiche di affidabilitá e precisione, riuscendo a convergere in tutti i casi trattati ed anche per valori estremamente bassi del residuo di uscita richiesto, ma richiedendo al contempo tempi di esecuzione normalmente piú elevati rispetto a metodi come il BiCGStab oppure il Gauss-Seidel.

Sará infine fatto un sintetico confronto con i risultati ottenuti (sempre in [11]) con gli algoritmi BiCGStab e Gauss-Seidel.

4.2.1 Rete mesh 2×2 con preemption

Inizialmente sono state prese in esame le matrici ottenute dalla rete relativa al modello mesh 2x2 con preemption. Per una prima serie di misure si é tenuto fisso il residuo di uscita ad un valore di 10^{-7} e si é fatto variare il parametro k della rete, misurando il numero di iterazioni ed il tempo di calcolo richisto dall'algoritmo GMRES precondizionato. Nella tabella 4.5 sono riportati il valore ottimo della dimensione m del sottospazio di Krylov utilizzato, il tempo di esecuzione ed il numero di iterazioni impiegate sia dalla versione precondizionata del GMRES(colonne di destra) sia, a titolo di confronto, di quella non precondizionata.

Dai dati riportati in tabella 4.5 si puó osservare come il numero di iterazioni necessario decresca regolarmente al crescere di k: questo é dovuto al fatto che contemporaneamente si verifica una crescita del valore ottimo del parametro m, come era ragionevole aspettarsi. Si puó rilevare poi come la versione precondizionata sia generalmente piú veloce dell'altra, con una differenza percentuale dei tempi di esecuzione che cresce all'aumentare di k. Anche il numero di iterazioni ed il valore di

	GMRES non precond.			**GMRES precondizionato**		
k	N^o iter.	Tempo(sec.)	m ottimo	N^o iter.	Tempo(sec.)	m ottimo
1	4	0.01	4	3	0.01	3
2	4	0.04	6	2	0.04	4
3	4	0.13	8	3	0.14	4
4	3	0.44	12	2	0.39	5
5	3	1.25	16	2	0.88	8
6	2	3.40	23	1	1.87	8
7	2	7.15	26	1	3.61	9
8	2	14.1	30	1	6.91	9
9	2	25.5	33	1	9.98	9
10	2	44.8	37	1	15.8	10
11	2	87.7	41	1	23.8	11

Tabella 4.5: Rete mesh 2x2 con preemption: confronto dei tempi di esecuzione al variare di k, $\epsilon = 10^{-7}$.

m sono inferiori nella versione precondizionata che quindi, per questo tipo di rete ha mostrato una buona efficienza. Nella tabella 4.6 sono invece riportati, al variare di m, il numero di iterazioni richieste, il tempo di esecuzione ed il residuo di uscita prodotto relativi alla matrice con k=9. Si puó vedere come anche per valori di m molto piccoli, in particolare fino ad m=3, l'algoritmo riesca a convergere, e sempre con tempi inferiori alla versione non precondizionata, ad ulteriore riprova dei benefici effetti del precondizionatore con il problema in esame. Infine nella tabella 4.7 si é misurato il comportamento dell'algoritmo al variare della precisione richiesta: sono state confermate le ottime capacitá del GMRES di ottenere anche precisioni elevate; in particolare si é ottenuto un residuo di uscita inferiore a 10^{-13} con un'aumento del tempo di esecuzione del 29% rispetto al caso $\epsilon = 10^{-7}$.

m	N^o iterazioni	Tempo(secondi)	residuo di uscita
2	200 (*)	213	1.41e-02(*)
3	11	17.9	5.91e-08
4	4	12.3	1.04e-08
5	3	11.8	5.42e-10
6	2	11.2	2.67e-08
7	2	11.8	4.56e-10
8	2	12.2	8.91e-12
9	1	9.98	9.38e-08
10	1	10.4	7.54e-09
11	1	11.1	1.59e-09
12	1	11.4	1.42e-09
15	1	12.4	1.42e-09
20	1	14.6	1.42e-09
30	1	19.8	1.42e-09

Tabella 4.6: Rete mesh 2x2 con preemption: tempi di esecuzione al variare di m per il GMRES precondizionato, $\epsilon = 10^{-7}$ e k=9.

Residuo	N^o iterazioni	m ottimo	Tempo(secondi)
10^{-5}	1	7	9.42
10^{-6}	1	8	9.66
10^{-7}	1	9	9.98
10^{-8}	1	10	10.4
10^{-9}	2	8	12.3
10^{-10}	2	8	12.3
10^{-11}	2	8	12.3
10^{-12}	2	9	12.9
10^{-13}	2	9	12.9

Tabella 4.7: Rete mesh 2x2 con preemption: numero di iterazioni al variare di ϵ per il GMRES precondizionato, k=9.

4.2.2 Rete mesh 2×2 senza preemption

Ripetendo la procedura giá utilizzata nel paragrafo precedente verranno riportati prima i tempi di esecuzione ed il numero di iterazioni impiegati mantenendo fisso ϵ

	GMRES non precond.			GMRES precondizionato		
k	N^o iter.	Tempo(sec.)	m ottimo	N^o iter.	Tempo(sec.)	m ottimo
1	16	0.02	4	3	0.02	4
2	16	0.79	7	2	0.47	7
3	12	9.25	11	1	3.94	13
4	12	56.9	15	1	17.1	14
5	8	210	22	2	51.9	15

Tabella 4.8: Rete mesh 2x2 senza preemption: confronto dei tempi di esecuzione al variare di k, $\epsilon = 10^{-7}$.

al valore di 10^{-7} e variando k da 1 a 5. In seguito saranno esaminati i risultati al variare di m e quindi al variare di ϵ.

Dai dati della tabella 4.8 si puó dedurre che anche per questo tipo di reti le prestazioni dell'algoritmo precondizionato sono sostanzialmente migliori di quello non precondizionato: in particolare per i valori massimi di k considerati i tempi di esecuzione si riducono a circa un quarto di quelli necessari alla versione non precondizionata. Contrariamente al caso precedente invece risulta che i valori ottimi di m della versione precondizionata sono spesso uguali o comunque prossimi a quelli della versione non precondizionata.

L'effetto delle variazioni del parametro m é mostrato in tabella 4.9, relativa al caso k=4 (141537 stati), dove si puó notare come il tempo di esecuzione presenti un minimo al variare di m. Del resto é immediato osservare come per m superiore o uguale a 14 l'algoritmo richieda una sola iterazione per convergere, mentre ulteriori aumenti di m rendono solo piú complessa l'esecuzione del processo di Arnoldi senza portare altro vantaggio se non una riduzione del residuo di uscita. La tabella 4.10 mostra il comportamento al variare del residuo richiesto, relativamente al caso k=3: anche in questo caso l'algoritmo precondizionato non ha avuto problemi a raggiungere elevate precisioni senza un peggioramento drastico dei tempi di esecuzione, che aumentano di circa il 30% passando da $\epsilon = 10^{-5}$ ad $\epsilon = 10^{-13}$.

m	N^o iterazioni	Tempo(secondi)	residuo di uscita
12	2	22.4	4.54e-12
13	2	22.5	2.92e-13
14	1	17.1	3.46e-8
15	1	17.4	1.00e-8
16	1	18.3	2.46e-9
17	1	18.5	1.71e-9
20	1	20.4	1.66e-9
30	1	27.5	1.66e-9

Tabella 4.9: Rete mesh 2x2 senza preemption: tempi di esecuzione al variare di m per il GMRES precondizionato, $\epsilon = 10^{-7}$ e k=4.

Residuo	N^o iterazioni	m ottimo	Tempo(secondi)
10^{-5}	1	10	9.42
10^{-6}	1	11	9.66
10^{-7}	1	13	10.0
10^{-8}	1	14	10.4
10^{-9}	2	10	12.3
10^{-10}	2	12	12.3
10^{-11}	2	12	12.3
10^{-12}	2	12	12.9
10^{-13}	2	12	12.9

Tabella 4.10: Rete mesh 2x2 senza preemption: numero di iterazioni al variare di ϵ per il GMRES precondizionato, k=3.

4.2.3 Rete multipoll

In questo paragrafo si continua l'analisi dei risultati prodotti dal GMRES precondizionato, utilizzando le matrici relative alla rete multipoll. Come termine di confronto si é scelto in questo caso il GMRES precondizionato mediante un semplice precondizionatore diagonale: per applicare tale precondizionatore é sufficiente dividere gli elementi di ogni riga della matrice per l'elemento diagonale della riga stessa. Tale versione é risultata piú efficiente della versione non precondizionata [11]; é parso allora opportuno confrontare i risultati dei due diversi precondizionatori. La tabella 4.11 raccoglie i tempi di esecuzione ed il numero di iterazioni al variare di k e con ϵ fissato a 10^{-7}. In questo caso il miglioramento dei tempi ottenuti dal precondizionatore basato sulla fattorizzazione incompleta si riduce rispetto ai casi precedentemente descritti, rimanendo peró piuttosto consistente (superiore ad un fattore 2 per i valori piú grandi di k). Un'altra differenza é costituita dal valore ottimo del parametro m utilizzato dal GMRES con precondizionatore ILU: mentre nei casi finora trattati tale valore risultava sempre inferiore rispetto al GMRES non precondizionato in questo caso risulta sempre superiore al GMRES con precondizionatore diagonale, ad eccezione del caso k=7.

	GMRES + prec. diagonale			**GMRES + prec. ILU0**		
k	N^o iter.	Tempo(sec.)	m ottimo	N^o iter.	Tempo(sec.)	m ottimo
1	13	0.01	3	3	0.01	4
2	21	0.10	3	5	0.10	5
3	16	0.53	5	3	0.48	8
4	14	1.92	7	2	1.45	18
5	6	6.67	16	1	3.91	19
6	12	23.1	15	1	8.20	19
7	2	37.7	41	1	15.5	20

Tabella 4.11: Rete multipoll: confronto dei tempi di esecuzione del GMRES con precondizionatore diagonale e GMRES con precondizionatore ILU(0) al variare di k, $\epsilon = 10^{-7}$.

4.2.4 Rete benchprod

L'ultimo caso trattato é relativo alla rete benchprod. Il confronto viene fatto ancora con il GMRES con precondizionatore diagonale. La tabella 4.12 riassume i risultati dei tempi di esecuzione, numero di iterazioni ed m ottimo per $\epsilon = 10^{-7}$.

Con questo tipo di rete si sono osservati i miglioramenti piú significativi ottenuti

	GMRES + prec. diagonale			GMRES + prec. ILU0		
k	N^o iter.	Tempo(sec.)	m ottimo	N^o iter.	Tempo(sec.)	m ottimo
1	14	0.03	12	1	0.01	2
2	62	0.60	6	2	0.12	4
3	33	4.01	11	3	0.76	4
4	67	23.8	10	3	2.93	6
5	26	102	25	2	7.48	8
6	16	265	40	1	15.0	14
7	16	763	47	1	23.9	16

Tabella 4.12: Rete benchprod: confronto dei tempi di esecuzione del GMRES con precondizionatore diagonale e GMRES con precondizionatore ILU(0) al variare di k, $\epsilon = 10^{-7}$.

m	N^o iterazioni	Tempo(secondi)	residuo di uscita
5	9	38.7	7.16e-8
10	3	30.0	8.72e-8
12	2	27.3	7.05e-8
13	2	28.8	2.00e-9
14	2	30.8	9.77e-10
15	2	31.3	6.21e-10
16	1	23.9	5.34e-8
17	1	23.9	4.09e-9
18	1	24.6	8.48e-10
19	1	25.5	2.09e-10
20	1	26.2	1.99e-10
30	1	40.9	4.0e-10
40	1	49.7	4.0e-10

Tabella 4.13: Rete bencprod: tempi di esecuzione al variare di m per il GMRES precondizionato, $\epsilon = 10^{-7}$ e k=4.

grazie all'impiego del precondizionatore basato sulla fattorizzazione incompleta di tipo LU: per k=7 i tempi di esecuzione sono diminuiti di ben 31 volte, ed inoltre é stata sufficiente una iterazione per ottenere la soluzione, contro le 16 necessarie al

GMRES con precondizionatore diagonale. Anche i valori ottimi di m sono risultati sempre inferiori per il GMRES+ILU, al contrario del caso della rete multipoll trattato in precedenza. Nella tabella 4.13 si é fatto variare m osservando come questo influenzasse i tempi di calcolo ed il residuo di uscita. Si puó osservare la presenza di minimi locali che si spiegano in questo modo: i tempi di esecuzione tendono ad aumentare al crescere di m fintanto che non si ha una diminuzione del numero di iterazioni necessarie; quando tale numero diminuisce si ha contemporaneamente una diminuzione dei tempi di esecuzione.

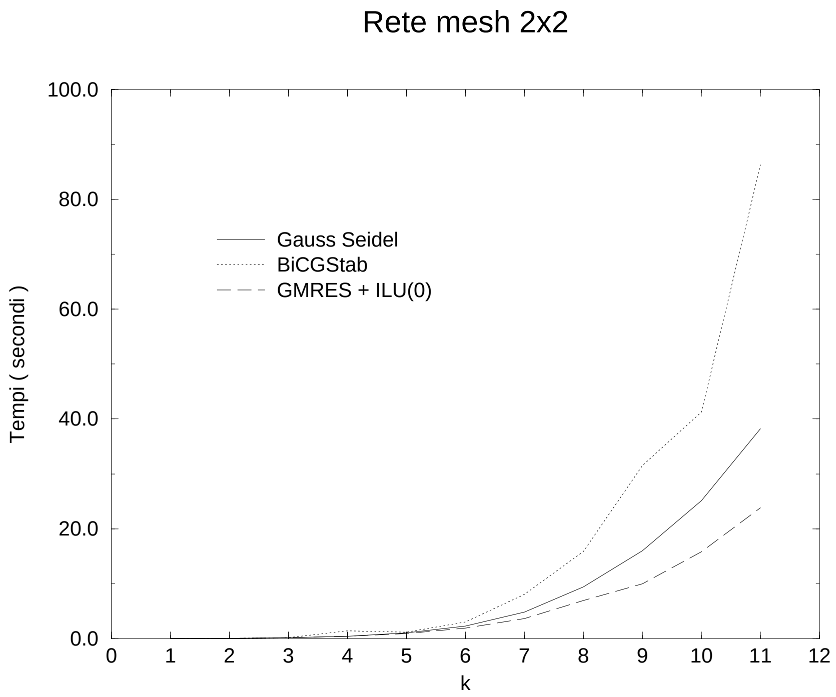
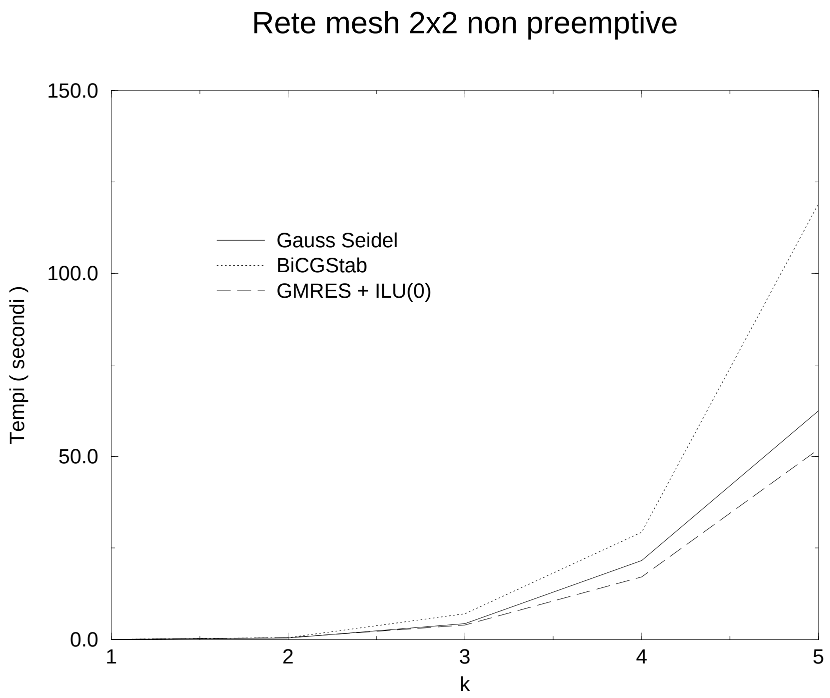


Figura 4.5: Tempi di esecuzione al variare di k

4.2.5 Confronto con altri metodi

Per concludere l'analisi sul comportamento del GMRES con precondizionatore ILU0 vengono riportati i grafici 4.5, 4.6, 4.7 e 4.8 dove i risultati ottenuti con tale algoritmo vengono posti a confronto con quelli relativi al BiCGStab ed al Gauss-Seidel.



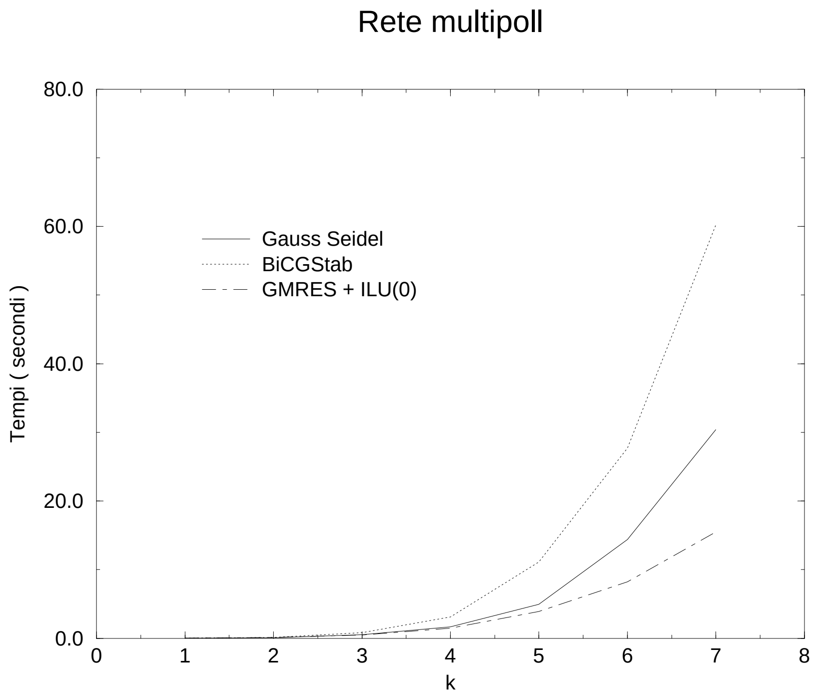


Figura 4.7: Tempi di esecuzione al variare di k

In tutti i casi il GMRES con ILU0 é risultato il metodo piú veloce. Considerando il numero limitato di casi presi in esame non si vogliono trarre conclusioni generali, ma é comunque interessante il confronto con i risultati ottenuti con gli altri due metodi, in particolare il Gauss-Seidel.

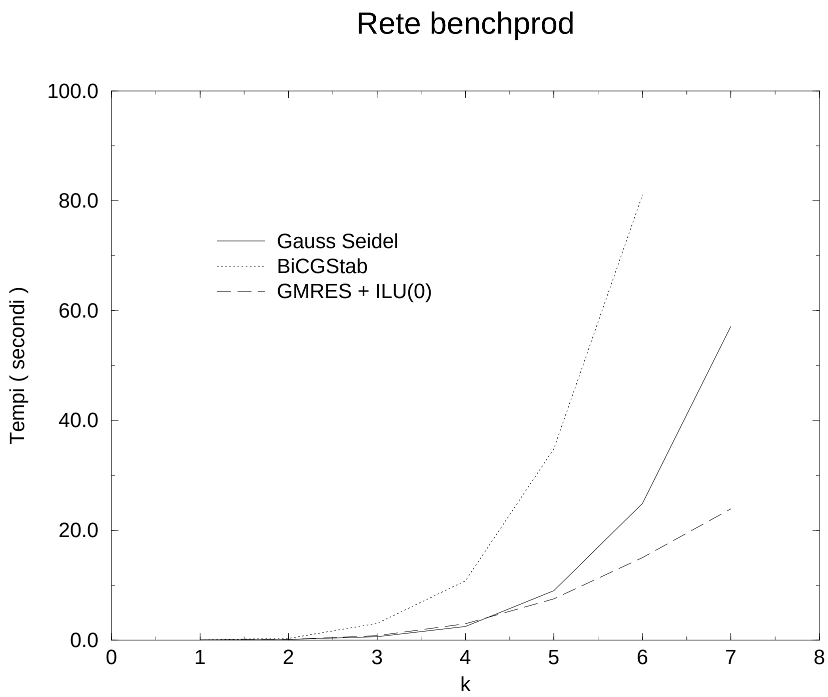


Figura 4.8: Tempi di esecuzione al variare di k

4.3 GMRES con precondizionatore ILUK

In questo paragrafo vengono descritti i risultati ottenuti con il precondizionatore ILUK applicato al GMRES. Le prime due tabelle riportate sono relative alla rete benchprod con k=3 (caratterizzata da 11386 stati e 51986 elementi diversi da zero) ed ai parametri di esecuzione $m = 9$ e precisione richiesta $\epsilon = 10^{-7}$. In tabella 4.14 é mostrato l'andamento del residuo di uscita e del numero di iterazioni al variare del

ilK	N^o iterazioni	residuo di uscita
3	18	4.291e-08
5	2	3.220e-8
7	2	5.192e-9
9	1	3.800e-8
15	1	2.959e-08
30	1	1.343e-08
60	1	4.736e-9

Tabella 4.14: Rete benchprod con k=3: numero di iterazioni e residuo di uscita al variare di ilK.

parametro ilK [1]. Nella successiva tabella 4.15 é descritto l'andamento del tempo di esecuzione sempre al variare di ilK; nell'ultima colonna é messa in evidenza la variazione della dimensione della matrice di precondizionamento al variare di ilK. Queste prime due tabelle hanno lo scopo di mettere in evidenza gli effetti della

ilK	Tempo di esecuzione (s)	dimensione precondizionatore (migliaia di elementi)
3	28.7	34
5	28.8	57
7	36.8	79
9	44.8	102
15	66.0	171
30	125	341
60	280	683

Tabella 4.15: Rete benchprod con k=3: tempi di esecuzione e dimensioni precondizionatore al variare di ilK.

scelta del parametro ilK, e quindi delle dimensioni del precondizionatore, sui tempi

[1] Si è qui indicato con ilK il parametro specifico del precondizionatore ILUK, chiamato semplicemente K nel capitolo dedicato ai precondizionatori, per distinguerlo dal parametro k delle reti.

di esecuzione e sul residuo di uscita ottenuto. Per maggior chiarezza tali dati sono stati rappresentati nei grafici delle figure 4.9 e 4.10.

Inanzitutto si puó notare come il tempo di esecuzione, dominato per questo tipo di precondizionatore dalla fase di costruzione del precondizionatore stesso, abbia un andamento pressoché lineare al crescere di ilK. Questo risultato é concorde con le aspettative, in quanto anche il numero di elementi del precondizonatore (e quindi il numero di elementi da "manipolare" durante la fase di costruzione e di utilizzo del precondizionatore) cresce linearmente con ilK.

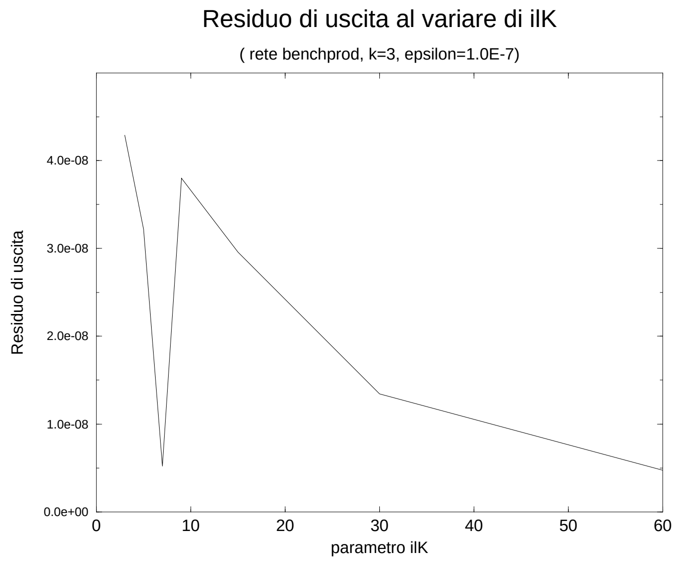


Figura 4.9: Rete benchprod con k=3: residuo di uscita al variare del parametro ilK

Anche l'andamento del residuo di uscita al variare di ilK mostra un andamento prevedibile. Inizialmente scende al crescere di ilK, quindi torna a salire in corrispondenza di $ilK = 9$, perché nel passaggio da $ilK = 7$ ad $ilK = 9$ si ha una diminuzione del numero di iterazioni. Infine continua a scendere al crescere di ilK. Tale comportamento si puó spiegare considerando che al crescere di ilK aumenta il numero di elementi del precondizionatore (diminuisce il numero di elementi scartati dalla fattorizzazione incompleta) e quindi, almeno qualitativamente, dovrebbe migliorare l'approssimazione $U^{-1}L^{-1} \simeq A^{-1}$. É opportuno notare che la diminuzione del residuo al crescere di ilK é comunque modesta: quasi un ordine di grandezza passando da $ilK = 9$ ad $ilK = 60$.

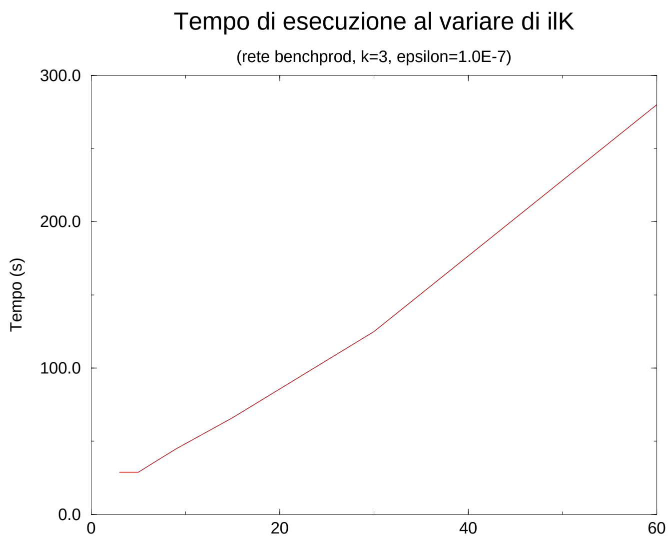


Figura 4.10: Rete benchprod con k=3: tempi di esecuzione al variare del parametro ilK

Nella tabella 4.16 sono riportati i risultati relativi alle prime quattro matrici generate dal modello benchprod. Per ciascuna matrice, il parametro ilK é stato scelto uguale alla media di elementi non nulli di ogni riga, in modo da rendere le dimensioni del precondizionatore analoghe a quelle del caso ILU0. Come si puó osservare da un confronto con i risultati del paragrafo precedente, il precondizionatore ILUK risulta nettamente piú dispendioso in termini di tempo di esecuzione di quello ILU0.

k	ilK	m	Tempo (s)	N iterazioni
1	3	2	0.02	1
2	4	7	1.22	2
3	5	9	28.6	2
4	5	14	234	3

Tabella 4.16: Reti benchprod: tempi di esecuzione GMRES+precond.ILUK al variare di k (scelta ilK = numero medio elementi per riga; m = valore ottimo)

Il numero di iterazioni si mantiene sempre basso e, poiché le dimensioni del precondizionatore ILUK sono in questo caso rese analoghe a quelle dell'ILU0, anche il costo della singola iterazione é sostanzialmente uguale; nel caso ILUK é peró determinante il tempo impiegato dalla fase di costruzione del precondizionatore.

Risultati analoghi, e che mostrano come il precondizionatore ILUK risulti meno efficace di quello ILU0 per questa categoria di problemi, sono stati ottenuti anche con le altre reti.

4.4 GMRES parallelo

Nel paragrafo 4.2 si é visto come l'uso del precondizionatore sia in grado di migliorare notevolmente le prestazioni del programma seriale, ma si é anche osservato in §2.1 che la fase di costruzione del precondizionatore e le operazioni di soluzione dei sistemi triangolari necessarie per il suo uso non sono efficacemente parallelizzabili. Nella seguente tabella 4.17 vengono misurati i tempi di esecuzione delle porzioni di codice non parallelizzate: le misure sono relative a due diverse matrici ed i risultati percentuali sono calcolati come media fra le due.

Contributi		rete mesh 2x2 preemp., k=7	rete mesh 2x2 preemp., k=9	percentuale
T_{tot}		4.25	11.67	100 %
	T_{IO}	2.80	7.55	64 %
	T_{calc}	1.45	4.11	36 %
T_{calc}		1.45	4.11	100 %
	T_{iter}	1.18	3.32	81 %
T_{iter}		1.18	3.32	100 %
	parall.	0.74	2.01	62 %
	non parall.	0.44	1.31	38 %

Tabella 4.17: GMRES: contributi sui tempi di esecuzione di singole parti del codice.

Il tempo totale di esecuzione del programma, indicato con T_{tot}, viene suddiviso in tre termini distinti riferiti a specifiche fasi dell'elaborazione: il T_{IO} rappresenta il tempo speso in operazioni di I/O (lettura da disco della matrice), il $T_{calc} = T_{tot} - T_{IO}$ é il tempo totale di calcolo, mentre T_{iter} é il tempo di calcolo destinato al processo iterativo e quindi T_{calc} meno il tempo speso nella costruzione del precondizionatore.

A sua volta T_{iter} é formato da due contributi, uno dovuto alla parte di codice parallelizzato e l' altro relativo alla risoluzione dei sistemi triangolari ed al calcolo delle rotazioni piane (quest'ultima funzione ha peró un costo trascurabile), che costituiscono la parte non parallelizzata del procedimento iterativo.

numero processori	**speedup (su T_{calc})**	**speedup (su T_{iter})**
1	1.00	1.00
2	1.33	1.44
3	1.50	1.70
4	1.60	1.86
6	1.71	2.06
8	1.77	2.18

Tabella 4.18: Stima del massimo speedup ottenibile dal programma GMRES parallelo realizzato.

Come si vede i contributi delle porzioni di codice non parallelizzato sono significativi: se da un lato l'uso del precondizionatore ha permesso di ottenere la convergenza in pochissime iterazioni (quasi sempre una o due) ed in tempi estremamente brevi, dall'altro esso limita fortemente il livello di parallelismo sfruttabile. Adoperando i valori riportati nella tabella appena citata é possibile calcolare lo speedup massimo teorico ottenibile con il programma parallelo realizzato, ed i cui valori sono riportati nella tabella 4.18.

Nei paragrafi successivi verrano riportati i risultati sperimentali ottenuti con le versioni del GMRES parallelo basate su MPI ed OpenMP. I dati sono relativi ad un gruppo di matrici scelte perché rappresentative di diverse dimensioni del problema trattato. Tutti i dati delle seguenti tabelle sono stati ottenuti con $\epsilon = 10^{-7}$, mentre i valori di m sono gli stessi utilizzati per la versione seriale (elencati nelle tabelle del paragrafo precedente).

4.4.1 Risultati con MPI

La tabella 4.19 mostra i risultati ottenuti con un uso esclusivo della macchina e relativi al programma *message passing* basato su MPI. Vengono confermate le previsioni fatte in precedenza: la rilevanza delle parti non parallele limita fortemente lo speedup, che rimane sempre inferiore a 2. Si puó notare come i risultati migliori si siano ottenuti con la matrice di dimensioni maggiori fra le tre riportate nella tabella (la benchprod con k=7 che possiede 223894 stati), il che é comprensibile se si considera che tutti gli overhead dovuti a sincronizzazioni ed alla gestione delle comunicazioni sono indipendenti dalle dimensioni delle matrici e quindi percentualmente piú significativi per le matrici piú piccole.

I dati mostrano anche come il programma non scali bene con il numero dei processori: si raggiunge un valore massimo di speedup con 4 o 6 processori, mentre con 8 i risultati non migliorano.

rete	**Numero processori**	T_{tot} **(s)**	T_{calc} **(s)**	T_{iter} **(s)**	**Speedup (su T_{iter})**
mesh 2x2 con preemption,k=7	1	4.10	1.43	1.19	1.00
	2	3.98	1.29	1.01	1.17
	3	3.90	1.23	0.95	1.25
	4	3.83	1.20	0.92	1.29
	6	3.84	1.21	0.93	1.28
	8	4.02	1.29	1.00	1.19
mesh 2x2 senza preemption,k=9	1	11.29	4.06	3.36	1.00
	2	10.95	3.74	2.95	1.14
	3	10.84	3.61	2.79	1.20
	4	10.59	3.50	2.67	1.26
	6	10.63	3.54	2.70	1.24
	8	11.27	3.85	2.91	1.15
benchprod,k=7	1	28.48	14.68	13.37	1.00
	2	24.46	11.66	10.29	1.33
	3	23.49	10.69	9.25	1.48
	4	22.83	10.31	8.80	1.56
	6	22.67	10.21	8.71	1.57
	8	23.84	10.87	9.41	1.45

Tabella 4.19: GMRES con MPI, tempi di esecuzione e speedup ottenuti con un uso esclusivo della macchina.

Nella tabella 4.20 sono invece riportati i risultati ottenuti in ambiente multiutente. Come era logico attendersi i valori di speedup peggiorano. Si puó notare come l'esecuzione con 8 processi, che corrisponde al numero di processori presenti sull'elaboratore utilizzato (descritto in §1.5) provochi un notevole peggioramento dei tempi: in ambiente multiutente é rarissimo che tutti i processori disponibili siano liberi ed inoltre é difficile che il sistema operativo allochi tutte le risorse ad un'unica applicazione (nonostante il sistema Irix 6.4 preveda il *gang scheduling* per le applicazioni parallele).

rete	**Numero processori**	T_{tot} **(s)**	T_{calc} **(s)**	T_{iter} **(s)**	**Speedup (su T_{iter})**
mesh 2x2 con preemption,k=7	1	5.22	2.49	2.23	1.00
	2	5.03	2.19	1.88	1.19
	3	4.78	1.95	1.65	1.35
	4	4.44	1.84	1.54	1.44
	6	5.23	2.37	2.05	1.09
	8	20.08	16.65	16.20	0.14
mesh 2x2 con preemption,k=9	1	14.38	7.09	6.35	1.00
	2	14.12	6.52	5.63	1.12
	3	14.50	6.71	5.69	1.11
	4	12.98	5.68	4.76	1.33
	6	12.85	5.59	4.72	1.34
	8	30.21	20.91	19.77	0.32
benchprod,k=7	1	28.36	15.14	13.63	1.00
	2	24.61	11.72	10.26	1.32
	3	24.39	10.90	9.36	1.45
	4	23.59	10.98	9.41	1.45
	6	23.85	11.35	9.80	1.39
	8	51.80	35.61	33.45	0.40

Tabella 4.20: GMRES con MPI, tempi di esecuzione e speedup ottenuti in ambiente multiutente.

4.4.2 Risultati con OpenMP

I risultati mostrati in questo paragrafo sono tutti relativi all'uso in modalitá multiutente della macchina su cui sono state effettuate le prove. Anche in questo caso viene presentata una selezione dei risultati piú significativi. Mentre con MPI la misura dei tempi é stata fatta con la funzione specifica di tale libreria MPI_Wtime(), in questo caso si é usata la funzione timex() del sistema operativo Irix che consente una valutazione accurata sia dei tempi reali, che dei tempi "utente". I valori di speedup,

rete	**Numero processori**	**T. totale real (s)**	**T. totale user (s)**	**T. calc. user (s)**	**T. iter. user (s)**	**Speedup**
multipoll, k=5	1	4.53	4.32	2.23	2.09	1.00
	2	4.15	3.78	1.69	1.55	1.34
	3	4.91	3.94	1.84	1.68	1.24
	4	4.58	3.99	1.90	1.76	1.19
	6	5.67	4.27	2.18	2.04	1.02
	8	8.93	5.10	3.01	2.87	0.78
multipoll, k=7	1	19.70	18.90	11.16	10.52	1.00
	2	15.87	14.97	7.22	6.66	1.57
	3	15.97	14.52	6.70	6.12	1.71
	4	15.34	14.19	6.44	5.88	1.78
	6	20.99	16.00	8.26	7.70	1.36
	8	71.22	16.13	8.39	7.83	1.34
mesh 2x2 senza preem., k=5	1	58.72	56.66	29.15	26.69	1.00
	2	52.02	49.99	22.45	20.02	1.33
	3	52.10	49.98	22.45	19.88	1.33
	4	62.81	50.79	23.27	20.83	1.28
	6	75.15	53.66	26.15	23.70	1.26
	8	91.88	54.68	27.17	24.71	1.08

Tabella 4.21: GMRES con OpenMP, tempi di esecuzione e speedup ottenuti.

seppure lievemente migliori di quelli ottenuti con MPI confermano il basso livello di parallelismo sfruttabile con il GMRES con precondizionatore ILU.

4.5 BiCGStab parallelo

Come é stato fatto con il GMRES, anche per l'algoritmo BiCGStab si é studiato l'impatto delle parti di programma non parallelizzate sui tempi di esecuzione. Il risultato, ottenuto da misure ripetute relative ad alcune matrici (le mesh 2x2 con preemption con k=9 e k=11, la multipoll con k=7 e la benchprod con k=7) é mostrato in tabella 4.22.

Contributi	Incidenza sul tempo di calcolo (percentuale)
Parte parallela	85 %
Parte sequenziale	15 %

Tabella 4.22: BiCGStab: contributi sui tempi di esecuzione delle parti parallele e non.

Basandosi su questi dati é possibile stimare lo speedup massimo teorico ottenibile con il programma parallelo realizzato, i cui valori sono riportati nella tabella 4.23, e sono calcolati sul tempo di calcolo (esclusa quindi la parte di I/O).

numero processori	**massimo speedup sul tempo di calcolo**
1	1.00
2	1.73
3	2.30
4	2.75
6	3.42
8	3.90

Tabella 4.23: Stima del massimo speedup ottenibile dal BiCGStab parallelo realizzato.

4.5.1 Risultati con MPI

Nella tabella 4.24 sono riportati i risultati ottenuti dalla versione MPI del BiCGStab parallelo ed operando sull'elaboratore in modalitá esclusiva. I valori di speedup sono relativi al tempo di calcolo, e quindi non si é considerata la fase di I/O. Dai dati riportati si puó osservare che si ha un certo miglioramento rispetto all'algoritmo GMRES per quel che riguarda lo speedup, che raggiunge valori fino a 2.

rete	**Numero processori**	T_{tot} (s)	T_{calc} (s)	**Speedup (su T_{calc})**
mesh 2x2 con preemption,k=9	1	26.01	18.67	1.00
	2	21.23	13.97	1.33
	3	18.50	11.45	1.63
	4	19.27	11.86	1.57
	6	18.04	10.63	1.75
	8	17.23	10.01	1.86
multipoll, k=5	1	9.48	7.31	1.00
	2	7.42	5.17	1.41
	3	6.37	4.22	1.73
	4	5.91	3.63	2.01
	6	6.18	4.07	1.79
	8	6.98	4.75	1.53
multipoll, k=7	1	53.83	45.73	1.00
	2	43.58	35.55	1.28
	3	36.97	28.96	1.57
	4	33.54	25.68	1.78
	6	31.32	23.48	1.95
	8	34.11	25.96	1.76
mesh 2x2 senza preem. k=5	1	99.17	70.39	1.00
	2	90.10	59.61	1.18
	3	73.75	45.29	1.55
	4	69.39	41.54	1.69
	6	67.18	39.35	1.78
	8	75.68	46.08	1.52

Tabella 4.24: BiCGStab con MPI, tempi di esecuzione e speedup ottenuti con un uso esclusivo della macchina.

Anche in questo caso si vede che all'aumentare del numero dei processori impiegati lo speedup tende a raggiungere un massimo fra 4 e 6, per poi calare con 8 processori (tranne che con la rete *preemptive mesh 2x2 , k=9*).

4.5.2 Risultati con OpenMP ed mp

In questo paragrafo sono riportati i risultati ottenuti dai due programmi di tipo *shared memory* derivati dall'algoritmo BiCGStab in versione parallela.

In tabella 4.25 i dati, relativi alla rete multipoll con k=5, sono stati ottenuti mediante esecuzione in modalitá esclusiva del programma che fa uso della libreria proprietaria *mp* di SGI.

rete	**Numero processori**	**Tempo totale real (s)**	**Tempo totale user (s)**	**Tempo calc. user (s)**	**Speedup**
multipoll, k=5	1	9.87	8.97	6.88	1.00
	2	6.65	6.24	4.17	1.65
	3	6.39	5.75	3.68	1.86
	4	5.23	4.84	2.77	2.48
	6	4.19	4.37	2.29	3.00
	8	5.28	4.76	2.69	1.55

Tabella 4.25: BiCGStab con mp, tempi di esecuzione e speedup ottenuti con un uso esclusivo della macchina.

Nel grafico della figura 4.11 sono posti a confronto i valori di speedup ottenuti con il BiCGStab parallelo di tipo message passing basato su MPI e con il programma di tipo shared memory realizzato con *mp*. I valori sono quelli riportati nelle tabelle 4.24 e 4.25 e relativi alla matrice della rete multipoll con k=5. Si puó osservare come il programma basato su *mp* sia risultato piú efficiente, riuscendo a produrre uno speedup di 3 in corrispondenza dell'esecuzione con sei processori, mentre con MPI non si é riusciti a superare il valore 2. Anche con il programma shared memory si é peró osservata la diminuzione dello speedup passando da 6 ad 8 processori, evidenziando quindi un comportamento critico per quanto riguarda la scalabilitá. Sarebbe a tal proposito interessante verificare con altre misure (eventualmente su altre macchine) se questo calo é dovuto all'uso del numero totale di processori disponibili, ad errori sperimentali nella misura oppure ad un'intrinseca caratteristica del programma che non riesce a scalare adeguatamente oltre i 6 processori.

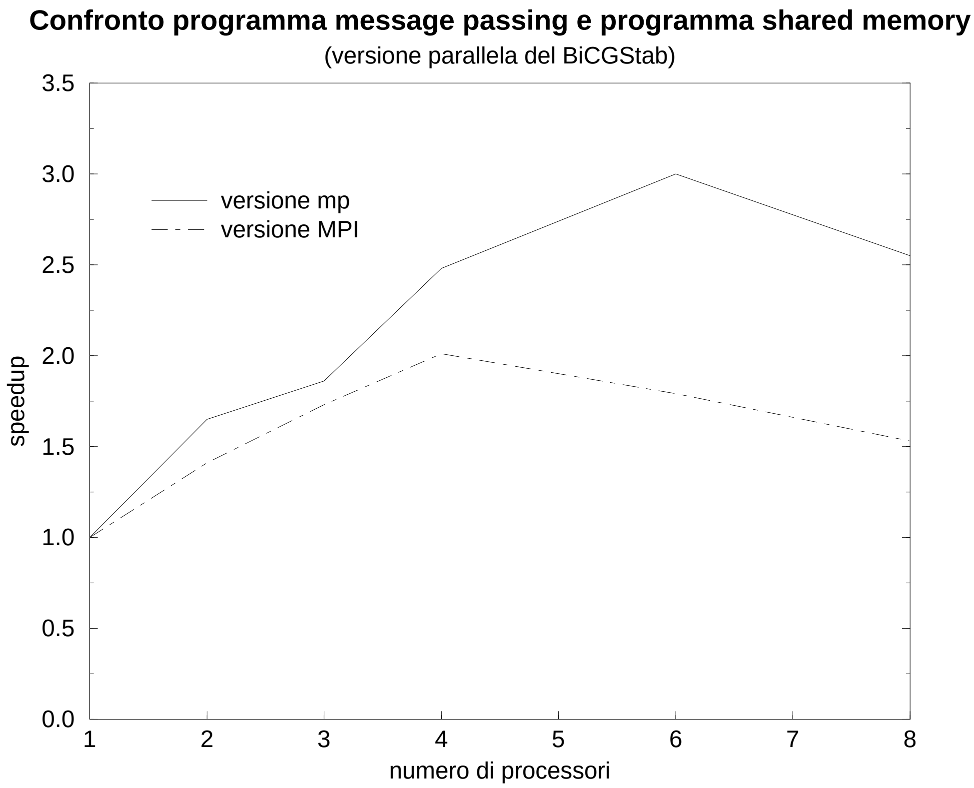


Figura 4.11: Confronto degli speedup ottenuti con MPI e con mp per l'algoritmo BiCGstab

In tabella 4.26 sono riportati i risultati ottenuti sempre con *mp* ma operando in normali condizioni multiutente. Nella figura 4.12 tali risultati sono posti a confronto con quelli ottenuti usando l'elaboratore in maniera esclusiva. Si puó osservare che fino a 4 processori i tempi di calcolo si sovrappongono con buona approssimazione, mentre nel caso di esecuzione con 6 ed 8 in modalitá condivisa peggiorano notevolmente. La spiegazione é che, come giá accennato nel capitolo 2.1, con mp ed OpenMP l'utente puó scegliere il numero dei processi ma il sistema operativo decide quanti processori allocare ad un unica applicazione, e difficilmente tutti i processori risultano liberi e vengono allocati alla stessa applicazione.

rete	Numero processori	Tempo totale real (s)	Tempo totale user (s)	Tempo calc. user (s)	Speedup
multipoll, k=5	1	9.39	9.10	7.00	1.00
	2	6.94	6.22	4.12	1.69
	3	5.87	5.60	3.49	2.00
	4	5.28	4.88	2.78	2.51
	6	10.70	7.51	5.41	1.29
	8	13.97	7.93	5.82	1.20

Tabella 4.26: BiCGStab con mp, tempi di esecuzione e speedup ottenuti in ambiente multiutente.

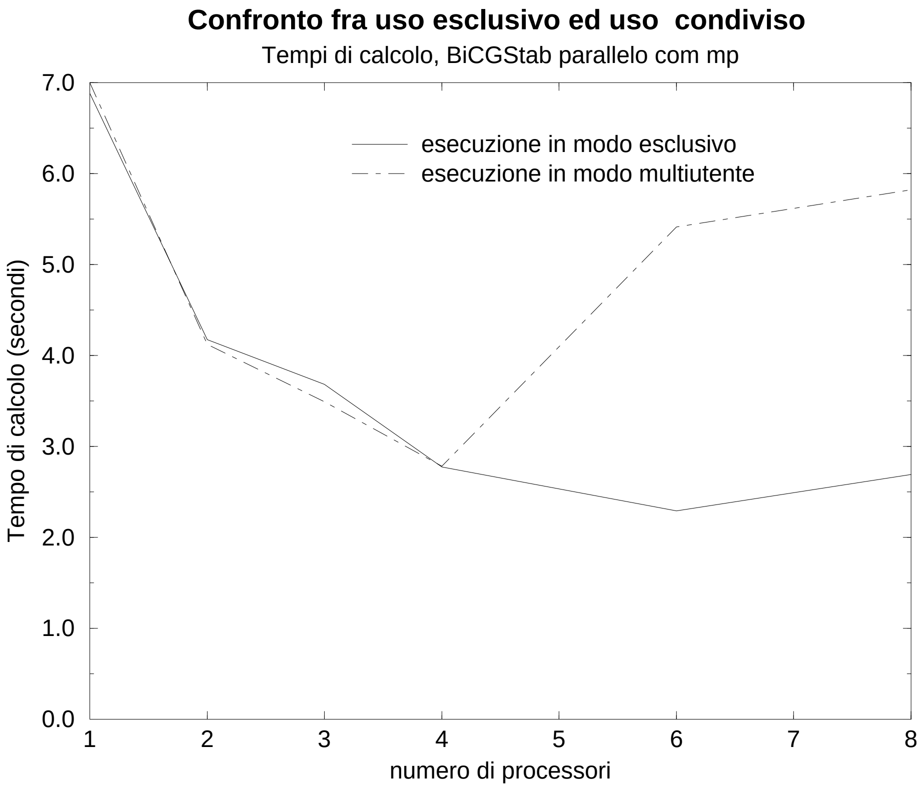


Figura 4.12: Confronto dei tempi di esecuzione ottenuti in modalita monoutente e multiutente con mp

Nell'ultima tabella del paragrafo, la 4.27, sono riportati i risultati ottenuti con il programma basato su OpenMP. Le misure sono state tutte eseguite in condizioni di uso condiviso della macchina e rispecchiano abbastanza fedelmente i risultati ottenuti con *mp*, come era logico attendersi dato che le due API sono molto simili.

rete	**Numero processori**	**Tempo totale real (s)**	**Tempo totale user (s)**	**Tempo calc. user (s)**	**Speedup**
multipoll, k=5	1	9.21	9.02	6.93	1.00
	2	7.15	6.71	4.62	1.50
	3	5.84	5.52	3.43	2.02
	4	6.06	5.38	3.26	2.12
	6	8.52	6.09	4.00	1.73
	8	12.62	7.27	5.18	1.33
multipoll, k=7	1	51.20	50.57	42.81	1.00
	2	37.65	37.01	29.24	1.46
	3	32.88	30.37	22.61	1.89
	4	30.55	28.46	20.70	2.07
	6	47.26	33.89	26.13	1.63
	8	63.43	35.04	27.20	1.57

Tabella 4.27: BiCGStab con OpenMP, tempi di esecuzione e speedup ottenuti in ambiente multiutente.

4.6 Algoritmo TFQMR

In questo paragrafo vengono riportati i risultati sperimentali ottenuti impiegando l'algoritmo TFQMR per la risoluzione di sistemi lineari la cui matrice dei coefficenti é un generatore infinitesimale di una catena di Markov associato ad una rete di Petri. La sintetica descrizione teorica dell'algorimo TFQMR é riportata nell'appendice A.

Innanzitutto é opportuno ricordare che i sistemi lineari trattati sono omogenei, cioé del tipo $Ax = 0$ e che la matrice A^T é il generatore infinitesimale di una CTMC ed é una matrice singolare. Per questo motivo, l'applicazione immediata al problema in esame di algoritmi solutori, generalmente definiti per matrici A non singolari, puó non essere possibile. Mentre infatti un algoritmo come il GMRES non precondizionato, grazie alla particolaritá delle matrici relative a CTMC isomorfe ad una rete di Petri, se converge allora converge alla soluzione corretta (avente cioé la somma delle componenti di x uguale ad 1), altri algoritmi come il BiCGStab ed anche il TFQMR non godono di questa caratteristica. É allora indispensabile integrare l'algoritmo riportato in precedenza con la condizione che la somma delle componenti del vettore soluzione sia unitaria. Sono possibili due approcci:

- Normalizzare il vettore soluzione ad ogni iterazione, aggiungendo in pratica un'equazione alla matrice A
- Sostituire un'equazione della matrice A con l'equazione (vettoriale) $1^T \cdot x = 1$

É da notare che per l'attuazione della seconda strategia é anche necessario scegliere quale riga sostituire : una riga scelta in modo arbitrario non necessariamente produce una matrice modificata non singolare e per essere sicuri di eliminare la singolaritá sarebbe necessario determinare una riga che risulta essere combinazione lineare delle altre. Un altro svantaggio é rappresentato dal fatto che la nuova riga, dovendo rappresentare l'equazione $1^T \cdot x = 1$, é una riga di tutti 1, che riduce in una qualche misura la sparsitá della matrice: per le matrici esaminate tale aumento nel numero di elementi non nulli varia dal 10% ad oltre il 20%, e pertanto non puó esser considerato trascurabile.

Anche la strategia alternativa d'altro canto non é esente da inconvenienti: l'aggiunta della normalizzazione della soluzione ad ogni iterazione costituisce una alterazione sostanziale dell'algoritmo e potrebbe influire negativamente sul processo di convergenza. Uno degli aspetti che si é voluto analizzare con le prove effettuate é proprio quale di queste due strategie sia migliore dal punto di vista della convergenza del metodo e del numero di iterazioni richieste.

Un altro argomento di indagine é costituito dall'espressione utilizzata per il test di arresto del processo iterativo dell'algoritmo TFQMR: il membro di destra dell'equazione (A.24) puó essere utilizzato al posto della norma del residuo $r_m = \|b - Ax_0\|$ che é molto piú dispendiosa da calcolare. Come si é giá sottolineato, la (A.24) rappresenta peró una maggiorazione della norma del residuo e non una sua approssimazione e puó pertanto risultare troppo restrittiva come condizione per la verifica della convergenza. Nel programma realizzato é prevista la possibilitá di scegliere quale di queste due espressioni utilizzare per effettuare il test di arresto.

4.6.1 Rete mesh 2×2 con preemption

Per la prima serie di prove a cui è stato sottoposto l'algoritmo TFQMR si sono utilizzate matrici ottenute dal modello mesh 2x2 con preemption descritto all'inizio del capitolo. Inizialmente si é tenuto fisso il residuo di uscita ad un valore di 10^{-7}e si é fatto variare il parametro k della rete, misurando il numero di iterazioni ed il tempo di calcolo richiesto dall'algoritmo TFQMR per determinare la soluzione. La scelta del residuo di uscita richiesto è stata fatta in modo arbitrario. Le misure sono state effettuate sia normalizzando la soluzione ad ogni iterazione (tabella 4.28), sia sostituendo un'equazione (tabella 4.29); in entrambi i casi le misure sono state ripetute utilizzando l'espressione completa del residuo o la forma semplificata descritta dalla maggiorazione dell'equazione (A.24). Dai dati riportati nella tabella 4.28 si

	$\|r_n\|$ semplificato		$\|r_n\|$ completo	
k	N^o iterazioni	Tempo(secondi)	N^o iterazioni	Tempo(secondi)
1	6	0.01	6	0.01
2	14	0.04	13	0.04
3	18	0.16	16	0.17
4	22	0.52	17	0.53
5	23	1.27	23	1.46
6	25	3.06	25	3.53
7	*	*	30	8.29
8	*	*	*	*
9	38	26.9	36	29.5
10	43	39.7	41	49.1
11	*	*	*	*

Tabella 4.28: Rete mesh 2x2 con preempition: esecuzione TFQMR con $\epsilon = 10^{-7}$ e normalizzazione ad ogni iterazione; nei casi (*) non é stata raggiunta la convergenza.

possono trarre alcune osservazioni. Innanzitutto si verifica che l'uso dell'espressione semplificata per il test di uscita comporta un significativo risparmio di tempo, fino al 21% nel caso di k=10, nonostante richieda in tutti i casi un numero maggiore di iterazioni; inoltre, nel caso k=7, l'uso dell'espressione semplificata per il test di arresto provoca una mancata convergenza del metodo, cosa che non si verifica utilizzando l'espressione non approssimata. In due casi, per k=8 e per k=11, l'algoritmo ha fallito, non riuscendo a convergere né impiegando l'espressione semplificata, né quella completa.

Le misure appena descritte sono state ripetute nel caso in cui la singolaritá della matrice sia stata risolta sostituendo una riga, precisamente la prima riga della matrice; i risultati sono riportati nella tabella 4.29. Effettuando la sostituzione di una riga l'algoritmo riesce a convergere per tutti i valori di k, e sia utilizzando l'espressione semplificata del residuo, sia quella completa, mostrando perció un comportamento molto piú regolare rispetto alla risoluzione con normalizzazione ad ogni iterazione.

	$\|r_n\|$ semplificato		$\|r_n\|$ completo	
k	N^o iterazioni	Tempo(secondi)	N^o iterazioni	Tempo(secondi)
1	8	0.01	8	0.01
2	17	0.04	17	0.05
3	22	0.18	19	0.19
4	24	0.55	24	0.64
5	27	1.40	24	1.53
6	34	3.65	34	4.40
7	35	7.91	34	10.7
8	41	15.7	39	19.4
9	54	32.7	46	36.7
10	55	49.5	53	62.3
11	60	84.5	58	99.3

Tabella 4.29: Rete mesh 2x2 con preempition: TFQMR con $\epsilon = 10^{-7}$ e sostituzione della riga 1.

Si puó notare come stavolta l'uso dell'espressione semplificata del residuo comporta

riga sostituita	N^o iterazioni
1	41
5	45
10000	44
20000	44
30000	45
40000	48
50000	44
60000	45
68567	45

Tabella 4.30: Rete mesh 2x2 con preempition e k=8: effetto della scelta della riga sostituita sul numero di iterazioni nel caso di esecuzione con $\epsilon = 10^{-7}$, sostituzione di una riga e uso della forma approssimata del residuo.

in tutti i casi un miglioramento dei tempi di esecuzione e risulta perció vantaggiosa. La maggior affidabilitá é stata peró ottenuta a scapito dei tempi di esecuzione che risultano superiori rispetto al caso di normalizzazione ad ogni iterazione, per tutti i valori di k per i quali si era ottenuta convergenza: ad esempio per k=10 sono necessarie 41 iterazioni per ottenere la precisione richiesta utilizzando l'espressione completa del residuo e la normalizzazione ad ogni iterazione, mentre ne servono ben 53 (corrispondenti ad un incremento del 29%) se si usa la sostituzione di una riga e sempre utilizzando l'espressione completa del residuo.

La sostituzione di una riga presenta inoltre un grado di libertá: la scelta di sostituire la prima riga operata nelle misure precedenti é infatti del tutto arbitraria e quasi sicuramente non quella ottimale.

Questo aspetto ha suggerito l'effettuazione di ulteriori misure per verificare l'impatto della scelta della riga da sostituire sulla convergenza del processo iterativo. Nella tabella 4.30 é riportato il numero di iterazioni necessario per ottenere la convergenza al variare della riga sostituita; i dati sono relativi al caso k=8, scelto perché era uno dei due casi in cui non si era raggiunta la convergenza con la normalizzazione ad ogni iterazione, e quindi probabilmente piú critico dal punto di vista della scelta della condizione di normalizzazione. Per l'ultima serie di misure relativa a matrici ottenute dal modello di rete "mesh 2x2 con preemption" si é valutato il comportamento dell'algorimo al variare della precisione richiesta. I dati della tabella

residuo d'uscita	N^o iterazioni	Tempo(secondi)
10^{-5}	35	17.7
10^{-6}	38	18.9
10^{-7}	39	19.4
10^{-8}	41	20.1
10^{-9}	41	20.1
10^{-10}	42	21.6
10^{-11}	*	*

Tabella 4.31: Rete mesh 2x2 con preempition e k=8: numero di iterazioni necessarie al TFQMR al variare del residuo di uscita richiesto. Esecuzione con sostituzione della riga 1 ed uso dell' espressione completa del residuo.

4.31 mostrano che l'algoritmo é in grado di raggiungere un residuo inferire a 10^{-10}, valore discreto considerando che il caso k=8 é relativo ad una matrice di dimensione 68657 .

4.6.2 Rete mesh 2×2 senza preemption

Anche per la rete mesh 2×2 senza preemption é risultato che la sostituzione di una riga ha consentito la convergenza del metodo per tutti i valori di k considerati, al contrario della soluzione con normalizzazione ad ogni passo con la quale spesso non si é avuta convergenza. Per questo motivo sono riportati solo i risultati relativi al caso di sostituzione di una riga che é risultata la scelta piú affidabile per questo tipo di rete. Come nel paragrafo precedente, si é inanzitutto fissato il risiduo di uscita al valore di 10^{-7} e si sono misurati tempi di esecuzione e numero di iterazioni effettuate.

	$\|r_n\|$ semplificato		$\|r_n\|$ completo	
k	N^o iterazioni	Tempo(secondi)	N^o iterazioni	Tempo(secondi)
1	22	0.03	22	0.03
2	47	0.87	47	1.08
3	59	11.1	56	13.4
4	69	54.3	64	69.1
5	89	204	72	326

Tabella 4.32: Rete mesh 2x2 senza preempition: esecuzione TFQMR con $\epsilon = 10^{-7}$, sostituzione della riga 1 ed uso della forma completa del residuo

Anche per questo tipo di reti il comportamento mostrato dall'algoritmo é piuttosto regolare: i tempi di esecuzione ed il numero di iterazioni, riportati nella tabella 4.32, crescono regolarmente al crescere del parametro k, e quindi al crescere del numero degli stati della rete. Viene inoltre confermato che l'uso dell'espressione semplificata del residuo produce una significativa riduzione dei tempi di calcolo e questo nonostante comporti sempre un numero superiore di iterazioni: come si era osservato nel commento alla tabella A.1, anche in questo caso si verifica che il calcolo dell'espressione $r_m = \|b - Ax_0\|$ non é trascurabile in termini di tempo di esecuzione. Anche per questo tipo di rete é stato studiato il comportamento dell'algoritmo al

residuo d'uscita	N^o iterazioni	Tempo(secondi)
10^{-5}	49	55.0
10^{-6}	63	66.5
10^{-7}	64	90.7
10^{-8}	68	71.1
10^{-9}	74	77.9
10^{-10}	77	80.0
10^{-11}	84	95.2
10^{-12}	*	*

Tabella 4.33: Rete mesh 2x2 senza preempition con k=4: numero di iterazioni necessarie al TFQMR al variare del residuo di uscita richiesto. Esecuzione con sostituzione della riga 1 ed uso dell' espressione completa del residuo.

variare del residuo di uscita: nella tabella 4.33 sono riportati tempi e numero di iterazioni richiesta al variare di ϵ fra 10^{-5} e 10^{-12}. Si puó notare che il numero di iterazioni impiegate cresca in modo piuttosto lento passando da un residuo di 10^{-5} ad uno di 10^{-11}, mentre non é stato possibile raggiungere valori inferiori a 10^{-12}, a conferma di quanto visto nel paragrafo precedente: nel problema in esame l'algoritmo non si presta a raggiungere valori di $\|r_n\|$ notevolmente piccoli.

4.6.3 Rete multipoll

Nella tabella 4.34 sono riportati i dati relativi a tempo di esecuzione e numero di iterazioni al variare di k per la rete multipoll. Come per la rete mesh 2×2 senza preeemption sono stati riportati solo i risultati relativi alla sostituzione di una riga perché tale tipo di scelta si é rivelata piú affidabile per quanto concerne la convergenza. Dall'analisi dei tempi si puó vedere come tale tipo di rete sia risultata particolarmente ostica per l'algoritmo impiegato. Oltre ai tempi piuttosto elevati,

	$\|r_n\|$ semplificato		$\|r_n\|$ completo	
k	N^o iterazioni	Tempo(secondi)	N^o iterazioni	Tempo(secondi)
1	33	0.02	32	0.02
2	94	0.25	80	0.29
3	151	2.2	123	2.03
4	171	6.37	159	8.09
5	277	42.1	260	41.4
6	353	95.6	271	102
7	322	235	306	220

Tabella 4.34: Rete multipoll: esecuzione TFQMR con $\epsilon = 10^{-7}$, sostituzione della riga 10 ed uso della forma completa del residuo. Nel caso k=7 si é sostituita la riga 3456.

se raffrontati alle dimensioni delle matrici, nel caso k=7 per ottenere la convergenza é stato necessario cambiare la riga da sostituire rispetto a quella usata per gli altri valori di k (si é sostituita la riga 3456 anziché la 10).

Un'altra osservazione possibile é relativa alle variazioni irregolari del numero di iterazione richieste per ottenere la soluzione: considerando la terza colonna della tabella 4.34, si vede che passando da k = 4 a k = 5 il numero di iterazioni aumenta da 159 a 260 (+63%), mentre passando da k=5 a k=6 il numero di iterazioni passa da 260 a 271 (+4%) nonostante in entrambi i casi le dimensioni della matrice approssimativamente raddoppino. Con le reti precedenti, il numero di iterazioni, oltre ad essere inferiore a paritá di numero di stati della rete, variava in modo molto piú regolare (pochi punti percentuali passando da un valore di k al successivo).

4.6.4 Rete benchprod

Come ultima rete presa in esame si é utilizzato il tipo benchprod. Le modalitá delle misure sono le stesse usate in precedenza, ed anche i risultati, riportati in tabella 4.35 confermano alcune osservazioni giá fatte. La sostituzione di una riga é risultata piú affidabile rispetto alla normalizzazione ad ogni iterazione, mentre l'uso dell'espressione approssimata del residuo é risultata vantaggiosa per tutti i

	$\|r_n\|$ semplificato		$\|r_n\|$ completo	
k	N^o iterazioni	Tempo(secondi)	N^o iterazioni	Tempo(secondi)
1	75	0.03	72	0.03
2	149	0.67	138	0.83
3	315	7.6	269	8.91
4	512	52.1	387	55.4
5	357	149.5	338	141.9
6	447	255	410	331
7	627	645	478	766

Tabella 4.35: Rete benchprod: esecuzione TFQMR con $\epsilon = 10^{-7}$, sostituzione della riga 3456 ed uso della forma completa del residuo

valori di k, tranne k=5. Proprio in corrispondenza al crescere di k da k=4 a k=5 si verifica una inaspettata diminuzione del numero di iterazioni richieste: fra tutte le reti considerati questo é l'unico caso in cui aumentando il numero di stati si verifica una diminuzione del numero di iterazioni.

4.6.5 Confronto con altri metodi

In questo paragrafo viene effettuato un sintetico confonto tra i risultati ottenuti con l'algoritmo TFQMR e quelli ottenuti mediante l'impiego degli algoritmi BiCGStab e GMRES(m) non precondizionato. Per il TFQMR si sono utilizzati i dati esposti nel paragrafo precedente e relativi alla sostituzione di una riga, con calcolo dell'espressione approssimata del residuo. Per il GMRES(m) si sono utilizzati i dati relativi alla versione non precondizionata dell'algoritmo e all'uso del valore ottimo di m. Per il BiCGStab si sono considerati di volta in volta la sostituzione di una riga, o la normalizzazione ad ogni iterazione, a seconda di quale risultasse migliore per il particolare tipo di rete. Sostanzialmente si é cercato di considerare la condizione migliore per ogni algoritmo. Tale confronto ha principalmente il fine di valutare le potenzialitá dei vari metodi applicati alla risoluzione di catene di Markov isomorfe a reti di Petri. In figura 4.13 sono riportati i tempi di esecuzione ottenuti dai 3 algo-

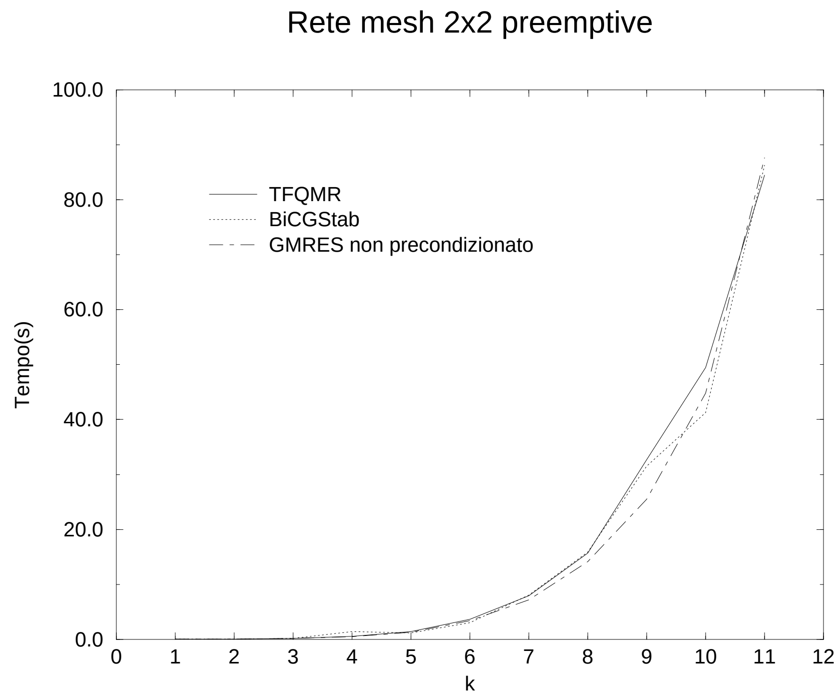


Figura 4.13: Tempi di esecuzione al variare di k

ritmi presi in esame ed applicati alla rete di tipo *preeemptive mesh* $2x2$: si osserva immediatamente che le 3 curve tendono a sovrapporsi, in particolare per i valori k<9 dove le differenze fra GMRES e TFQMR sono inferiori agli errori di misura sui tempi; anche la curva relativa al BiCGStab risulta molto vicina alle prime due. Per k=10 si ha invece una differenziazione piú marcata delle tre curve, differenziazione che si torna a ridurre per k=11. Complessivamente si puó concludere che per questo tipo di rete l'algoritmo TFQMR ha un'efficienza equivalente agli altri due; come si vedrá dai risultati successivi questo risulta esser il caso migliore per quest'algoritmo.

La sostanziale uniformitá delle prestazioni per la rete mesh 2x2 preemptive non si ripresenta per gli altri tre tipi di rete considerati. Nella figura 4.14, relativa alla rete mesh 2x2 preemptive, si vede come il TFQMR produca tempi di esecuzione analoghi o leggermente migliori di quelli realizzati con il GMRES non precondizionato, ma notevolmente superiori a quelli ottenuti con il BiCGStab.

Per i restanti due tipi di reti i tempi di esecuzione del GMRES si riferiscono alla versione con precondizionatore diagonale. In figura 4.15 si vede come, con la rete

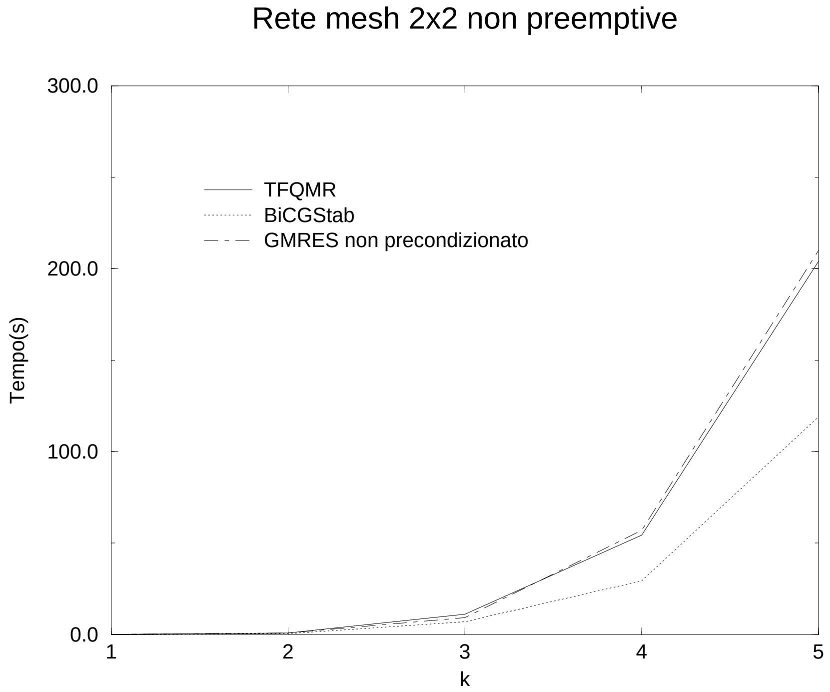


Figura 4.14: Tempi di esecuzione al variare di k

multipoll, il TFQMR risulti sensibilmente piú lento degli altri due per tutti i valori di k. Sono confermate quindi le considerazioni fatte in §4.6.3 dove si era notato il numero elevato di iterazioni e il loro aumento a volte molto significativo passando da un valore di k al successivo, unitamente alla difficoltá di convergenza per k=7. Infine in figura 4.16, relativa al modello benchprod, si puó osservare una situazione simile a quella verificata per le reti mesh 2x2 con e senza preemption, in quanto il BiCGStab risulta sempre piú veloce, tranne nel caso k=7 dove non converge, mentre GMRES e TFQMR realizzano tempi abbastanza simili: il GMRES risulta piú veloce del TFQMR per k inferire a 5 e piú lento nei casi k =6 e k=7. Si puó notare infine che, dal punto di vista qualitativo, i risultati ottenuti sono coerenti con quanto riportato in [9], pur se in un ambito diverso da quello delle reti di Petri.

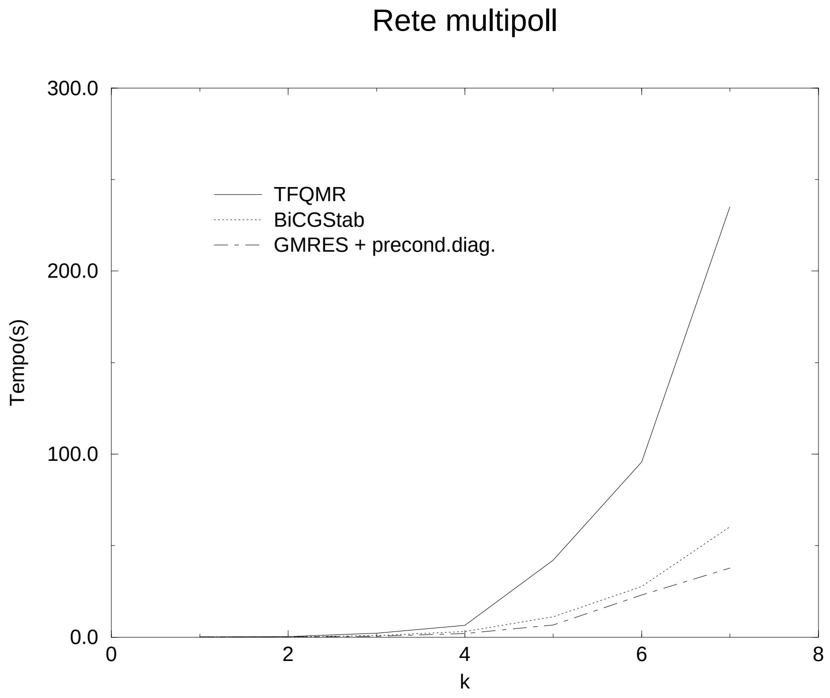


Figura 4.15: Tempi di esecuzione al variare di k

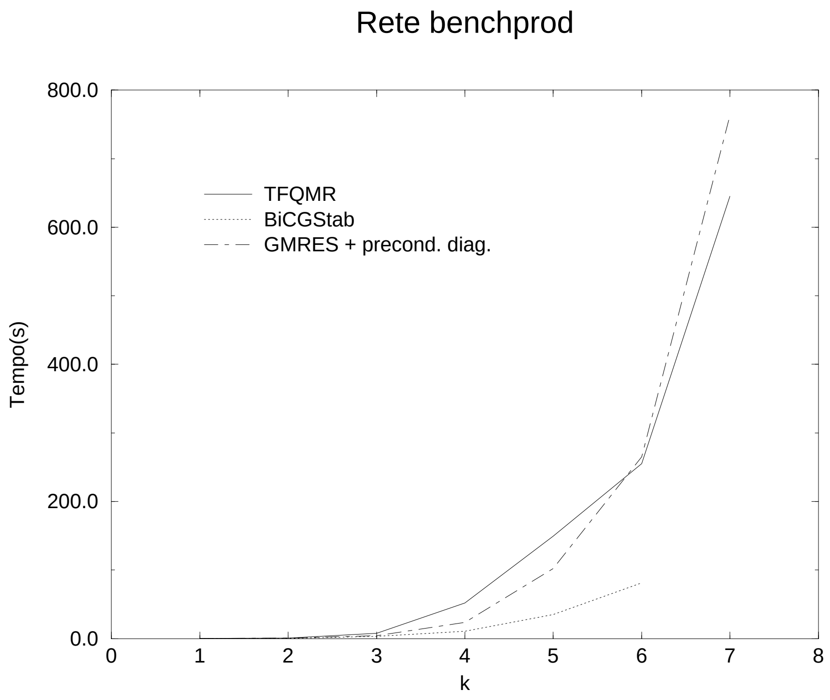


Figura 4.16: Tempi di esecuzione al variare di k

4.7 Osservazioni sui vettori soluzione

Durante lo studio dei vari modelli utilizzati e delle matrici da essi generate è stato possibile compiere alcune interessanti analisi sulle caratteristiche dei vettori soluzione calcolati. Si ricorda innanzitutto che il vettore soluzione x è il vettore distribuzione di probabilità sulle marcature della GSPN: ogni componente del vettore rappresenta la probabilità che la rete si trovi nella marcatura corrispondente; la somma delle componenti di x è perciò unitaria.

Nella tabella 4.36 sono riportati alcuni parametri statistici dei vettori soluzione relativi alla rete mesh 2x2 con preemption. I risultati sono stati ottenuti con l'algoritmo GMRES precondizionato richiedendo un ϵ di uscita di 10^{-12}.

k	numero	componenti di valore $> 10^{-5}$		
	di stati	%	somma	numero
1	57	100.0	1.000	57
2	461	34.06	0.999	157
3	1837	9.42	0.999	173
4	5145	3.36	0.999	173
5	11681	1.48	0.999	173
6	23077	0.75	0.999	173
7	41301	0.42	0.999	173
8	68657	0.25	0.999	172
9	107785	0.16	0.999	172
10	161661	0.11	0.999	179
11	233597	0.07	0.999	164

Tabella 4.36: Alcune statistiche relative ai vettori soluzione ottenuti dalla rete mesh 2x2 con preemption.

Si può subito notare come il numero di componenti che assume un valore superiore a 10^{-5} si mantenga pressochè costante al valore 172 al variare delle dimensioni delle matrici e la somma delle probabilità di tali stati raggiunge lo 0.999.

Questo risultato suggerisce che, se fosse possibile caratterizzare questi stati "dominanti", si potrebbe tentare di risolvere problemi semplificati potendo poi generalizzare i risultati anche ai problemi di dimensioni maggiori. Per la determinazione dei problemi semplificati sono ipotizzabili due approcci distinti:

- l'intervento sulla rete, scoprendo una relazione fra gli stati dominanti e la struttura della rete ed arrivando eventualmente a ridefinire il modello.
- l'intervento sulla matrice di sistema, riducendo il numero di equazioni da risolvere.

Le considerazioni appena esposte sono da considerarsi sostanzialmente come uno spunto per sviluppi futuri del lavoro svolto in questa sede.

Conclusioni

L'attivitá svolta in questo lavoro di tesi é consistita nello studio di due distinti aspetti della risoluzione di sistemi lineari generati da modelli di reti di Petri: il precondizionamento e la parallelizzazione di algoritmi solutori.

Sono state studiate due tecniche di precondizionamento basate sulla fattorizzazione incompleta, la ILU(0) e la ILU(K). Entrambe sono state applicate all'algoritmo proiettivo GMRES(m) che, fra gli algoritmi solutori studiati in precedenti tesi, ha dimostrato ottime caratteristiche di affidabilitá, riuscendo in tutti i casi esaminati a raggiungere valori di precisione sul residuo di 10^{-10} e spesso anche superiori.

Il GMRES(m) con precondizionatore ILU0 realizzato si é dimostrato molto efficiente consentendo di ottenere riduzioni dei tempi di esecuzione fino a 30 volte rispetto alla versione non precondizionata. I tempi di esecuzione sono risultati i piú bassi anche se confontati a quelli ottenuti con gli algoritmi BiCGStab e Gauss-Seidel (sviluppati in [11, 12]). Per una migliore comprensione degli effetti del precondizionatore ILU0 é stata compiuta un' analisi degli autovalori della matrice di iterazione: i risultati ottenuti evidenziano una consistente diminuzione del modulo degli autovalori (eccettuato naturalmente quello unitario) motivando cosí i positivi effetti del precondizionatore.

L'uso del precondizionatore ILUK è invece risultato penalizzato dal costo computazionale dovuto alla costruzione della matrice contenente la fattorizzazione incompleta. Per le matrici trattate, l'uso del GMRES(m) con tale precondizionatore é risultato decisamente meno conveniente del caso ILU0.

Il GMRES(m) con ILU0 é stato il primo algoritmo parallelizzato. Riguardo a questo aspetto i risultati ottenuti sono peró relativamente modesti, perché se da un lato il precondizionatore ha consentito un netto miglioramento dei tempi di esecuzione e del numero di iterazioni, dall'altro ha ridotto il livello di concorrenza sfruttabile nello sviluppo del programma parallelo. Le porzioni di codice parallelizzate rappresentano meno del 50% del tempo di calcolo ed infatti i risultati sperimentali, sia del programma *messagge passing* (basato su MPI) sia di quello di tipo *shared memory* (basato su OpenMP), hanno sempre dato uno speedup inferiore a due.

Anche in considerazione di questi risultati si é scelto come secondo algoritmo da parallelizzare il BiCGStab non precondizionato. Il livello di parallelismo sfruttabile é significativamente superiore e ció ha permesso di ottenere risultati piú interessanti anche nel confronto fra i programmi realizzati con MPI e quelli realizzati con OpenMP ed mp. Riguardo a quest'ultimo aspetto si puó concludere che l'uso di un paradigma di programmazione a memoria condivisa, consentito da OpenMP ed mp, si adatta in modo migliore rispetto ad MPI alle caratteristiche dell'architettura ccNUMA utilizzata. Nel primo caso, infatti, i valori di speedup massimi misurati sono stati di 3 mentre per MPI si é raggiunto il valore 2.

Il confronto fra risultati ottenuti con un uso esclusivo o meno della macchina ha infine evidenziato come l'ambiente multiutente possa influenzare i risultati delle misure provocando in alcuni casi significativi peggioramenti dei tempi di esecuzione: questo aspetto é sicuramente accentuato dal modello data-parallel necessario per la parallelizzazione degli algoritmi solutori, che é caratterizzato da frequenti sincronizzazioni.

A chiusura del capitolo sono proposti alcuni sviluppi futuri di questo lavoro di tesi:

- la sperimentazione dei programmi paralleli sviluppati su altre architetture,
- lo studio dei precondizionatori realizzati applicati nella risoluzione di matrici ottenute da modelli diversi da quelli proposti,
- lo studio (accennato nel paragrafo 4.7) di tecniche che consentano di generalizzare risultati ottenuti con modelli di piccole dimensioni.

Appendice A

L'algoritmo TFQMR

A.1 Introduzione

Il metodo del gradiente coniugato (CG) é uno dei piú efficaci metodi iterativi per risolvere un sistema di equazioni $Ax = b$ nel caso in cui la matrice A sia simmetrica e definita positiva. Sono state sviluppate numerose varianti del metodo CG in grado di risolvere sistemi lineari per i quali la matrice dei coefficenti A sia una generica matrice N x N, non simmetrica e non definita positiva. Fra di esse ricordiamo il gradiente biconiugato (Bi-CG) ed il gradiente quadro coniugato (CG-S), i quali peró, a differenza del metodo CG, non prevedono nei passi di iterazione una proprietá di minimizzazione della norma del residuo. Questa caratteristica puó spiegare il comportamento irregolare, caratterizzato da ampie oscillazioni della norma del residuo, mostrato in numerose situazioni da tali metodi.

Per migliorare le caratteristiche di convergenza dell'algoritmo Bi-CG, é stato proposto il metodo QMR ($quasi - minimal\ residual\ method$) simile al Bi-CG ma con passi di iterazione che prevedono una quasi-minimizzazione della norma del residuo al fine di ottenere un comportamento piú regolare per quanto concerne la convergenza del procedimento iterativo. Come il Bi-CG, anche il metodo QMR richiede ad ogni iterazione il calcolo di prodotti matrice per vettore che coinvolgono sia la matrice dei coefficenti A, sia la sua trasposta A^T. Quest'aspetto rappresenta uno svantaggio non trascurabile in numerose applicazioni ed anche nei problemi trattati in questa tesi, nei quali la matrice dei coefficenti A é una matrice sparsa di grandi dimensioni, memorizzata in una formato compatto le cui caratteristiche rendono la matrice trasposta A^T non immediatamente disponibile. Un'ulteriore evoluzione dell'algoritmo Bi-CG é rappresentata dal metodo del gradiente quadro coniugato (CG-S), che non richiede il calcolo di prodotti che coinvolgano A^T. L'algoritmo TFQMR ($transpose - free\ quasi - minimal\ residual$) proposto in [7] utilizza un approccio basato sulla quasi-minimizzazione della norma del residuo simile a quello usato nel metodo QMR, ma differentemente da esso non trae le sue origini dall'algoritmo Bi-CG ma dall'algoritmo CG-S: il TFQMR non richiede moltiplicazioni per la matrice A^T, ed almeno in teoria, dovrebbe ereditare sia le buone caratteristiche di velocitá di convergenza dell'algoritmo CG-S, sia la regolaritá ed affidabilitá mostrate dall'algoritmo QMR.

A.1.1 L'algoritmo CGS

Il processo iterativo dell'algoritmo Bi-CG utilizza due vettori residuo r_j ed $\hat{r}_j$, definiti dalle espressioni:

$$r_j = \phi_j(A)r_0 \quad \text{e} \quad \hat{r}_j = \phi_j(A^T)\hat{r}_0 \text{ , con } \phi_j \text{ polinomio di grado} j$$

e soddisfacenti la condizione di biortogonalitá $(\hat{r}_j, r_i)= 0$ se $i \neq j$, e due vettori di "ricerca della direzione", v_j ed $\hat{v}_j$, che soddisfano la condizione di biconiugazione $\hat{v_j}^T A v_i = 0$ se $i \neq j$. Le approssimazioni successive della soluzione sono determinate mediante una relazione ricorsiva a tre termini. Sfruttando la condizione di biortogonalitá appena descritta si puó modificare il passo di iterazione in modo che la matrice A^T non sia piú coinvolta nel calcolo del residuo; é possibile infatti affermare che:

$$\begin{aligned}(r_j, \hat{r}_i) &= (\phi_j(A)r_0, \phi_i(A^T)\hat{r}_0) = \\ &= (\phi_i(A)\phi_j(A)r_0, \hat{r}_0) = 0 \qquad \text{per } i < j\end{aligned}$$

Al posto dei vettori $\hat{r}_j$ possono essere calcolati i vettori $\tilde{r}_j = \phi_j^2(A)r_0$ senza dover effettuare moltiplicazioni per la matrice A^T. Il risultato di questa modifica é l'algoritmo CG-S, di cui viene riportata la descrizione dettagliata:

1. Inizio:

   - si sceglie x_0 vettore iniziale
   - si calcola $p_0 = u_0 = r_0 = b - Ax_0$ e $v_0 = Ap_0$;
   - si sceglie un vettore $\tilde{r}_0$ tale che $\rho_0 = (r_0, \tilde{r}_0) \neq 0$, ad esempio: $\tilde{r}_0 = r_0$

2. Ciclo: for $i = 1, 2, 3, \ldots$ finchè x_i non è abbastanza accurata

   - $\sigma_{n-1} = (\tilde{r_0}, v_{n-1})$
   - $\alpha_{n-1} = \rho_{n-1}/\sigma_{n-1}$
   - $q_n = u_{n-1} - (\alpha_{n-1}, v_{n-1})$
   - $x_n = x_{n-1} + \alpha_{n-1}(u_{n-1} - q_n)$
   - $r_n = r_{n-1} - \alpha_{n-1}A(u_{n-1} + q_n)$
   - se x_n é abbastanza accurata allora fine
   - $\rho_n = (\tilde{r}_0, r_n)$
   - $\beta_n = \rho_n/\rho_{n-1}$
   - $u_n = r_n + \beta_n q_n$
   - $p_n = u_n + \beta_n(q_n + \beta p_{n-1})$
   - $v_n = Ap_n$

In aritmetica esatta l'algoritmo CG-S termina dopo al piú n iterazioni, dove n é la dimensione di A. Purtroppo in generale non é possibile escludere che l'algoritmo termini all'iterazione n_* (con $n_* < n$) prima di aver trovato la soluzione del sistema: quest'arresto prematuro del processo iterativo si puó avere nel caso in cui $\sigma_{n*} = 0$ o $\rho_{n*} = 0$; se si verifica una di queste due condizioni la continuazione del processo iterativo porterebbe ad una divisione per zero con conseguente *breakdown* dell'algoritmo.

A.1.2 Il metodo QMR

Come si é accennato in precedenza l'algoritmo CG-S non utilizza nessuna proprietá di minimizzazione del residuo. É peró possibile modificare l'iterazione di questo algoritmo in modo tale da soddisfare una proprietá di quasi-minimo. La descrizione di tale proprietá richiede alcune definizioni preliminari:

$$y_m = \begin{cases} u_{n-1} & \text{se } m = 2n-1 \ , \\ q_n & \text{se } m = 2n \end{cases} \tag{A.1}$$

$$w_m = \begin{cases} (\phi_n(A))^2 r_0 & \text{se } m = 2n-1, \\ \phi_n(A)\phi_{n-1}(A) r_0 & \text{se } m = 2n \end{cases} \tag{A.2}$$

Sono inoltre utili le seguenti relazioni [7]:

$$u_{n-1} = \phi_{n-1}(A)\psi_{n-1}(A) r_0 \tag{A.3}$$

$$q_n = \phi_n(A)\psi_{n-1}(A) r_0 \tag{A.4}$$

Dove $\phi_j(x)$ é il medesimo dell'espressione $r_j{}^{\text{Bi-CG}} = \phi_j(A) r_0$ con $\phi_n(0) = 1$, mentre il polinomio ψ_n é definito ricorsivamente da [1]

$$\psi_n(\lambda) = \phi_n(\lambda) + \beta_n \psi_{n-1}(\lambda), \qquad \psi_0(\lambda) = 1 \tag{A.5}$$

Avendo definito in questo modo ψ_n, si puó anche scrivere che

$$\phi_n(\lambda) = \phi_{n-1}(\lambda) - \alpha_{n-1}\lambda\psi_{n-1}(\lambda). \tag{A.6}$$

Ricordando che $r_j^{CG-S} = \phi_j^2(A) r_0$ si puó anche dedurre l'uguaglianza: $w_{2n+1} = r_n$. Dalle definizioni ed espressioni appena elencate si ottiene la seguente uguaglianza

$$A y_m = \frac{1}{\alpha_{\lfloor (m-1)/2 \rfloor}} (w_m - w_{m-1}) \tag{A.7}$$

[1] Le seguenti grandezze α_n e β_n sono le stesse definite per l'algoritmo CG-S

che stabilisce una relazione fra i vettori y e w (la notazione $\lfloor x \rfloor$ indica la parte intera inferire di x).

Quest'ultima equazione puó essere riscritta nella forma matriciale

$$AY_m = W_{m+1}{B_m}^{(e)} \tag{A.8}$$

che fa uso delle seguenti definizioni

$$\begin{aligned} Y_m &= [y_1, y_2, \ldots, y_m] \\ W_{m+1} &= [w_1, w_2, \ldots, w_m, w_{m+1}] \end{aligned}$$

$${B_m}^{(e)} = \begin{bmatrix} 1 & 0 & \ldots & 0 \\ -1 & 1 & \ddots & \vdots \\ 0 & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 1 \\ 0 & \ldots & 0 & -1 \end{bmatrix} \cdot (diag(\alpha_0, \alpha_0, \alpha_1, \ldots, \alpha_{\lfloor (m-1)/2 \rfloor}))^{-1}$$

La matrice ${B_m}^{(e)}$ é quindi una matrice bidiagonale inferiore $(m+1) \times m$. Infine, sempre come conseguenza delle equazioni (A.6),(A.5) e dalla (A.1) si puó scrivere [2]

$$K_m(r0, A) = span\{y_1, y_2, \ldots, y_m\} = \{Y_m z | z \in \mathbf{C}^m\} \tag{A.9}$$

Da quest'ultima equazione deriva il fatto che ogni possibile soluzione x_m relativa all' m-esima iterazione puó esser scritta come

$$x_m = x_0 + Y_m z \quad \text{per un certo } z \in \mathbf{C}^m \tag{A.10}$$

Utilizzando l'equazione matriciale (A.8) si puó calcolare il residuo relativo ad un certo x_m:

$$r_m = r_0 - AY_m z = W_{m+1}\left(e_1^{(m+1)} - B_m^{(e)} z\right) \tag{A.11}$$

[2]Si definisce con $K_m(v, A) = span\{v, Av, A^2v, ..., A^{m-1}v\}$ lo spazio di Krylov di dimensione m

dove si definisce $e_1^{(m+1)} = [1 \quad 0 \quad \dots \quad 0]^T \in \mathbf{N}^{m+1}$. Se si volesse minimizzare la norma di r_m bisognerebbe trovare il valore del vettore z risolvendo un problema ai minimi quadrati di dimensione $N \times (m+1)$, perché in generale la matrice W_{m+1} é densa e le sue colonne non sono ortogonali fra loro. La tecnica di "quasi" minimizzazione prevede invece un approccio diverso. Si definisce una matrice di riscalamento nel seguente modo:

$$\Omega_{m+1} = diag(\omega_1, \omega_2, \dots, \omega_{m+1}), \quad \omega_k > 0, k = 1, 2, \dots, m+1 \tag{A.12}$$

e le seguenti grandezze

$$f_{m+1} = \omega_1 e_1^{(m+1)} \text{ e } H_m^{(e)} = \Omega_{m+1} B_m^{(e)} \tag{A.13}$$

L'equazione (A.11) viene a questo punto riscritta come

$$r_m = W_{m+1} \Omega_{m+1}^{-1} \left(f_{m+1} - H_m^{(e)} z \right) \tag{A.14}$$

Si puó ora definire il passo di aggiornamento della soluzione nell'algoritmo TFQMR:

$$x_m = x_0 + Y_m z_m \tag{A.15}$$

dove $z_m \in \mathbf{C}$ é la soluzione del problema ai minimi quadrati

$$\tau_m := \|f_{m+1} - H_m^{(e)} z_m\| = \underset{z \in \mathbf{C}}{min} \quad \|f_{m+1} - H_m^{(e)} z\| \tag{A.16}$$

In quest'ultima equazione e nelle successive si é utilizzato il simbolo $\|.\|$ per denotare la norma euclidea.

La soluzione x_m, relativa all'iterazione m, non é unica in quanto dipende dalla scelta dei pesi ω_k. Sono possibile varie strategie per la scelta di tali coefficenti: quella seguita nell'implementazione dell'algoritmo TFQMR realizzata prevede l'uso della seguente espressione

$$\omega_k = \|w_k\|, \quad k = 1, 2, \dots, m+1 \tag{A.17}$$

Questa scelta fa sí che le colonne della matrice $W_{m+1}\Omega_{m+1}^{-1}$ siano riscalate per diventare dei vettori a norma unitaria. Si puó dimostrare che la (A.16) puó essere risolta in modo rapido semplicemente calcolando τ_m in modo ricorsivo. Il risultato di quanto visto fino ad ora é che l'approccio QMR si basa non tanto sulla minimizzazione dell'espressione completa del residuo r_m come definito dalla (A.11) ma sulla minimizzazione del solo termine vettore della moltiplicazione matrice per vettore definita nell'uguaglianza di destra dell' equazione (A.11); da questo fatto deriva il nome *quasi minimal residual*. Viene di seguito riportato il lemma, la cui dimostrazione é riportata in [7], che spiega come il problema ai minimi quadrati (A.16) puó essere risolto semplicemente aggiornando il valore di τ_m ad ogni passo di iterazione, sfruttando la definizione di un vettore di "appoggio" $\tilde{z}_m$ definito nel modo seguente

$$\tilde{x} = x_0 + Y_m \tilde{z}, \text{ dove } \tilde{z} = H_m^{-1} f_m \tag{A.18}$$

Lemma 1 *Siano* $\omega_1 > 0$, $m \geq 0$ *e*

$$H_m{}^{(e)} = \begin{bmatrix} H_m \\ h_{m+1,m} e_m^T \end{bmatrix} = \begin{bmatrix} H_{m-1}^{(e)} & * \\ 0 & h_{m+1,m} \end{bmatrix}$$

Una matrice di Hessenberg superiore di rango m, dove $e_m^T = [0 \quad \ldots \quad 0 \quad 1]$. *Sia poi* z_k, *con* $z_k \in \mathbf{C}$ *e* $k = m-1, m$ *la soluzione del problema ai minimi quadrati*

$$\tau_k := \min_{z \in \mathbf{C}^k} \quad \|f_{k+1} - H_k^{(e)} z\|, \ \textit{dove}\ f_{k+1} = \omega_1 e_1^{(k+1)} \in \mathbf{R}^{k+1} \tag{A.19}$$

Sia infine la matrice $m \times m$ H_m *non singolare e definendo*

$$\tilde{z}_m = H_m^{-1} f_m \tag{A.20}$$

Allora risulta

$$z_m = (1 - c_m^2) \begin{bmatrix} z_{m-1}^{(e)} \\ 0 \end{bmatrix} + c_m^2 \tilde{z_m} \quad \textit{e} \quad \tau_m = \tau_{m-1} \theta_m c_m \tag{A.21}$$

dove

$$\theta_m = \frac{1}{\tau_{m-1}} \|f_{m+1} - H_m^{(e)} \tilde{z}_m\|, \qquad c_m = \frac{1}{\sqrt{1 + \theta_m^2}} \tag{A.22}$$

L'applicazione del lemma 1 appena enunciato al problema ai minimi quadrati (A.16) rende possibile modificare la procedura iterativa utilizzata nell'algoritmo CG-S in modo tale da ottenere l'algoritmo TFQMR. In particolare é immediatamente verificabile la relazione fra x_m ed $\tilde{x}_m$:

$x_m = (1 - c_m^2)x_{m-1} + c_m^2\tilde{x}_m$

Definendo la quantitá d_m come segue:

$d_m = \frac{1}{\alpha_{\lfloor (m-1)/2 \rfloor}}(\tilde{x}_m - x_{m-1})$

si puó scrivere l'equazione che aggiorna la soluzione x ad ogni iterazione:

$$x_m = x_{m-1} + (c_m^2 \alpha_{\lfloor (m-1)/2 \rfloor})d_m \tag{A.23}$$

Con semplici passaggi ed utilizzando le varie quantitá ausiliarie definite si possono infine riscrivere le espressioni dei vettori y_m e w_m:

$$v_n = Ay_{2n+1} + \beta_n(Ay_{2n} + \beta_n v_{n-1}) \qquad \text{e} \qquad w_{m+1} = w_m - \alpha_{\lfloor (m-1)/2 \rfloor} Ay_m.$$

A.2 L'algoritmo TFQMR con pesi standard

La descrizione che segue prevede l'impiego di una strategia di "riscalamento" in accordo con l'espressione (A.17).

1. Inizio:
    - si sceglie x_0 vettore iniziale
    - si calcola $w_1 = y_1 = r_0 = b - Ax_0$ e $v_0 = Ay_1$;
    - si sceglie $d_0 = 0$, $\tau_0 = \|r_0\|, \theta_0 = 0, \eta_0 = 0$
    - si sceglie $\tilde{r}_0$ vettore arbitrario tale che $\rho_0 = (r_0, \tilde{r}_0) \neq 0$, ad esempio: $\tilde{r}_0 = r_0$
2. Ciclo: for $i = 1, 2, 3, \ldots$ finchè x_i non è abbastanza accurata
    - $\sigma_{n-1} = (\tilde{r_0}, v_{n-1})$

- $\alpha_{n-1} = \rho_{n-1}/\sigma_{n-1}$
- $y_{2n} = y_{2n-1} - \alpha_{n-1}v_{n-1}$
- Ciclo: for $m =$ 2n-1,2n
  - $w_{m+1} = wm - \alpha n - 1Ay_m$
  - $\theta_m = \|w_{m+1}\|/\tau_{m-1}$
  - $c_m = 1/\sqrt{1+\theta_m{}^2}$
  - $\tau_m = \tau_{m-1}\theta_m c_m$
  - $\eta_m = c_m{}^2\alpha_{n-1}$
  - $d_m = y_m + (theta_m{}^2\eta_{m-1}/\alpha_{n-1})d_{m-1}$
  - $x_m = x_{m-1} + \eta_m d_m$
  - se x_n é abbastanza accurata allora fine
- $\rho_n = (\tilde{r}_0, w_{2n-1})$
- $\beta_n = \rho_n/\rho_{n-1}$
- $y_{2n+1} = w_{2n+1} + \beta_n y_{2n}$
- $v_n = Ay_{2n+1} + \beta_n(Ay_2 n) + \beta_n v_{n-1}$

Il test sulla convergenza della soluzione x_m può essere effettuato calcolando la norma del residuo r_m relativo ad x_m, utilizzando quindi l'usuale formula $r_m = \|b - Ax_0\|$. Per calcolare tale norma é necessario eseguire due volte per ogni iterazione del ciclo principale (indice n) un prodotto matrice per vettore, una sottrazione vettoriale, ed il calcolo della norma di un vettore. Inoltre il calcolo di $\|r_m\|$ non é necessario nel procedimento iterativo se non per il test sulla convergenza. Un metodo alternativo per determinare se si é ottenuta la convergenza consiste nel servirsi della seguente espressione, che definisce un limite superiore al valore del residuo e puó essere calcolata con un costo computazionale trascurabile:

$$\|r_m\| \leq \|W_{m+1}\Omega_{m+1}{}^{-1}\| \cdot \|f_{m+1} - H_m{}^{(e)}z_m\| \leq \sqrt{m+1}\tau_m \tag{A.24}$$

É importante ripetere che tale relazione rappresenta una maggiorazione del residuo: l'utilizzo per il test di uscita dal ciclo iterativo dell'espressione approssimata $r_m \simeq \sqrt{m+1}\tau_m$ puó risultare in alcuni casi troppo restrittivo come si vedrá dall'analisi dei risultati sperimentali.

A.3 Alcune note sull'efficienza del TFQMR

L'algoritmo TFQMR é stato sviluppato in tempi recenti principalmente con l'obbiettivo di mantenere o migliorare le buone caratteristiche di velocitá di convergenza mostrate dall'algoritmo CG-S in numerose applicazione, migliorandone al contempo l'affidabilitá e la regolaritá del processo di convergenza. Sono riportati in [7, 9] risultati prodotti da quest'algoritmo in alcune applicazioni per le quali il TFQMR si é dimostrato in alcuni casi (problemi alle derivate parziali) piú regolare e veloce nel processo di convergenza rispetto agli algoritmi CG-S e BiCGStab, mentre in altri no (analisi di catene di Markov originate da SANs,*Stochastic Automa Networks*).

Si puó notare peró che la riduzione del numero di iterazioni richieste non é sufficiente per garantire anche un risparmio di tempo di calcolo, in quanto la singola iterazione dell'algoritmo TFQMR é piú dispendiosa di quelle del CG-S e del BiCGStab. Nella tabella A.1 é elencato il numero delle principali operazioni richieste da ogni algoritmo: si puó notare come l'algoritmo TFQMR, soprattutto nel caso in cui si effettui il calcolo della norma del residuo per il test di uscita (seconda colonna), sia sostanzialmente piú dispendioso degli altri due in ciascuna iterazione.

Tipo operazione	TFQMR	TFQMR+$\|r_n\|$	CG-S	BiCGStab
matrice · vettore	3	5	2	2
vettore · vettore	2	3	2	4
vettore + vettore	9	11	8	6
scalare · vettore	10	10	6	6

Tabella A.1: Numero di operazioni fondamentali eseguite per ogni iterazione

Bibliografia

[1] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains.* Princeton University Press, 1994.

[2] G. Monegato. *Fondamenti di calcolo Numerico.* Levrotto e Bella, 1990.

[3] A. Quarteroni, R. Sacco, and F. Saleri. *Matematica numerica.* Springer-Verlag, 1998.

[4] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing.* Benjamin/Cummings, 1994.

[5] W. J. Stewart, Y. Saad, and B. Philippe. Numerical methods in Markov chain modeling. *Operations Research*, 40(6):1157–1179, November-December 1992.

[6] J. Erhel. A parallel GMRES version for general sparse matrices. *Electronic transactions on Numerical Analysis*, 3:160–176, 1995.

[7] R. W. Freund. A transpose-free quasi-minimal residual algorithm for non-hermitian linear systems. *SIAM J. Sci. Comput.*, 14:470–482, 1993.

[8] P. Marenzoni. *Algoritmi Paralleli per la Soluzione di Reti di Petri Stocastiche.* Tesi di Dottorato di Ricerca in Tecnologie dell'Informazione, Universitá di Parma, 1998.

[9] B. Plateau, W. J. Stewart, and M. Silva. *Numerical Solution of Markov Chains (NSMC 99).* Prensas Universitas de Zaragoza, 1999.

[10] P. S. Pacheco. *Parallel programming with MPI.* Morgan Kaufmann Publishers, 1997.

[11] L. Amighetti. *Algoritmi proiettivi per la risoluzione di sistemi lineari sparsi derivati dall'analisi di reti di Petri.* Tesi di laurea in Ingegneria Elettronica, Universitá di Parma, 1998.

[12] E. Riccardi. *Realizzazione e confronto di algoritmi per la risoluzione di catene di Markov di grandi dimensioni generate da modelli GSPN.* Tesi di laurea in Ingegneria Elettronica, Universitá di Parma, 1998.

[13] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastics Petri Nets.* John Wiley & Sons, 1995.

[14] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Solving Linear System on Vector and Shared Memory Computers.* SIAM, 1991.

[15] Message passing interface forum. *MPI: A message passing Interface forum.* University of Tennessee, 1994.

[16] Message passing interface forum. *MPI-2:Extension to the Message-Passing Interface.* University of Tennessee, 1997.

[17] A. A. Desrochers. *Modelling and Control of Automated Manufacturing Sustems.* IEEE Computer Society Press Tutorial, 1990.