

POLITECNICO DI MILANO

Facoltà di Ingegneria dei Processi Industriali

Corso di Laurea Magistrale in Ingegneria Chimica



METODI NUMERICI DI CALCOLO PARALLELO PER LA  
SOLUZIONE DI RETI DI REATTORI

Relatore: Prof. Ing. Tiziano FARAVELLI

Correlatori: Ing. Alberto CUOCI

Prof. Ing. Antonello BARRESI

Tesi di Laurea Magistrale di:

Carlo Maria CAMPELLI Matr. 751135

Alessandro STAGNI Matr. 749216

Anno Accademico 2010-2011



*Due valgon meglio di uno solo, perché hanno una buona ricompensa per la loro fatica.*

*Se infatti cadono, l'uno rialza l'altro; ma guai a chi è solo e cade, perché non ha nessun altro che lo rialzi!*

(Ecclesiaste 4:9-10)



# Indice

Indice delle figure .....	v
Indice delle tabelle .....	vii
Abstract .....	ix
Abstract (English) .....	xi
Capitolo 1 Il KPP nell'analisi delle emissioni di inquinanti.....	1
1.1. Risoluzione di sistemi non lineari di grandi dimensioni .....	1
1.1.1. Panoramica sui metodi risolutivi.....	1
1.1.2. Criteri di convergenza .....	4
1.1.3. Caratteristiche del KPP .....	6
1.2. La struttura del sistema.....	9
1.3. Analisi CFD e schemi cinetici dettagliati .....	11
1.4. Analisi delle emissioni inquinanti: gli NO <sub>x</sub> .....	13
Capitolo 2 Calcolo parallelo: utilità e applicazioni.....	17
2.1. L'importanza crescente del calcolo parallelo .....	17
2.2. Memoria condivisa e distribuita .....	18
2.3. Memoria condivisa e OpenMP .....	21
2.4. Memoria distribuita: il protocollo MPI .....	23
2.5. Prestazioni di un algoritmo parallelo.....	25
2.6. Speedup ed efficienza.....	26
2.7. Legge di Amdahl .....	27
2.8. Scalabilità e legge di Gustafson .....	28
Capitolo 3 Metodi numerici .....	31
3.1. Potenzialità e limiti dello schema risolutivo originale .....	31
3.2. Dal sistema globale alla soluzione locale.....	32

3.3.	Risoluzione della sequenza di CSTR.....	34
3.4.	Splitting.....	36
3.4.1.	Risoluzione tramite splitting: l'esempio di Strang .....	37
3.4.2.	Predictor-Corrector .....	41
3.5.	Implementazione nel KPP .....	43
Capitolo 4	Approccio alla parallelizzazione .....	55
4.1.	Prodotto tra matrici .....	55
4.2.	Analisi di scalabilità.....	59
4.3.	Parallelizzazione del prodotto tra matrici .....	62
4.4.	Parallelizzazione di un sistema con trasporto e reazione.....	69
Capitolo 5	Dal seriale al parallelo: la nuova forma del KPP.....	75
5.1.	Struttura originale del Post-Processore Cinetico .....	75
5.2.	Design parallelo dell'algoritmo .....	77
5.2.1.	Scelta del modello di parallelizzazione adeguato.....	78
5.2.2.	Distribuzione dei dati tra i processi .....	79
5.2.3.	Approccio parallelo ai metodi risolutivi.....	81
5.2.4.	Load Balancing e distribuzione ottimale delle celle.....	84
5.3.	Ottimizzazione della convergenza della sequenza di CSTR .....	86
5.4.	Il KPP in sistemi a memoria distribuita .....	88
5.5.	Prestazioni dell'algoritmo parallelo.....	90
5.5.1.	Confronto con l'algoritmo originale.....	91
5.5.2.	Scalabilità dell'algoritmo .....	92
5.6.	Efficienza dell'algoritmo e schema cinetico.....	97
Capitolo 6	Conclusioni e prospettive .....	107
6.1.	Bilanciamento del carico e allocazione processi .....	107
6.2.	Sequenza di CSTR e metodi globali .....	109

6.3. Confronto con altre librerie parallele .....	114
6.4. Conclusioni.....	117
Glossario .....	119
Bibliografia .....	125
Ringraziamenti .....	127



## Indice delle figure

Figura 1: Jacobiano per le specie presenti in un singolo reattore della rete...	7
Figura 2: Struttura booleana della matrice Jacobiana del sistema non lineare per una griglia bidimensionale di reattori (sinistra) e un ingrandimento (destra).....	7
Figura 3: Schema logico per l'algoritmo di risoluzione di sistemi non lineari di grandi dimensioni.....	8
Figura 4: Griglia monodimensionale .....	9
Figura 5: Schema di comunicazione tra celle adiacenti in una griglia monodimensionale .....	10
Figura 6: Matrice dei coefficienti in una griglia monodimensionale.....	10
Figura 7: Griglia bidimensionale .....	11
Figura 8: Andamento della frequenza di calcolo dei processori commercializzati nel periodo 1993-2006 (fonte: HP).....	17
Figura 9: Potenza dissipata da un singolo processore in funzione della frequenza di <i>clock</i> .....	18
Figura 10: Differenza tra processi e <i>thread</i> .....	20
Figura 11: Il modello <i>fork-join</i> supportato da OpenMP.....	22
Figura 12: <i>Speedup</i> previsto dalla legge di Amdahl in funzione della frazione di codice non parallelizzabile.....	28
Figura 13: Confronto tra procedura sequenziale e <i>predictor-corrector</i> al variare del <i>time-step</i> .....	43
Figura 14: Media della norma dei residui al crescere delle iterazioni .....	49
Figura 15: Struttura <i>cube-connected</i> del terzo ordine (un nodo comunica con 3 adiacenti).....	58
Figura 16: Schema del modello <i>master-slave</i> .....	62

Figura 17: Confronto tra lo <i>speedup</i> massimo teorico (in blu) e quello ottenuto dalle analisi di scalabilità per l’algoritmo di moltiplicazione tra matrici .....	67
Figura 18: Andamento del tempo totale di esecuzione dell'algoritmo .....	72
Figura 19: Allocazione della griglia a blocchi (sinistra) o in maniera alternata (destra) con 3 <i>slave</i> a disposizione .....	80
Figura 20: Logica di comunicazione delle frazioni massive.....	83
Figura 21: Schema della rete creata tramite protocollo <i>ssh</i> .....	89
Figura 22: Andamento <i>speedup</i> nei quattro casi considerati .....	93
Figura 23: Andamento dell'efficienza al variare del numero di processori per ogni caso di studio.....	94
Figura 24: <i>Speedup</i> raggiunto nei casi <i>1X</i> .....	100
Figura 25: <i>Speedup</i> raggiunto nei casi <i>2X</i> .....	101
Figura 26: <i>Speedup</i> raggiunto nei casi <i>4X</i> .....	101
Figura 27: Efficienza raggiunta nei casi <i>1X</i> .....	102
Figura 28: Efficienza raggiunta nei casi <i>2X</i> .....	103
Figura 29: Efficienza raggiunta nei casi <i>4X</i> .....	103
Figura 30: Andamento della media della norma 1 dei residui con soluzione in sequenza, ODE globale e Newton globale.....	111
Figura 31: Confronto qualitativo tra la velocità di convergenza con il solo metodo della sequenza di CSTR e l’approccio misto .....	113

## Indice delle tabelle

Tabella 1: Confronto tra frazioni massive ottenute con i due diversi algoritmi .....	44
Tabella 2. Riassunto delle caratteristiche dei casi studiati.....	47
Tabella 3. Valori per il criterio di stop della risoluzione della sequenza di CSTR.....	47
Tabella 4: Tempi di esecuzione per il casi di studio 1 .....	48
Tabella 5: Tempi di calcolo medi per un'iterazione .....	49
Tabella 6: Tempi di esecuzione per il casi di studio 2 .....	51
Tabella 7. Confronto tra i tempi di calcolo dell'algoritmo tradizionale e dell'approccio misto sequenza di CSTR e <i>predictor-corrector</i> .....	53
Tabella 8: Analisi di scalabilità dell'algoritmo parallelo per il prodotto tra matrici .....	66
Tabella 9: Frazione di codice parallelizzabile nei due casi.....	68
Tabella 10. Risultati dell'analisi di scalabilità .....	71
Tabella 11: Prestazioni dell'algoritmo parallelo con distribuzione delle celle a blocchi .....	84
Tabella 12: Prestazioni dell'algoritmo parallelo con distribuzione delle celle in maniera alternata.....	84
Tabella 13: <i>Speedup</i> ottenuto con l'algoritmo a blocchi .....	85
Tabella 14: <i>Speedup</i> ottenuto con l'algoritmo alternato.....	85
Tabella 15: Confronto tra i tempi di esecuzione dei due algoritmi con e senza ottimizzazione della risoluzione della sequenza .....	88
Tabella 16: <i>Benchmark</i> dell'algoritmo parallelo al variare del caso e del numero di processori.....	90

Tabella 17: Tempi di esecuzione dell'algoritmo originale per i quattro casi di studio .....	91
Tabella 18: Rapporti tra i tempi relativi di esecuzione dei due algoritmi ....	91
Tabella 19: <i>Speedup</i> ottenuto nei vari casi di studio.....	93
Tabella 20: Andamento dell'efficienza al variare del numero di processori nei differenti casi di studio .....	94
Tabella 21: Andamento tempi iterazioni e comunicazione.....	95
Tabella 22: Tempi di esecuzione del Post-Processore su 12 casi di studio .	99
Tabella 23: <i>Speedup</i> ottenuto per ciascun caso di studio.....	100
Tabella 24: Efficienza ottenuta per ciascun caso .....	102
Tabella 25: Analisi dei tempi caratteristici dei dodici casi di studio .....	104
Tabella 26: Tempo di una singola iterazione nel caso <i>IX-1step-gri30</i> adottando rispettivamente 1 e 2 slave .....	107
Tabella 27: Confronto tra i tempi per iterazione all'inizio e alla fine della procedura risolutiva.....	110

## Abstract

La valutazione delle emissioni inquinanti in seguito a combustione in bruciatori o motori è attualmente fonte di grande interesse tra i produttori di tali apparecchiature. Normative di recente introduzione pongono limiti precisi a tali emissioni, perciò è necessario elaborare un metodo di calcolo efficiente ed economico, e che fornisca risultati affidabili.

Il Post Processore Cinetico (KPP) è uno strumento numerico nato con lo scopo di gestire schemi cinetici completi e particolarmente complessi (fino a 500 specie chimiche), e a fornire un'analisi fluidodinamica dettagliata, grazie ad una griglia di calcolo fitta. Ogni punto della griglia può essere considerato un reattore continuo perfettamente miscelato (CSTR), per cui l'intero problema si può riformulare come la soluzione numerica di una rete di reattori. Un tale calcolo è decisamente impegnativo dal punto di vista computazionale e richiede inoltre un'enorme quantità di memoria RAM per allocare i dati e i risultati; non è possibile risolvere il problema con i metodi tradizionali, a causa dei limiti tecnologici propri della dimensione della memoria, del tempo di calcolo richiesto eccessivo e poiché i metodi di calcolo globali che convergono più rapidamente, come il metodo di Newton, non sono in grado di trovare la soluzione se i residui dei dati di primo tentativo sono troppo elevati.

Per risolvere le difficoltà citate l'algoritmo risolutivo per la rete di reattori è ripensato nell'ottica del calcolo parallelo su una macchina a memoria distribuita. Questo consente di avere a disposizione sufficiente memoria e di tentare di ridurre in modo significativo il tempo di calcolo, a seconda del numero di processi paralleli usati.

Nel presente lavoro di Tesi viene proposto un algoritmo parallelo per il KPP caratterizzato da un metodo risolutivo locale. Vengono analizzati diversi schemi risolutivi per la rete di reattori ed anche formulazioni alternative della matrice del sistema globale. La validità della soluzione proposta è valutata su una macchina multiprocessore a memoria condivisa. Infine, vengono proposte alcune soluzioni per aumentare l'efficienza dell'algoritmo mediante l'introduzione di metodi globali.



## **Abstract (English)**

The estimation of pollutant emissions from combustion devices, such as burners or engines is nowadays a source of major interest among manufacturers. Newly introduced regulations set precise limits to emissions, hence the need for an efficient and cost-effective way to perform the calculations and provide reliable results.

The Kinetic Post Processor (KPP) is a numerical tool aimed at addressing the need to manage complete and very big kinetic schemes (with up to 500 chemical species), as well as providing a detailed fluid dynamics study thanks to a thick mesh. Every grid point belonging to the mesh can be considered a Continuous Stirred Tank Reactor (CSTR), therefore the entire problem can be formulated as the numerical solution of a reactor network. Such a computation is very compelling from a numerical point of view and also requires a huge amount of RAM memory to store the data and results; addressing the solution with traditional methods is unfeasible, due to memory size technological constraints, the excessive computation time needed and because faster converging global methods, such as Newton method, are not able to find the solution if the first guess inputs' residuals are too high.

A new approach is attempted to meet the aforementioned challenges: rethinking the reactor network solution algorithm as a parallel computation on a distributed memory machine. This allows to grant that enough memory is available and attempts at reducing significantly the computation time, according to the number of parallel processes used.

In this Thesis a parallel algorithm for the KPP is proposed, featuring a local solution method. Different reactor network solution patterns are analyzed together with alternative formulation of the global system matrix. The solution proposed is benchmarked on a shared memory, multi-core machine. Finally, some proposals are made to further enhance the efficiency of the algorithm through the introduction of global methods.



# Capitolo 1

## Il KPP nell'analisi delle emissioni di inquinanti

### 1.1. Risoluzione di sistemi non lineari di grandi dimensioni

#### 1.1.1. Panoramica sui metodi risolutivi

In molti problemi caratteristici dell'ingegneria chimica vi è l'esigenza di risolvere sistemi non lineari di equazioni. Alcuni esempi classici possono essere il calcolo di equilibri chimici, la soluzione di reti di reattori, il dimensionamento di scambiatori di calore o anche la risoluzione di alcuni tipi di equazioni di stato. Da quando gli strumenti di calcolo informatici sono diventati disponibili ad un numero di utenti estremamente più vasto e calcolatori relativamente potenti sono divenuti economicamente accessibili ai privati, sono proliferati numerosi strumenti risolutivi che fanno uso di svariati algoritmi, ciascuno più o meno adatto a risolvere una determinata tipologia di problema. Ciò significa che oggi l'attenzione posta alla precisione del metodo numerico usato non può prescindere dalle caratteristiche della *routine* risolutiva di robustezza ed efficienza.

Il problema che si vuole affrontare nel presente lavoro di Tesi ha la forma di un sistema di equazioni differenziali non lineari sparso, con struttura a blocchi. Una caratteristica non meno importante sono le dimensioni del problema considerato: ovvero oltre le 100 000 equazioni e relative incognite. Il numero di equazioni coinvolto nel sistema è uno dei fattori che maggiormente influenzano l'efficienza della risoluzione, sia in termini di una maggiore richiesta di tempo di calcolo ed occupazione di memoria, che di effetto sulla soluzione finale: in sistemi estremamente grandi anche metodi consolidati in letteratura e nella pratica possono non convergere verso la soluzione cercata. Lo scopo del presente lavoro di Tesi è quello di stabilire una procedura risolutiva sufficientemente precisa per la soluzione di sistemi non lineari di grandi dimensioni ed implementarla in un algoritmo robusto ed efficiente. La tipologia di problemi che richiedono di essere affrontati in questa ottica hanno generalmente delle caratteristiche comuni: si tratta di sistemi *stiff*, sparsi e strutturati, quindi un'ideale *routine* risolutiva potrà

trarre vantaggio dalla struttura del sistema. Per sistema *stiff* si intende un sistema differenziale ben condizionato con rapporto tra massimo e minimo autovalore grande. Fisicamente parlando, ciò avviene quando nel sistema sono presenti equazioni differenziali caratterizzate da scale di tempo molto differenti tra di loro, oppure da soluzioni che differiscono per diversi ordini di grandezza.

Quando si analizza un problema non lineare, il metodo di Newton è uno dei più utilizzati per la sua caratteristica di approssimare la soluzione al problema risolvendo una serie di problemi lineari. La struttura generale del metodo è esposta nel seguito; data un'equazione non lineare del tipo:

$$F(x) = 0 \quad (1.1)$$

ed essendo  $F$  derivabile in modo continuo almeno una volta. Avendo a disposizione una soluzione di primo tentativo  $x^0$  della soluzione cercata  $x^*$ , procedendo per linearizzazioni successive si ottiene il metodo di Newton generalizzato:

$$F'(x^k)\Delta x^k = -F(x^k), x^{k+1} = x^k + \Delta x^k, \quad k = 0, 1, \dots \quad (1.2)$$

Condizione necessaria per la risoluzione dell'Equazione (1.2) è che esista e sia invertibile la derivata  $F'(x)$ . Tuttavia è bene ricordare che in analisi numerica è sempre sconsigliabile effettuare inversioni, specie quando si tratti di matrici, per cui il rispetto di questa condizione viene valutato implicitamente. Dal momento che il problema considerato non è una singola equazione, bensì come si è detto, un sistema, dalla condizione di derivabilità si ha che deve esistere per il sistema considerato una matrice Jacobiana. Lo Jacobiano si può ottenere per approssimazione con metodi numerici, ma talvolta è vantaggioso calcolarlo una sola volta analiticamente e utilizzare l'espressione ottenuta all'interno dell'algoritmo. L'espressione riportata nell'Equazione (1.2) rimanda alla sequenza operativa di ciascuna iterazione: prima si calcolano le correzioni di Newton  $\Delta x^k$ , quindi si migliora la  $k$ -esima iterazione ottenendo  $x^{k+1}$ , evitando anche in questo modo di incorrere nella perdita di cifre significative, che potrebbe avvenire se si

risolvesse di direttamente in  $x^{k+1}$ . Dal metodo generale si sono sviluppate numerose varianti, tra cui ad esempio il metodo di Newton semplificato:

$$F'(x^0)\overline{\Delta x}^k = -F(x^k), \quad x^{k+1} = x^k + \overline{\Delta x}^k, \quad k = 0, 1, \dots \quad (1.3)$$

nel quale si mantiene la derivata iniziale per tutta l'iterazione. Rispetto al metodo di Newton tradizionale si risparmia il costo computazionale per ciascuna iterazione, però questo a scapito del numero di iterazioni necessarie per giungere a convergenza. Esiste poi un'intera famiglia di metodi *Newton-like*, che hanno in comune il fatto che lo Jacobiano viene sostituito da una sua approssimazione  $F'(z)$  in cui  $z \neq x^0$ , oppure è costruito in modo tale per cui vale:

$$M(x^k)\delta x^k = -F(x^k), \quad x^{k+1} = x^k + \delta x^k, \quad k = 0, 1, \dots \quad (1.4)$$

Questa metodologia può essere vantaggiosamente usata per sistemi sparsi riscrivendo opportunamente lo Jacobiano in modo da trascurare quelle parti in cui i contributi sono poco significativi rispetto al resto. In questo modo è possibile ridurre il peso di ciascuna iterazione ed utilizzare un solutore sparso diretto; se si scelgono opportunamente i contributi da trascurare, l'esito dell'iterazione non viene compromesso in modo apprezzabile. Infine, laddove sia possibile utilizzare metodi di eliminazione diretta si ha il metodo di Newton esatto; librerie di algebra numerica che fanno largo uso di questa tipologia di metodo, sia per solutori pieni che sparsi sono ad esempio LAPACK e SPARSPACK. La prima, *Linear Algebra PACKage*, è una raccolta di *routine* scritte in Fortran 90, nata con lo scopo di rendere efficiente l'esecuzione in parallelo su macchine a memoria condivisa delle precedenti EISPACK e LINPACK. Questi sono altri solutori per sistemi lineari, o dei minimi quadrati, oppure problemi agli autovalori. Lo scheletro algebrico delle librerie nominate sono le BLAS, *Basic Linear Algebra Subprograms*, una serie di sottoprogrammi che svolgono in modo efficiente operazioni di moltiplicazione tra matrici o risoluzione di sistemi triangolari. Il secondo strumento, SPARSPACK, è una serie di *subroutine* per la soluzione sistemi lineari sparsi di grandi dimensioni.

Inoltre, è opportuno distinguere tra i metodi locali e quelli globali. La principale differenza consiste nel fatto che per implementare opportunamente un

metodo locale è necessario disporre di una soluzione di primo tentativo abbastanza vicina alla soluzione cercata, mentre i metodi globali sono in grado di compensare una non ottimale soluzione di primo tentativo tramite opportune strategie (*damping*, o *adaptive trust region*). Tuttavia quando le dimensioni del sistema non lineare da risolvere diventano molto grandi, non è più possibile risolvere direttamente, ovvero in modo “esatto” i problemi lineari generati dalla risoluzione secondo Newton, ma devono essere risolti iterativamente. Per tale motivo i metodi appartenenti a questa famiglia sono detti “inesatti”, e sono particolarmente adatti per problemi di dimensioni enormi. In questo caso lo schema risolutivo si avvale di una “iterazione interna” al  $k$ -esimo passo:

$$\begin{aligned} F'(x^k) \delta x_i^k &= -F(x^k) + r_i^k, \quad k = 0, 1, \dots \\ x_i^{k+1} &= x_i^k + \delta x_i^k \quad i = 0, 1, \dots, i_{\max}^k \end{aligned} \quad (1.5)$$

in cui compaiono i residui  $r_i^k$ . Vi è poi una “iterazione esterna” in cui, data la soluzione di partenza, il risultato dell’iterazione è dato da:

$$x^{k+1} = x_i^{k+1}, \quad i = i_{\max}^k, \quad k = 0, 1, \dots \quad (1.6)$$

Rispetto all’impostazione del metodo di Newton esatto si può osservare l’insorgere di errori dovuti alla differenza  $\delta x_i^k - \Delta x^k$ .

### 1.1.2. Criteri di convergenza

Dato che, come si è detto, alcuni problemi portano alla scrittura di sistemi di equazioni di dimensioni notevoli, anche dell’ordine dei milioni di equazioni in milioni di incognite, stante l’elevato peso computazionale che la soluzione comporta, è importante giungervi in modo efficiente. Quindi, poiché la risoluzione avviene in modo iterativo, devono essere stabiliti degli opportuni criteri per valutare se ci si sta avvicinando alla soluzione, e quando interrompere la computazione perché si è raggiunto un grado di precisione sufficiente.

Per quanto riguarda i criteri di convergenza, dato un generico sistema non lineare in  $N$  equazioni, come nella (1.7):

$$\bar{f}(\bar{x}) = 0 \quad (1.7)$$

Si può utilizzare la norma euclidea come criterio per l'accettazione della soluzione, in modo tale che la soluzione è accettata se vale la (1.8):

$$\|\bar{f}(\bar{x}_{i+1})\|_2 < \|\bar{f}(\bar{x}_i)\|_2 \quad (1.8)$$

la quale afferma che la soluzione all'iterazione successiva è più vicina alla soluzione rispetto a quella dell'iterazione  $i$ -esima.

Al criterio generale bisogna aggiungere ulteriori considerazioni se le equazioni presentano ordini di grandezza diversi tra di loro: in tal caso conviene introdurre delle espressioni pesate per fare sì che le equazioni con ordini di grandezza inferiori non influenzino l'accettazione o meno della soluzione. Naturalmente è opportuno calcolare i pesi in modo automatico, per questo si considera la matrice Jacobiana  $J$  in modo tale che il criterio diviene:

$$\|J_i^{-1} \bar{f}(\bar{x}_{i+1})\|_2 < \|J_i^{-1} \bar{f}(\bar{x}_i)\|_2 \quad (1.9)$$

Questo metodo è vantaggioso se utilizza il metodo di Newton per risolvere il sistema, dal momento che in tal caso lo Jacobiano è già stato fattorizzato per risolvere il sistema lineare.

Per il criterio di stop vanno invece considerati più aspetti contemporaneamente. Innanzitutto è conveniente imporre un numero massimo di iterazioni dopo il quale la procedura si interrompe; oltre a ciò si considera la soluzione raggiunta se la funzione di merito che raccoglie i termini pesati assume un valore sufficientemente piccolo. Infine si possono fermare le iterazioni se tutte le componenti del vettore differenza tra la soluzione all'iterazione successiva e precedente assumono un valore sufficientemente piccolo. Il criterio prende matematicamente (Manca et al., 2009) la forma della (1.10):

$$\max_{1 \leq i \leq N} \left| \frac{\bar{x}_{i+1} - \bar{x}_i}{\max\{x_i, z_i\}} \right| \leq \varepsilon \quad (1.10)$$

in cui il termine  $z_i$  al denominatore serve ad evitare di dividere involontariamente per zero.

Nel caso considerato nel presente lavoro di Tesi, benché il numero di incognite e di equazioni sia molto elevato, generalmente si possono individuare *set* di equazioni che dipendono da un numero ridotto di variabili. Sfruttare la sparsità della matrice dei coefficienti consente perciò non solo una maggiore efficienza dell'algoritmo, perché non vengono fatte operazioni inutili laddove gli elementi sono nulli, ma permette anche un notevole risparmio di memoria dal momento che vengono registrati solo i termini diversi da zero. I termini nulli non vengono considerati automaticamente se lo Jacobiano viene calcolato in modo analitico. Infine anche l'aggiornamento dello Jacobiano conviene sia fatto in modo efficiente, ad esempio aggiornando solo le parti significativamente diverse ad ogni iterazione, invece di ricalcolarlo ogni volta.

### **1.1.3. Caratteristiche del KPP**

Le considerazioni fatte finora sono di carattere generale, ma sono un valido punto di partenza per affrontare il problema considerato nel Post-Processore Cinetico, il quale altro non è che, per l'appunto, un sistema non lineare di grandi dimensioni con struttura sparsa e di carattere *stiff*. Si vedrà dunque nel seguito come sia possibile applicare tali considerazioni al problema specifico, e quali aspetti risultino più critici. Prima di affrontare la soluzione dal punto di vista numerico ci sono alcuni aspetti che possono avvantaggiarne il raggiungimento. Innanzitutto l'analisi a rete di reattori (*Reactor Network Analysis*) consente di raggruppare celle in condizioni simili, e quindi di ridurre le dimensioni del sistema. Inoltre in ciascun punto nodale, o ipotetico reattore della rete, si considera una temperatura media fissata, in questo modo la non linearità del problema si riduce significativamente.

Per comporre il sistema si scrivono le equazioni di bilancio di massa per ciascuna specie, dando adito ad una struttura che può essere del tipo di Figura 1:

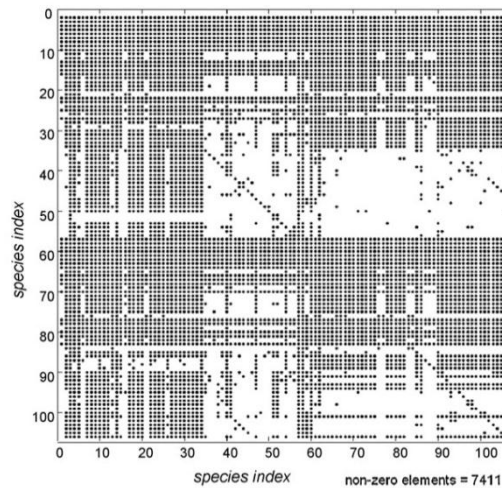


Figura 1: Jacobiano per le specie presenti in un singolo reattore della rete

Tale singolo blocco rappresenta lo Jacobiano per un singolo reattore della rete, perciò questa sotto-struttura viene ripetuta per ciascuna cella rispettando il *pattern* di comunicazione imposto dalla griglia. Per una griglia strutturata bidimensionale l'aspetto del sistema globale può essere del tipo di Figura 2:

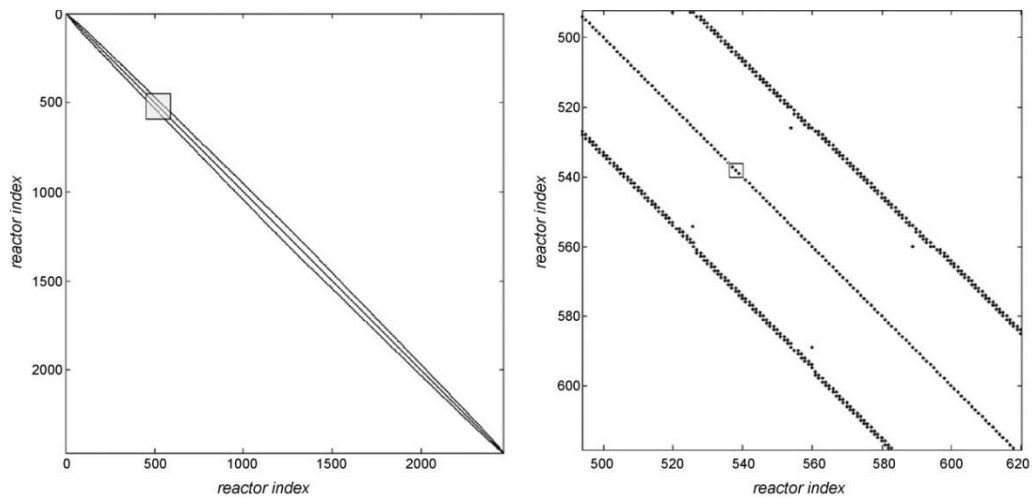


Figura 2: Struttura booleana della matrice Jacobiana del sistema non lineare per una griglia bidimensionale di reattori (sinistra) e un ingrandimento (destra)

Quindi la struttura globale del sistema è diagonale a blocchi. Come si può osservare la gran parte degli elementi della matrice sono nulli, anche se ciascuno dei puntini della Figura 2 rappresenta in realtà una matrice quadrata di dimensione  $N_{specie} \times N_{specie}$ , come mostrato in Figura 1.

Non è conveniente utilizzare un metodo di Newton globale a partire dalla soluzione di primo tentativo per il KPP, cioè quella data dall'analisi CFD con lo

schema cinetico non dettagliato: tale metodo non è abbastanza robusto. La procedura utilizzata allo stato dell'arte prevede di avvicinarsi alla soluzione tramite un procedimento iterativo reattore per reattore utilizzando un metodo di Newton locale. Si utilizza un “falso transitorio”, ovvero la soluzione di un sistema di equazioni differenziali ordinarie (ODE) per migliorare la soluzione di primo tentativo mentre si raggiunge la convergenza. Ad ogni successivo passaggio si valutano i residui sostituendo la soluzione trovata nelle equazioni del sistema e, quando i residui di tutte le equazioni hanno raggiunto valori sufficientemente piccoli si applica un metodo di Newton globale modificato all'intero sistema, che può quindi convergere verso la soluzione. Utilizzando il metodo di Newton globale si garantisce la convergenza verso la soluzione ed inoltre si rende molto più veloce la risoluzione rispetto al metodo locale iterativo, che è computazionalmente pesante. Per questo è importante innescare il metodo di Newton globale appena possibile, non appena la robustezza della soluzione lo consente, evitando di perdere tempo in iterazioni superflue. Una panoramica della logica dell'algorithm descritto si può osservare nella Figura 3:

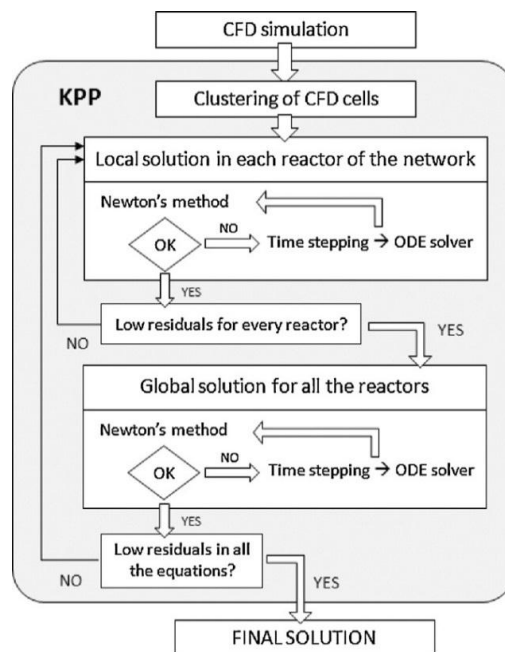


Figura 3: Schema logico per l'algorithm di risoluzione di sistemi non lineari di grandi dimensioni

Attualmente i limiti della metodologia finora descritta consistono nell'elevato peso computazionale che la procedura comporta, e nel grande fabbisogno di

memoria che un sistema di grandi dimensioni presenta. Nello svolgimento del lavoro di Tesi si vuole rispondere a queste esigenze proponendo un approccio atto a diminuire il tempo di calcolo per giungere alla soluzione del problema sfruttando tecniche di calcolo parallelo ed inoltre di affrontare gli aspetti numerici del calcolo rendendo l'algoritmo più efficiente grazie a criteri di bilanciamento del carico di lavoro e di innesco del metodo di Newton globale.

### 1.2. La struttura del sistema

La discretizzazione del *set* di equazioni porta alla scrittura di un sistema di equazioni algebriche che descrivono lo stato del sistema fisico. Il sistema di equazioni si può generalmente scrivere nella forma data dalla (1.11):

$$A\phi = B \quad (1.11)$$

in cui  $\phi$  è la variabile incognita di una generica proprietà di trasporto, la matrice  $A$  contiene i coefficienti delle equazioni algebriche, mentre  $B$  è il vettore dei termini noti dati, ad esempio, dalle condizioni al contorno o da eventuali termini di sorgente o scomparsa.

Nel caso di una griglia strutturata, visto lo schema di comunicazione discusso precedentemente, anche la matrice dei coefficienti assume una struttura ordinata. Si consideri, a titolo di esempio, un caso monodimensionale in cui una cella (o punto) può comunicare solo con quelle adiacenti, come esemplificato in Figura 4:



Figura 4: Griglia monodimensionale

La matrice dei coefficienti, che ha dimensione  $N_{celle} \times N_{celle}$ , avrà componenti non nulle dove la proprietà di trasporto si comunica da una cella all'altra. Quindi, nel caso esemplificato in Figura 4 si avrà che la prima cella  $P_1$  comunica solo con la  $P_2$ , mentre la  $P_2$  può comunicare sia con la  $P_1$  che con la  $P_3$ , e così via per le successive, come illustrato in Figura 5:

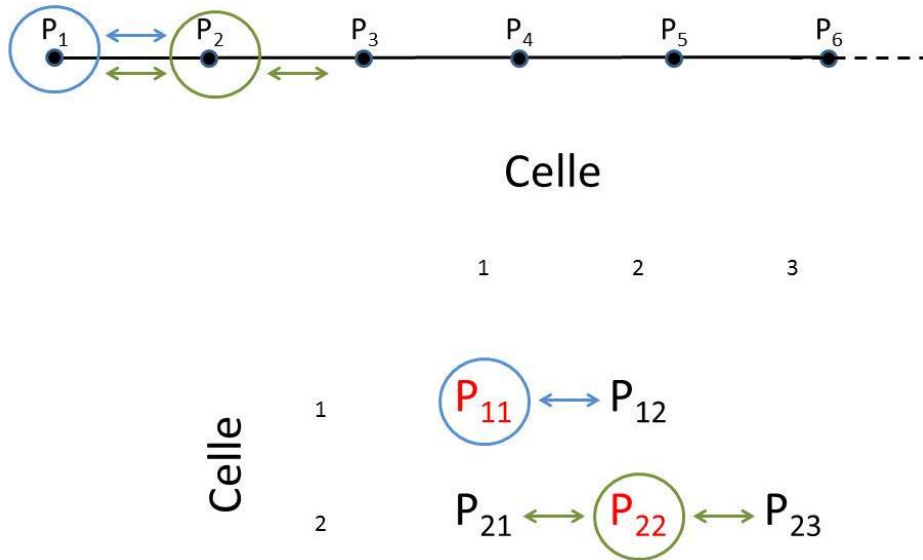


Figura 5: Schema di comunicazione tra celle adiacenti in una griglia monodimensionale

Ripetendo lo schema descritto per tutte le equazioni e su ciascuna cella si otterrebbe la matrice dei coefficienti rappresentata in Figura 6:

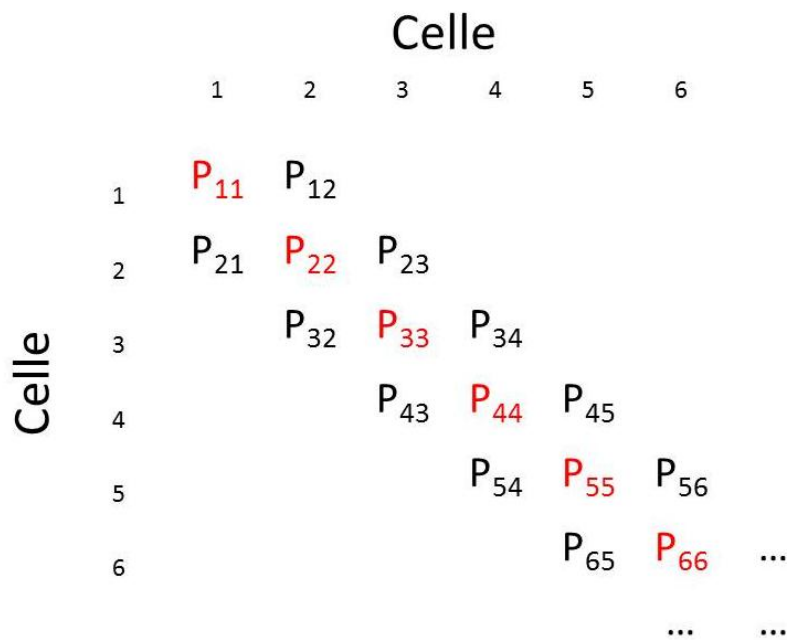


Figura 6: Matrice dei coefficienti in una griglia monodimensionale

Si può notare che la struttura della matrice è tridiagonale, e ciò scaturisce direttamente dal *pattern* di comunicazione tra le celle. Quindi, considerando il KPP, la forma della rete di reattori ha una notevole influenza sulla forma del sistema risolutivo, e di conseguenza sui metodi numerici che si usano per

risolverlo, dato che per aumentare l'efficienza conviene implementare metodi *ad hoc* che tengano conto della struttura o della sparsità della matrice.

Come è logico, aumentando la complicazione della rete di reattori, cresce il numero degli elementi non nulli e cambia la forma della matrice. Se in una griglia monodimensionale una cella può comunicare solo con le due celle adiacenti, dando adito a tre diagonali, in due dimensioni le celle attraverso cui si può trasmettere una variabile di trasporto sono quattro, intorno ad una centrale, come si vede in Figura 7:

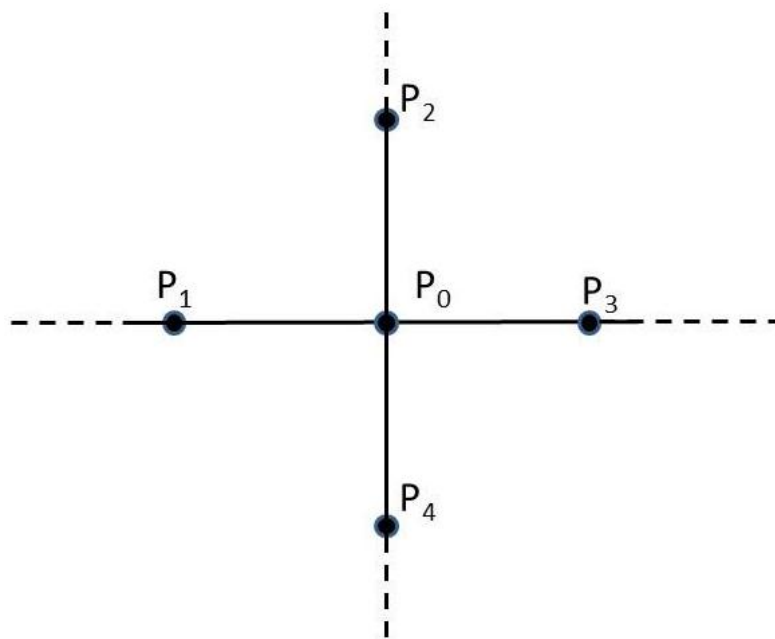


Figura 7: Griglia bidimensionale

Dalla configurazione bidimensionale si avrebbe una matrice dei coefficienti pentadiagonale, quindi con un notevole incremento dell'informazione ed un maggiore peso computazionale. Analogamente, ad una griglia tridimensionale corrisponde una matrice eptadiagonale.

### **1.3. Analisi CFD e schemi cinetici dettagliati**

In seguito sia alla spinta politica di numerose istituzioni, quali l'Unione Europea, che allo sviluppo di nuove tecnologie in ambito industriale, il controllo delle emissioni inquinanti è ormai una prerogativa nella progettazione di processi o prodotti che possono dare adito ad immissione nell'atmosfera di sostanze

indesiderate. Nel caso di grandi processi integrati, come le centrali di potenza, si può ricorrere a sistemi di cattura od abbattimento, ma quando si tratta di veicoli non è possibile implementare tali soluzioni a causa della scala ridotta e del costo che esse comportano.

A questo proposito è fondamentale conoscere nel dettaglio il processo che porta dal combustibile ai gas di scarico, perché ciò consente di progettare la combustione nel modo ottimale, controllando così la formazione degli inquinanti o inibendola con opportuni accorgimenti.

La fluidodinamica computazionale (*Computational Fluid Dynamics*, CFD) è un ottimo strumento per la progettazione di un processo di combustione. In primo luogo essa consente di condurre analisi multiple con parametri differenti in modo relativamente economico, consentendo di realizzare nuovi sistemi e migliorare quelli esistenti in un tempo ridotto e con costi operativi più bassi rispetto ad una procedura sperimentale. Inoltre, l'analisi CFD consente di acquisire una conoscenza più approfondita di come il sistema che si sta studiando lavori, favorendone in questo modo una progettazione più efficace. La modellazione numerica delle emissioni inquinanti tramite CFD è essenzialmente lo studio di un sistema di fluidi che si modifica nel tempo e reagisce. Le caratteristiche fisiche di fluidi in movimento possono essere descritte tramite equazioni matematiche fondamentali, generalmente nella forma di equazioni differenziali alle derivate parziali che vengono discretizzate per poter essere risolte. Accanto a ciò è necessario utilizzare un adeguato schema cinetico che traduca in termini matematici la chimica del sistema, ovvero la parte reattiva.

Per risolvere la fisica del flusso all'interno del dominio, l'approccio CFD ne prevede la suddivisione in una serie di sottodomini non sovrapposti tra loro. In questo modo si crea una griglia, o *mesh*, di celle che costituiscono a loro volta dei volumi di controllo nei quali si calcolano i valori discreti delle proprietà del flusso, quali la velocità, la pressione, la temperatura, la concentrazione delle specie e altri parametri di interesse. Naturalmente, maggiore è il numero dei sottodomini e più accurata risulta la descrizione fluidodinamica dell'intero volume di controllo. Quindi, all'interno di ogni cella saranno risolte le equazioni

fondamentali della fluidodinamica e le equazioni chimiche che ne costituiscono la caratteristica reattiva.

Si consideri a titolo d'esempio lo schema cinetico del meccanismo completo per pirolisi, ossidazione parziale e combustione di carburante formato da molecole aventi da uno a sedici atomi di carbonio. Il meccanismo è formato da 396 specie interessate da 11933 reazioni. Considerando una *mesh* sufficientemente accurata, formata da  $10^5 \div 10^6$  celle, si osserva facilmente che il sistema costituente il bilancio di massa globale, da risolvere per calcolare la concentrazione delle varie specie, assume dimensioni dell'ordine di  $10^7 \div 10^8$  equazioni in altrettante incognite. Un'ulteriore complicazione deriva dal fatto che il contributo reattivo al bilancio è generalmente non lineare, e la forma del sistema considerato è *stiff*.

Nella fattispecie, l'analisi delle emissioni inquinanti si presta a questo tipo di difficoltà, perché accanto a prodotti di combustione presenti in termini di percentuali, come  $\text{CO}_2$  o  $\text{H}_2\text{O}$ , compaiono specie presenti in quantità di poche parti per milione, come gli ossidi di azoto ( $\text{NO}_x$ ) e i radicali.

Come si è accennato, l'applicazione di schemi cinetici dettagliati a normali analisi CFD risulta ancora eccessivamente pesante dal punto di vista computazionale, nonostante i recenti e continui sviluppi nella potenza di calcolo dei *computer*. Infatti, un sistema delle dimensioni sopracitate può impiegare diversi giorni per essere risolto. L'obiettivo del presente lavoro di Tesi è quello di proporre una metodologia risolutiva per analisi di fluidodinamica computazionale coinvolgenti schemi cinetici dettagliati che riduca significativamente il tempo di calcolo richiesto dal computo del risultato. Per ottenere ciò si sfruttano le potenzialità del calcolo parallelo su *cluster* e sistemi *multi-core* ed un uso opportuno di algoritmi risolutivi in sequenza. Il risultato ottenuto è un'analisi precisa ed accurata delle emissioni inquinanti in seguito alla combustione di un carburante.

#### **1.4. Analisi delle emissioni inquinanti: gli $\text{NO}_x$**

Si consideri il caso di una fiamma turbolenta non premiscelata, come nel caso dei bruciatori industriali, o di motori *turbo-jet*. Tali fiamme di tipo diffusivo

producono maggiori quantità di inquinanti rispetto alle premiscelate, con particolare riferimento a ossidi di azoto e particolato (*soot*). Nella progettazione di un'apparecchiatura o di un processo di combustione non si può ormai più prescindere dalla valutazione delle emissioni inquinanti, sia perché in molti casi le normative vigenti lo impongono, sia perché si sono sviluppati schemi cinetici sufficientemente dettagliati da cogliere i sottoprodotti della combustione, anche se prodotti in quantità di diversi ordini di grandezza inferiori rispetto a quelli principali. Tuttavia, come si è accennato, questo livello di dettaglio, associato ad un'analisi CFD con griglia molto fitta, comporta un carico computazionale eccessivo anche per i moderni calcolatori. Questo è a maggior ragione vero nella progettazione di apparecchiature per la combustione, nelle quali generalmente i flussi sono turbolenti, determinando una stretta interazione tra chimica del sistema e miscelazione. L'applicazione di una chimica dettagliata a fenomeni turbolenti risulta eccessivamente complessa, ma sono possibili approcci alternativi per superare questo ostacolo.

Se si osserva che le reazioni di produzione di componenti inquinanti, come ad esempio gli ossidi di azoto ( $\text{NO}_x$ ), hanno generalmente tempi caratteristici superiori rispetto alle altre reazioni, e che inoltre hanno scarsa influenza sui campi di temperatura e di flusso, allora si può affrontare il problema utilizzando una procedura di post-processo. Si esegue prima un'analisi fluidodinamica sulla griglia completa, ma con uno schema cinetico semplificato, per ottenere risultati di primo tentativo sulle specie principali più diffuse. Le specie inquinanti, come gli ossidi di azoto, che non vengono considerate in questa analisi preliminare hanno scarsa influenza sui campi di temperatura e di velocità, che quindi vengono registrati durante l'analisi CFD. Avendo così ottenuto una serie di valori di primo tentativo, e tenendo costanti i campi di temperatura e fluidodinamici, si post-processano i risultati dell'analisi CFD utilizzando lo schema cinetico completo, dal quale si computano anche i valori per le specie di interesse, quali gli  $\text{NO}_x$ .

Questo approccio consente di introdurre due semplificazioni principali: in primo luogo di considerare la griglia della CFD come una rete di reattori, in cui la fluidodinamica è data dalla comunicazione tra celle adiacenti, in ciascuna delle quali valgono le condizioni microcinetiche: non ci sono gradienti di

concentrazione e temperatura. Grazie a queste considerazioni le celle simili tra di loro possono essere raggruppate (*lumping*) laddove il dettaglio della descrizione non è fondamentale, ovvero in celle non critiche, riducendo in tal modo le dimensioni totali della griglia. Il secondo aspetto consiste nell'imposizione di un campo di temperatura costante, come risultato dell'analisi CFD. Il sistema considerato perde così buona parte della sua non linearità dal momento che le costanti di reazione, dipendenti dall'Arrhenius modificata, come mostrato nell'equazione (2.1), risultano delle costanti, evitando il ripetersi del calcolo di un esponenziale:

$$k = A \cdot T^\beta \cdot \exp\left(-\frac{E_{att}}{RT}\right) \quad (2.1)$$

Così si possono accoppiare una griglia CFD dettagliata ed uno schema cinetico complesso rimediando a buona parte dei problemi relativi al peso computazionale che un'analisi di questo tipo porterebbe con sé, in caso fosse svolta direttamente.



## Capitolo 2

### Calcolo parallelo: utilità e applicazioni

#### 2.1. L'importanza crescente del calcolo parallelo

La crescente potenza di calcolo messa a disposizione dai moderni calcolatori ha permesso, nel campo della modellazione, di fronteggiare problemi numerici dalla complessità sempre maggiore (e con risultati sempre più dettagliati e verosimili) in tempi relativamente brevi. Fino ai primi anni 2000, ciò è stato reso possibile dall'aumento del numero di *transistor*, nonché della frequenza di *clock* dei processori (Figura 8), che ha raggiunto punte di 3.5 – 4 GHz.

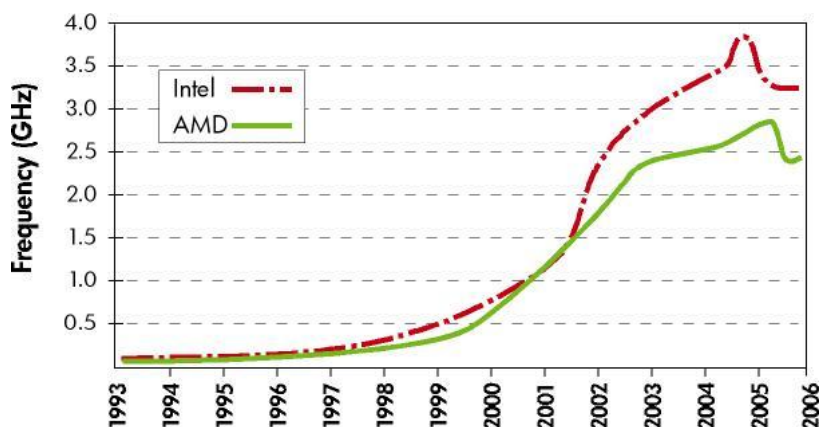


Figura 8: Andamento della frequenza di calcolo dei processori commercializzati nel periodo 1993-2006 (fonte: HP)

Tale incremento, tuttavia, ha dovuto fare i conti con un contemporaneo aumento, non lineare ma esponenziale, della potenza dissipata dai processori stessi (Figura 9); in sostanza ci si è trovati davanti ad un limite fisico che ha impedito di migliorare ulteriormente le prestazioni dei sistemi a singolo processore.

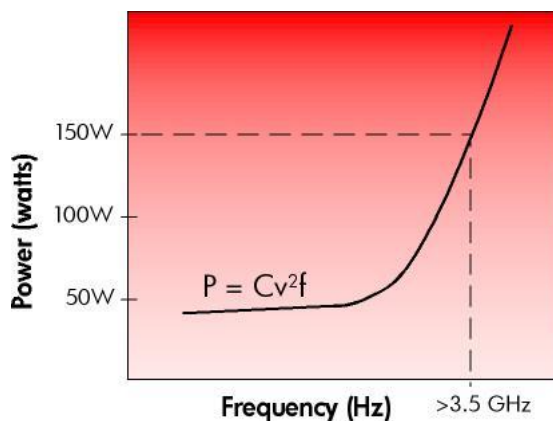


Figura 9: Potenza dissipata da un singolo processore in funzione della frequenza di *clock*

Per tale motivo, negli ultimi anni l'attenzione delle aziende produttrici di microprocessori si è concentrata sui sistemi *multi-core*, basati su un nucleo di più processori fisici che condividono la stessa memoria, aggirando così il problema della potenza dissipata descritto in precedenza: negli ultimi anni i sistemi *dual-core* e *quad-core* sono di serie anche sulle normali configurazioni desktop e laptop.

Dall'altra parte, un cambiamento così significativo *nell'hardware* ha imposto anche un'evoluzione della struttura dei *software*, da sempre pensati per essere eseguiti "sequenzialmente" su un singolo processore: per sfruttare le maggiori risorse computazionali messe a disposizione da un numero di processori più elevato il software esistente deve essere riprogettato in una forma adeguata alla struttura parallela della CPU, in modo tale da ottenere un'efficienza maggiore tramite l'esecuzione contemporanea sulle singole unità di più parti dello stesso programma.

## 2.2. Memoria condivisa e distribuita

Il progetto di un algoritmo parallelo parte dalla suddivisione delle istruzioni dell'algoritmo iniziale in più parti, dette *task*, che possono essere eseguite in parallelo sui singoli processori. Ovviamente la modalità di ripartizione delle istruzioni nei vari *task* non è univoca, così come la loro dimensione (detta anche granularità). Ciononostante quest'operazione risulta fondamentale per l'efficienza finale del software e va pertanto effettuata anche in funzione della tipologia di

macchina, in termini di memoria e processori, che si prevede di utilizzare. In particolare, l'organizzazione della memoria all'interno del calcolatore gioca un ruolo fondamentale per la parallelizzazione.

In un sistema a memoria condivisa un'unica memoria raccoglie i dati di un'applicazione, a cui possono accedere tutti i processori; in questa situazione non è necessaria alcuna comunicazione tra le varie unità computazionali, in quanto il trasferimento di variabili avviene tramite un semplice accesso alla memoria condivisa. Esempio di questi sistemi sono i processori commercialmente disponibili *Intel® Core™ i7* di nuova generazione, *dual-core* o *quad-core*.

Invece, in un sistema a memoria distribuita ciascun processore possiede una memoria privata cui ha accesso esclusivo; lo scambio di dati avviene tramite operazioni di comunicazione, più dispendiose dal punto di vista temporale rispetto a semplici accessi di memoria. D'altra parte gli attuali limiti tecnologici sulla massima dimensione della memoria del sistema (difficilmente si trovano configurazioni desktop con RAM che supera i 16 GB) rendono questi sistemi indispensabili in problemi che richiedono un utilizzo notevole di risorse. Ciò accade, per esempio, per i modelli meteorologici o per i codici di fluidodinamica computazionale (per esempio *Fluent*).

La distinzione tra memoria distribuita e condivisa va di pari passo con quella tra processi e *thread*: generalmente ad ogni unità di memoria (alla quale possono afferire uno o più processori) è associato un singolo processo, ovvero un'istanza del programma in esecuzione in maniera sequenziale. Ad esso è associato uno spazio di memoria, per consentire l'esecuzione del programma, e un controllo di flusso, che consente di sapere quale parte del programma è in esecuzione sotto il processo stesso in ogni istante. All'interno del singolo processo possono a loro volta essere eseguiti contemporaneamente più *thread*, che condividono variabili e dati sulla stessa memoria.

Il vantaggio dell'esecuzione parallela di più *thread* risulta evidente nei sistemi a memoria condivisa: ciascun processore, indipendentemente dagli altri, può gestire un singolo *thread*, aumentando la velocità di esecuzione dell'intero processo al massimo di un fattore pari al numero di processori, dal momento che i

costi di comunicazione sono praticamente nulli. In Figura 10 è schematizzata la differenza tra processi e *thread*:

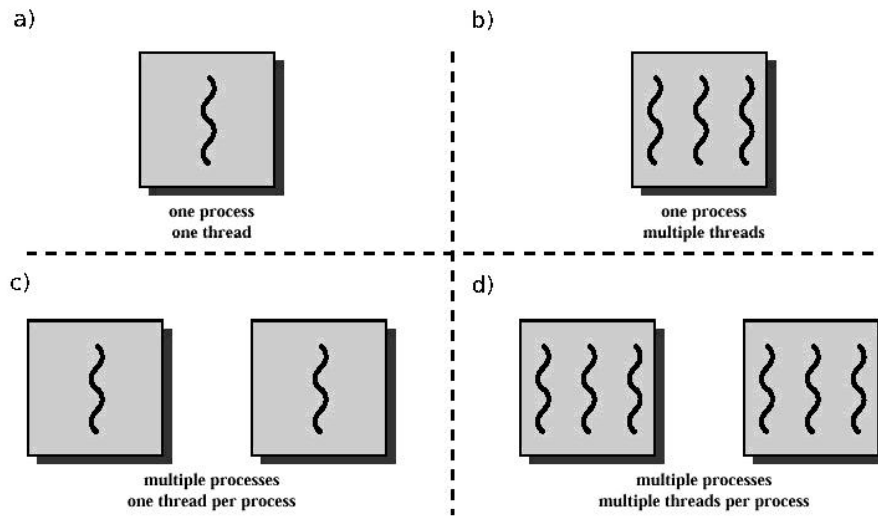


Figura 10: Differenza tra processi e *thread*

I quattro sistemi schematizzati nella figura sono rappresentativi delle possibili configurazioni *hardware* presenti negli odierni calcolatori:

- a) Sistema a singolo processore; esecuzione del programma in maniera puramente sequenziale;
- b) Sistema a memoria condivisa; i processori operano sulla stessa memoria e a ciascuno di essi è associato uno (o più) *thread*;
- c) Sistema a memoria distribuita; ad ogni processore corrisponde una differente memoria, per cui su ciascuno di essi è avviato un processo che scambia dati con gli altri tramite comunicazione;
- d) Sistema ibrido; più processori condividono differenti memorie, e su ciascuna di esse sono avviati differenti *thread*.

Come sarà spiegato nel seguito, la parallelizzazione dei *thread* all'interno del singolo processo è gestita tramite l'interfaccia OpenMP. Invece la creazione e gestione di più processi in parallelo in sistemi a memoria distribuita avviene mediante lo standard MPI, implementato tramite differenti librerie come *OpenMPI* o *mpich2*.

### **2.3. Memoria condivisa e OpenMP**

Nel caso di calcolatori a memoria condivisa il modello di parallelizzazione più opportuno è quello basato sulla gestione di più *thread*, aventi accesso allo stesso spazio condiviso all'interno dello stesso processo. OpenMP (*Open Multi-Processing*) è uno standard creato nel 1997 che permette un approccio agevole ed efficiente nella creazione di programmi paralleli in sistemi a memoria condivisa, oltre che portabile, in quanto applicabile ad un elevato numero di architetture SMP (*Symmetric Multi-Processing*).

OpenMP non è un nuovo linguaggio di programmazione: il suo principio di funzionamento è basato su una serie di istruzioni che possono essere aggiunte a un programma sequenziale scritto in Fortran, C o C++ per gestire la distribuzione del lavoro tra i vari *thread*, nonché per regolare l'accesso alle variabili condivise. Il programma originale viene pertanto modificato spesso in minima parte, in quanto la gestione dei *thread* paralleli nel dettaglio spetta totalmente al compilatore, senza ulteriori sforzi per l'utente.

Ad ogni modo, spetta al programmatore riconoscere quelle parti di programma che possono essere eseguite contemporaneamente dai differenti *thread*. A volte questo può richiedere il ripensamento dell'algoritmo iniziale in una forma che possa sfruttare maggiormente la possibilità di gestire il problema in maniera parallela su più *thread*.

I programmi *multi-thread* possono essere scritti in una molteplicità di modi differenti, alcuni dei quali permettono anche complesse interazioni tra i *thread* stessi. Il vantaggio offerto da OpenMP consiste proprio nel facilitare la programmazione, evitando a monte una serie di potenziali errori, fornendo un approccio strutturato alla programmazione parallela: lo standard è infatti basato sul modello *fork-join* (Figura 11).

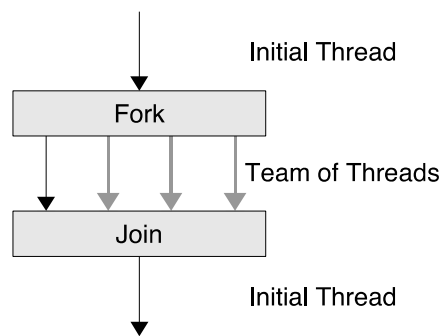


Figura 11: Il modello *fork-join* supportato da OpenMP

Secondo quest'approccio, il programma avvia inizialmente un singolo *thread* (*Initial thread*). Una volta che esso incontra un'istruzione OpenMP, apre una serie di nuovi *thread* (*fork*), di cui diventa il *master*, con i quali contribuisce all'esecuzione delle istruzioni del programma. Completata questa fase, il *master* continua l'esecuzione del programma, mentre gli altri *thread* si chiudono (*join*).

Come già detto, all'utente spetta la scelta di un algoritmo che possa sfruttare per quanto possibile il grado di parallelismo del problema; il modello descritto sopra viene invece applicato automaticamente dal compilatore.

Di seguito è presentato un algoritmo esemplificativo, in cui viene eseguito il prodotto scalare tra due vettori, con l'applicazione della struttura OpenMP:

```

int main(int argc, char *argv[])
{
//Inizializzo le variabili
double sum;
double a[256], b[256];
int status;
int n=256;

//Riempio i due vettori
for (i = 0; i < n; i++)
{
    a[i] = i * 0.5;
    b[i] = i * 2.0;
}

sum = 0;

//Aggiunta dell'istruzione per l'esecuzione parallela
#pragma omp for reduction(+:sum)

```

```
//Esecuzione del prodotto scalare
for (i = 1; i <= n; i++ )
{
    sum = sum + a[i]*b[i];
}

//Stampa a video dei risultati
printf ("sum = %f \n", sum);
}
```

#### **2.4. Memoria distribuita: il protocollo MPI**

Il protocollo MPI (*Message Passing Interface*) è un sistema standardizzato di comunicazione ideato da un gruppo di ricercatori provenienti da ambito accademico e industriale per funzionare su un'ampia gamma di calcolatori paralleli. Esso definisce la sintassi e il significato di una serie di *routine* basilari per la scrittura di programmi paralleli in *Fortran*, C o C++.

Il processo di standardizzazione, avviato nel 1992 e finalizzato nel 1994, fu essenzialmente guidato da tre parole chiave: portabilità, efficienza e funzionalità; in poche parole, l'obiettivo ultimo è stato la creazione di un'infrastruttura che permettesse all'utente di realizzare programmi in grado di essere eseguiti in qualsiasi sistema a memoria distribuita, a prescindere dalla sua architettura (i.e. portabili) in maniera ottimizzata (i.e. efficiente) per l'*hardware* sul quale il programma stesso è eseguito. Tutto ciò garantendo sincronizzazione e comunicazione (i.e. funzionalità) tra i vari processi indipendentemente dal linguaggio adottato (per quanto la sintassi sia specifica per i vari linguaggi di programmazione).

Come già accennato, il modello *message-passing* si fonda sull'ipotesi di base che i processi operanti in parallelo abbiano indirizzi di memoria separati. Pertanto la comunicazione avviene quando una porzione di memoria relativa a un processo viene copiata sulla memoria di un altro; quest'operazione avviene quando il primo processo esegue un'operazione di *send*, mentre il secondo ne esegue una di *receive*.

La comunicazione di una porzione di dati richiede un *set* minimo di informazioni che garantiscano che il determinato messaggio arrivi correttamente a destinazione. Il processo che invia i dati deve infatti specificare quale parte di

memoria copiare, definendo un indirizzo di partenza e la lunghezza del messaggio, oltre al processo a cui inviare i dati. Ad ogni messaggio viene inoltre assegnata un'identità (nel seguito *tag*), in modo tale che questo possa essere "riconosciuto" dal processo che riceve. Da parte sua, il destinatario deve indicare l'identità del processo che invia il messaggio (nel seguito *source*), il *tag* e la dimensione totale del messaggio stesso. Per cui le operazioni elementari di comunicazione di un messaggio saranno rispettivamente:

```
send(send_address, length, destination, tag)
```

per il mittente, e:

```
receive(recv_address, length, source, tag, actlen)
```

per il destinatario (*actlen* è la dimensione effettiva del messaggio).

Il trasferimento di un messaggio in MPI viene eseguito in tre passaggi:

1. I dati da inviare sono copiati dal *buffer* specificato nella riga di *send* (*send\_address*) in un *buffer* di sistema proprio del protocollo MPI; ad essi è associata una "etichetta" di riconoscimento contenente informazioni sul processo che invia, quello che riceve, il *tag* e il comunicatore utilizzato;
2. Il messaggio è inviato dal *sender* al *receiver*.
3. I dati del messaggio inviato sono copiati dal buffer di sistema a quello specificato nella riga di *receive* (*recv\_address*).

Nello *standard* MPI tutte le operazioni di scambio dati, identificazione dei messaggi, dei tipi di dati scambiati e dei processi sono gestite da un oggetto detto comunicatore (*MPI::COMM\_WORLD*), ovvero un dominio di comunicazione costituito da un *set* di processi che si scambiano messaggi. Questo, reso disponibile una volta che l'interfaccia MPI è stata inizializzata, riceve come *input* dall'utente il numero di processi e associa a ciascuno di essi un *rank*, ovvero li identifica tramite un intero che va da 0 a  $n-1$ , dove  $n$  è il numero di processi assegnato.

Di seguito è mostrata una semplice applicazione dell'interfaccia MPI, scritta in linguaggio C++, in cui ogni processore stampa su periferica video l'espressione "Hello, World!".

```

#include <iostream>
#include "mpi.h"
using namespace std;

int main(int argc, char *argv[])
{
int rank, size, tag, i;
char message[20];

MPI::Status status; //Oggetto contenente informazioni sul
                    //messaggio trasmesso (rank, tag ed
                    //eventuali errori)

MPI::Init(argc, argv); //Inizializzo MPI

size = MPI::COMM_WORLD.Get_size(); //Ricezione del numero di
//processi
rank = MPI::COMM_WORLD.Get_rank(); //Assegnazione di un rank a
//ciascun processo

tag = 7; //Intero che identifica il messaggio scambiato

if (rank==0)
{
    strcpy(message, "Hello, World");
    for (int i=1; i < size; i++)
        MPI::COMM_WORLD.Send(&message, 13, MPI::CHAR, i, tag);
}

else
MPI::COMM_WORLD.Recv(&message, 13, MPI::CHAR, 0, tag, status);
cout << "node " << rank << ": " << message << endl;
MPI::Finalize(); //Chiusura di MPI
}

```

L'esecuzione di un programma in MPI avviene da terminale, digitando l'istruzione:

```
mpiexec -n 4 nomeprogramma
```

dove il numero che segue  $n$  indica il numero di processi impiegati. Nel caso precedente l'esecuzione fornirebbe un output del tipo:

```

node 1: Hello, World!
node 2: Hello, World!
node 3: Hello, World!
node 4: Hello, World!

```

## **2.5. Prestazioni di un algoritmo parallelo**

L'obiettivo fondamentale del calcolo parallelo è la risoluzione di problemi di grandi dimensioni in tempi ragionevolmente brevi. All'ottenimento di ciò

concorrono una serie di fattori, dipendenti innanzitutto dal tipo di *hardware* e *software* impiegato, dal grado di parallelizzabilità del problema nonché dal modello adottato per la parallelizzazione dell'algoritmo originale. Si tratta, in sostanza, di un sistema con un numero elevato di variabili, spesso dipendenti tra di loro, pertanto difficili da considerare singolarmente.

Per agevolare l'analisi delle prestazioni fornite da un algoritmo parallelo sono introdotti dei concetti basilari che da una parte confrontano l'algoritmo ottenuto con quello originale, verificandone (e quantificandone) l'effettivo aumento di prestazioni ottenuto, dall'altra analizzano il comportamento dell'algoritmo al variare del numero di processi e/o *thread* impiegati.

Al riguardo è stata sviluppata anche una notevole letteratura con l'obiettivo di prevedere i miglioramenti massimi attesi dalla parallelizzazione di un algoritmo sequenziale: come sarà illustrato nel seguito, le leggi di Amdahl e Gustafson vanno esattamente in questa direzione.

## **2.6. Speedup ed efficienza**

Il concetto di *speedup* è finalizzato al confronto tra l'algoritmo originale e quello scritto in forma parallela, eventualmente riscritto in una forma che possa sfruttare maggiormente il grado di parallelismo del problema. Quantitativamente si calcola come:

$$S_p(n) = \frac{t_s(n)}{t_p(n)} \quad (3.1)$$

in cui, per un numero di processori  $p$  definito e una dimensione del problema fissata pari a  $n$ ,  $t_s(n)$  è il tempo di esecuzione del miglior algoritmo sequenziale che risolve il problema, mentre  $t_p(n)$  è il tempo di esecuzione in parallelo.

Come risulta intuibile, teoricamente il massimo *speedup* ottenibile non può essere maggiore del numero di processori  $p$  utilizzati. Se così fosse, verrebbe meno l'ipotesi iniziale di "miglior algoritmo sequenziale" e il problema potrebbe essere riscritto con un nuovo algoritmo sequenziale più efficiente. Per ovviare a questa eventualità si può utilizzare nel calcolo dello *speedup* come algoritmo

sequenziale di riferimento una versione sequenziale dell'implementazione parallela.

In realtà può comunque capitare che pur utilizzando questo accorgimento lo *speedup* osservato risulti superlineare (ovvero  $S_p(n) > p$ ). Una possibile causa di questo comportamento è individuabile nel cosiddetto “effetto *cache*”, legato ai tempi di accesso alla memoria da parte dei vari processori: nei moderni calcolatori paralleli, all'aumentare del numero dei processori aumenta anche la quantità totale di dati accumulati nelle *cache* dei vari processori; per cui i tempi di accesso alla memoria si riducono drasticamente, al punto che talvolta è possibile osservare uno *speedup* superlineare.

Accanto al concetto di *speedup* è in uso anche quello, praticamente analogo, di efficienza. Essa è un indice di quanto efficacemente siano utilizzati i  $p$  processori per eseguire l'algoritmo, al netto quindi dei tempi di comunicazione e sincronizzazione tra i differenti processi. Matematicamente essa è definita come:

$$E_p(n) = \frac{t_s(n)}{p \cdot t_p(n)} = \frac{S_p(n)}{p} \quad (3.2)$$

Ovviamente a uno *speedup* lineare corrisponde un'efficienza unitaria.

## 2.7. Legge di Amdahl

Come già affermato in precedenza, ogni problema possiede un grado di parallelismo intrinseco che non è possibile modificare. Ciò equivale a dire che esiste, piccola o grande che sia, una parte di codice che deve essere eseguita sequenzialmente.

La legge di Amdahl razionalizza proprio questo concetto e fornisce un'espressione quantitativa del massimo *speedup* ottenibile in funzione della parte da eseguire sequenzialmente e del numero di processori. Definita infatti  $f$  la frazione di codice non parallelizzabile, e  $p$  il numero di processori, il tempo totale di esecuzione del programma è dato dalla somma della frazione di tempo impiegato per eseguire la parte sequenziale, pari a  $f \cdot t_s(n)$ , e della frazione di

tempo necessaria per eseguire la parte parallela, pari a  $(1-f)/p \cdot t_s(n)$ , nell'ipotesi che questa porzione di codice sia parallelizzata con efficienza unitaria. Per cui, eseguite le dovute semplificazioni, lo *speedup* massimo effettivamente raggiungibile diventa:

$$S_p(n) = \frac{1}{f + \frac{1-f}{p}} \quad (3.3)$$

tendente a  $1/f$  al tendere a infinito del numero di processori. Si ottiene quindi l'andamento mostrato in Figura 12:

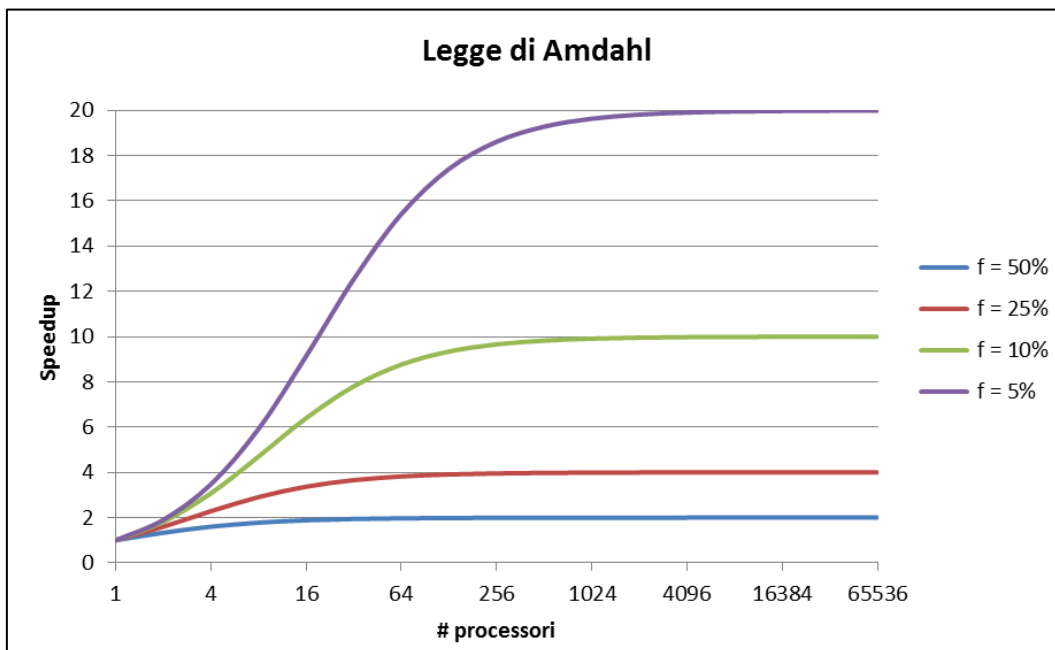


Figura 12: *Speedup* previsto dalla legge di Amdahl in funzione della frazione di codice non parallelizzabile

La parte non parallelizzabile diventa quindi determinante oltre un certo numero di processori; essa va quindi, per quanto possibile, minimizzata all'interno del codice, onde evitare che la parallelizzazione della restante porzione di programma risulti tanto laboriosa quanto inutile.

## 2.8. Scalabilità e legge di Gustafson

L'analisi del comportamento di un programma parallelo al variare del numero di processori impiegati va sotto il nome di analisi di scalabilità. In particolare, un

algoritmo parallelo si definisce scalabile se un incremento delle risorse *hardware* (ovvero, del numero di processori) dà luogo ad un aumento proporzionale dell'efficienza dell'algoritmo, ovvero del suo *speedup*.

Generalmente si osserva che, fissate le dimensioni del problema, un aumento del numero di processori oltre un certo limite non apporta più benefici significativi, e lo *speedup* raggiunge un valore asintotico. Aumentando tuttavia le dimensioni del problema si osserva un incremento dello *speedup* massimo ottenibile, col valore asintotico che si sposta verso un numero più elevato di processori. Variando quindi di pari passo il numero di processori e la dimensione del problema, l'efficienza dell'algoritmo può essere mantenuta costante e in linea teorica problemi di dimensione elevata possono essere risolti nello stesso tempo di problemi più contenuti, posto di utilizzare un numero più alto di processori.

Come nel caso della legge di Amdahl, dal comportamento appena descritto è stata formulata una legge di validità generale (sotto determinate ipotesi), che va sotto il nome di legge di Gustafson. Essa considera la scalabilità di un sistema avente una parte sequenziale il cui tempo di esecuzione  $\tau_s$  è indipendente dalle dimensioni del problema, e una restante parte perfettamente parallelizzabile, che viene eseguita in un tempo  $\tau_p$  inversamente proporzionale al numero di processori. Il relativo *speedup* è allora pari a:

$$S_p(n) = \frac{\tau_s + \tau_p(n,1)}{\tau_s + \tau_p(n,p)} \quad (3.4)$$

Con l'ipotesi di perfetta parallelizzabilità della seconda parte, il suo tempo di esecuzione sarà pari a  $\tau_p(n,p) = (t_s(n) - \tau_s) / p$ , con  $t_s(n)$  tempo totale di esecuzione dell'intero algoritmo sequenziale. L'espressione di cui sopra si riduce infine a:

$$S_p(n) = \frac{\frac{\tau_s}{t_s(n) - \tau_s} + 1}{\frac{\tau_s}{t_s(n) - \tau_s} + \frac{1}{p}} \quad (3.5)$$

Se, come è logico attendersi,  $t_s(n)$  aumenta monotonamente all'aumentare delle dimensioni del problema, tale rapporto tende a  $p$  al tendere di  $n$  a infinito.

A prima vista, l'andamento previsto dalla legge di Gustafson sembrerebbe contraddittorio rispetto al modello elaborato da Amdahl. Tra le due leggi, tuttavia, c'è un'importante differenza di fondo: Gustafson assume infatti che la frazione di tempo impiegata per eseguire la parte sequenziale non sia più costante al variare delle dimensioni del problema, ma diminuisca all'aumentare di queste, in quanto il suo tempo di esecuzione  $\tau_s$  viene posto costante e indipendente da  $n$ . Ciò risulta per esempio vero, o quantomeno verosimile, in un prodotto tra matrici: trascurando i costi di comunicazione, il numero di operazioni (*flop*) che viene parallelizzato in questo caso è proporzionale a  $n^3$ , mentre le operazioni non parallelizzabili, come l'allocazione di memoria iniziale e il riempimento delle due matrici, richiedono uno sforzo computazionale che cresce molto più lentamente all'aumentare delle dimensioni del problema, e che quindi diventa sempre meno importante all'aumentare di  $n$ .

Il *background* teorico appena descritto inquadra perfettamente il problema in esame nel presente lavoro di Tesi, ovvero la parallelizzazione del preesistente post-processore cinetico: anche in questo caso sarà presente una parte di codice non parallelizzabile, e sarà necessaria una scrittura alternativa dell'algoritmo in una forma che possa distribuire più agevolmente i calcoli su più processi e/o *thread*. Inoltre c'è da attendersi che traslando il problema su una scala più realistica, come previsto da Gustafson, la rilevanza della parte sequenziale debba diminuire.

Nei capitoli che seguono viene illustrata la procedura seguita per raggiungere l'obiettivo proposto.

## Capitolo 3

### Metodi numerici

#### *3.1. Potenzialità e limiti dello schema risolutivo originale*

Come descritto nel Capitolo 1, l'algoritmo risolutivo per un sistema non lineare di grandi dimensioni utilizzato nel KPP prevede l'implementazione di un metodo di Newton globale. La preferenza per questo metodo è dovuta soprattutto alla velocità di convergenza, che è di tipo quadratico; tuttavia per casi di dimensioni particolarmente elevate possono insorgere dei problemi. Nella fattispecie, il metodo prevede il calcolo e la fattorizzazione della matrice Jacobiana (Figura 2), che per quanto sia sparsa a blocchi e simmetrica (come struttura, ma non come valori), ha comunque un'occupazione di memoria non indifferente. Il problema infatti non è di poco conto se si considerano le dimensioni in gioco: per una griglia composta da  $10^6$  celle e uno schema cinetico con 50 specie, sapendo che un numero in doppia precisione occupa 8 byte e che in una struttura bidimensionale ciascuna cella comunica con (al massimo) altre quattro per cui in totale la riga del reattore corrispondente ha cinque blocchi non nulli, per memorizzare anche solo una matrice di coefficienti si necessita di:

$$8 \cdot 50 \times 50 \cdot 1000000 = 1000000000000 \text{ Byte} \approx 93 \text{ Gigabyte}$$

Inoltre il costo computazionale di una fattorizzazione della matrice secondo Gauss diviene proibitivo dal momento che è proporzionale a  $o(n^3/3)$ , essendo  $n$  la dimensione della matrice.

Per tali ragioni, nel presente Lavoro di Tesi è stato necessario riformulare l'algoritmo risolutivo originale, creandone una versione in grado di superare le problematiche appena descritte e considerando anche che esso sarà infine eseguito su macchine a memoria distribuita, in cui i dati sono delocalizzati sulle singole unità computazionali. Come sarà illustrato nel seguito, è stata verificata l'efficienza di algoritmi diversi, eseguiti in maniera combinata e non, nello schema risolutivo. In questa fase l'algoritmo, per quanto sia stato rivisto in

un'ottica parallela, è stato comunque mantenuto in forma seriale, dal momento che senza un'architettura di base solida qualunque processo di parallelizzazione risulta inutile.

### 3.2. Dal sistema globale alla soluzione locale

Il problema in questione è stato riformulato in modo opportuno considerando i bilanci di massa per ciascuna cella che compone il sistema discretizzato, per ipotesi considerata un reattore perfettamente miscelato.

La scrittura generalizzata di un qualsiasi bilancio (di materia, energia o quantità di moto) ha la seguente forma:

$$[IN] - [OUT] + [PROD] = [ACC] \quad (4.1)$$

Quella mostrata nell'Equazione (4.1) è una forma compatta dei rispettivi principi di conservazione che rappresentano i pilastri portanti della fluidodinamica. Nel caso della conservazione della massa, il bilancio può essere espresso nel seguente modo: per un sistema aperto, costituito da un volume di controllo delimitato da una superficie di controllo, la variazione della massa di una generica specie contenuta nel suddetto sistema  $[ACC]$  è pari alla somma della massa entrante netta nel sistema stesso  $[IN] - [OUT]$  e della quantità netta  $[PROD]$  prodotta in seguito a reazioni chimiche. Per cui, per la singola  $j$ -esima specie si avrà:

$$m_{tot} \frac{d\omega_j}{dt} = \dot{m}_j^{in} - \dot{m}_j^{out} + R_j V + f_j^{in} \quad (4.2)$$

Nell'equazione (4.2) compaiono i termini di trasporto  $\dot{m}_j^{in} - \dot{m}_j^{out}$ , reazione  $R_j V$  e le condizioni di bordo  $f_j^{in}$ , mentre le incognite sono le frazioni massive  $\omega_j$ .

Riscrivendo l'equazione (4.2) ed esplicitando le portate massive della singola specie si ottiene:

$$m_{tot} \frac{d\omega_j}{dt} = \sum_{i=1}^{N_{in}} c_i^{in} \omega_j^{in(i)} - \sum_{i=1}^{N_{in}} c_i^{out} \omega_j - \sum_{i=1}^{N_{in}+N_{out}} d_i (\omega_j - \omega_j^{diff(i)}) + R_j V + f_j^{in} \quad (4.3)$$

Sviluppando le parentesi e raccogliendo in modo opportuno si ottiene poi:

$$m_{tot} \frac{d\omega_j}{dt} = -\omega_j (c_{tot}^{out} + d_{tot}) + \sum_{i=1}^{N_{in}} c_i^{in} \omega_j^{in(i)} + \sum_{i=1}^{N_{in}+N_{out}} d_i \omega_j^{diff(i)} + R_j V + f_j^{in} \quad (4.4)$$

Quindi si hanno tante equazioni quante sono le specie chimiche ( $NS$ ) coinvolte, della forma:

$$m_{tot} \frac{d\omega_j}{dt} = -M \omega_j + \sum_{i=1}^{N_{in}} c_i^{in} \omega_j^{in(i)} + \sum_{i=1}^{N_{in}+N_{out}} d_i \omega_j^{diff(i)} + R_j V + f_j^{in} \quad (4.5)$$

Nell'equazione (4.5) si possono mettere in evidenza i termini di convezione e diffusione, quelli di reazione e le condizioni di ingresso:

$$m_{tot} \frac{d\omega_j}{dt} = - \left[ M \omega_j + \underbrace{\sum_{i=1}^{N_{in}} c_i^{in} \omega_j^{in(i)} + \sum_{i=1}^{N_{in}+N_{out}} d_i \omega_j^{diff(i)}}_{\text{convezione+diffusione}} \right] + \underbrace{R_j V}_{\text{reazione}} + \underbrace{f_j^{in}}_{\text{ingressi}} \quad (4.6)$$

Le  $NS$  equazioni differenziali così formulate per la singola cella di calcolo (o reattore) devono essere scritte per tutti i reattori che compongono la griglia, originando un sistema globale di equazioni differenziali ordinarie di  $NS \times NR$  equazioni in altrettante incognite.

Introducendo la matrice  $C$  dei termini convettivi e diffusivi, la matrice  $R(\bar{\omega})$  dei contributi reattivi e scrivendo in forma vettoriale anche il termine degli ingressi  $\bar{f}$  e quello delle incognite  $\bar{\omega}$  si ottiene la forma compatta per il sistema globale:

$$\bar{m}_{tot} I \frac{d\bar{\omega}}{dt} = -C \bar{\omega} + R(\bar{\omega}) + \bar{f} \quad (4.7)$$

Il sistema riscritto nella forma dell'equazione (4.7) evidenzia alcune proprietà che possono essere sfruttate per cambiare la logica di soluzione. Si verifica infatti che il termine di reazione è disaccoppiato tra le equazioni relative a reattori diversi, mentre all'interno di uno stesso reattore i termini relativi a convezione e diffusione sono disaccoppiabili tra le equazioni. Queste considerazioni confermano la struttura diagonale a blocchi che si era descritta nel Capitolo 1, e suggeriscono la possibilità di risolvere più sistemi localmente sui singoli reattori. Un primo approccio per consentire la risoluzione di un sistema di dimensioni

molto elevate senza appesantire eccessivamente l'archiviazione in memoria, come si è visto accadere per la matrice Jacobiana del metodo di Newton globale, consta nella risoluzione sequenziale delle singole celle di calcolo.

### 3.3. Risoluzione della sequenza di CSTR

Vista la proprietà espressa alla fine del Paragrafo 3.2, è possibile scrivere tanti sistemi ODE quanti sono i reattori della griglia. Ciascun sistema, di dimensioni  $NS \times NS$ , viene risolto indipendentemente dagli altri ed assume la forma mostrata nell'equazione (4.8):

$$\begin{cases} m \frac{d\bar{\omega}}{dt} = [-C\bar{\omega} + \bar{f}]_{old} + R(\bar{\omega}) \\ \bar{\omega}(t=0) = \bar{\omega}_{old} \end{cases} \quad (4.8)$$

La procedura è iterativa e ad ogni iterazione si utilizza il valore del contributo convettivo e diffusivo trovato nella precedente. Dati i tempi caratteristici tipici dei fenomeni di combustione, si può considerare il transitorio completamente esaurito nell'ordine di tempo delle decine di secondi, per cui l'integrazione viene fatta su questo intervallo di tempo. Si possono implementare due strategie di risoluzione: nel primo caso ciascuna cella viene risolta in modo totalmente indipendente dalle altre e l'aggiornamento delle composizioni su tutta la griglia viene fatta solamente dopo aver risolto tutti gli  $NR$  sistemi ODE; nel secondo approccio, una cella aggiorna tutte le altre con cui comunica con le nuove composizioni non appena viene risolta.

Le due modalità descritte presentano vantaggi e svantaggi che vanno valutati nell'ottica più ampia dell'algoritmo risolutivo che si vuole impiegare. Il secondo approccio, infatti, presenta velocità di convergenza superiore al primo, grazie all'aggiornamento continuo delle composizioni nella griglia. Dal momento che l'obiettivo del lavoro è quello di costruire un algoritmo efficiente che sfrutti le potenzialità del calcolo parallelo, si deve prendere in considerazione il fatto che la struttura *hardware* a memoria distribuita obbligherebbe ad un continuo scambio di dati tra le celle appartenenti a processi differenti, accrescendo notevolmente la complicazione del codice e il costo *overhead* di comunicazione. Al contrario la

prima procedura, seppure apparentemente meno performante, si presta meglio ad una struttura parallela del codice, dato che sfrutta appieno la *concurrency* tra operazioni, ovvero la possibilità di operare contemporaneamente su dati differenti svolgendo operazioni simili: una condizione pertanto ottimale per implementare il parallelismo nel codice. Un ulteriore sviluppo, per sfruttare i vantaggi di entrambe le configurazioni, può essere quello di sfruttare la prima procedura per suddividere le celle di calcolo su più processi e, ricordando che non necessariamente ad un processo corrisponde un solo processore anzi, spesso i *cluster* sono formati da nodi di *multi-core*, si può innestare la seconda procedura sulle celle appartenenti allo stesso nodo utilizzando tecniche di parallelismo annidato (*nested parallelism*).

Il set di equazioni differenziali ordinarie presentato nell'equazione (4.8) viene risolto applicando il metodo di Eulero *backward*, tramite il quale i valori approssimati della soluzione sono ottenuti risolvendo una griglia di punti. Essendo un metodo implicito, per calcolare il valore dell'incognita all'iterazione successiva è necessario calcolare il valore della funzione all'iterazione corrispondente, come mostrato nell'equazione (4.9):

$$y_{n+1} = y_n + hf(x_{n+1}, y_{n+1}) \quad n = 0, 1, 2, \dots \quad (4.9)$$

La (4.9) rispecchia esattamente la scrittura del sistema ODE come è stato rappresentato nella (4.8), in cui il termine di convezione e diffusione viene preso dall'iterazione precedente, mentre il termine di reazione è valutato a quella corrente:

$$\bar{\omega}_{n+1} = \bar{\omega}_n + m^{-1} \Delta t \left[ (-C\bar{\omega}_n + \bar{f}) + R(\bar{\omega}_{n+1}) \right] \quad (4.10)$$

Dopo ogni iterazione si calcolano i residui sostituendo il valore della soluzione approssimata all'interno delle equazioni; la procedura iterativa continua fino a che la media della norma 1 dei residui è inferiore ad un valore stabilito. Si definisce perciò la grandezza:

$$F_0 = \frac{\sum_{j=1}^{N_{\text{equazioni}}} |r_j|}{N_{\text{equazioni}}} \quad (4.11)$$

in cui  $r_j$  sono i residui di ciascuna equazione, e il numero di equazioni vale  $N_{\text{equazioni}} = NS \times NR$ . Si considera raggiunta la convergenza quando si verifica la condizione:

$$F_0 < 10^{-12} \quad (4.12)$$

Una volta raggiunta la condizione (4.12) la procedura iterativa si conclude perché i residui sono sufficientemente piccoli da ritenere che la soluzione ottenuta sia esatta con sufficiente precisione.

### **3.4. Splitting**

Tra i metodi esistenti per ridurre il costo computazionale di sistemi composti da schemi cinetici molto complessi applicati a fluidi reagenti, le tecniche di *splitting* forniscono una modalità pratica e che si presta ad applicazioni in parallelo per risolvere problemi nel dominio del tempo. L'approccio è particolarmente efficace se applicato a casi stazionari (Schwer et al., 2003).

Nella tecnica di *operator-splitting* la soluzione viene suddivisa in diverse integrazioni separate. Nel caso dei fluidi reagenti si separa generalmente il contributo di trasporto da quello di reazione, che è *stiff* e non lineare. Questo consente una maggiore flessibilità nel risolvere il problema. Ad esempio il passo di integrazione può essere scelto indipendentemente nei due casi, inoltre lo *splitting* consente di sfruttare al meglio la struttura del sistema: i due contributi, trasporto e reazione possiedono caratteristiche tra loro differenti. Suddividere la matrice del sistema consente perciò di avere un sistema più strutturato per ciascuna delle due integrazioni.

Nella risoluzione tradizionale infatti, per eseguire l'integrazione si utilizzano metodi impliciti perché il sistema è *stiff*. Per ogni step, quindi, va risolto un sistema non lineare, ad esempio tramite il metodo di Newton-Raphson. Prendendo invece in considerazione soltanto la parte relativa al trasporto, il sistema da

risolvere per ogni step diventa lineare e non più *stiff*. Per di più, tenendo fisso il termine di reazione (come accade nel *Predictor-Corrector*, descritto nel Paragrafo 3.4.2) il problema diventa puramente di trasporto, e dal momento che i flussi convettivi e diffusivi di ogni specie sono indipendenti l'uno dall'altro è possibile disaccoppiare le equazioni delle varie specie, risolvendo un sistema lineare per ogni singola specie. L'ulteriore vantaggio si riscontra nel dover fattorizzare la matrice corrispondente una volta sola, se lo step di integrazione non varia. Nella seconda integrazione invece si agisce solo sul termine reattivo: i flussi in uscita dipendono unicamente dalle variabili locali, per cui ogni cella può essere risolta separatamente. Come risultato finale, se il problema originale richiedeva l'integrazione di un sistema differenziale di  $NS \times NR$  equazioni, tramite *splitting* si risolvono dapprima  $NS$  sistemi lineari di  $NR$  equazioni, e quindi  $NR$  sistemi non lineari, ciascuno composto da  $NS$  equazioni.

Ovviamente questa situazione risulta molto più leggera dal punto di vista computazionale rispetto al problema originario, per quanto nella pratica si tratti di una risoluzione approssimata del sistema che introduce inevitabilmente un errore.

### **3.4.1. Risoluzione tramite *splitting*: l'esempio di Strang**

Ricordando la struttura dell'equazione (4.7), si è già detto che, discretizzando nello spazio, le equazioni alle derivate parziali divengono un sistema ODE nelle variabili in ciascun punto della griglia. Esistono delle notevoli differenze dal punto di vista matematico tra il termine di reazione e quello di trasporto: il termine di reazione, così formulato, è indipendente dal tempo poiché il processo è separato in ciascun punto della griglia; l'operatore di trasporto al contrario può dipendere dal tempo a causa di interazioni con le condizioni di bordo che possono a loro volta essere funzioni del tempo. Oltretutto il processo di trasporto non è separato per ciascun punto della griglia, dato che la somma delle frazioni massive deve mantenersi unitaria, per ogni cella, su tutto il dominio. Ciò non si applica alla componente reattiva, poiché il principio di conservazione della massa garantisce il rispetto dei vincoli fisici sulla frazione massiva di ogni singola specie.

Per risolvere numericamente l'equazione (4.7) si discretizza il tempo in intervalli del tipo  $\Delta t \equiv (t_f - t_0)/n_t$ , essendo  $n_t + 1$  il numero totale di intervalli di tempo. Quindi, un istante di tempo si può rappresentare come:

$$t_n = t_0 + n\Delta t \quad n = 0, 1, 2, \dots, n_t \quad (4.13)$$

Utilizzando uno *splitting* tale intervallo viene suddiviso in sottointervalli generando un transitorio fittizio in cui solo uno dei termini viene integrato di volta in volta. Si consideri ad esempio lo schema di *splitting* di Strang (Ren e Pope, 2008) per fluidi reagenti, in cui ogni passo temporale viene suddiviso in due sottopassi. Per un generico *set* di ODE che descrivono la variazione di una grandezza a causa dei contributi di trasporto e reazione, si può scrivere l'equazione (4.14):

$$\frac{dr}{dt} = S(r, u(r)) + M(r, u(r), t) \quad (4.14)$$

nella quale compaiono un termine di reazione  $S$  che dipende dalla variabile  $r$  e, nel caso più generale, da opportune funzioni  $u(r)$ . Il termine di trasporto  $M$ , oltre a dipendere da  $r$  e da sue funzioni, presenta anche dipendenza dal tempo, come si è detto in precedenza. Avendo effettuato lo *splitting*, invece di muoversi dal tempo  $t_n$  al tempo  $t_{n+1}$  in ogni passo, si integra prima su metà passo fino a  $t_{n+1/2}$  l'equazione che presenta solo il termine reattivo:

$$\frac{dr^a}{dt} = S(r^a, u(r^a)) \quad (4.15)$$

Si considera come condizione iniziale quella proveniente dall'iterazione precedente. Nella seconda fase dello *splitting* si integra il termine di trasporto da  $t_{n+1/2}$  a  $t_{n+1}$ :

$$\frac{dr^b}{dt} = M(r^b, u(r^b), t) \quad (4.16)$$

nella quale la condizione iniziale corrisponde allo stato del sistema alla fine del primo passaggio. Infine, si compie un ultimo passo per aggiornare i termini reattivi integrando su metà intervallo di tempo, ma utilizzando come condizione

iniziale lo stato del sistema alla fine del secondo passaggio al tempo  $t_{n+1}$ . Il valore delle variabili del sistema calcolate in questo ultimo passaggio vanno a costituire le condizioni di partenza per il successivo passo di integrazione. Come si può osservare, l'integrazione è così svolta su tutto l'intervallo discreto e la tecnica di *splitting* consente di semplificare i termini che di volta in volta vengono integrati, dal momento che possono essere affrontati in sottopassi separati, consentendo perciò di utilizzare algoritmi risolutivi ottimizzati per la struttura e il *time-step* massimo numericamente consentito a seconda dei casi. Tuttavia si rinuncia ad una descrizione precisa del transitorio, nel quale gli effetti di trasporto e reazione non vengono mai applicati contemporaneamente, al contrario di quanto avviene nel reale sistema fisico. Se però il sistema si trova in condizioni stazionarie ed è tale stato ad interessare in ultima analisi, allora la tecnica di *splitting* può essere uno strumento efficace per scalare la risoluzione a sistemi di dimensioni notevoli.

Esistono diverse varianti per la tecnica presentata, in cui le principali diversificazioni risiedono nel numero di volte o nel modo in cui una determinata funzione viene valutata per giungere alla soluzione (Ren e Pope, 2008). Ad esempio è possibile introdurre nella soluzione un'approssimazione per estrapolazione lineare di una delle variabili secondarie calcolata all'istante di tempo  $p$ :

$$u^p = u^0 + (r^p - r^0) \nabla_r u \Big|_{r=r^0} \quad (4.17)$$

in cui  $\nabla_r u$  è il gradiente della variabile secondaria del tipo  $\partial u_i / \partial r_j$ . Come è evidente, con questa soluzione si richiede di valutare la condizione iniziale  $u^0 = u(r^0)$ , ma la valutazione di  $u^p$  risulta più immediata, a patto di conoscere il gradiente.

L'accuratezza relativa di tali metodi, il costo computazionale e la facilità di implementazione sono strettamente dipendenti dal caso che si studia. Si consideri a titolo di esempio il caso dell'ossidazione di una miscela di gas di sintesi (CO e H<sub>2</sub>) in un reattore a miscelazione perfetta non isoterma. La dinamica del sistema è rappresentata dal *set* di equazioni differenziali (4.18):

$$\begin{bmatrix} \frac{d\bar{z}}{dt} \\ \frac{dh}{dt} \end{bmatrix} = \begin{bmatrix} \bar{S}(\bar{z}, T(\bar{z}, h)) \\ 0 \end{bmatrix} + \begin{bmatrix} (\bar{z}^{in} - \bar{z})/t_{res} \\ (h^{in} - h)/t_{res} + (T_a - T(\bar{z}, h))Q/\rho(\bar{z}, T(\bar{z}, h)) \end{bmatrix} \quad (4.18)$$

in cui le variabili primarie  $\bar{z}$  e  $h$  sono rispettivamente il vettore delle frazioni molari e l'entalpia specifica; le variabili secondarie  $T$  e  $\rho$  sono la temperatura e la densità di miscela. Sono inoltre definite le condizioni di ingresso  $\bar{z}^{in}$  e  $h^{in}$ , il tempo di residenza  $t_{res}$ , la temperatura ambiente  $T_a$  e il coefficiente di scambio termico  $Q$ . Il vettore  $\bar{S}$  è determinato dallo schema cinetico e rappresenta il termine di reazione. La soluzione diretta del sistema (4.18) risulta computazionalmente impegnativa a causa del costo dovuto alla valutazione delle funzioni secondarie di temperatura e densità, tuttavia esso può essere efficacemente risolto utilizzando uno schema di *splitting* in cui la reazione sia separata dal termine di trasporto. L'applicazione di uno schema del tipo Strang risulta in questo caso banale grazie alla struttura del problema che porta a scrivere i sistemi separati come nella (4.19):

$$\begin{bmatrix} \frac{d\bar{z}^a}{dt} \\ \frac{dh^a}{dt} \end{bmatrix} = \begin{bmatrix} \bar{S}(\bar{z}, T(\bar{z}, h)) \\ 0 \end{bmatrix} \quad (4.19)$$

$$\begin{bmatrix} \frac{d\bar{z}^b}{dt} \\ \frac{dh^b}{dt} \end{bmatrix} = \begin{bmatrix} (\bar{z}^{in} - \bar{z})/t_{res} \\ (h^{in} - h)/t_{res} + (T_a - T(\bar{z}, h))Q/\rho(\bar{z}, T(\bar{z}, h)) \end{bmatrix}$$

In questo caso invece, la tipologia di *splitting* con estrapolazione lineare non risulta vantaggiosa dal punto di vista computazionale poiché è costoso ricostruire il gradiente, dato che non può essere ottenuto direttamente dal precedente sottopassaggio di reazione.

Perciò è importante conoscere a fondo il problema che si vuole affrontare in modo da scegliere la tecnica più adeguata a risolverlo, anche nel caso dello *splitting*, tenendo presente dei fattori che guidino la scelta: il peso computazionale

di ciascun passaggio, l'eventuale errore introdotto e la facilità di implementazione.

### 3.4.2. *Predictor-Corrector*

Il problema affrontato nel KPP, descritto dall'equazione (4.7), richiede una notevole accuratezza nella soluzione ed è possibile associare questa necessità ai vantaggi illustrati nel Paragrafo 3.4 riguardo alle tecniche di *splitting*. Infatti la tipologia di algoritmo risolutivo *predictor-corrector* si presta alla suddivisione in sottopassaggi del *time-step* permettendo l'integrazione separata di alcuni termini.

In linea di principio, gli algoritmi *predictor-corrector* utilizzano soluzioni dagli intervalli precedenti per proiettare la soluzione alla fine dell'intervallo successivo in modo più accurato. La parte *predictor* dell'algoritmo serve per trovare una stima iniziale della variabile dipendente alla fine dell'intervallo, successivamente la parte *corrector* ripete l'iterazione sullo stesso intervallo fino a che la soluzione non converge entro un limite di tolleranza accettabile. Lo svantaggio di questa tecnica è che, per gli intervalli iniziali, se i valori trovati dal *predictor* sono ancora lontani dalla soluzione, allora il *corrector* può non essere efficiente in termini di tempi di calcolo nel giungere a convergenza. Per questa ragione spesso si utilizza un algoritmo *predictor-corrector* dopo un metodo più robusto che avvicini alla soluzione.

Il sistema (4.7) può essere riscritto in modo da trattare i termini di trasporto e di reazione alternativamente in modo implicito ed esplicito, come mostrato nell'equazione (4.20):

$$\begin{cases} m_{tot} \frac{\bar{\omega}^{n+1/2} - \bar{\omega}^n}{\Delta t} = [-C\bar{\omega} + \bar{f}]_{n+1/2} + R(\bar{\omega}^n) \\ m_{tot} \frac{\bar{\omega}^{n+1} - \bar{\omega}^n}{\Delta t} = [-C\bar{\omega} + \bar{f}]_{n+1/2} + R(\bar{\omega}^{n+1}) \end{cases} \quad (4.20)$$

Si può osservare che nella prima delle due equazioni dell'algoritmo ci si sposta di mezzo passo sull'intervallo di integrazione, ma il termine di reazione, rimanendo esplicito, è riferito al passo precedente; questo è lo *step predictor*. Nella seconda equazione invece, il termine di trasporto è riferito allo *step*  $n + 1/2$ , mentre il

termine di reazione è ricavato alla fine dell'intervallo di integrazione secondo un algoritmo implicito, e ciò rappresenta lo *step corrector*. Quest'ultima equazione ha la stessa struttura di quella della procedura di risoluzione sequenziale della rete di CSTR mostrata nell'equazione (4.8): in entrambi i casi il termine di reazione è implicito. Quindi l'algoritmo di risoluzione della sequenza di CSTR si differenzia per la mancanza della prima delle due equazioni in (4.20) ed inoltre il tempo di integrazione è di 10 secondi che, visti i tempi caratteristici delle reazioni di combustione, corrisponde nella pratica al raggiungimento dello stato stazionario. Dal momento che è proprio lo stato stazionario ad interessare, mentre non è necessaria alcuna accuratezza nella descrizione del transitorio, si può riadattare la (4.20) in modo da svolgere l'integrazione della parte *corrector* fino al raggiungimento dello stato stazionario, mentre si vuole integrare la parte *predictor* utilizzando il *time-step* massimo possibile in modo tale che l'informazione si trasporti più velocemente incrementando la velocità di convergenza. L'algoritmo diviene dunque:

$$\begin{cases} m_{tot} \frac{\bar{\omega}^{n+1/2} - \bar{\omega}^n}{\Delta t} = [-C\bar{\omega} + \bar{f}]_{n+1/2} + R(\bar{\omega}^n) \\ 0 = [-C\bar{\omega} + \bar{f}]_{n+1/2} + R(\bar{\omega}^{n+1}) \end{cases} \quad (4.21)$$

Come prima implementazione, per semplicità, il sistema non lineare che rappresenta il secondo passaggio dell'algoritmo viene risolto come un ODE su un tempo infinito, nella quale però le iterazioni vengono interrotte oltre un *time-step* stabilito dall'utente per risparmiare iterazioni. Nel grafico di Figura 13 vengono messe a confronto le curve dei residui medi in funzione del numero di iterazioni svolte al variare del *time-step*:

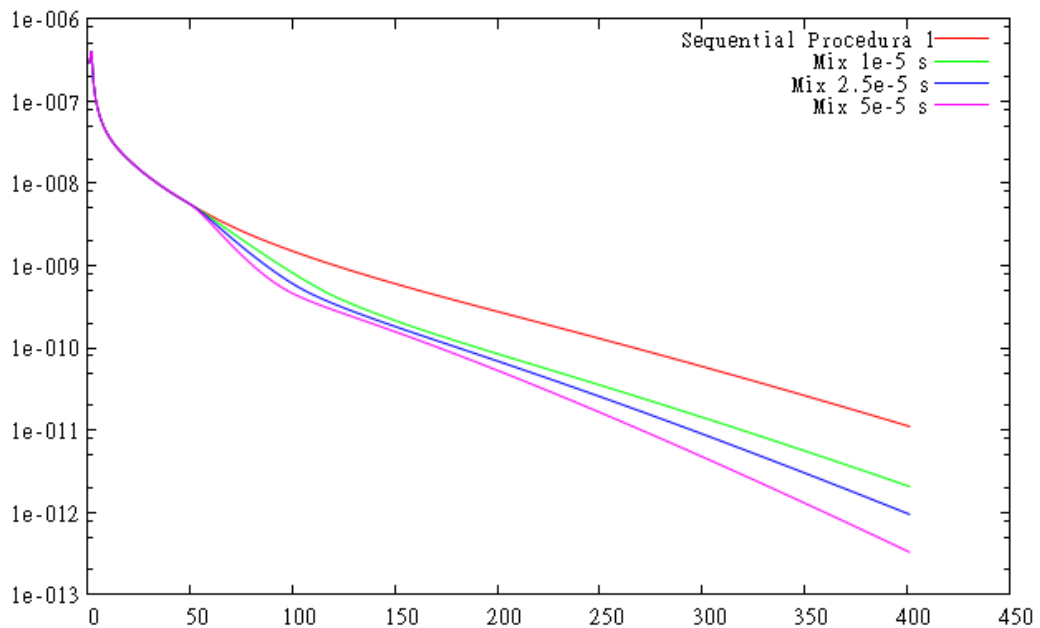


Figura 13: Confronto tra procedura sequenziale e *predictor-corrector* al variare del *time-step*

Dopo le prime iterazioni, svolte con l'algoritmo risolutivo dei CSTR in sequenza in tutti i casi studiati in modo da far partire più agevolmente il *predictor-corrector*, si osserva che, come ci si attendeva, tale metodo aumenta la velocità di convergenza. L'effetto è tanto più accentuato quanto maggiore è il *time-step* utilizzato. Rimane perciò da stabilire come si possa calcolare il *time-step* massimo utilizzabile senza rendere instabile il problema e quale sia il peso computazionale effettivo di un iterazione. Infatti il *predictor-corrector* necessita di meno iterazioni perché converge più velocemente della pura sequenza, tuttavia risulta computazionalmente più impegnativo in quanto è caratterizzato da più passaggi in serie. È necessario quantificare questi due parametri: il *time-step* e il costo per iterazione, allo scopo di ottimizzare la procedura risolutiva.

### 3.5. Implementazione nel KPP

Si è implementato l'algoritmo misto di soluzione in sequenza dei CSTR e *predictor-corrector* con *splitting* in modo da poter confrontare i risultati ottenuti con questa strategia con quelli ottenuti con il KPP classico, in cui la soluzione dei CSTR serviva ad abbassare i residui abbastanza da poter lanciare un metodo di Newton globale per la soluzione del sistema di ODE. Tuttavia quest'ultimo metodo non è applicabile per sistemi di dimensioni enormi (sopra le 100.000 celle

di calcolo) ed inoltre si presta meno alla parallelizzazione del codice. Il nuovo algoritmo porta con sé alcuni gradi di libertà che vanno opportunamente considerati. In primo luogo si è introdotto il criterio di stop delle iterazioni, utilizzando quello espresso dall'equazione (4.12). Confrontando i risultati ottenuti dal KPP con le due tipologie di algoritmo si è verificato che il criterio è valido e consente di ottenere risultati confrontabili. Ad esempio, i risultati per le frazioni massive finali ottenuti sullo stesso caso di una griglia composta da 2360 celle di calcolo con i due algoritmi (identificati da ora in avanti come “KPP *old*” e “KPP *new*”) sono riportati in Tabella 1:

**Tabella 1: Confronto tra frazioni massive ottenute con i due diversi algoritmi**

<b>Specie</b>	<b>KPP <i>old</i></b>	<b>KPP <i>new</i></b>	$\Delta =  old - new $	$\Delta\% = \frac{ old - new }{old} \times 100$
CH <sub>4</sub>	1.811593E-13	1.317750E-10	1.32E-10	7.263985E+04
CO	9.264982E-07	9.297130E-07	3.21E-09	3.469839E-01
CO <sub>2</sub>	2.302318E-02	2.302330E-02	1.20E-07	5.212138E-04
H <sub>2</sub> O	1.095708E-01	1.095710E-01	2.00E-07	1.825304E-04
O <sub>2</sub>	1.769722E-01	1.769720E-01	2.00E-07	1.130121E-04
OH	3.834035E-07	3.833670E-07	3.65E-11	9.519997E-03
HO <sub>2</sub>	1.025718E-08	1.027570E-08	1.85E-11	1.805564E-01
H <sub>2</sub>	3.100973E-09	3.104470E-09	3.50E-12	1.127711E-01
O	2.425848E-09	2.425640E-09	2.08E-13	8.574321E-03
CH <sub>2</sub> O	1.183641E-14	7.502400E-12	7.49E-12	6.328408E+04
H <sub>2</sub> O <sub>2</sub>	1.098170E-07	1.098050E-07	1.20E-11	1.092727E-02
N <sub>2</sub>	6.904225E-01	6.904230E-01	5.00E-07	7.241942E-05
NO	9.792577E-06	9.755990E-06	3.66E-08	3.736197E-01
HCN	7.271748E-11	7.222050E-11	4.97E-13	6.834395E-01
HNCO	9.848638E-10	9.843430E-10	5.21E-13	5.288041E-02

N <sub>2</sub> O	3.027455E-08	3.026890E-08	5.65E-12	1.866254E-02
------------------	--------------	--------------	----------	--------------

Come si può osservare, i risultati ottenuti con i due algoritmi, posto un criterio di stop opportuno, sono diversi a meno di una precisione accettabile. Fanno eccezione i casi di CH<sub>4</sub> e CH<sub>2</sub>O, per i quali nonostante si riscontri una differenza di alcuni ordini di grandezza, tuttavia il valore finale dovrebbe essere vicino allo zero, come effettivamente avviene in entrambi i casi. La differenza non è quindi da imputare alla precisione della soluzione, che risulta invece accurata su specie critiche come il CO o NO, ma piuttosto al fatto che si stanno confrontando tra loro valori molto vicini allo zero: preso il caso *old* come riferimento, si può notare infatti come la concentrazione di CH<sub>4</sub> e CH<sub>2</sub>O sia la più bassa tra tutte le specie (rispettivamente 10<sup>-13</sup> e 10<sup>-14</sup>), minore del valore di 10<sup>-12</sup> usato come criterio di stop nel valore della norma 1.

Nel caso di studio preso in considerazione viene monitorato il tempo di calcolo necessario per risolvere il sistema con la precisione voluta, al variare di alcuni parametri. Lo schema cinetico utilizzato è il FLUENT DRM 22 POLIMI composto da 35 specie e 155 reazioni. Per monitorare il comportamento dell'algoritmo allo scalare delle dimensioni della griglia in termini di numero di celle di calcolo, lo stesso caso è stato applicato a tre griglie composte rispettivamente da 590, 2360 e 9440 reattori CSTR. A causa dei problemi di convergenza propri delle prime iterazioni del metodo *predictor-corrector* quando i residui sono ancora troppo grandi, le prime iterazioni vengono svolte utilizzando la soluzione in sequenza di reattori CSTR.

Nell'ottimizzazione dell'algoritmo di soluzione è opportuno valutare anche con che criterio passare da un metodo all'altro in modo da innescare la procedura di *splitting* quando i residui sono abbastanza piccoli da non rendere eccessivamente pesante la singola iterazione, ma non troppo piccoli in modo da sfruttare appieno l'accelerazione della velocità di convergenza che l'approccio *predictor-corrector* comporta. Questo ulteriore grado di libertà viene analizzato nei casi seguenti.

Ad ogni caso di studio corrisponde un *file* di *input* che riassume le caratteristiche del caso stesso. Si riporta di seguito la parte significativa del primo caso:

```
#Kinetics          /home/./Fluent_DRM22_Polimi_NOX
#Input             /home/./Input-NoReordered-4x
#Output            /home/./Output

#EndSequenceCSTR 2e-10          //Stop SequenceCSTR if
                                //Norm1Resid < value
#CSTRIntegrationTime 10.    s

#StopFactorODESingleContinuousReactor 1e-8

#InitialTimeStep   1e-6 s

#TimeStepIncrementFactor 1.

#MaxTimeStep       1e2 s

#VerboseSingleContinuousReactorStatistics on

#END
```

Nel codice riportato si assegna lo schema cinetico, mentre il comando *#Input* consente di selezionare il caso da cui verrà letta la rete di reattori. Il comando *#EndSequenceCSTR* rappresenta il criterio con cui si passa dalla soluzione della sequenza di reattori al *predictor-corrector*: si definisce un valore della media della norma 1 dei residui di tutte le equazioni, come definito nella (4.11) raggiunto il quale si innesca automaticamente il secondo metodo. Altro comando molto importante è *#InitialTimeStep*, che definisce la dimensione iniziale del passo temporale su cui integrare. Se il fattore di incremento del *time-step* è maggiore di 1, allora il passo si ingrandisce all'approssimarsi della convergenza, per rendere l'integrazione più rapida, altrimenti si può porre un fattore unitario e svolgere tutta la soluzione con il passo di integrazione definito, come è il caso dell'*input file* di esempio riportato sopra.

Si sono svolte analisi su due casi di studio principali, le cui caratteristiche salienti sono riportate in Tabella 2:

Tabella 2. Riassunto delle caratteristiche dei casi studiati

Caratteristiche	CASO 1	CASO 2
# Reattori	590 - 2360 - 9440	590 - 2360 - 9440
Schema Cinetico	Fluent DRM 22	Fluent DRM 22
# Specie	35	35
CSTRIntegrationTime	10 s	10 s
StopFactorODESingleContinuousReactor	1.00E-8	1.00E-7
InitialTimeStep	1.00E-6	1.00E-8
TimeStepIncrementFactor	1	1.5
MaxTimeStep	1.00E+2 s	1.00E+2 s
Break Predictor-Corrector if Norm1Resid <	1.00E-12	1.00E-12

Entrambi i casi sono stati applicati alle tre reti di dimensioni crescenti. Inoltre per studiare come il criterio di passaggio da un tipo di algoritmo all'altro influenzasse il tempo di calcolo, per ogni griglia di ciascun caso si è ripetuta la simulazione imponendo che si interrompesse la risoluzione della sequenza di CSTR quando il valore della norma 1 dei residui raggiungeva i valori  $\varepsilon$  riassunti in Tabella 3:

Tabella 3. Valori per il criterio di stop della risoluzione della sequenza di CSTR

$\varepsilon$	1.00E-09	3.00E-10	2.00E-10	1.00E-10	1.00E-11	1.00E-12
---------------	----------	----------	----------	----------	----------	----------

L'ultimo caso, quello tale per cui  $\varepsilon = 1.00E - 12$ , corrisponde a compiere tutte le iterazioni con la procedura della sequenza di CSTR, in modo da avere un caso base di confronto per valutare l'efficienza dell'introduzione dello *splitting*.

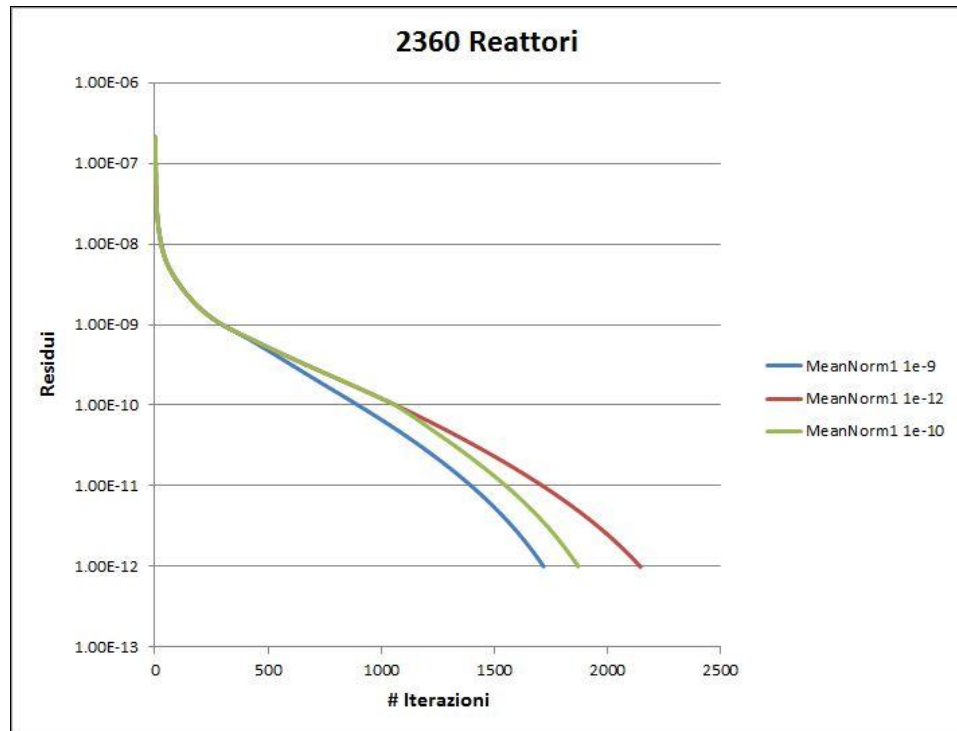
Sono stati misurati i tempi di calcolo per completare la parte di sequenza dei CSTR, quella di *predictor-corrector* e il tempo totale. Per il Caso 1 i risultati sono riportati in Tabella 4:

Tabella 4: Tempi di esecuzione per il casi di studio 1

# Celle		Limite sul valore della norma 1 dei residui per il cambio di algoritmo					
		1.00E-09	3.00E-10	2.00E-10	1.00E-10	1.00E-11	1.00E-12
590	SequenceCSTR Time [s]	30.59	31.11	32.39	35.47	44.19	46.06
	Predictor-Corrector Time [s]	29.73	24.35	19.68	16.38	6.61	0.06
	Total CPU Time [s]	60.32	55.46	52.07	51.86	50.81	46.11
2360	SequenceCSTR Time [s]	203.81	327.88	363.33	416.84	554.19	630.32
	Predictor-Corrector Time [s]	467.45	306.77	294.87	284.22	83.94	0.28
	Total CPU Time [s]	671.25	634.65	658.19	701.06	638.13	630.61
9440	SequenceCSTR Time [s]	-	1652.12	1944.96	2580.62	4219.69	5716.78
	Predictor-Corrector Time [s]	-	5700.03	5152.77	4442.00	1976.64	1.05
	Total CPU Time [s]	-	7352.15	7097.73	7022.62	6196.33	5717.83

Nel caso con 9440 reattori non si sono effettuate simulazioni in cui l'uscita dalla risoluzione in sequenza dei CSTR avveniva al valore di  $1.00E-9$  della norma dei residui, perché si è verificato che, data la richiesta computazionale elevata, i residui erano ancora troppo lontani da zero per permettere l'innescio efficiente della procedura *predictor-corrector*, che quindi non riusciva a convergere.

Si consideri l'andamento dei residui al crescere delle iterazioni per la griglia composta da 2360 reattori, come mostrato in Figura 14: si può osservare che, rispetto al caso base nel quale la risoluzione della sequenza di CSTR è condotta fino al criterio di stop finale, l'introduzione del metodo *predictor-corrector* incrementa la velocità di convergenza. L'effetto è tanto più evidente quanto prima si innesca il metodo, infatti la curva per la quale il *predictor-corrector* parte quando il valore della media della norma dei residui è  $10^{-9}$  compie un numero significativo di iterazioni in meno rispetto a quella ottenuta con la pura sequenza di CSTR. Si può altresì distinguere la discontinuità nella curva dove avviene il cambiamento di metodo che porta ad un repentino cambiamento della curvatura.



**Figura 14: Media della norma dei residui al crescere delle iterazioni**

Purtroppo i risultati presentati non incoraggiano l'uso del *predictor-corrector*: osservando i tempi di calcolo presentati nella Tabella 4 ci si rende infatti conto che tale metodo consente di fare molte iterazioni in meno, ma che in realtà richiedono più tempo, facendo sì che il metodo risulti svantaggioso dal punto di vista computazionale. Osservando i dati sui tempi si nota come in questi casi il *predictor-corrector* sia tanto meno favorito quanto più fitta è la rete di reattori. In prima battuta si può spiegare questo risultato osservando il tempo medio necessario a compiere un'iterazione per i due diversi algoritmi, mostrato in Tabella 5:

**Tabella 5: Tempi di calcolo medi per un'iterazione**

<b># Reattori</b>	<b>590</b>	<b>2360</b>	<b>9440</b>
<b>t medio Sequenza CSTR [s]</b>	0.16	0.42	1.44
<b>t medio P-C [s]</b>	0.07	0.25	1.1

Si nota che al crescere del numero delle celle di calcolo i tempi di calcolo medi per i due algoritmi sono sempre più simili tra di loro, togliendo margine al vantaggio teorico del *predictor-corrector*. Le iterazioni di quest'ultimo metodo,

infatti, hanno circa tutte lo stesso costo a prescindere dal numero di iterazioni svolto. Il contrario avviene per la sequenza di CSTR, man mano che si risolve la quale le iterazioni divengono sempre più veloci e quindi competitive con il metodo alternativo. Come risultato si ha che le iterazioni finali calcolate con la sequenza di CSTR sarebbero molto più veloci del *predictor-corrector* rendendolo svantaggioso.

Si è quindi affrontato il problema di come poter migliorare l'efficienza del *predictor-corrector*. I gradi di libertà su cui intervenire sono principalmente due: il tempo di integrazione del CSTR fino a cui iterare nella fase *corrector* e il *time-step* della singola integrazione. Si è verificato che il primo parametro non ha un'influenza significativa sui tempi di calcolo, a meno di renderlo molto piccolo, ma questo non è fattibile perché potrebbe creare problemi di stabilità della soluzione, oltre a non garantire il raggiungimento dello stato stazionario. Il fattore di primaria importanza è quindi il passo temporale di integrazione. Esso deve conciliare aspetti contrastanti: è necessario che sia sufficientemente piccolo da non creare instabilità nell'integrazione, ma d'altro canto, quanto più è piccolo tante più iterazioni sono necessarie per coprire l'intervallo di tempo, rendendo pesante soprattutto la parte iterativa, quindi nuovamente lo *step corrector* dell'algoritmo interessato. Per rispondere a queste esigenze si sono ripetute le prove sulle tre griglie, ma partendo da un *time-step* di due ordini di grandezza più piccolo e introducendo un fattore di amplificazione che lo aumentasse via via che si raggiunge la convergenza. È previsto anche un fattore di riduzione nel caso ci si allontanasse dalla convergenza. I dettagli del caso di studio sono stati mostrati nella Tabella 2; tutte le successive simulazioni sono state lanciate sulla stessa macchina per confrontare i tempi in modo uniforme. Si è utilizzata la macchina multiprocessore del dipartimento di Chimica, Materiali e Ingegneria Chimica "G. Natta" con processori Xeon E7340, 2.4GHz, 8MB, 1066 FSB e 32 GB di RAM a 667 MHz condivisa. Lanciando le simulazioni nelle stesse condizioni del caso precedente sono stati raccolti i tempi di esecuzione nella Tabella 6:

Tabella 6: Tempi di esecuzione per il casi di studio 2

		Limite sul valore della norma 1 dei residui per il cambio di algoritmo					
# Celle		1.00E-09	3.00E-10	2.00E-10	1.00E-10	1.00E-11	1.00E-12
590	SequenceCSTR Time [s]	15.35	21.82	23.63	26.49	33.71	38.00
	Predictor-Corrector Time [s]	24.48	18.35	16.29	12.94	5.02	0.06
	Total CPU Time [s]	39.83	40.17	39.92	39.44	38.73	38.06
2360	SequenceCSTR Time [s]	134.956	269.461	312.604	370.832	494.798	548.285
	Predictor-Corrector Time [s]	405.567	266.998	236.906	178.072	55.8821	0.229949
	Total CPU Time [s]	540.523	536.459	549.51	548.904	550.6801	548.5149
9440	SequenceCSTR Time [s]	-	1474.09	1708.29	2189.83	3615.85	5381.92
	Predictor-Corrector Time [s]	-	5840.45	4998.61	4394.33	2520.01	1.33462
	Total CPU Time [s]	-	7314.54	6706.9	6584.16	6135.86	5383.255

In questo caso di studio le condizioni sono leggermente più favorevoli per il *predictor-corrector*, infatti la distanza tra i tempi ottenuti nei casi in cui il *predictor-corrector* è stato innescato e quelli in cui il caso è stato risolto come pure sequenza di CSTR si assottiglia. Questo comportamento è spiegabile con le modifiche che sono state apportate al *time-step*: partire da un valore più piccolo consente di rendere più accurata l'integrazione all'inizio, quando si è ancora lontani dalla convergenza. Il fattore di amplificazione del passo permette poi di rendere molto più veloci le iterazioni successive, dando un vantaggio soprattutto allo *step corrector*.

Tuttavia nel caso che si è studiato, nonostante gli accorgimenti introdotti, non si è rilevata evidenza che l'introduzione del *predictor-corrector* portasse ad un effettivo miglioramento della procedura risolutiva in termini di tempo di calcolo richiesto. La peculiarità del metodo consiste nell'aumentare la velocità di convergenza a pari numero di iterazioni, rispetto alla pura sequenza di CSTR, tuttavia le singole iterazioni richiedono più tempo. Si ha quindi un vantaggio significativo laddove si riesca a raggiungere la convergenza con un numero minore di iterazioni, in modo da guadagnare tempo di calcolo. Perciò il passo d'integrazione deve poter crescere rispetto a quello iniziale così che il *time-step* lungo consenta di giungere alla fine dell'intervallo usando meno iterazioni.

Tuttavia, per ragioni di stabilità della soluzione non è possibile aumentare il passo d'integrazione indefinitamente, bensì esiste un limite teorico. Infatti ci si è resi conto che, nonostante il controllo per applicare il fattore di amplificazione fosse eseguito ad ogni iterazione, il passo di integrazione a convergenza raggiunta era rimasto comunque molto piccolo: dell'ordine di grandezza di  $10^{-5}$  secondi. Per spiegare il perché dell'impossibilità di aumentare il passo di integrazione mantenendo la soluzione stabile nel caso studiato si è preso in considerazione il numero di Courant, che si definisce come il rapporto tra il *time-step* e il tempo caratteristico di convezione:

$$c = u \frac{\Delta t}{\Delta x} \quad (4.22)$$

in cui  $u$  è la velocità del flusso,  $\Delta t$  è il *time-step* e  $\Delta x$  la dimensione caratteristica della cella. Tale rapporto è uno dei parametri fondamentali della fluidodinamica computazionale per stabilire se è mantenuta la stabilità o meno dell'integrazione. Ricordando che la portata massiva si può esprimere come:

$$\rho u A = \dot{m} \quad (4.23)$$

da cui si ricava la velocità:

$$u = \frac{\dot{m}}{\rho A} = \frac{\dot{m}}{\rho \times \Delta x^2} \quad (4.24)$$

ed esprimendo la massa presente in ogni reattore:

$$m = \rho \times V = \rho \times \Delta x^3 \quad (4.25)$$

da cui:

$$\Delta x^3 = \frac{m}{\rho} \quad (4.26)$$

sostituendo, si può riscrivere l'espressione (4.22):

$$c = \Delta t \frac{\dot{m}}{\rho \times \Delta x^3} = \frac{\dot{m}}{m} \quad (4.27)$$

si può dimostrare che, perché la soluzione sia stabile deve valere:

$$c < 1 \tag{4.28}$$

Quindi il limite teorico per il numero di Courant corrisponde a  $c = 1$ , per cui si può sostituire tale vincolo per calcolare il *time-step* massimo teorico per il sistema:

$$\Delta t_{\max} \frac{\dot{m}}{m} = 1 \tag{4.29}$$

da cui:

$$\Delta t_{\max} = \frac{m}{\dot{m}} \tag{4.30}$$

Stanti le caratteristiche del sistema in esame, sostituendo i valori della massa per ciascun reattore e della portata nell'espressione (4.30) si ottiene un *time-step* dell'ordine di  $10^{-5}$  secondi, come si era trovato anche dall'analisi del caso di studio. Quindi la possibilità di accelerare la convergenza è strettamente legata alle caratteristiche fisiche del sistema, perché esse determinano il passo d'integrazione massimo. Inoltre, come mostra l'equazione (4.22) il *time-step* è direttamente proporzionale alla dimensione caratteristica della cella, per cui infittendo la griglia l'integrazione richiede un maggiore costo computazionale, come si era verificato nei casi di studio, anche per tale ragione.

Infine si sono confrontati i tempi di calcolo richiesti dall'algorithm tradizionale che comprende il metodo di Newton globale sull'intero sistema con quelli richiesti dall'algorithm proposto nel presente capitolo, ovvero con approccio misto di soluzione in sequenza di CSTR e *predictor-corrector*. I risultati ottenuti sono mostrati in Tabella 7:

**Tabella 7. Confronto tra i tempi di calcolo dell'algorithm tradizionale e dell'approccio misto sequenza di CSTR e *predictor-corrector***

# Reattori	590	2360	9440
Tempo KPP <i>old</i> [s]	20.49	197.94	3010.34
Tempo KPP <i>new</i> [s]	38.06	536.46	5383.26
Rapporto <i>new/old</i>	1.86	2.71	1.79

Si può notare che al variare delle dimensioni della griglia si ha un rapporto tra i tempi dei due algoritmi variabile tra  $\approx 2$  e  $3$ . La nuova procedura implementata è più lenta, come ci si attendeva, ma si presta molto meglio ad essere convertita al calcolo parallelo ed inoltre consente di utilizzare griglie con un numero di reattori molto superiore a quello attuale. Nell'ipotesi di una perfetta scalabilità in parallelo basterebbero tre processori per rendere la nuova procedura veloce quanto quella utilizzata fino ad ora. Dal momento che si prevede di utilizzare un *cluster* con decine di nodi, le prospettive di miglioramento dei tempi di calcolo riscrivendo il codice in parallelo sono notevoli. Studiati gli aspetti numerici dell'algoritmo risolutivo, si rende ora necessaria un'analisi delle operazioni e del possibile scambio di dati tra i processi in modo da sovrapporre una struttura parallela al codice del KPP scritto.

## Capitolo 4

### Approccio alla parallelizzazione

#### 4.1. Prodotto tra matrici

Prima di affrontare un problema complesso come il KPP dal punto di vista del calcolo parallelo, è opportuno analizzare un caso di esempio. In questo modo si può strutturare la logica di soluzione in un contesto in cui non sono presenti altri disturbi; una volta provata l'efficacia della logica la si può applicare con successo a casi più complessi e più impegnativi dal punto di vista computazionale. Nel caso presente si è voluto affrontare il problema del prodotto tra matrici perché, benché sia matematicamente semplice, è ricco di possibili approcci alla soluzione.

Uno degli algoritmi più conosciuti per la moltiplicazione matrice per matrice utilizza la definizione di prodotto scalare: date due matrici  $A \in \mathbb{R}^{m \times r}$  e  $B \in \mathbb{R}^{r \times n}$ , si può calcolare il prodotto  $C = AB$  considerando la matrice  $C$  come un insieme di prodotti scalari. La struttura logica dell'algoritmo può essere schematizzata come segue:

```
m = righe(A); r = colonne(A)
n = colonne(B)
c(1:m,1:n) = 0
for i = 1:m
    for j = 1:n
        for k = 1:r
            C(i,j) = C(i,j) + A(i,k)B(k,j)
        end
    end
end
```

In questa procedura l'elemento  $c_{ij}$  della matrice prodotto viene calcolato come il prodotto scalare della  $i$ -esima riga di  $A$  e della  $j$ -esima colonna di  $B$ . Seguendo tale logica è stato scritto un codice in C++ la cui parte centrale, che viene riportata di seguito, rispecchia l'algoritmo presentato:

```
{
  for ( int i = 0; i < a; i++ )
    for ( int j = 0; j < c; j++ )
      { double sigma = 0;

        for ( int k = 0; k < b; k++ )
          { sigma += A[i][k] * Bt[j][k]; }
        C[i][j] = sigma;
      }
}
```

In questo caso gli elementi della matrice prodotto sono ottenuti tramite una variabile di servizio chiamata *sigma*. La matrice B è stata trasposta per ragioni di efficienza dell'algoritmo, in tal modo infatti si svolgono i calcoli in sequenza, il che risulta preferibile rispetto alla versione *scattered* dell'algoritmo. Per aumentare ulteriormente l'efficienza dell'algoritmo si è sfruttata l'allocazione dinamica di memoria consentita dal C++ introducendo dei puntatori per velocizzare l'accesso:

```
for(int i = 0; i < a; i++)
{
  for(int j = 0; j < c; j++)
  {
    double sigma = 0;
    double *A_ptr = myArray_ptr + i*b;
    double *B_ptr = myFactor_ptr + j*c;
    for(int k = 0; k < b; k++)
    {
      sigma += (*A_ptr)*(*B_ptr);
      A_ptr++;
      B_ptr++;
    }
    *myProduct_ptr++ = sigma;
  }
}
```

In questo modo le operazioni sono svolte in celle di memoria contigue rendendole più veloci.

Si può riformulare il problema del prodotto tra matrici per esplicitarne gli aspetti parallelizzabili. Innanzitutto l'algoritmo risolutivo implica la ripetizione di operazioni: infatti la parte computazionalmente rilevante si racchiude in poche righe di codice entro due cicli *for*. Inoltre le operazioni non sono necessariamente consequenziali: la riga *i*-esima della matrice prodotto non dipende dalle

precedenti, ma è ottenuta dalla  $i$ -esima riga della prima matrice e dalla  $j$  colonne della seconda matrice moltiplicata.

Ciò suggerisce la presenza di *concurrency* nell'operazione del calcolo del prodotto: righe diverse della matrice risultante possono essere calcolate contemporaneamente. Esistono diverse tecniche per implementare il calcolo del prodotto tra matrici su *cluster* di processori a memoria distribuita; questi si differenziano principalmente per come sfruttano la comunicazione e la *concurrency*. Ad esempio l'algoritmo di Bernstein ha un *overhead* minore rispetto all'algoritmo di Cannon, quindi tende ad ottimizzare la comunicazione; tuttavia ha un grado di *concurrency* inferiore, ovvero la frazione di codice parallelizzabile è minore (Gupta e Kumar, 1993). Anche lo schema di comunicazione tra i processori è rilevante e l'algoritmo scelto deve essere ottimizzato per la tipologia di architettura *hardware* presente. Si può infatti implementare un algoritmo che preveda un intensivo scambio di informazione tra i processori se la struttura della macchina è tale per cui tutti i processi, o nodi, sono in grado di comunicare tra loro, come avviene in uno schema *Infiniband*, un protocollo di comunicazione nel quale tutti i nodi sono connessi tramite degli *switch* ad alta velocità per la comunicazione puntuale bidirezionale. Quindi ogni nodo è in grado di comunicare un pacchetto di dati a qualsiasi altro della griglia senza coinvolgere nessun altro nodo che non sia il *sender* e il *receiver*. Se invece il *cluster* avesse una struttura *cube-connected* come quella schematizzata in Figura 15:

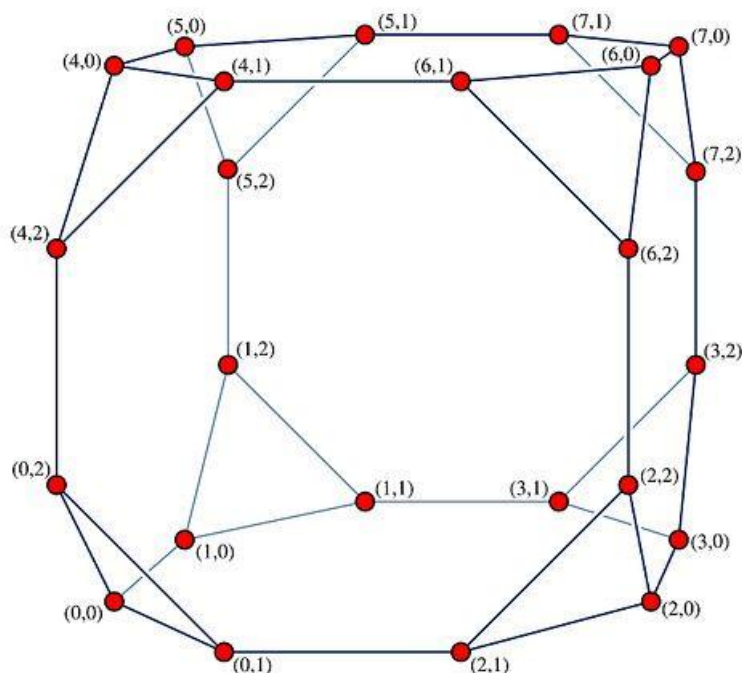


Figura 15: Struttura *cube-connected* del terzo ordine (un nodo comunica con 3 adiacenti)

in cui è rappresentato uno schema del terzo ordine tale che consenta la comunicazione diretta di un nodo solo con i 3 adiacenti, allora un algoritmo con un passaggio di messaggi intensivo potrebbe divenire non efficiente a causa di elevati costi del tempo di *overhead*. La scelta della struttura della rete è solitamente dettata da considerazioni economiche, poiché gli *switch* sono elementi molto costosi; spesso infatti si preferisce uno schema di comunicazione puntuale tra i nodi anche se non è la soluzione più efficiente. Considerando infatti la rete schematizzata in Figura 15, se il nodo (0,0) dovesse comunicare con il nodo (7,0), anche seguendo il percorso più breve dovrebbero essere coinvolti altri 5 nodi, con altrettante istruzioni di *send/receive*.

Non esiste quindi un algoritmo migliore in assoluto, ma esistono delle condizioni in cui alcuni algoritmi risultano più efficienti di altri. Riconoscere tali condizioni e schematizzarle in una strategia risolutiva integrata con l'algoritmo è parte integrante del lavoro di ottimizzazione di un processo di calcolo parallelo.

Uno strumento efficace per valutare le prestazioni di diverse combinazioni di architettura e algoritmi è l'analisi di scalabilità. La funzione di isoefficienza è un parametro quantitativo per misurare la scalabilità, definita come la capacità di un

algoritmo di sfruttare efficacemente un numero crescente di processi in un architettura parallela. Il vantaggio dell'analisi consiste nel rappresentare in modo sintetico l'impatto dell'*overhead* di comunicazione, la *concurrency*, i colli di bottiglia seriali, ovvero quelle parti di codice che non possono essere parallelizzate e lo sbilanciamento del carico, in una singola espressione. L'efficienza di un algoritmo parallelo è stata definita nel Capitolo 2, con l'equazione (3.2). Introducendo il tempo di *overhead* e il tempo totale di esecuzione seriale l'equazione si può riscrivere come:

$$E = 1 / \left( 1 + \frac{T_0}{W} \right) \quad (5.1)$$

dove  $T_0$  è il tempo di *overhead*, ovvero la somma dei costi di comunicazione, dei tempi morti nella sincronizzazione tra processi e dell'esecuzione delle parti seriali dell'algoritmo; mentre  $W$  è il tempo impiegato dal migliore algoritmo seriale per risolvere il problema con un singolo processore.

L'analisi della funzione di isoefficienza è solo uno dei modi per valutare quantitativamente le prestazioni di un sistema parallelo e si può riassumere nella formula:

$$W = \frac{E}{1-E} T_0(W, p) \quad (5.2)$$

#### 4.2. Analisi di scalabilità

Data un'architettura parallela ed un problema di dimensioni fissate è noto che lo *speedup* di un algoritmo parallelo non aumenta indefinitamente all'aumentare del numero di processori utilizzati, bensì raggiunge asintoticamente un determinato valore. Questo comportamento si può spiegare principalmente in due modi: o per via del fatto che il peso degli *overhead* diventa preponderante all'aumentare del numero dei processori coinvolti, o perché tale numero infine supera il grado di parallelizzabilità dell'algoritmo, rendendo quindi inutile ogni ulteriore processo parallelo introdotto non potendo sfruttare alcuna operazione concorrente. Una modalità di verifica dell'efficienza di un algoritmo parallelo consiste nel considerare la funzione di isoefficienza, ovvero il legame tra la

dimensione del problema e il numero di processori; quando tale relazione si mantiene tale per cui l'efficienza, come definita nell'equazione (3.2), è costante allora si può chiamare funzione di isoefficienza. Se si utilizza un sistema parallelo per risolvere un problema di dimensione fissata, all'aumentare del numero di processori l'efficienza diminuisce perché il tempo totale di *overhead* cresce insieme al numero di processori. Si ha poi che, fissato il numero dei processori, se si aumenta la dimensione del problema l'efficienza aumenta perché  $T_0$  cresce più lentamente di  $W$ . Dunque in sistemi paralleli che possiedono le caratteristiche descritte si può mantenere l'efficienza ad un valore scelto (ovviamente compreso tra 0 e 1, come impone la definizione) facendo sì che all'aumentare del numero dei processori la dimensione del problema sia opportunamente accresciuta. I sistemi paralleli per cui vale quanto formulato prendono il nome di sistemi scalabili.

Lo studio della scalabilità è uno strumento di utilità generale che, applicato a casi di dimensioni ridotte o facilmente monitorabili, permette di valutare l'efficienza dell'approccio parallelo allo scopo di prevedere il comportamento di sistemi molto più complessi. Per questo nel presente lavoro di Tesi si è scelto di approfondire in primo luogo l'analisi per l'algoritmo di moltiplicazione matrice per matrice. I risultati ottenuti servono da termine di confronto e da base per approcciare la parallelizzazione del Post-Processore Cinetico.

L'algoritmo per la moltiplicazione di matrici presentato nel paragrafo 4.1 può essere analizzato in modo da evidenziare quali contributi richiedano tempo in un'eventuale risoluzione parallela. Si ipotizzi per semplicità una griglia quadrata di  $p$  processori usata per moltiplicare due matrici di dimensione  $n \times n$  ed essendo  $n \geq p$ . Le matrici possono essere suddivise in blocchi che sono inviati a ciascun processore in cui viene calcolata una parte della matrice prodotto di  $n^2/p$  elementi. Ciò implica un invio a tutti i nodi di  $n^2/p$  elementi per la matrice A attraverso i  $\sqrt{p}$  processi di ciascuna riga della griglia e lo stesso per gli elementi della matrice B. Tali operazioni si possono completare in un tempo di:

$$2t_s \log p + 2t_w \frac{n^2}{\sqrt{p}} \quad (5.3)$$

in cui si esplicita il tempo di comunicazione dei messaggi composto dal termine di preparazione del messaggio  $t_s$ , e dal tempo di comunicazione per “parola” per messaggio  $t_w$ . Le ultime due grandezze introdotte dipendono dalle caratteristiche *hardware* della rete di processori, in particolare  $t_w$  equivale al rapporto tra il numero di *byte* che compongono una “parola” del messaggio e la larghezza di banda in *byte* al secondo. Dopo aver ricevuto la sua frazione dei dati di partenza, ogni nodo di calcolo esegue la moltiplicazione impiegando, come si è detto nel Capitolo 1, un tempo dell’ordine di  $n^3/p$ . Quindi il tempo totale per l’esecuzione parallela del codice con  $p$  processori risulta (Gupta e Kumar, 1993):

$$T_p = \frac{n^3}{p} + 2t_s \log p + 2t_w \frac{n^2}{\sqrt{p}} \quad (5.4)$$

Questo tipo di analisi permette di conoscere a priori importanti informazioni sul comportamento dell’algoritmo, come l’utilizzo di memoria. Dall’equazione (5.4) si può inferire ad esempio che la richiesta di memoria per ciascun processo è dell’ordine di  $O(n^2/\sqrt{p})$ , quindi la richiesta di memoria totale è  $O(n^2\sqrt{p})$ , maggiore di quella per l’algoritmo sequenziale, che è di  $O(n^2)$ . Come spesso accade, la parallelizzazione dell’algoritmo porta ad una diminuzione delle prestazioni, che dovrebbero essere recuperate grazie allo *speedup* dovuto all’uso di un numero sufficiente di processori.

Possono essere scritti altri algoritmi paralleli per il prodotto di matrici, anche più efficienti di quello proposto, ma non interessa in questa sede approfondire ulteriormente questo aspetto. Basti sapere che sulla stessa architettura gli algoritmi possono distinguersi per il modo in cui suddividono i blocchi delle matrici di partenza, cambiando il numero di messaggi che è necessario mandare per computare la matrice prodotto completa. In ogni caso, per quasi tutti gli algoritmi il tempo di *overhead* è composto dalle due componenti che sono funzione di  $t_s$  e  $t_w$ . Spesso si può individuare la componente dominante, quella

che richiede alla dimensione del problema di crescere maggiormente per mantenere l'efficienza costante all'aumentare del numero di processi utilizzati. Confrontando in letteratura (Gupta e Kumar, 1993) le prestazioni di diversi algoritmi si verifica che per ciascuno esiste un intervallo di dimensioni del problema e numero di processori utilizzati che rende consigliabile l'uso di una tipologia piuttosto che di un'altra; senza dimenticare l'importanza dell'architettura *hardware*. Avendo considerato questi aspetti, si è potuto implementare una struttura parallela al codice in C++ che era stato scritto ed è stata testata conducendo un'analisi di scalabilità.

### 4.3. Parallelizzazione del prodotto tra matrici

Nel paragrafo 4.2 si è visto che lo schema logico dell'algoritmo parallelo verte intorno all'invio di blocchi delle matrici di partenza, i quali vengono processati separatamente per dare una frazione della matrice risultante. Una tecnica efficiente per sfruttare la *concurrency* di dati sulle stesse operazioni è lo schema *master-slave*. In questo modello di struttura parallela si ha un processo "master" che controlla l'esecuzione del programma e crea dei *thread* di altri processi detti "slave" o "worker" che hanno il compito di svolgere i calcoli nella parte opportuna del programma, come illustrato schematicamente in Figura 16:

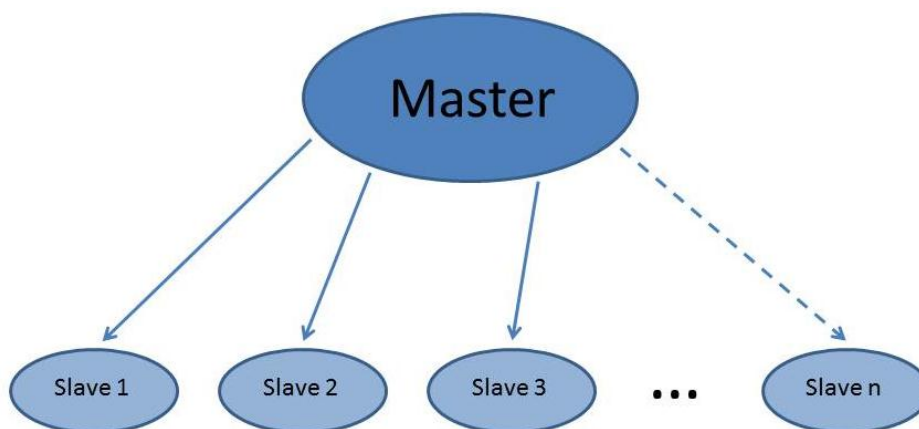


Figura 16: Schema del modello *master-slave*

Nel caso dell'algoritmo che si vuole parallelizzare, il *master* ha il compito di suddividere i dati e di distribuirli tra i vari processi *worker*, come si vede dal codice di esempio:

```

/*-----*/
/****** master task *****/
/*-----*/

/*-----*/
/* Misura tempo di clock del processore */
/*-----*/

double startTime = MPI::Wtime();

/*-----*/
/* Invio dei dati ai processi worker */
/*-----*/

averow = NRA/numworkers;
extra = NRA%numworkers;
offset = 0.;
mtype = FROM_MASTER;
for (dest=1; dest<=numworkers; dest++)
{
    rows = (dest <= extra) ? averow+1 : averow;
/*-----*/
/* Si indica il valore di offset a cui ciascun processo */
/* comincia a cercare dati nella matrice A */
/*-----*/

    MPI::COMM_WORLD.Send(&offset,1, MPI::INT, dest, mtype);

/*-----*/
/* Invio del numero di righe che ciascun processo deve computare */
/*-----*/

    MPI::COMM_WORLD.Send(&rows, 1, MPI::INT, dest, mtype);

/*-----*/
/* Invio a ciascun processo di rows*NCA frazioni di dati a partire */
/* dal valore di offset */
/*-----*/

MPI::COMM_WORLD.Send(&a[offset][0], rows*NCA, MPI::DOUBLE, dest,
mtype);

/*-----*/
/* Invio a ciascun processo della matrice B */
/*-----*/

MPI::COMM_WORLD.Send(&b, NCA*NCB, MPI::DOUBLE, dest, mtype);
    offset = offset + rows;
}

/*-----*/
/* Attesa dei risultati da parte di tutti i processi worker */
/*-----*/

mtype = FROM_WORKER;
for (i=1; i<=numworkers; i++)
{
    source = i;

```

```

/*-----*/
/* Ricezione del valore di offset con cui si è concluso il processo */
/*-----*/

MPI::COMM_WORLD.Recv(&offset,1,MPI::INT,source,mtype,status);

/*-----*/
/* Ricezione del numero di righe computate */
/*-----*/

MPI::COMM_WORLD.Recv(&rows,1,MPI::INT,source,mtype,status);

/*-----*/
/* Ricezione del valore finale della matrice C a partire */
/* dal corrispondente valore di offset */
/*-----*/

MPI::COMM_WORLD.Recv(&c[offset][0],rows*NCB,MPI::DOUBLE,source
,mtype,status);
}

```

In un codice poco complesso come quello proposto si può far risaltare la struttura parallela. Innanzitutto si può osservare che all'interno del processo *master* non viene svolto nessun calcolo per il computo del risultato, ma i comandi fanno sì che i dati vengano distribuiti in modo automatico tra i vari processi; a questo scopo si crea una variabile detta *offset* che costituisce il fattore di frazionamento dei dati: esso è regolato dal numero di processori coinvolti e si aggiorna per distribuire tutti i dati a disposizione. Il secondo compito del *master* è quello di ricevere i risultati e anche per fare ciò si introduce il contatore *offset* che consente di ordinarli per assemblare la matrice finale.

La struttura dei processi *slave* è invece diversa: ognuno è individuato da un numero di riconoscimento univoco (*rank*) maggiore di 0, essendo il valore 0 proprio del *master*. Essi in un primo momento ricevono i dati, quindi svolgono le computazioni e infine inviano i risultati in modo ordinato. La struttura è riportata nel codice:

```

/*-----*/
/****** worker task *****/
/*-----*/

if (taskid > MASTER)
{
    mtype = FROM_MASTER;

/*-----*/
/* Ricezione del valore iniziale di offset a cui inizia il processo*/
/*-----*/

```

```

MPI::COMM_WORLD.Recv(&offset,1,MPI::INT,MASTER,mtype,status);

/*-----*/
/* Ricezione del numero di righe da computare */
/*-----*/

    MPI::COMM_WORLD.Recv(&rows, 1, MPI::INT, MASTER, mtype,
status);

/*-----*/
/* Ricezione della matrice A a partire da offset */
/*-----*/

    MPI::COMM_WORLD.Recv(&a, rows*NCA, MPI::DOUBLE, MASTER,
mtype, status);

/*-----*/
/* Ricezione della matrice B e calcolo di C */
/*-----*/

    MPI::COMM_WORLD.Recv(&b,NCA*NCB,MPI::DOUBLE,MASTER,mtype,
status);

    for (k=0; k<NCB; k++)
        for (i=0; i<rows; i++)
            {
                c[i][k] = 0.0;
                for (j=0; j<NCA; j++)
                    c[i][k] = c[i][k] + a[i][j] * b[j][k];
            }
        mtype = FROM_WORKER;

/*-----*/
/* Invio del valore di offset alla fine della computazione */
/*-----*/

    MPI::COMM_WORLD.Send(&offset,1,MPI::INT,MASTER,mtype);

/*-----*/
/* Invio del numero di righe computate al master */
/*-----*/

    MPI::COMM_WORLD.Send(&rows,1,MPI::INT,MASTER,mtype);

/*-----*/
/* Invio della frazione finale di C */
/*-----*/

MPI::COMM_WORLD.Send(&c,rows*NCB,MPI::DOUBLE,MASTER,mtype);
}

/*-----*/
/* Chiusura di MPI */
/*-----*/

double stopTime = MPI::Wtime();

    MPI::Finalize();
}

```

Al centro del processo *worker* si possono riconoscere gli stessi due cicli utilizzati per il calcolo degli elementi della matrice prodotto anche nell'algoritmo sequenziale. Tuttavia è evidente come il codice abbia subito una notevole complicazione a causa della sovrastruttura di comunicazione tra i processi che si deve imporre affinché si possa sfruttare il parallelismo. Inoltre non sono stati utilizzati i puntatori, nonostante nell'algoritmo seriale si fosse dimostrata una tecnica utile ad accorciare i tempi di calcolo, perché si è verificato che l'algoritmo così ottimizzato non risultava poi scalabile se lanciato in parallelo. Tale comportamento è tipico dei programmi paralleli, per i quali spesso l'algoritmo che sfrutta meglio la *concurrency* risulta di solito molto meno efficiente se eseguito in seriale.

Il confronto con il miglior algoritmo seriale e l'analisi di scalabilità sono gli strumenti che consentono di determinare se la complicazione apportata risulti vantaggiosa date le dimensioni del problema e l'architettura di comunicazione.

L'algoritmo parallelo è stato lanciato sulla macchina multiprocessore del Dipartimento di Chimica, Materiali e Ingegneria Chimica "G. Natta" del Politecnico di Milano. La macchina è composta da 4 nodi *quadcore* e architettura SMP. I processori sono 4 Xeon E7340, 2.4GHz, 8MB, 1066 FSB e la macchina ha 32 GB di RAM a 667 MHz condivisa. Per valutare la dipendenza dalla dimensione del problema si è effettuata l'analisi di scalabilità su due casi: nel primo vengono moltiplicate tra loro due matrici quadrate di dimensioni  $1000 \times 1000$ , mentre nel secondo sono state utilizzate dimensioni  $5000 \times 5000$ . In entrambi i casi gli elementi delle matrici sono numeri reali in doppia precisione. I risultati delle prove sono riportati in Tabella 8:

**Tabella 8: Analisi di scalabilità dell'algoritmo parallelo per il prodotto tra matrici**

	$1000 \times 1000$			$5000 \times 5000$		
# Processi	Tempo [s]	Speedup	Efficienza	Tempo [s]	Speedup	Efficienza
<b>1</b>	13.86	1.00	100.00	1903.61	1.00	100.00
<b>2</b>	7.20	1.92	96.18	957.61	1.99	99.39

<b>3</b>	4.93	2.81	93.79	641.30	2.97	98.95
<b>4</b>	3.78	3.67	91.70	485.04	3.92	98.12
<b>5</b>	2.98	4.65	92.95	386.33	4.93	98.55
<b>6</b>	2.56	5.41	90.10	325.48	5.85	97.48
<b>7</b>	2.32	5.96	85.18	280.16	6.79	97.07
<b>8</b>	2.12	6.55	81.83	247.12	7.70	96.29

In entrambi i casi si ottiene una buona scalabilità, ma risulta evidente come essa sia favorita nel caso computazionalmente più impegnativo, ovvero quello in cui la matrice ha dimensione maggiore. Osservando il grafico in Figura 17:

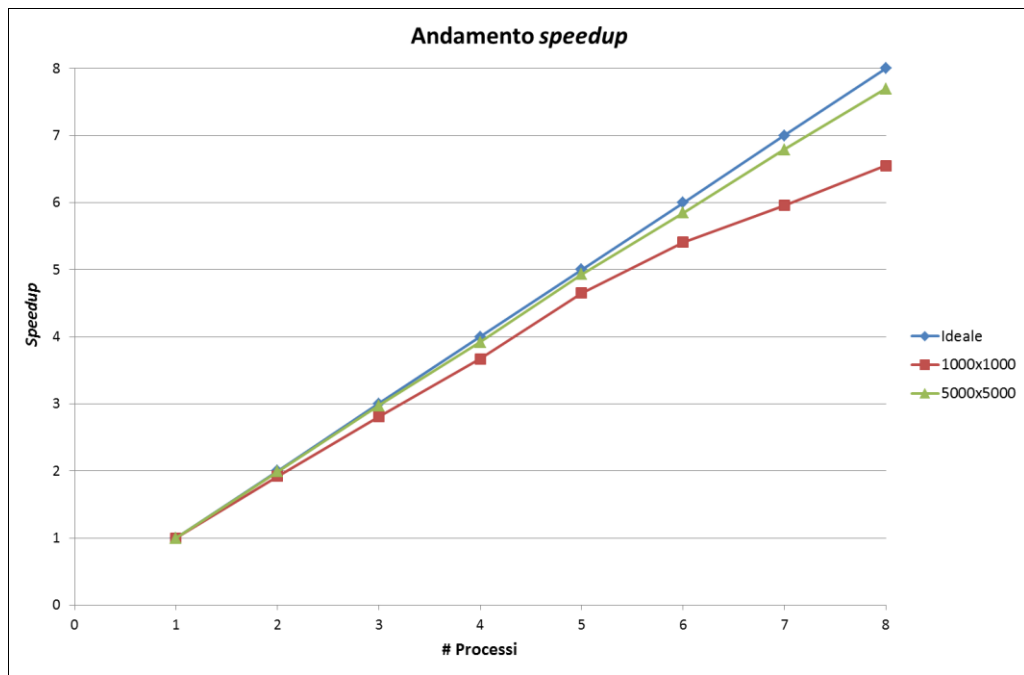


Figura 17: Confronto tra lo *speedup* massimo teorico (in blu) e quello ottenuto dalle analisi di scalabilità per l’algoritmo di moltiplicazione tra matrici

si nota che, come ci si aspettava, l’efficienza dell’algoritmo e conseguentemente lo *speedup*, diminuiscono all’aumentare del numero dei processi paralleli coinvolti, rimanendo fissata la dimensione del problema. La retta verde indica il limite teorico massimo per lo *speedup* e va come la bisettrice del quadrante, mentre la curva blu rappresenta lo *speedup* dell’algoritmo di moltiplicazione applicato alle matrici di dimensione 1000×1000 e quella rossa alle matrici

5000×5000. Dai risultati ottenuti si può affermare che il modello *master-slave* consente di sfruttare efficacemente la *concurrency* nel problema proposto, minimizzando le parti di codice non parallelizzate. Noto lo *speedup* e il numero di processi, si può ricavare un parametro rappresentativo della parallelizzabilità dell'algoritmo calcolando il peso della frazione di codice parallelizzabile come:

$$Par_{\%} = \frac{N_{\text{Procs}} - N_{\text{Procs}} / \text{Speedup}}{N_{\text{Procs}} - 1} \quad (5.5)$$

Tale parametro assume valore 1 se il codice è completamente parallelizzabile, e quindi se la scalabilità è lineare. La consistenza delle prove di scalabilità effettuate è ben confermata dai risultati del parametro di parallelizzabilità percentuale, come si vede in Tabella 9:

**Tabella 9: Frazione di codice parallelizzabile nei due casi**

# Processi	$Par_{\%}$ (1000×1000)	$Par_{\%}$ (5000×5000)
2	0.958333	0.994975
3	0.966192	0.994949
4	0.970027	0.993197
5	0.981183	0.99645
6	0.978189	0.994872
7	0.970917	0.994845
8	0.968375	0.994434

Infatti, a parità di dimensioni del problema il parametro si mantiene circa uguale al variare del numero di processi, a meno dell'incertezza sul *clock* dei processori, più evidente sul caso più piccolo perché i tempi di calcolo ne risentono molto di più, essendo decisamente più brevi. Questi risultati sono anche in linea con la legge di Amdahl, esposta nel paragrafo 2.7, laddove si mostrava nella Figura 12 come anche solo una frazione di codice non parallelizzabile del 5% bastasse a limitare significativamente lo *speedup*. Gli effetti sono evidenti anche nell'analisi di scalabilità descritta: l'algoritmo con peso computazionale minore

arriva ad avere una frazione non parallelizzabile del 3-4%, e oltre i primi processi mostra un crollo dell'efficienza rispetto al caso di dimensioni maggiori, per il quale la frazione non parallelizzabile, che fungerebbe da collo di bottiglia per la scalabilità risulta sempre inferiore all'1%.

I risultati delle prove dimostrano l'importanza della scelta dell'algoritmo parallelo se si vuole garantire l'efficienza del calcolo; basta infatti che non sia parallelizzabile una piccola parte di codice, o che la *concurrency* non sia pienamente sfruttata, che si perde il vantaggio a causa della forzata complicazione introdotta. Visti i buoni risultati ottenuti con la tecnica proposta si è poi voluto implementarla su di un caso più significativo rispetto al problema che è oggetto del presente lavoro di Tesi.

#### 4.4. *Parallelizzazione di un sistema con trasporto e reazione*

Prima di prendere in considerazione la parallelizzazione del KPP si è voluto verificare l'efficacia del modello *master-slave* su un caso più complesso del prodotto tra matrici, ma più gestibile in termini di dimensioni dell'algoritmo finale. Il caso considerato è un sistema con una griglia in due dimensioni in cui sono presenti due specie sottoposte ad un campo di moto e ad un termine di reazione molto semplificato. Il problema si può formulare come:

$$\begin{cases} \frac{du}{dt} = -\nabla(\nabla u - \beta u) + \frac{(u-v)}{\tau} = 0 \\ \frac{dv}{dt} = -\nabla(\nabla v - \beta v) + \frac{(v-u)}{\tau} = 0 \end{cases} \quad (5.6)$$

in cui  $\beta = [1,0]$ , il campo di moto è a divergenza nulla, mentre sui bordi valgono condizioni di flusso diffusivo nullo. Fisicamente si può interpretare come un sistema in cui il fluido entra per moto convettivo su uno dei lati ed esce, sempre solo per moto convettivo dal lato opposto. All'interno del dominio sussiste anche un campo di moto diffusivo. Il flusso della specie  $i$ -esima entrante o uscente dal dominio si può scrivere, a meno del segno, come:

$$F_i = \beta \times u_i \times n \quad (5.7)$$

in cui  $n$  è il versore normale uscente dalla frontiera in ogni punto dei lati di entrata o uscita. Tale sistema, sebbene formulato matematicamente in modo corretto, non rispecchia le condizioni reali di un bruciatore industriale, ma lo scopo dell'esempio studiato in questa fase non è quello di restituire dei risultati realistici, bensì verificare il comportamento del modello di comunicazione parallela su un sistema reagente e sottoposto ad un campo di moto. Il codice è stato scritto in C++ prima come algoritmo sequenziale e i risultati sono stati confrontati con un algoritmo di risoluzione preesistente scritto in ambiente Octave. Questo a sua volta ha fatto uso di tre librerie (De Falco e Culpo, 2010) scritte nel medesimo linguaggio e liberamente disponibili sul web:

- *Bim*, pacchetto per la soluzione di sistemi di equazioni differenziali alle derivate parziali relativi a sistemi caratterizzati da trasporto (convettivo e diffusivo) e reazione;
- *Fpl*, raccolta di routine per il salvataggio dei dati in formato grafico;
- *Msh*, libreria che crea e gestisce le griglie (strutturate e non) per sistemi a elementi finiti o a volumi finiti.

L'algoritmo di risoluzione formulato in Octave era basato sullo *splitting* dei termini convettivo/diffusivo da quello reattivo (in una forma del tipo *predictor-corrector* come descritto nei paragrafi 3.4 e 3.4.2), quindi comprendeva la soluzione di  $NS$  sistemi lineari sulla prima parte e di  $NR$  non lineari relativamente alla seconda.

Dal momento che il caso in questione era particolarmente semplice, la riscrittura in C++ è avvenuta in maniera procedurale anziché a oggetti, senza sfruttarne quindi appieno le potenzialità. Le funzioni contenute nelle librerie appena indicate, necessarie per il caso in questione, sono state anch'esse "tradotte" nel medesimo linguaggio e poste in un unico file di testo a valle del *main* (la parte centrale del codice, al cui interno tali funzioni sono richiamate). In totale è stato ottenuto un file di circa 1000 righe di codice, che ha riprodotto esattamente il comportamento (e i risultati) dell'algoritmo in Octave.

Infine, utilizzando una logica simile a quella utilizzata per il prodotto di matrici, si è convertito l'algoritmo per sfruttare le potenzialità del calcolo

parallelo. In questo caso la *concurrency* è stata individuata nella risoluzione del sistema non lineare del contributo reattivo. Infatti, come si è detto nel Capitolo 1, le frazioni massive delle specie coinvolte possono essere disaccoppiate tra i diversi reattori, perché nello *step corrector* si itera su ciascun reattore separatamente. Invece di risolvere un sistema globale composto da  $N_{nodi} \times N_{specie}$  incognite, è possibile risolvere  $N_{nodi}$  sistemi composti da  $N_{specie}$  reazioni ciascuno. Questa impostazione si presta al modello *master-slave* la cui efficacia è stata provata nell'esempio precedente: si può avere un processo adibito a distribuire i dati e ricevere i risultati in modo ordinato, mentre gli altri processi svolgono la computazione vera e propria per la risoluzione di una parte del sistema non lineare. Nel caso in esame rientra tra i compiti del processo *master* anche la parte *predictor* dell'algoritmo, che consiste nella risoluzione di un sistema lineare e che in questo caso non è stata parallelizzata.

Nel caso semplificato in cui sono presenti due sole specie, è stata condotta un'analisi di scalabilità sull'algoritmo così ottenuto, che ha fornito i risultati presentati in Tabella 10 (N.B.: data la semplicità del caso considerato, per diminuire il peso dell'incertezza di *clock* l'esecuzione del caso è stata iterata per 50 volte):

**Tabella 10. Risultati dell'analisi di scalabilità**

# Processi	$t_{tot}$ [s]
1 (seriale)	45.33
2	45.83
3	32.71
4	27.86
5	26.39
6	25.66
7	25.28
8	25.05

Per come è stato concepito il caso di esempio, nonostante la soluzione del sistema non lineare sia stata parallelizzata, la porzione di codice relativa al sistema lineare è rimasta sequenziale (ovvero affidata al *master*) per due motivi:

innanzitutto la delocalizzazione della risoluzione dei sistemi lineari in oggetto richiede il trasferimento di questi ultimi (ciascuno di dimensione  $N_{nodi} \times N_{nodi}$ ) da un processore all'altro, con un aumento significativo degli *overhead* di comunicazione. D'altro canto, trattandosi di un caso con due sole specie reattive, ciò diventa inutile perché per ogni iterazione sono risolti solo due sistemi lineari, per cui questi non possono essere distribuiti su più di due processori.

Ovviamente però questo rappresenta un limite alla parallelizzabilità dell'algoritmo: come affermato dalla già citata legge di Amdahl, basta che una piccola parte dell'algoritmo non sia parallelizzata (o parallelizzabile) perché si raggiunga rapidamente un asintoto nella diminuzione dei tempi di esecuzione.

Questo è tanto più vero nel caso considerato, in cui lo schema cinetico risulta semplificato e la risoluzione del sistema lineare non è ottimizzata. Come è possibile notare dalla Figura 18, si raggiunge un valore di "saturazione" ad un tempo pari a circa metà del tempo dell'esecuzione seriale:

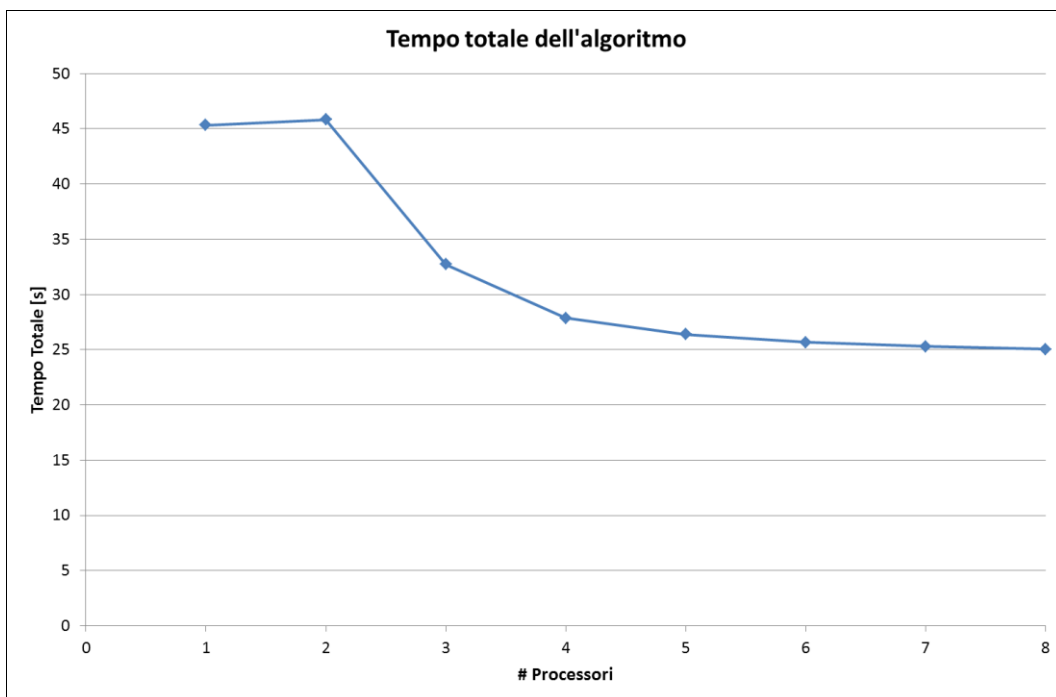


Figura 18: Andamento del tempo totale di esecuzione dell'algoritmo

L'andamento visualizzato mostra in maniera chiara (anche se enfatizzata, data la particolarità del caso considerato) i limiti dal punto di vista parallelo del metodo *Predictor-Corrector*: a parte le problematiche di convergenza già discusse nel

Capitolo 3, è evidente come la parte non parallelizzabile condizioni inesorabilmente la scalabilità dell'algoritmo. Ovviamente, in un caso più realistico caratterizzato da una griglia più grande e un numero maggiore di specie l'effetto negativo viene attenuato dalla delocalizzazione dei sistemi lineari su più processori, ma va anche considerato che per all'aumentare delle dimensioni della griglia la comunicazione per la trasmissione dei sistemi lineari aumenta significativamente.

In conclusione, dai due problemi-tipo affrontati nel presente Capitolo si evince come la struttura parallela prima creata per il prodotto tra matrici, e poi applicata allo step *corrector* del secondo algoritmo, si possa perfettamente adattare a processi iterativi, come quelli utilizzati nel KPP per giungere a convergenza. Tuttavia, in forma parallela essi saranno efficienti solo se si riuscirà a rendere minima sia la comunicazione che la parte non parallelizzabile, avvicinandosi per quanto possibile al caso *embarrassingly parallel* quale è il prodotto tra matrici. Nel Capitolo 5 l'applicazione del modello creato al KPP è descritto nel dettaglio.



## Capitolo 5

### Dal seriale al parallelo: la nuova forma del KPP

#### 5.1. *Struttura originale del Post-Processore Cinetico*

Come è stato già precisato nel Capitolo 1, il punto di partenza del presente lavoro di Tesi è consistito in uno strumento numerico preesistente, il *Kinetic Post Processor* (Cuoci et al., 2007). Tale strumento, applicando schemi cinetici dettagliati ad una *mesh* di reattori e partendo da una simulazione di primo tentativo ottenuta con schemi cinetici semplificati (dalla quale viene anche importato il campo di moto e di temperatura che viene resta fissato nel post-processamento), è in grado di prevedere con accuratezza soddisfacente le frazioni massive dei vari prodotti di reazione in condizioni stazionarie. Molti di questi infatti (tra cui spiccano gli  $\text{NO}_x$ ), che non sono presi in considerazione nello schema cinetico semplificato, sono prodotti in quantità dell'ordine delle parti per milione (*ppm*), tali da non influire significativamente sulla fluidodinamica del sistema.

Questo strumento è scritto in linguaggio C++ e ne sfrutta appieno le potenzialità e i vantaggi: come ben noto infatti, la sua caratteristica principale, che lo distingue dagli altri linguaggi ideati per il calcolo scientifico e numerico come il FORTRAN, è il suo essere *object-oriented*, ovvero basato su strutture di dati (gli oggetti) che interagiscono tra loro scambiandosi informazioni. Ciascun oggetto, definito all'interno di una classe, contiene un set standard di dati (o attributi) che ne definiscono completamente lo stato. Questi possono essere poi resi accessibili e/o modificabili dall'esterno tramite opportuni metodi a seconda delle esigenze, permettendo così un completo controllo della struttura creata.

I benefici di un tale approccio risultano evidenti in un problema altamente strutturato, quale è quello preso in considerazione nel presente lavoro di Tesi. Il sistema modellato è infatti costituito da una rete di reattori, ciascuno dei quali assimilabile a un CSTR (*Continuous Stirred Tank Reactor*), e pertanto

funzionante allo stesso modo, con le medesime variabili caratteristiche, sebbene in condizioni operative differenti l'uno dall'altro.

D'altro canto l'utilizzo del C++ risulta ottimale anche per via della gestione della memoria (che nella versione seriale dell'algoritmo è spesso risultata *lo stadio lento*) in maniera ottimizzata. A titolo di esempio, la scrittura di una matrice sparsa risulta molto più conveniente se si memorizzano solo gli elementi non nulli, in quanto ciascuno di essi è completamente definito da tre valori: i due indici che ne localizzano la posizione nella matrice e il suo valore numerico. Ancora una volta, con una logica "ad oggetti", quanto sopra diventa immediatamente applicabile con ottimi risultati dal punto di vista computazionale.

Non per niente lo "scheletro algebrico" del Post Processore è costituito dalle librerie *BzzMath* (Buzzi-Ferraris, 1994), anch'esse scritte in C++ con una logica decisamente *object-oriented*, le cui classi permettono di gestire in maniera robusta ed efficiente problemi numerici di vario tipo, tra cui la risoluzione di sistemi algebrici, differenziali ed algebrico-differenziali. Esse sono un progetto in continuo miglioramento, e sono liberamente scaricabili dal web.

Per quanto riguarda l'algoritmo in sé, nella sua versione originale è costituito da circa 6000 righe di codice, distribuite in una serie di classi (una decina in totale) e in un nucleo centrale chiamato *main*. Le classi:

- Gestiscono la rete di reattori, importando i valori di composizione di primo tentativo e la topologia del sistema da file di testo ottenuti tramite *Fluent*, assemblando la rete finale e regolando la comunicazione tra le celle, portando infine il sistema a convergenza tramite i vari metodi descritti nel Capitolo 3;
- Caratterizzano completamente il singolo reattore, il quale riceve dalla rete le sue condizioni iniziali e la topologia, dopo di che provvede, nel caso delle metodologie risolutive descritte nei Paragrafi 3.3 e 3.4.2 (per quanto riguarda lo step *corrector*), al raggiungimento della convergenza locale, mediante risoluzione del sistema lineare o, qualora ciò sia insufficiente, tramite imposizione di un falso transitorio sino al raggiungimento della convergenza;
- Calcolano i residui del sistema, valutandone norma infinita, norma 1 e norma 2 e verificando quindi quanto si sia distanti dalla convergenza;

- Gestiscono infine la cinetica del sistema, nonché i metodi risolutivi (globali e locali) e creano un'interfaccia con l'utente mediante l'utilizzo di un file di input, al cui interno possono essere modificati i percorsi dei file da importare e possono essere regolate varie opzioni (come è già stato descritto nel Paragrafo 3.5).

Questa struttura è stata mantenuta anche nella versione parallela del codice: dal momento che nella classe che gestisce la rete di reattori sono presenti tutte le informazioni relative alla comunicazione tra celle, essa si è rivelata particolarmente flessibile ed utile anche per una distribuzione dei dati su più processi (o processori) comunicanti.

## ***5.2. Design parallelo dell'algoritmo***

Nel Capitolo 2 è stato spiegato come, a monte del ripensamento in forma parallela di un qualsiasi algoritmo, quest'ultimo vada analizzato a fondo in modo da individuare le dipendenze delle singole operazioni dai dati (*data dependency*), e di conseguenza le operazioni che possono essere eseguite contemporaneamente ottenendo gli stessi risultati del programma sequenziale per tutti i possibili valori di input. Fatto ciò, le operazioni tra loro indipendenti possono essere distribuite a più processi, i quali condividono la stessa memoria e/o si scambiano informazioni mediante lo standard MPI.

Ciò è avvenuto anche nel caso del Post-Processore: il problema affrontato, in questo caso, ben si presta ad un approccio parallelo *object-oriented* in quanto coinvolge una griglia di reattori, a ciascuno dei quali corrisponde un rispettivo oggetto. Inoltre, come è stato già discusso nel Capitolo 3, l'algoritmo sequenziale prevede, per avvicinarsi alla convergenza, l'utilizzo di metodi locali che operano indipendentemente su ogni singola cella, aggiornando tra un'iterazione e l'altra le condizioni locali del sistema (ovvero le frazioni massive). Quest'ultima osservazione è stata alla base della strategia adottata per la parallelizzazione, che è di seguito descritta.

### 5.2.1. Scelta del modello di parallelizzazione adeguato

La struttura di un programma parallelo è basata su processi e *thread*, e sulla loro tipologia di organizzazione, scelta dall'utente in fase di progettazione dell'algoritmo. Per fare un esempio, come è stato già esposto nel Paragrafo 2.3, l'utilizzo di OpenMP in sistemi a memoria condivisa prevede una creazione (e distruzione) dinamica dei *thread*, secondo il modello *fork-join*. Questo non è possibile, invece, nella versione originale dello standard MPI (dalla 1.0 alla 1.2): la creazione e gestione dei processi è di tipo statico, e il loro numero non può essere modificato in corso d'opera. Diverso è il discorso per le versioni successive di MPI (dalla 2.0 in poi), che permettono invece ciò, ma la cui implementazione è più complessa della precedente.

D'altro canto, per quanto non presentino problemi di *overhead* di comunicazione, i soli sistemi a memoria condivisa possiedono dei limiti tecnologici legati alla dimensione della memoria RAM tali per cui problemi come quello in considerazione nel presente lavoro di Tesi rendono spesso indispensabile l'utilizzo di sistemi a memoria distribuita, e quindi di più processi tra loro comunicanti. Non va comunque dimenticato che il protocollo MPI è compatibile anche con sistemi a memoria condivisa, per cui un primo *benchmark* dell'algoritmo in forma parallela (e su casi di test dalle dimensioni ridotte) è possibile anche su questi sistemi (come è avvenuto nell'esempio del prodotto tra matrici in forma parallela, descritto nel Paragrafo 4.3).

Fatta questa scelta, è opportuno valutare le gerarchie relative tra i vari processi in esecuzione. In linea di principio infatti, lo standard MPI prevede che i processi siano tra loro indipendenti e allo stesso livello. Tramite opportune operazioni di sincronizzazione e comunicazione è comunque possibile creare una dipendenza relativa di un processo dall'altro, ottenendo modelli di parallelismo differenti (Rauber et al., 2010).

Nel caso in cui sia possibile distribuire equamente il carico di lavoro tra i vari processi, il già menzionato modello *Master-Slave* risulta particolarmente efficiente: in questo caso, uno solo dei processi esegue le funzioni principali del

programma, controlla e coordina gli altri processi ricevendo ed inviando loro le informazioni necessarie per proseguire l'esecuzione dell'algoritmo.

Per tale motivo la scelta del modello *Master-Slave*, già rivelatasi vincente nel caso del prodotto tra matrici, è stata mantenuta anche nel più complesso Post-Processore Cinetico. Come sarà meglio descritto nel Paragrafo 5.2.3, esso ha tratto vantaggio proprio dalla possibilità di avvicinarsi alla soluzione tramite metodi "locali", distribuendo così dati della *mesh* e carico computazionale sui singoli *Slave*.

### 5.2.2. *Distribuzione dei dati tra i processi*

L'unità principale della *mesh* modellata dal Post-Processore è costituita dal reattore: questo, definito in C++ come un oggetto all'interno di una rete (un oggetto a sua volta), è completamente caratterizzato nella sua composizione iniziale e nella sua topologia all'interno della griglia tramite dati ottenuti da due file di testo esportati da *Fluent*. Ciascun reattore possiede inoltre un indice relativo alla sua posizione all'interno dell'*array* di oggetti "reattore" e un indice assoluto corrispondente alla sua posizione nella griglia. Come è logico attendersi, in un algoritmo puramente sequenziale essi coincidono.

La parallelizzazione dell'algoritmo parte proprio da qui: anziché creare un unico *array* di reattori, in ogni processo viene allocato un numero di reattori inferiore, ottenuto distribuendo equamente il numero totale tra i diversi *slave* secondo la stessa logica del prodotto tra matrici:

```
int numworkers = nprocs - 1;           //Calcolo degli slave
avecells = NR_/numworkers;             //Numero medio di reattori
extra = NR_%numworkers;                //Resto della divisione
NR_P[0] = 0;
offset[1] = 0;
offset[nprocs_] = NR_;
for(dest = 1; dest <= numworkers_; dest++)
{
    NR_P[dest] = (dest <= extra) ? avecells + 1 : avecells;
    if(dest != numworkers_)
        offset[dest+1] = offset[dest] + NR_P[dest];
}
```

Per esempio nel caso di 7 processori e 2360 celle, sui primi 2 *slave* sono allocati 394 reattori, mentre sugli altri 4 ne sono allocati 393.

La distribuzione della griglia sui diversi processi può avvenire in due modi differenti: a blocchi o in maniera alternata. In Figura 19 sono rappresentate le due differenti logiche di distribuzione.

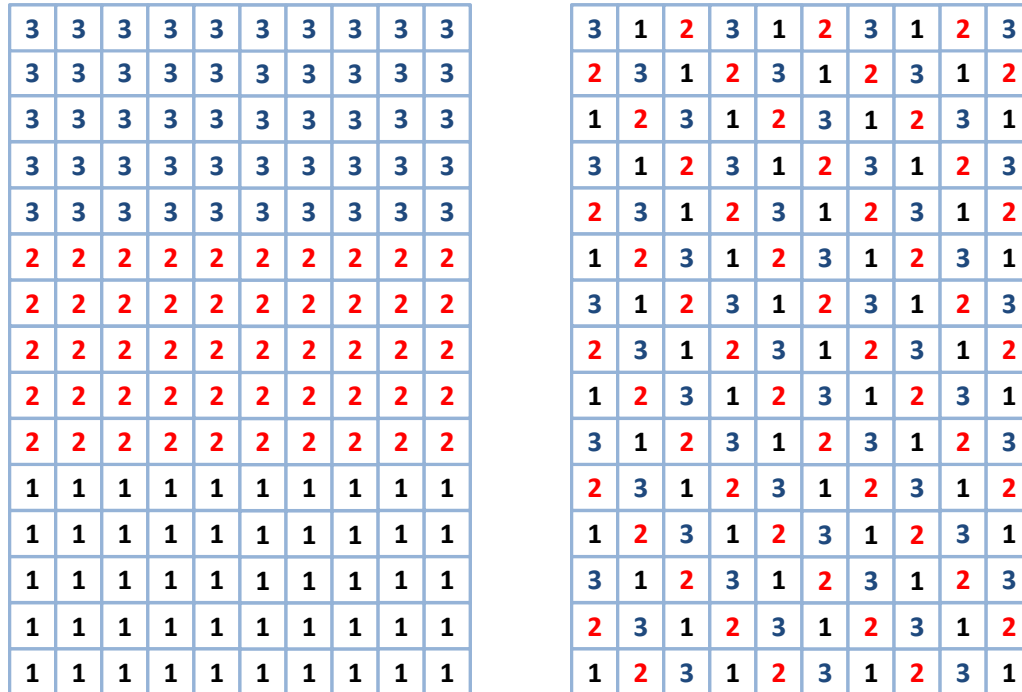


Figura 19: Allocazione della griglia a blocchi (sinistra) o in maniera alternata (destra) con 3 slave a disposizione

Entrambe le procedure sono state implementate; la scelta finale è stata presa dopo averle provate entrambe e averne valutato l'efficienza, come sarà descritto nel seguito.

Per quanto riguarda la costruzione della rete, ovvero l'assemblaggio della matrice di convezione-diffusione, la definizione della comunicazione cella-cella, la correzione dei flussi massivi e il calcolo dei residui iniziali, tutto ciò è gestito dal *master*, il quale tramite operazioni di comunicazione riceve le informazioni necessarie dai vari *slave*, li rielabora e, ove sia necessario (come nel caso della topologia della rete), re-invia nuovi dati ai relativi processi.

Allo scopo, nell'algorithm in forma parallela si fa uso del *class template* "vector", facente parte della *Standard Template Library (STL)* del C++ e funzionante in maniera dinamica. Le ragioni di tale scelta sono multiple: innanzitutto, funzionando in maniera dinamica la sua capacità non è prefissata in

fase di compilazione, ma varia a seconda delle necessità. Inoltre, in questo modo viene meno il rischio di *Stack Overflow* per variabili di dimensioni molto grandi, che andrebbero altrimenti dichiarate come *static* o al di fuori del *main*. Ultimo, ma non meno importante, i vettori dichiarati in questa forma ben si prestano ad essere trasmessi da un processo all'altro mediante comunicazione: come già detto, il KPP utilizza gli oggetti appartenenti alle classi delle librerie *BzzMath* per vettori e matrici. Dall'altra parte invece, lo standard MPI accetta soltanto tipi predefiniti di dati, tra cui figurano ovviamente *int* e *double*; non è possibile trasmettere, in maniera immediata, oggetti appartenenti a classi definite dall'utente. Per tale motivo, nella versione parallela del codice sono stati creati *array* "di servizio" per consentire la trasmissione delle informazioni contenute negli oggetti della sopraccitata libreria da un processore all'altro.

Una volta ottenuta una distribuzione adeguata della rete su più *slave* è possibile applicare i metodi risolutivi "locali" già citati nel Capitolo 3 per avvicinare il sistema alla convergenza. Nel seguito ciò è descritto con maggior dettaglio.

### **5.2.3. Approccio parallelo ai metodi risolutivi**

Come già spiegato nel Paragrafo 3.2, per sistemi fluidodinamici di grandi dimensioni la limitazione posta dai metodi globali è duplice: da un lato, infatti, l'occupazione di memoria richiesta dalla memorizzazione di una matrice per la risoluzione di un sistema *ODE* o algebrico non lineare va tenuta in conto (allo stato attuale, sistemi con più di 100.000 ÷ 150.000 celle non possono essere risolti "globalmente" per via dei limiti tecnologici sulla *RAM*); dall'altro, la fattorizzazione di una matrice richiede un costo computazionale proporzionale al cubo della dimensione della matrice, per cui i metodi di fattorizzazione diretti diventano sempre più pesanti all'aumentare delle dimensioni della matrice.

È possibile ovviare a queste limitazioni in due modi: se si vuole restare nell'ambito dei metodi globali è possibile utilizzare dei risolutori, diretti o iterativi, in grado di delocalizzare il carico, computazionale e di memoria, su più processori in sistemi a memoria distribuita. Allo scopo, è possibile utilizzare delle librerie, come *MUMPS* o *LIS*, in grado di risolvere i sistemi succitati

rispettivamente in maniera diretta o iterativa (si veda il Capitolo 6 per maggiori informazioni).

È comunque possibile (anzi, indispensabile) avvicinarsi alla convergenza mediante metodi iterativi locali, risolvendo i bilanci locali dei reattori in sequenza oppure mediante opportuni metodi di *splitting* come il *Predictor-Corrector*. Il primo metodo, in particolare, risulta perfettamente parallelizzabile in quanto la risoluzione del sistema (*ODE* o non lineare) è locale, ed ogni cella è indipendente dalle altre. Anche il secondo metodo si presta bene ad un approccio parallelo, per quanto richieda un *overhead* di comunicazione maggiore per via dello *step Predictor*, in cui vengono risolti  $NS$  sistemi lineari di  $NR$  equazioni in  $NR$  incognite, sistemi che vanno trasferiti per intero sugli *slave* se ne si vuole delocalizzare la risoluzione. D'altro canto, i risultati non soddisfacenti ottenuti col *Predictor-Corrector* sull'algoritmo seriale, già descritti nel Capitolo 3, hanno portato a concentrarsi sulla risoluzione della sequenza di reattori.

I passaggi fondamentali di questo metodo sono essenzialmente tre: dapprima, in base alla topologia della rete, ovvero in base alle composizioni delle celle circostanti, i flussi massivi delle singole specie sono aggiornati cella per cella; quindi i sistemi di equazioni locali sono portati a convergenza; infine sono calcolati i residui globali, in termini di norma infinita, norma 1 e norma 2, onde verificare la distanza dalla soluzione.

Le modifiche apportate a questa sequenza di operazioni sono state due. Per consentire l'aggiornamento dei flussi massivi, è necessario che ogni cella conosca le condizioni delle celle adiacenti in termini di composizioni; nel caso di una suddivisione della griglia a blocchi, le celle confinanti non conoscono le condizioni di parte delle celle adiacenti, che vanno quindi trasmesse tramite opportuna comunicazione. La strategia adottata in questo caso consiste nella trasmissione delle frazioni massive di tutte le celle al *master*, e nella successiva comunicazione da parte del *master* soltanto delle composizioni delle celle di bordo tra un processo e l'altro a tutti gli *slave*. In Figura 20 è rappresentata tale logica di comunicazione.

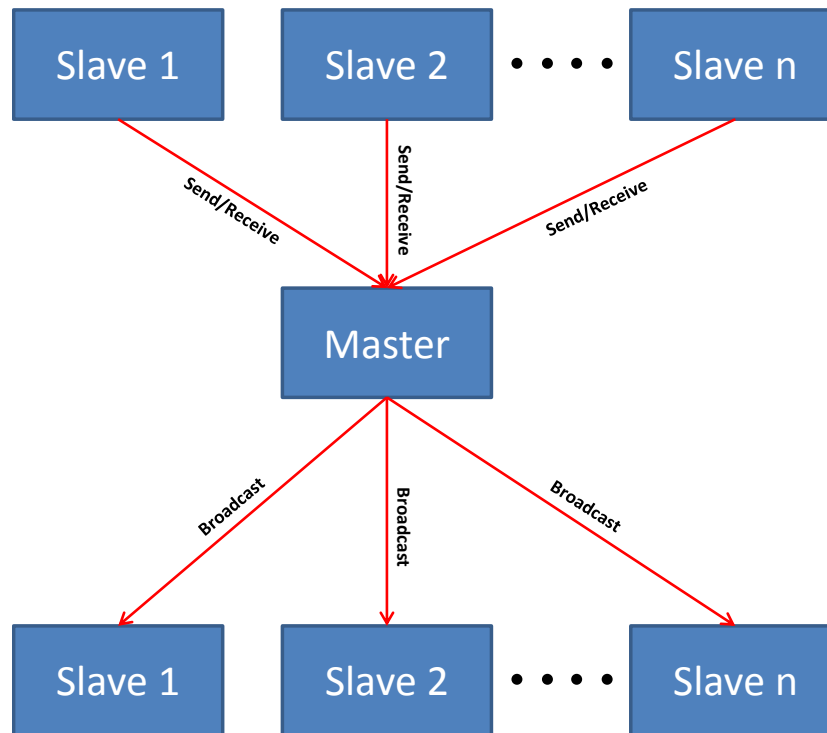


Figura 20: Logica di comunicazione delle frazioni massive

Nel caso in cui la griglia venga invece distribuita in maniera alternata sui diversi *slave*, la comunicazione richiesta sarà invece in quantità maggiore, in quanto ogni cella comunica con celle appartenenti a processi diversi ed ogni processore deve quindi conoscere le frazioni massive di tutte le altre celle. In quest'ultimo caso, la logica di Figura 20 resta invariata, con la differenza che il *Broadcast* riguarda le frazioni massive di tutte le celle del sistema.

La seconda modifica all'algoritmo sequenziale, meno significativa della precedente, riguarda il calcolo dei residui: poiché quest'operazione, se effettuata in parallelo necessita comunque di comunicazione tra processi differenti, si è scelto di non effettuarla per ogni iterazione della sequenza, ma ogni  $n$  iterazioni, con  $n$  valore prefissato inserito dall'utente in fase di compilazione.

In questo modo gli *overhead* di comunicazione sono ridotti al minimo, fermo restando la coerenza dei risultati finali ottenuti con quelli dell'algoritmo sequenziale.

#### 5.2.4. Load Balancing e distribuzione ottimale delle celle

La parallelizzazione della risoluzione della sequenza di CSTR ha posto due alternative per quanto riguarda l'allocazione dei reattori sui differenti processori. La scelta migliore è stata effettuata in base a prove di scalabilità dell'algoritmo creato, utilizzando un numero crescente di processori su una macchina dotata di processore *Intel® Core™ i7* con frequenza di *clock* pari a 3.07 GHz per ciascuno dei 4 *core*. Dal momento che il numero di processori è pari a 4, sono state valutate, in termini di tempo impiegato, le prestazioni dell'algoritmo al variare del numero di *slave* da 1 a 3.

Di seguito sono rappresentati i risultati forniti dai due algoritmi per quattro casi di studio: i primi tre, rispettivamente a 590, 2360 e 9440 celle, analoghi a quelli descritti nel Capitolo 3, e l'ultimo costituito da una fiamma con *Bluff Body* a 24575 celle. Lo schema cinetico adottato è il *Fluent\_DRM22\_Polimi\_NOx*, costituito da 35 specie e 155 reazioni. Il tempo è misurato in secondi.

**Tabella 11: Prestazioni dell'algoritmo parallelo con distribuzione delle celle a blocchi**

# Celle	# Slave		
	1	2	3
<b>590</b>	30.04	27.21	23.15
<b>2360</b>	436.60	404.22	339.41
<b>9440</b>	4196.32	3859.06	3330.36
<b>24575</b>	39234.40	24070.30	17769.00

**Tabella 12: Prestazioni dell'algoritmo parallelo con distribuzione delle celle in maniera alternata**

# Celle	# Slave		
	1	2	3
<b>590</b>	28.52	14.79	10.35
<b>2360</b>	435.24	224.13	153.56
<b>9440</b>	4104.63	2145.14	1476.71
<b>24575</b>	39234.40	19791.10	13521.50

Considerando come riferimento la versione sequenziale (*master* con un solo *slave*) in quanto si vuole verificare la scalabilità dell'algoritmo, è possibile valutare lo *speedup* in entrambi i casi secondo l'espressione fornita nel Capitolo 2:

**Tabella 13: *Speedup* ottenuto con l' algoritmo a blocchi**

# Celle	# Slave		
	1	2	3
590	1.00	1.10	1.30
2360	1.00	1.08	1.29
9440	1.00	1.09	1.26
24575	1.00	1.63	2.21

**Tabella 14: *Speedup* ottenuto con l' algoritmo alternato**

# Celle	# Slave		
	1	2	3
590	1.00	1.93	2.76
2360	1.00	1.94	2.83
9440	1.00	1.91	2.78
24575	1.00	1.98	2.90

È possibile notare come nel primo caso la scalabilità dell'algoritmo sia molto bassa, in particolare per la prima fiamma (e in tutti e tre i restanti casi). Risultati nettamente migliori si raggiungono invece con il secondo algoritmo, in cui lo *speedup* ottenuto non solo è vicino all'idealità, ma risulta molto simile per tutte e due le fiamme.

La ragione di questo differente comportamento risiede essenzialmente nel bilanciamento della distribuzione del carico computazionale tra i vari processori (noto come *Load Balancing*): suddividendo la *mesh* di reattori a blocchi, è più facile che un blocco di reattori impieghi un tempo maggiore rispetto agli altri per raggiungere la convergenza locale, probabilmente per via di condizioni fluidodinamiche localmente più "proibitive". Gli altri processori quindi, una volta completata la propria parte di lavoro restano inattivi in attesa dello stadio più lento. Poiché il tempo impiegato per concludere un'iterazione è governato dal processore più lento, la conseguenza è un notevole peggioramento dell'efficienza dell'algoritmo in forma parallela.

Ciò non accade nel secondo caso: allocando i reattori in maniera alternata sui vari processi, è "statisticamente" più facile che il carico computazionale sia maggiormente bilanciato su tutti i reattori, per cui il tempo di inattività dei processi che hanno terminato la propria parte di lavoro è ridotta al minimo, con

ottimi risultati in termini di *speedup* ed efficienza. Per tale motivo la scelta è caduta sul secondo algoritmo, che nel seguito sarà quello su cui saranno effettuati i test su sistemi di più grandi dimensioni (e dalla fluidodinamica più complicata) e su macchine con un numero maggiore di processori, a memoria condivisa ed anche distribuita.

### **5.3. Ottimizzazione della convergenza della sequenza di CSTR**

Oltre al nuovo approccio parallelo al problema, è possibile migliorare ulteriormente la velocità di convergenza del singolo reattore scrivendo la relativa funzione in maniera più ingegnosa.

Come già detto in precedenza, il sistema di  $NS$  equazioni caratterizzanti la singola cella è del tipo:

$$\left[ -C\bar{\omega} + \bar{f} \right]_{old} + R(\bar{\omega}) = 0 \quad (6.1)$$

sistema che, a meno che non si parta da un set di valori di primo tentativo sufficientemente vicino alla soluzione, difficilmente può essere risolto come un sistema non lineare, dal momento che sono in gioco specie radicaliche (dalle concentrazioni bassissime) e non.

Perciò la procedura risolutiva prevede di utilizzare un falso transitorio, passando dal precedente sistema lineare al corrispondente sistema di equazioni differenziali (4.8). Nella versione tradizionale del Post-Processore tale sistema viene integrato tramite un risolutore *stiff* per un intervallo di tempo standard, pari solitamente a  $10s$ , in grado di permettere il raggiungimento dello stazionario, fermandosi prima se il residuo raggiunto è minore di  $10^{-8}$ . Questo metodo, molto più robusto di un risolutore non lineare, risulta però estremamente più lento, con la conseguenza che l'efficienza finale della risoluzione della sequenza di reattori ne risente significativamente.

D'altra parte, non sempre tale metodo risulta indispensabile: partendo infatti da condizioni iniziali piuttosto vicine alla soluzione è possibile che un risolutore non lineare riesca a raggiungere la convergenza, e questo in un tempo di gran lunga inferiore rispetto a quello impiegato dal risolutore *ODE*.

Una strategia pertanto efficiente per la risoluzione dei singoli bilanci dei reattori consiste nell'aggiunta, all'interno della funzione risoltrice originale che prevedeva unicamente l'integrazione del sistema *ODE*, nella possibilità di sfruttare il metodo di Newton per giungere a convergenza. In particolare, nella sua ultima versione il Post-Processore risolve i bilanci locali tramite la seguente sequenza di operazioni:

1. Se si tratta della prima iterazione in assoluto, tutte le celle vengono risolte in maniera robusta, ovvero tramite integrazione del relativo sistema differenziale;
2. Vengono calcolati i residui della singola cella;
3. Se i residui locali sono inferiori rispetto ad un valore sufficientemente basso (e.g.  $norm1 < 10^{-16}$ ) la soluzione precedente è considerata soddisfacente e non viene pertanto aggiornata;
4. In caso contrario, si tenta il metodo di Newton e ne si verifica l'avvenuta convergenza;
5. Se il metodo di Newton non ha avuto successo, il sistema viene risolto tramite falso transitorio e relativa integrazione, per un intervallo di tempo definito nel file di *input*;
6. Si ritenta con il metodo di Newton e ne si verifica la convergenza;
7. Si aggiornano i residui;
8. Se la soluzione trovata non risulta ancora soddisfacente, si procede per una seconda volta con l'integrazione del sistema *ODE*.
9. Infine si tenta la risoluzione della singola cella mediante metodo di Newton iterativamente, per un numero massimo di iterazioni definito sempre nel file di *input*. Se non si ottiene un valore soddisfacente dei residui, la convergenza è considerata "non raggiunta", e ciò viene segnalato a schermo come *Convergence Failure* (anche se ciò non interrompe la risoluzione della sequenza).

I miglioramenti in termini di tempo ottenuti con la nuova forma di questa funzione sono significativi: di seguito sono illustrati i risultati ottenuti per i quattro casi di studio, comparati con i vecchi, eseguiti sul calcolatore già citato al Paragrafo 5.2.4.

**Tabella 15: Confronto tra i tempi di esecuzione dei due algoritmi con e senza ottimizzazione della risoluzione della sequenza**

	Tempi per Caso studiato [s]			
	590	2360	9440	<i>BluffBody</i> 24575
<b>Base</b>	32.2815	464.349	5186.26	42441.4
<b>Ottimizzata</b>	20.6149	290.411	2810.22	30537.8
Rapporto	1.56593	1.598937	1.8455	1.389799

È possibile notare come per la prima fiamma il miglioramento sia crescente col numero di celle del sistema modellizzato: nella sua nuova versione, l'algoritmo impiega circa un terzo del tempo in meno rispetto al precedente, a vantaggio dell'efficienza globale dell'algoritmo.

È opportuno infine sottolineare come la modifica della risoluzione del singolo reattore non abbia alcuna influenza sul grado di parallelismo dell'algoritmo; la risoluzione dei CSTR, comunque venga eseguita, avviene localmente e non ha nulla a che vedere con gli *overhead* di comunicazione o con il *Load Balancing*, dal momento che il carico computazionale è distribuito in modo "statisticamente" bilanciato tramite l'allocazione dei reattori in maniera alternata sui vari *slave*.

#### **5.4. Il KPP in sistemi a memoria distribuita**

La parallelizzazione del Post-Processore è volta, come già detto, a distribuire il carico computazionale su di un numero sufficientemente elevato di processori per abbattere il tempo di calcolo totale. Per tale motivo si è scelto di utilizzare le librerie MPI, progettate appunto per consentire lo scambio di informazioni tra processi che non condividono lo stesso spazio di memoria; ed è per lo stesso motivo che si è deciso di tralasciare l'utilizzo dell'interfaccia di comunicazione *OpenMP* che permette, attraverso i *thread*, di operare in memoria parallela su sistemi *multicore* con *overhead* di comunicazione trascurabile, in quanto trattasi di sistemi a memoria condivisa.

Ben diverso è il discorso che riguarda invece sistemi costituiti da più unità operative (ciascuna provvista di un proprio disco rigido, memoria e processore/i) tra loro interconnesse, ovvero a memoria distribuita: in questo caso la comunicazione diventa indispensabile, in quanto unico modo per trasferire dati tra memorie fisicamente separate tra di loro. Questi sistemi risultano gli unici in

grado di scalare il sistema su un numero sufficientemente elevato di processori, e gli unici pertanto utilizzabili per problemi di dimensioni enormi, non eseguibili su un singolo nodo in quanto limitati dalla massima RAM disponibile.

Sullo stesso principio possono essere creati sistemi a memoria distribuita su piccola scala, connettendo tra loro due comuni computer mediante le rispettive schede di rete e un cavo *ethernet* incrociato (*crossover*). In questo modo si può dare vita a una rete *peer-to-peer*: ciascuno dei due computer è identificato tramite un indirizzo IP locale ed è riconosciuto tramite questo dall'altro nodo. In particolare, la connessione e lo scambio dati avviene tramite il protocollo di rete *ssh*, che permette di stabilire una connessione remota tra due nodi. Il tutto può essere implementato, ad esempio, mediante *OpenSSH*, per cui uno dei due *host* opera come *master*, mentre l'altro fa da *slave*, come illustrato in Figura 21:

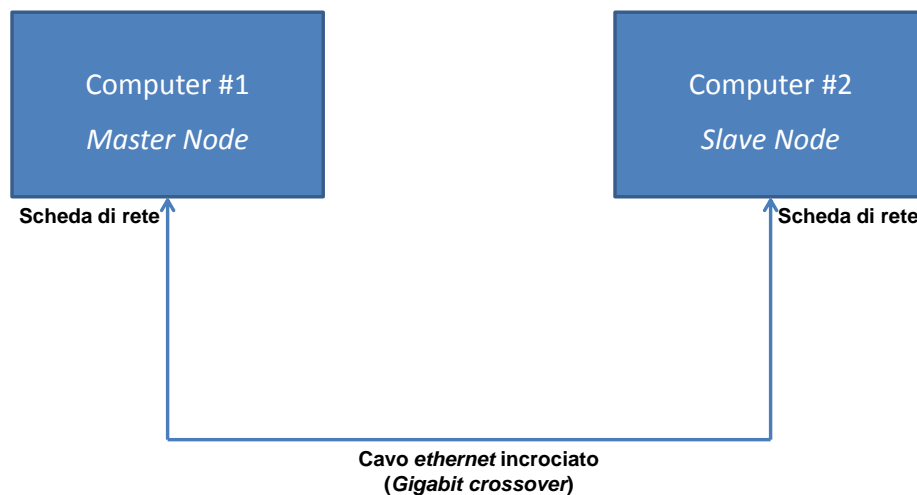


Figura 21: Schema della rete creata tramite protocollo *ssh*

In questo modo è possibile verificare il corretto funzionamento di un programma parallelo in un sistema reale a memoria distribuita, in quanto i dati vengono “fisicamente” trasmessi da un computer all'altro.

Quanto appena spiegato è stato provato con successo sul Post Processore per diversi casi fluidodinamici: si è potuto riscontrare come i risultati numerici siano identici a quelli ottenuti su sistemi a memoria condivisa, per cui si può concludere che l'algoritmo è perfettamente eseguibile anche su *cluster* di grandi dimensioni, in cui sarà finalmente possibile verificarne le prestazioni in termini di efficienza e scalabilità.

### 5.5. Prestazioni dell'algoritmo parallelo

Nella sua versione finale il Post Processore Cinetico distribuisce i reattori tra i vari processori, allocandoli in maniera alternata in base all'indice da essi posseduto nella griglia fluidodinamica precedentemente importata da *Fluent*. Seguendo un modello del tipo *master/slave*, i processori sui quali i reattori sono allocati (gli *slave* appunto) si scambiano informazioni inviandole ad un unico processore (il *master*), il quale provvede a ridistribuirli.

Il metodo utilizzato per raggiungere la convergenza è un metodo locale, consistente nella risoluzione dei bilanci dei singoli CSTR in sequenza e nell'aggiornamento iterativo delle frazioni massive di tutte le specie finché i residui ottenuti non diventano sufficientemente bassi (la condizione di stop di default è che la norma 1 sia minore di  $10^{-12}$ ). Tale risoluzione avviene in maniera intelligente, evitando di risolvere il sistema differenziale se è possibile risolvere il sistema non lineare corrispettivo, o addirittura evitando di risolverlo se le condizioni di partenza sono soddisfacenti.

Nella forma appena esposta, l'algoritmo è stato compilato e il relativo programma eseguito sulla stessa macchina impiegata per effettuare il *benchmark* del prodotto tra matrici descritta al Paragrafo 4.3. I casi di studio testati sono costituiti dalle stesse due fiamme considerate in precedenza (si veda, a proposito, il Paragrafo 5.2.4). Di seguito in Tabella 16 è indicato il tempo, in secondi, impiegato da ciascun programma per raggiungere la convergenza nei vari casi.

**Tabella 16: Benchmark dell'algoritmo parallelo al variare del caso e del numero di processori**

Caso	# Slave						
	1	2	3	4	5	6	7
590	39.48	20.10	13.64	10.65	9.21	7.63	7.04
2360	534.76	274.88	186.52	147.30	126.26	109.41	102.54
9440	5310.22	2769.58	1894.14	1513.72	1317.78	1145.58	1093.10
24575	57183.00	29237.70	19915.40	15209.40	12664.60	11105.10	9989.85

Da questi dati è possibile effettuare valutazioni sotto due differenti punti di vista: da un lato, infatti, essi possono essere confrontati con gli analoghi risultati ottenuti dal codice seriale preesistente, onde valutarne l'effettivo guadagno in termini di efficienza; dall'altro si può valutare, caso per caso, il comportamento

dell'algoritmo parallelo al variare del numero di *slave* in termini di *speedup* e di efficienza.

### 5.5.1. Confronto con l'algoritmo originale

Per quanto riguarda il primo aspetto, i dati di cui sopra vanno confrontati con quelli della Tabella 7, che per completezza sono qui riportati con l'aggiunta del caso della fiamma a 24575 celle:

**Tabella 17: Tempi di esecuzione dell'algoritmo originale per i quattro casi di studio**

Caso	Tempi [s]
590	21.03
2360	197.12
9440	2973.02
24575	19384.89

Come previsto, l'algoritmo in forma seriale risulta migliore per un numero relativamente basso di processori; tuttavia, già con tre processori si ottiene l'incrocio tra i tempi dell'algoritmo parallelo e quelli del seriale. In Tabella 18 sono riportati i rapporti relativi tra i tempi di esecuzione dei due codici.

**Tabella 18: Rapporti tra i tempi relativi di esecuzione dei due algoritmi**

Caso	# Slave						
	1	2	3	4	5	6	7
590	1.88	0.96	0.65	0.51	0.44	0.36	0.33
2360	2.71	1.39	0.95	0.75	0.64	0.56	0.52
9440	1.79	0.93	0.64	0.51	0.44	0.39	0.37
24575	2.95	1.51	1.03	0.78	0.65	0.57	0.52

In generale, da quattro processori in su il nuovo algoritmo risulta più rapido, e la differenza diventa sempre più evidente all'aumentare del numero di *slave*.

È possibile inoltre fare due ulteriori considerazioni sugli andamenti osservati: innanzitutto, esaminando i risultati della prima fiamma nelle tre differenti versioni non è possibile individuare un rapporto fisso, o quantomeno lineare, tra i tempi dei due algoritmi. In particolare il caso con 2360 celle risulta quello con il rapporto più alto (2.71), mentre gli altri due casi hanno rapporti più bassi. La ragione di questo comportamento irregolare su una stessa fluidodinamica risiede non tanto nel nuovo algoritmo, quanto piuttosto in quello originale: la procedura

risolutiva che esso segue, già illustrata in Figura 3, opera tramite cicli ripetuti, in cui dopo aver iterato per un certo numero di volte la risoluzione locale dei bilanci, si tenta il metodo di Newton globale, con o senza falso transitorio; nel caso in cui si sia sufficientemente vicini alla soluzione, questo porta ad un crollo repentino dei residui, accelerando notevolmente la convergenza. Non esiste quindi una linearità nella velocità di convergenza perché ogni ciclo ha la sua durata a seconda di quanto si sia ancora distanti dalla soluzione.

Il secondo aspetto da considerare riguarda invece il rapporto più elevato nel caso della fiamma con *Bluff Body*, pari con uno *slave* a 2.95: in questa situazione però, la fluidodinamica risulta di gran lunga più complicata rispetto alla prima fiamma, dal momento che il corpo interposto induce ricircolazioni che rendono più difficile la convergenza tramite sostituzioni successive, ovvero con la sola sequenza di CSTR, mentre un metodo globale non perde di efficienza anche in questi casi (anzi, per fluidodinamiche ancora più complicate può risultare indispensabile).

Queste osservazioni possono risultare utili per ciò che concerne l'individuazione di una strategia "universale" e automatica di risoluzione che possa essere in futuro implementata nel Post-Processore parallelo e che non si limiti alla sola risoluzione in sequenza dei singoli reattori; ciò è discusso in maniera più dettagliata nel capitolo successivo.

### **5.5.2. Scalabilità dell'algoritmo**

Nonostante il numero di processori a disposizione sia limitato, è comunque possibile valutare il comportamento dell'algoritmo ottenuto al variare del numero di processori, utilizzando come parametri quantitativi di valutazione lo *speedup* e l'efficienza, la cui definizione è stata già fornita nel Paragrafo 2.6. Vale comunque la pena sottolineare come, in questo caso, per le due definizioni di cui sopra sia stato scelto come tempo di riferimento quello valutato adottando un singolo *slave* anziché quello ottenuto con il miglior algoritmo sequenziale (ovvero, la versione originale del KPP già descritta nel Paragrafo 1.1).

Le ragioni di questa scelta sono facilmente intuibili, e vanno ricercate nella finalità per cui lo *speedup* e l'efficienza sono stati calcolati: ciò che interessa in

questo caso è verificare se all'aumentare del numero di processori le prestazioni non siano negativamente influenzate da *overhead* di comunicazione troppo alti, o se siano ancora presenti problemi di distribuzione eterogenea del carico (nonostante la distribuzione alternata delle celle sui processori) che possano dare luogo a parziali inutilizzazioni di alcuni degli *slave*, e quindi andare a discapito dell'efficienza totale.

Seguendo la definizione (3.1) è possibile quindi calcolare lo *speedup* per ogni caso di studio e per ogni numero totale di *slave*. Di seguito in Tabella 19 sono riportati i risultati:

Tabella 19: *Speedup* ottenuto nei vari casi di studio

Caso	# Slave						
	1	2	3	4	5	6	7
590	1.00	1.96	2.90	3.71	4.29	5.18	5.61
2360	1.00	1.95	2.87	3.63	4.24	4.89	5.21
9440	1.00	1.92	2.80	3.51	4.03	4.64	4.86
24575	1.00	1.96	2.87	3.76	4.52	5.15	5.72

Gli stessi andamenti possono essere meglio visualizzati graficamente; ciò è fatto in Figura 22:

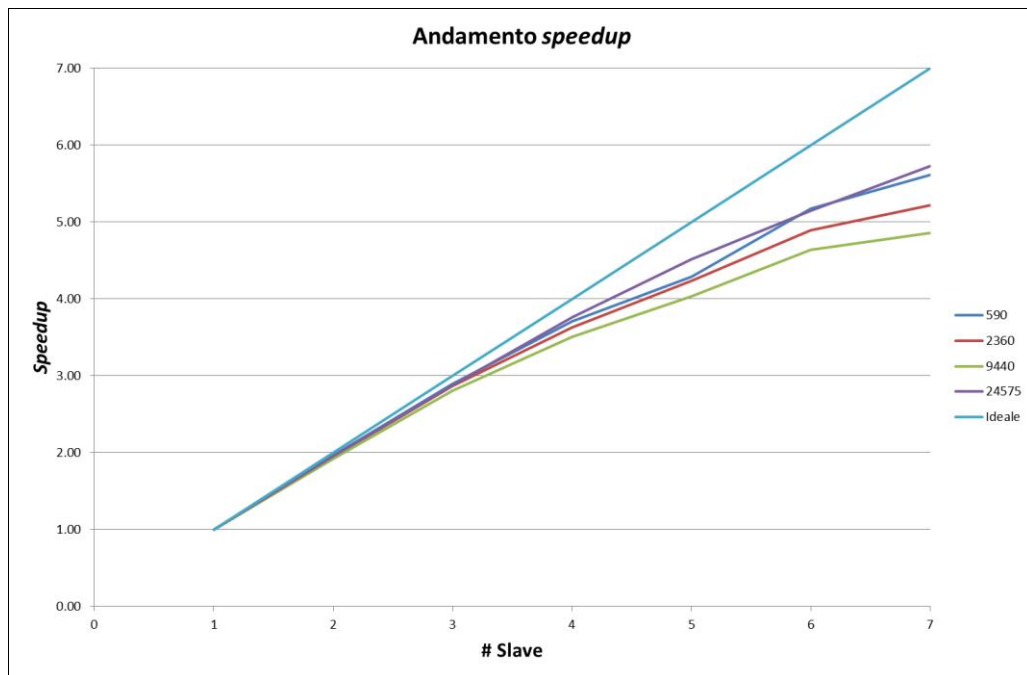
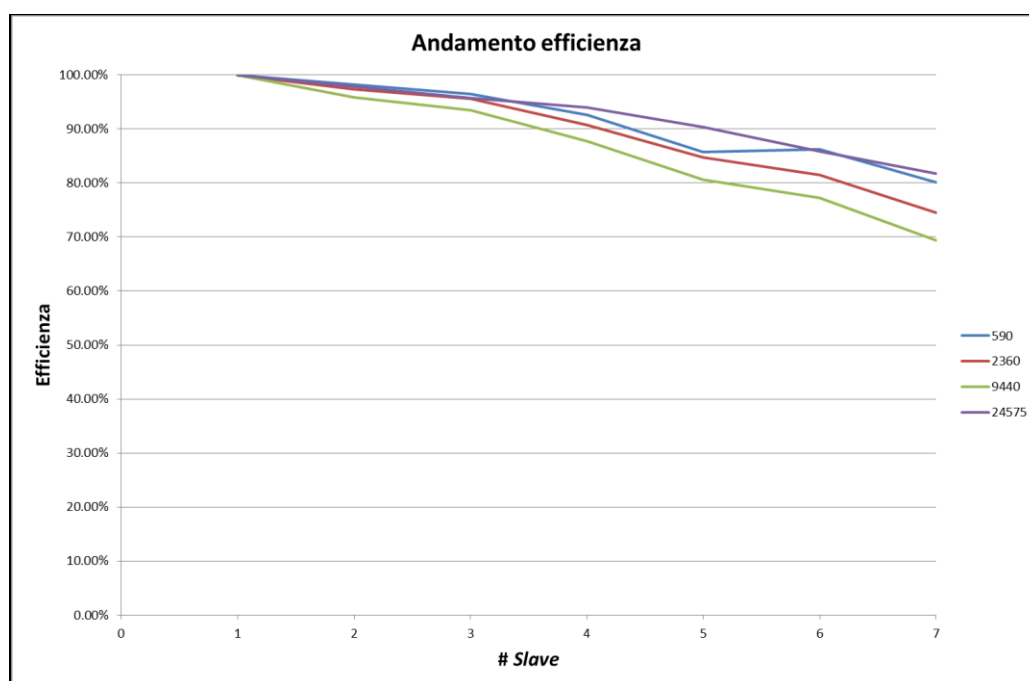


Figura 22: Andamento *speedup* nei quattro casi considerati

I medesimi risultati sono osservabili anche in termini di efficienza nella Tabella 20 e nella Figura 23 che seguono:

**Tabella 20: Andamento dell'efficienza al variare del numero di processori nei differenti casi di studio**

Caso	# Slave						
	1	2	3	4	5	6	7
590	100.00%	98.19%	96.50%	92.63%	85.71%	86.27%	80.11%
2360	100.00%	97.27%	95.57%	90.76%	84.71%	81.46%	74.50%
9440	100.00%	95.87%	93.45%	87.70%	80.59%	77.26%	69.40%
24575	100.00%	97.79%	95.71%	93.99%	90.30%	85.82%	81.77%



**Figura 23: Andamento dell'efficienza al variare del numero di processori per ogni caso di studio**

Importanti osservazioni possono essere dedotte dall'analisi degli ultimi risultati: in primo luogo, risulta evidente come i risultati migliori si ottengano nel caso più complicato dal punto di vista fluidodinamico, seguita a breve distanza dalla prima fiamma nella sua "versione" meno dettagliata da 590 celle.

Inoltre, per quanto a prima vista possa sembrare incoerente con i concetti espressi nel Capitolo 4 in termini di parallelismo e dimensione del problema, l'efficienza cala notevolmente nella stessa fiamma, passando da 590, a 2360 e 9440 reattori. Nel caso del prodotto tra matrici si è visto infatti come l'efficienza aumentava all'aumentare delle dimensioni del problema. Tuttavia le due

situazioni sono, in termini computazionali, enormemente diverse: il fattore discriminante è, in entrambi i casi, il rapporto tra il tempo impiegato dai processori per portare a termine il proprio lavoro e il tempo impiegato per la comunicazione con il *master*.

Nel prodotto tra matrici il carico computazionale assegnato a ciascun processore cresce proporzionalmente a  $n^3/p$ , con  $p$  numero di processori, mentre il peso della comunicazione, proporzionale al numero di elementi della matrice, cresce con  $n^2$ . All'aumentare di  $n$  pertanto si ottiene un progressivo decremento relativo degli *overhead* di comunicazione e un conseguente aumento dell'efficienza.

Invece, nel caso del Post-Processore, prendendo in considerazione la prima fiamma è possibile notare come i tempi medi per iterazione non aumentino sempre in maniera proporzionale al numero di celle, ma passando dal secondo al terzo caso aumentino con un andamento meno che lineare. In Tabella 21 sono evidenziati tali rapporti:

**Tabella 21: Andamento tempi iterazioni e comunicazione**

<b>Reattori</b>	590	2360	9440
<b>Rapporto tra le dimensioni normalizzato</b>	1	4	16
<b>Tempi iterazione [s]</b>	0.066	0.267	0.965
<b>Rapporto normalizzato</b>	1.00	4.07	14.70
<b>Tempo medio per cella [s]</b>	1.11E-04	1.13E-04	1.02E-04

Confrontando i dati di Tabella 20 e Tabella 21 risulta evidente la correlazione, nei tre differenti casi, tra i rispettivi andamenti di *speedup* e i tempi medi impiegati per la risoluzione della singola cella: per quanto quest'ultimo parametro non possa essere considerato un'informazione completamente esaustiva dell'andamento progressivo dei tempi delle singole iterazioni (dalle prime più pesanti, alle ultime di durata spesso comparabile con i tempi di comunicazione) e sulle rispettive, singole efficienze, esso conferma ancora una volta che la scalabilità del Post Processore Cinetico, applicato ad una determinata fiamma,

dipende dalla “difficoltà” che esso incontra nella risoluzione dei bilanci locali, difficoltà che va intesa in senso relativo, ovvero rapportata al peso della comunicazione, proporzionale come già detto alle dimensioni della griglia e al numero di specie.

Da tutto questo ragionamento è stato escluso l’effetto negativo legato al *Load Balancing*, che pur essendo stato ridotto al minimo mediante l’allocazione dei reattori in maniera alternata sui differenti processori, risulta comunque presente e in ultima analisi può essere considerato il motivo per cui il caso a 590 e quello a 2360 celle, pur avendo un tempo medio per cella e per iterazione quasi uguale ( $1.11E-4$  vs  $1.13E-4$ ) presentano scalabilità leggermente differenti (anzi, il caso con il tempo maggiore presenta una scalabilità minore, contrariamente a quanto previsto dalla logica esposta in precedenza). Purtroppo lo sbilanciamento residuo tra i processi non è un aspetto eliminabile, quantomeno con una gestione statica dei processi quale è quella attualmente implementata nel Post-Processore, in cui ogni reattore è allocato in maniera fissa sul singolo processore. Una gestione dinamica delle risorse computazionali potrebbe risolvere definitivamente il problema; maggiori informazioni al riguardo sono esposte nel Capitolo 6.

Ad ogni modo, il risultato certamente più positivo dell’analisi effettuata è costituito dalla buona scalabilità offerta dal caso più realistico della fiamma con *Bluff Body* a 24575 celle; ciò costituisce il punto di partenza per una verifica futura delle prestazioni dell’algoritmo su casi computazionalmente più pesanti, in cui il metodo risolutivo qui utilizzato (risoluzione dei CSTR in sequenza) potrebbe risultare non più sufficiente per ottenere una convergenza in tempi ragionevoli anche con più processori a disposizione, e l’implementazione di ulteriori approcci, stavolta di natura globale, si rivelerebbe indispensabile. Nel Capitolo 6 possibili strategie risolutive di tale tipo, approcciabili anche in un’ottica parallela, sono discusse nel dettaglio.

### 5.6. Efficienza dell'algoritmo e schema cinetico

Nel presente Capitolo è stato ribadito più volte che la scalabilità di un algoritmo dipende essenzialmente da due fattori: il *Load Balancing* e il peso relativo della comunicazione rispetto alla parte computazionale. Se il primo è stato (incertezze stocastiche a parte) risolto mediante un'allocazione dei reattori in maniera alternata sui processori, il secondo risulta inevitabile da affrontare, e come afferma la legge di Amdahl condiziona il massimo *speedup* raggiungibile.

In termini assoluti il tempo impiegato per la comunicazione dipende dalle dimensioni della griglia (i.e. dal numero di reattori) e dal numero di specie coinvolte, mentre il carico computazionale totale, sempre in termini assoluti, dipende dai fattori appena menzionati, oltre che dalla fluidodinamica del sistema, la cui complessità rende più o meno difficile il raggiungimento della convergenza.

In base a queste considerazioni risulta quindi evidente il ruolo giocato, all'interno di questo contesto, dallo schema cinetico adottato. Esso influenza infatti sia gli *overhead* di comunicazione che il carico computazionale; per quanto riguarda il primo, la correlazione è abbastanza semplice, in quanto la dimensione dei messaggi scambiati è proporzionale al numero di specie del suddetto schema. La dipendenza del carico computazionale dal numero di specie è invece più complessa: esso definisce la dimensione del sistema di equazioni che viene risolto cella per cella ad ogni iterazione. A prescindere dal fatto che si tratti di un sistema algebrico o differenziale è comunque richiesta a monte la fattorizzazione dello Jacobiano del sistema che richiede un numero di *flop* proporzionale a  $n^3$  (moltiplicato per un coefficiente che dipende dal metodo impiegato e che nel caso di una fattorizzazione *LU* è pari a  $2/3$ ).

Nell'ambito della presente Tesi la relazione tra numero di specie e scalabilità del problema è stata verificata analizzando il comportamento del Post Processore parallelo, in termini di tempi di esecuzione, al variare del sistema modellato e dello schema cinetico impiegato. Per quanto riguarda il sistema, è stata utilizzata la medesima fiamma già descritta nel Capitolo 3, nelle sue tre differenti versioni, rispettivamente a 590, 2360 e 9440 reattori, in modo tale da disporre di un termine

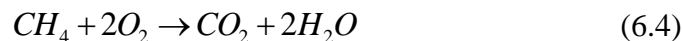
di paragone con i risultati già ottenuti. Invece gli schemi cinetici adottati sono stati tre, ovvero:

- *POLIMI\_DRM22*: 35 specie e 155 reazioni (già adottato nelle simulazioni precedenti);
- *GRI30*: 53 specie e 325 reazioni;
- *POLIMI\_C1C3HT\_1101*: 109 specie e 1764 reazioni;

Nei test che seguono, inoltre, è stata adottata un'ulteriore differenziazione circa le cinetiche ridotte utilizzate in *Fluent* per la valutazione delle condizioni di primo tentativo da cui partire con il Post-Processamento: queste, come detto più volte, fissano il campo di moto e di temperatura per tutta la durata della simulazione e forniscono dei valori iniziali di composizione per tutte le specie e per ogni cella. Nei casi considerati fino a questo punto, lo schema cinetico ridotto utilizzato è il seguente, costituito da due reazioni (*2-step kinetics*):



Invece in questa nuova serie di prove si è fatto uso, accanto alla cinetica appena descritta, di una cinetica globale (*1-step kinetics*) che ovviamente è costituita dall'unica reazione:



In particolare, lo schema cinetico *POLIMI\_DRM22* è stato testato su griglie provenienti da *Fluent* preprocessate tramite cinetiche sia a 1 che a 2 *step*, mentre per gli altri due si è partiti solo da cinetiche a 1 *step*. Ovviamente, non c'è da stupirsi se i risultati finali con la stessa cinetica dettagliata, ma con cinetiche ridotte di partenza tra loro differenti siano a loro volta differenti, in quanto il campo fluidodinamico importato da *Fluent* al termine del pre-processamento (che resta in seguito invariato) varia a seconda della cinetica ridotta adottata.

Nel complesso, la sensitività del Post-Processore parallelo al variare dello schema cinetico è stata valutata tramite prove di scalabilità su un totale di dodici

griglie computazionali, quattro per ogni dimensione di fiamma (una delle quali ottenuta mediante pre-processamento a 2 *step*, mentre le altre tre con una cinetica ridotta a 1 *step*).

Come nel caso precedente, tutti i casi sono stati eseguiti variando il numero di *Slave* da 1 a 7, misurandone i tempi di esecuzione e calcolando il relativo *speedup* ed efficienza. In Tabella 22 sono riportati tali risultati. I tempi di esecuzione sono misurati in secondi.

**Tabella 22: Tempi di esecuzione del Post-Processore su 12 casi di studio**

Caso	# Slave						
	1	2	3	4	5	6	7
<b>1X-1step-gri30</b>	83.84	42.15	29.40	22.25	19.10	15.75	13.84
<b>1X-1step-C1C3</b>	356.42	182.79	127.02	98.58	83.41	67.72	60.51
<b>1X-1step-drm22</b>	35.15	18.05	12.58	9.79	8.36	7.01	6.21
<b>1X-2step-drm22</b>	38.11	19.51	13.71	10.68	9.27	7.61	6.96
<b>2X-1step-gri30</b>	1131.64	575.34	393.19	309.66	253.34	215.96	193.91
<b>2X-1step-C1C3</b>	4311.68	2227.00	1494.96	1168.55	935.19	794.85	700.09
<b>2X-1step-drm22</b>	529.26	279.82	202.34	160.93	140.48	123.78	113.77
<b>2X-2step-drm22</b>	511.85	265.03	187.01	148.24	126.99	111.32	101.13
<b>4X-1step-gri30</b>	14991.70	7658.68	5247.24	4064.60	3305.36	2869.76	2486.91
<b>4X-1step-C1C3</b>	44462.80	23061.50	15931.60	12126.00	9660.00	8413.23	7161.75
<b>4X-1step-drm22</b>	5532.58	2909.51	2075.93	1645.71	1422.72	1252.30	1182.49
<b>4X-2step-drm22</b>	5037.49	2648.03	1903.42	1504.72	1314.56	1156.53	1098.75

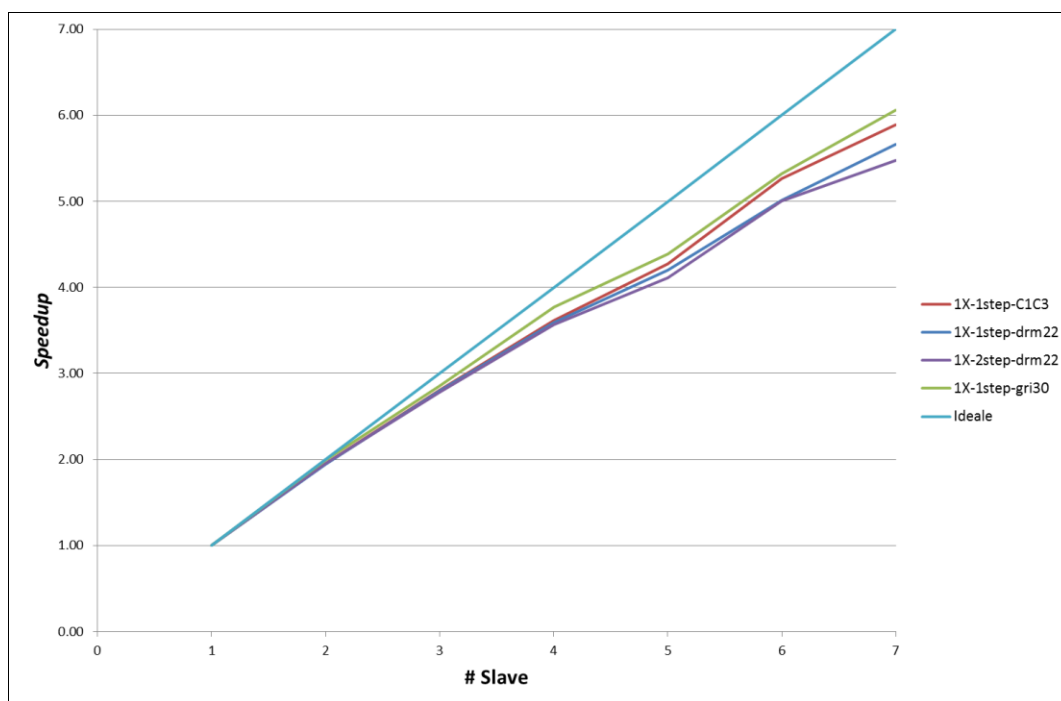
Si noti che i casi *1X-2step-drm22*, *2X-2step-drm22* e *4X-2step-drm22* corrispondono rispettivamente a 590, 2360 e 9440 di Tabella 16 e presentano infatti tempi di esecuzione, incertezze di *clock* a parte, praticamente analoghi.

Come da definizione (3.1), nel seguito si riporta lo *speedup* relativo, simulazione per simulazione, prendendo anche stavolta come riferimento il tempo di esecuzione con 1 *Slave*:

Tabella 23: *Speedup* ottenuto per ciascun caso di studio

Caso	# Slave						
	1	2	3	4	5	6	7
1X-1step-gri30	1.00	1.99	2.85	3.77	4.39	5.32	6.06
1X-1step-C1C3	1.00	1.95	2.81	3.62	4.27	5.26	5.89
1X-1step-drm22	1.00	1.95	2.80	3.59	4.20	5.01	5.66
1X-2step-drm22	1.00	1.95	2.78	3.57	4.11	5.01	5.48
2X-1step-gri30	1.00	1.97	2.88	3.65	4.47	5.24	5.84
2X-1step-C1C3	1.00	1.94	2.88	3.69	4.61	5.42	6.16
2X-1step-drm22	1.00	1.89	2.62	3.29	3.77	4.28	4.65
2X-2step-drm22	1.00	1.93	2.74	3.45	4.03	4.60	5.06
4X-1step-gri30	1.00	1.96	2.86	3.69	4.54	5.22	6.03
4X-1step-C1C3	1.00	1.93	2.79	3.67	4.60	5.28	6.21
4X-1step-drm22	1.00	1.90	2.67	3.36	3.89	4.42	4.68
4X-2step-drm22	1.00	1.90	2.65	3.35	3.83	4.36	4.58

Ciò che emerge immediatamente è che, adottando uno schema cinetico con un numero maggiore di specie e reazioni, si ottiene un andamento di gran lunga migliore: in particolare, per il caso più “pesante” in assoluto (*4X-1step-C1C3*) il valore di *speedup* raggiunto con 7 *Slave* è pari a 6.21. In forma grafica ciò è meglio rappresentato:

Figura 24: *Speedup* raggiunto nei casi *1X*

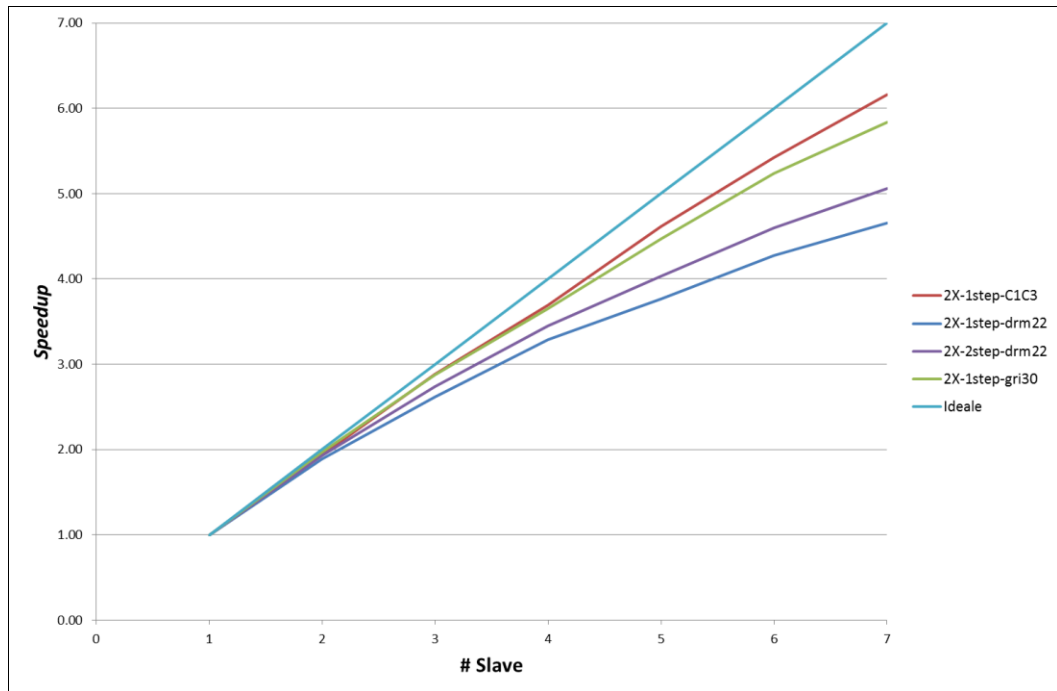


Figura 25: Speedup raggiunto nei casi 2X

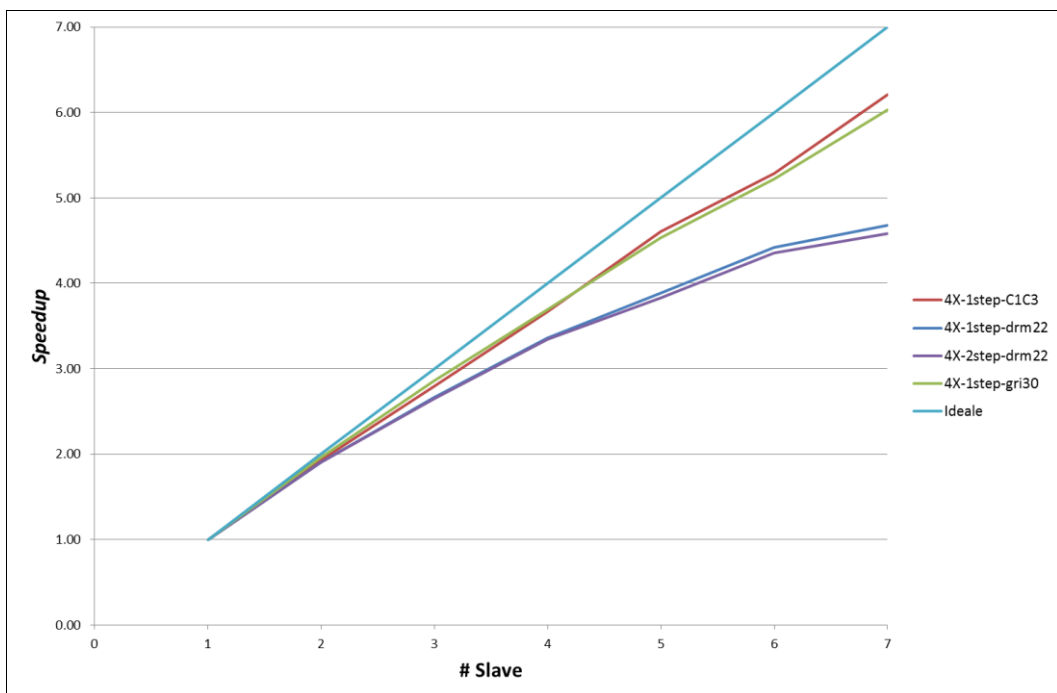
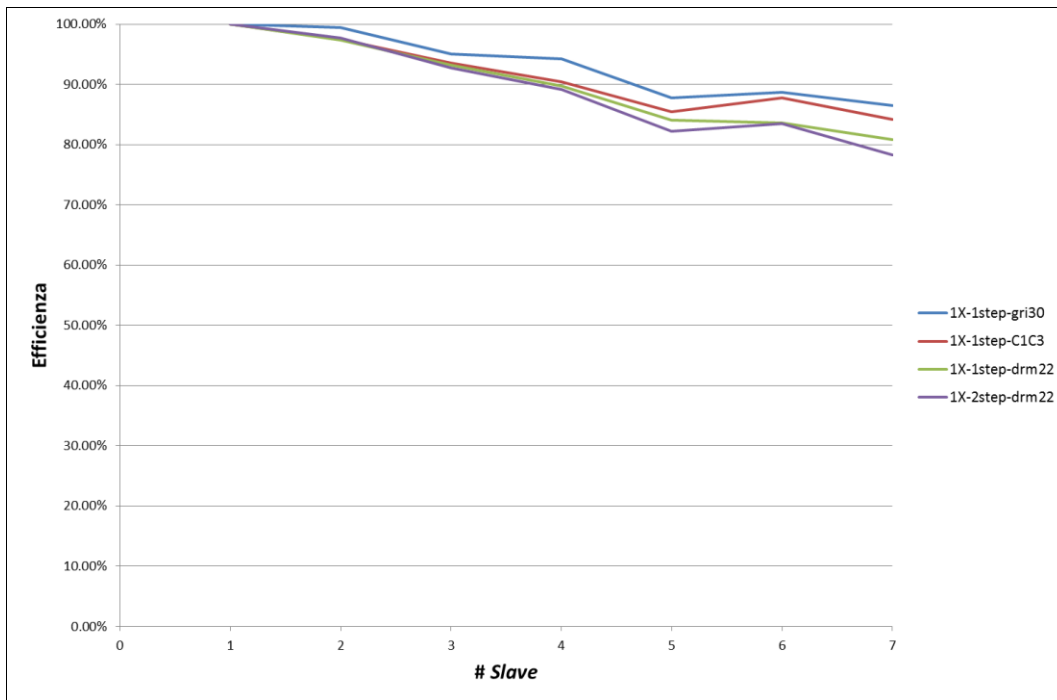


Figura 26: Speedup raggiunto nei casi 4X

Soprattutto nei casi 2X e 4X, si nota una notevole discrepanza tra lo schema cinetico originale (*drm22*) e gli altri due, i quali offrono prestazioni molto migliori. Gli stessi andamenti sono visualizzabili anche in termini di efficienza, come da Tabella 24 e Figura 27, Figura 28 e Figura 29.

**Tabella 24: Efficienza ottenuta per ciascun caso**

Caso	# Slave						
	1	2	3	4	5	6	7
1X-1step-gri30	100.00%	99.45%	95.06%	94.19%	87.80%	88.74%	86.54%
1X-1step-C1C3	100.00%	97.50%	93.53%	90.39%	85.47%	87.72%	84.15%
1X-1step-drm22	100.00%	97.37%	93.17%	89.76%	84.05%	83.55%	80.85%
1X-2step-drm22	100.00%	97.65%	92.68%	89.20%	82.27%	83.44%	78.25%
2X-1step-gri30	100.00%	98.35%	95.94%	91.36%	89.34%	87.33%	83.37%
2X-1step-C1C3	100.00%	96.80%	96.14%	92.24%	92.21%	90.41%	87.98%
2X-1step-drm22	100.00%	94.57%	87.19%	82.22%	75.35%	71.26%	66.46%
2X-2step-drm22	100.00%	96.56%	91.23%	86.32%	80.61%	76.64%	72.30%
4X-1step-gri30	100.00%	97.87%	95.24%	92.21%	90.71%	87.07%	86.12%
4X-1step-C1C3	100.00%	96.40%	93.03%	91.67%	92.06%	88.08%	88.69%
4X-1step-drm22	100.00%	95.08%	88.84%	84.05%	77.77%	73.63%	66.84%
4X-2step-drm22	100.00%	95.12%	88.22%	83.69%	76.64%	72.59%	65.50%



**Figura 27: Efficienza raggiunta nei casi 1X**

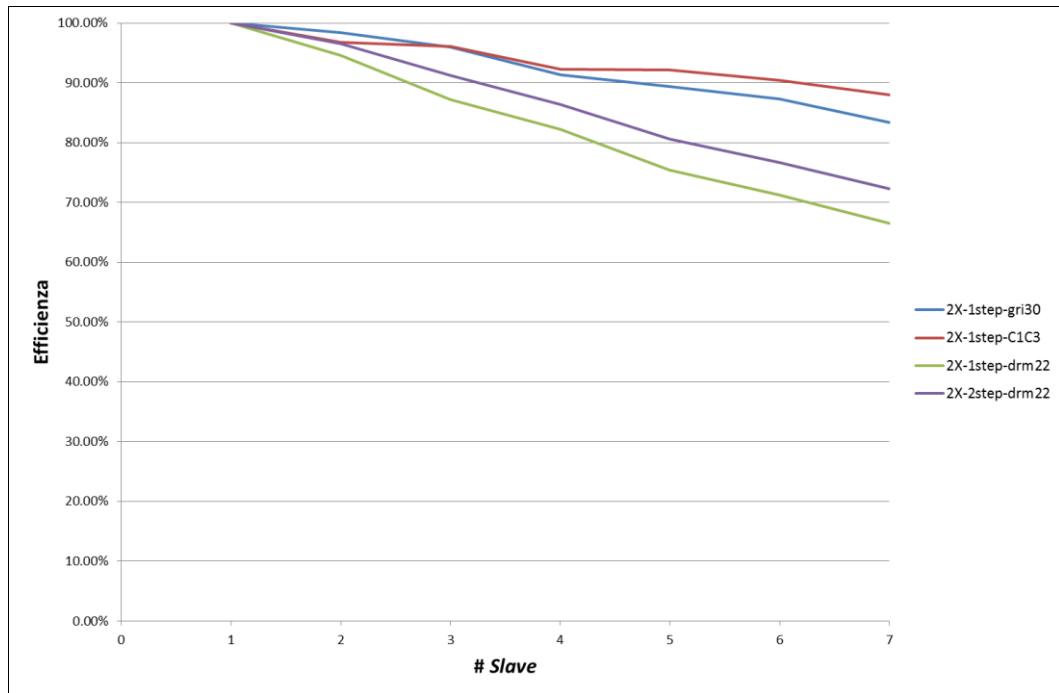


Figura 28: Efficienza raggiunta nei casi 2X

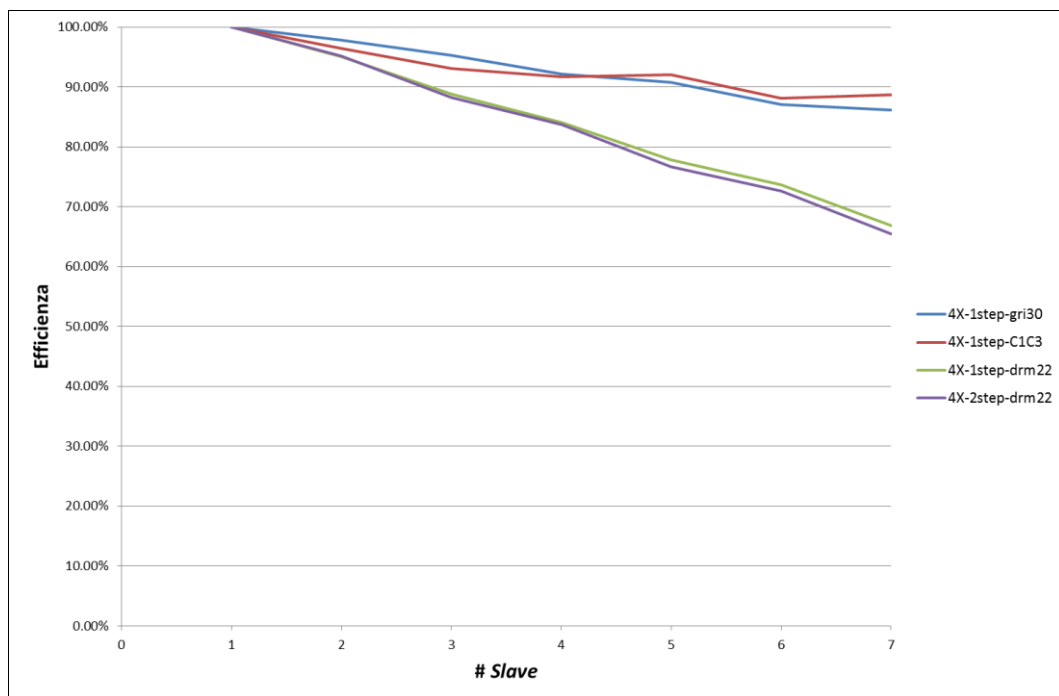


Figura 29: Efficienza raggiunta nei casi 4X

Anche i grafici precedenti mostrano il differente comportamento tra i due nuovi schemi cinetici e il *drm22*. Inoltre, l'andamento dell'efficienza al variare del numero dei processori risulta abbastanza irregolare, per quanto nel complesso tenda a diminuire. Nei grafici appena mostrati è infatti frequente che essa subisca

un temporaneo aumento aumentando di una unità il numero di *Slave*, contrariamente a quanto accade invece nel caso del prodotto tra matrici, in cui (incertezze di *clock* a parte) essa diminuisce monotonamente per via degli *overhead* di comunicazione. Tuttavia, a differenza del prodotto tra matrici, in questo caso è presente un altro fattore, il già citato *Load Balancing*, di natura aleatoria, che coesiste con il primo ed influisce in maniera spesso significativa sulle prestazioni del Post-Processore.

L'effetto combinato di *overhead* di comunicazione e distribuzione eterogenea del carico può essere visualizzato calcolando il tempo medio di risoluzione di ogni cella per ogni iterazione, per ciascuno dei dodici casi di studio in esame. In Tabella 25 ciò è descritto nel dettaglio.

**Tabella 25: Analisi dei tempi caratteristici dei dodici casi di studio**

Caso	N. celle	Tempo totale	Iterazioni	Tempo medio iterazione	Tempo medio cella-iterazione	Speedup max
1X-1step-gri30	590	83.84	501	1.67E-01	<b>2.84E-04</b>	6.06
1X-1step-C1C3	590	356.42	501	7.11E-01	<b>1.21E-03</b>	5.89
1X-1step-drm22	590	35.15	501	7.02E-02	<b>1.19E-04</b>	5.66
1X-2step-drm22	590	38.11	601	6.34E-02	<b>1.07E-04</b>	5.48
2X-1step-gri30	2360	1131.64	1801	6.28E-01	<b>2.66E-04</b>	5.84
2X-1step-C1C3	2360	4311.68	1601	2.69E+00	<b>1.14E-03</b>	6.16
2X-1step-drm22	2360	529.26	2001	2.64E-01	<b>1.12E-04</b>	4.65
2X-2step-drm22	2360	511.85	2001	2.56E-01	<b>1.08E-04</b>	5.06
4X-1step-gri30	9440	14991.70	5101	2.94E+00	<b>3.11E-04</b>	6.03
4X-1step-C1C3	9440	44462.80	4301	1.03E+01	<b>1.10E-03</b>	6.21
4X-1step-drm22	9440	5532.58	5701	9.70E-01	<b>1.03E-04</b>	4.68
4X-2step-drm22	9440	5037.49	5501	9.16E-01	<b>9.70E-05</b>	4.58

L'andamento dei tempi medi per cella e per iterazione conferma quanto previsto: confrontando in ciascuno dei tre casi (*1X*, *2X* e *4X*) questi ultimi con lo *speedup* massimo raggiunto con 7 *Slave* (a destra), si ottiene quasi sempre un andamento concorde: a un tempo medio maggiore corrisponde una scalabilità maggiore. Nelle griglie più grandi (*4X*), ordinando i tempi medi e gli *speedup* massimi si ottengono i medesimi risultati: il caso *4X-1step-C1C3*, con il maggior numero di specie in gioco, ha sia il tempo medio maggiore che uno *speedup* maggiore; a seguire, i due parametri vanno di pari passo anche per il *4X-1step-gri30* e per gli

ultimi due in cui lo schema cinetico adottato, il *DRM22*, possiede il minor numero di specie.

Nelle griglie più piccole tuttavia il parallelismo tempi-*speedup* non risulta sempre rispettato: per esempio, nel caso *IX* lo *speedup* massimo si ha nel caso *GRI30*, mentre il tempo medio più alto, per cella e per iterazione, appartiene al *CIC3* (come è legittimo attendersi dato il maggior numero di specie). Ma come è già stato detto in precedenza, va considerata la presenza della distribuzione eterogenea del carico sui processori, che è in grado di influire sulla scalabilità del problema a prescindere dallo schema cinetico adottato in quanto variabile stocastica, e quindi di modificare leggermente il succitato parallelismo. Va ricordato infatti che il tempo medio per cella e per iterazione può essere considerato un indicatore del costo computazionale rispetto agli *overhead* di comunicazione, ma non della scalabilità del problema, dacché la comunicazione non è l'unico fattore che influisce su di essa.

In conclusione, dall'analisi dei risultati presentati si evince come, in forma parallela, il Post-Processore riesca ad abbattere i tempi di calcolo della risoluzione dei reattori in sequenza tramite una distribuzione, di risorse e di potenza di calcolo, su un numero di processori sufficientemente elevato. Inoltre schemi cinetici più dettagliati permettono scalabilità maggiori per via del carico computazionale richiesto maggiore, per quanto non sia possibile avvicinarsi indefinitamente ad una scalabilità ideale a causa del *Load Unbalancing*. Ci si attende quindi che i risultati siano ancora migliori applicando gli schemi cinetici più dettagliati (e.g. *GRI30* o *CIC3*) a casi più realistici, come quello della fiamma con *Bluff Body* già analizzata al Paragrafo 5.5, che con lo schema meno dettagliato (*DRM22*) presentava la miglior scalabilità. Tuttavia, per griglie abbastanza grandi potrebbe non rivelarsi sufficiente la memoria RAM di una singola macchina (per quanto grande), risultando perciò indispensabile l'utilizzo di sistemi a memoria distribuita (*cluster*).

Ciò sarà tanto più vero nel momento in cui la risoluzione della sequenza di reattori non dovesse più essere l'unico metodo risolutivo adottato nel Post-Processore, in quanto per griglie molto grandi l'implementazione di metodi

globali (molto dispendiosi dal punto di vista della memoria) diventa fondamentale per ottenere una convergenza in tempi ragionevoli. Una descrizione di questi metodi nel dettaglio è fornita nel capitolo successivo.

## Capitolo 6

### Conclusioni e prospettive

#### 6.1. Bilanciamento del carico e allocazione processi

Come descritto anche nel Capitolo 5, i due aspetti che hanno il maggiore impatto sull'efficienza di processi paralleli sono il tempo di *overhead* e il bilanciamento del carico di lavoro del singolo nodo. L'esperienza dimostra che entrambi questi contributi sono dello stesso ordine di grandezza, in particolare l'insorgere di problemi dovuti al secondo è assai comune e difficile da arginare: basta infatti un solo processo che necessiti di un tempo di calcolo superiore rispetto agli altri per giungere a convergenza, perché si crei un collo di bottiglia che inficia le prestazioni di tutto l'algoritmo. Per valutare quantitativamente il peso dello sbilanciamento del carico sono state effettuate alcune simulazioni di una fiamma in cui si sono monitorati i tempi di esecuzione di ciascun processo per ogni iterazione. Alcuni risultati sono riportati in Tabella 26:

Tabella 26: Tempo di una singola iterazione nel caso *IX-1step-gri30* adottando rispettivamente 1 e 2 slave

# Iterazione	1 Slave	2 Slave	
	Tempo Slave 1 [s]	Tempo Slave 1 [s]	Tempo Slave 2 [s]
1	2.43805	1.2234	1.21647
2	1.08781	0.519673	0.56514
3	0.688707	0.361263	0.324576
...	...	...	...
497	0.064267	0.031983	0.037034
498	0.061956	0.030478	0.036061
499	0.068042	0.031308	0.035827

Calcolando le differenze di tempo, determinate dal processo più lento, si può valutare la perdita di efficienza del processo parallelo sfruttando la legge di Amdahl. Infatti il tempo di attesa per la sincronia dei processi è un intervallo nel quale non è possibile compiere operazioni in parallelo e opera quindi una sorta di serializzazione del codice. La legge di Amdahl, enunciata nell'equazione (3.3), restituisce lo *speedup* massimo teoricamente raggiungibile in base alla frazione di

codice non parallelizzabile. Osservando i dati della Tabella 26 si rileva uno scarto di variabile tra lo 0.5% e il 5% tra i tempi dei singoli processi. Ciò significa che, considerando uno scarto ad esempio dell'1%, per ogni iterazione si ha l'1% del tempo totale di calcolo che non fa parte dei processi paralleli: è equivalente ad una frazione di codice non parallelizzabile. Quindi si può sostituire nella legge di Amdahl verificando che, ad esempio utilizzando 10 processi:

$$S_{(10)} = \frac{1}{0.01 - \frac{1-0.01}{10}} = 9.174 \quad (6.5)$$

l'efficienza è già scesa al valore di 91.74%, senza considerare tutti gli altri fattori già ampiamente descritti che diminuiscono lo *speedup* in un caso reale.

La criticità che si è riscontrata per il bilanciamento di carico è dovuta in misura maggiore al contributo reattivo delle equazioni descrittive del sistema, perché a causa della disuniformità del campo di moto e delle condizioni di temperatura, la *stiffness* del problema da risolvere è variabile di cella in cella, per cui bisogna considerare questo aspetto nella distribuzione delle celle ai nodi di calcolo, onde non penalizzare eccessivamente la scalabilità. Si possono considerare principalmente due approcci per rispondere al problema: il primo è una distribuzione di tipo statico, il secondo prevede un'assegnazione dinamica. Nel primo caso si richiede di effettuare una valutazione della *stiffness* del problema per ciascuna cella prima che sia iniziata la computazione, in modo da poter distribuire il carico più uniformemente possibile; altrimenti si può scegliere un approccio stocastico, in cui le celle sono assegnate casualmente tra i processi per evitare la concentrazione di sistemi particolarmente *stiff* solo in alcuni, bensì cercare di suddividerli. L'altra opzione è l'assegnazione dinamica, in cui un processo che rimane indietro rispetto agli altri a causa del carico sbilanciato può distribuire dinamicamente agli altri processori le celle di troppo fino a ristabilire l'equilibrio.

Nel presente lavoro di Tesi si è scelto di seguire l'approccio stocastico, dal momento che unisce ad un miglioramento significativo dell'efficienza, l'immediatezza di implementazione. Si è infatti verificato un notevole aumento

dello *speedup* rispetto all'assegnazione "in sequenza" delle celle ai processi, permettendo di passare, ad esempio considerando due processi paralleli, da uno *speedup* di 1.10 ad uno di 1.95 circa, come indicato in Tabella 13 e

Tabella 14. Nonostante ciò, l'approccio stocastico mostra i suoi limiti all'aumentare del numero di processori, dal momento che dà adito ancora ad uno sbilanciamento che, per quanto piccolo, viene amplificato secondo la legge di Amdahl, come si è descritto nell'esempio precedente. Infine l'approccio stocastico è sensibile allo specifico caso simulato, perché può rivelarsi più o meno efficace in funzione della distribuzione delle celle *stiff*. Uno sviluppo futuro del lavoro di Tesi può riguardare l'aspetto del bilanciamento del carico tra i processi per migliorare ulteriormente lo *speedup* e possibilmente implementare una procedura che si adatti a ciascun caso.

## **6.2. Sequenza di CSTR e metodi globali**

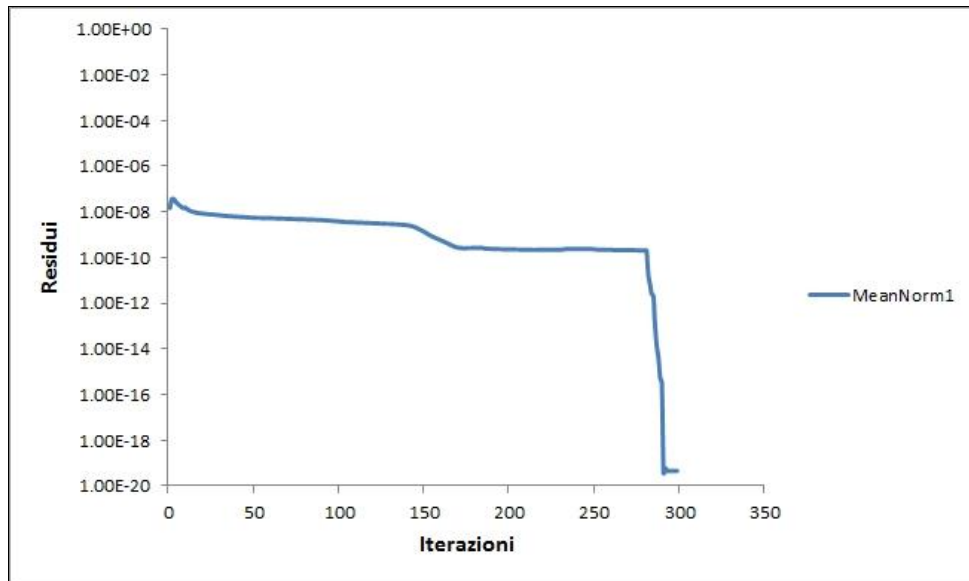
L'algoritmo presentato nei capitoli precedenti è caratterizzato da una buona scalabilità sulle architetture su cui è stato possibile testarlo. Si sono riscontrati risultati tanto migliori, quanto più i tempi di calcolo per singola iterazione risultavano maggiori dei tempi di comunicazione tra i processi paralleli. Nonostante ciò, lo scambio di messaggi che è prerogativa dello *standard* MPI e lo sbilanciamento di carico di lavoro dovuto ad un'allocazione stocastica delle celle di calcolo, ha causato una progressiva diminuzione dell'efficienza. Per questa ragione diviene importante monitorare il tempo di calcolo della singola iterazione, per assicurarsi che esso rimanga il tempo controllante; tuttavia tale richiesta temporale non rimane costante durante l'intera procedura risolutiva. Le iterazioni iniziali richiedono infatti un tempo superiore rispetto alle successive in cui i residui delle equazioni risolvono il sistema si sono abbassati avvicinandosi alla convergenza. La Tabella 1 riporta il tempo di calcolo per giungere rispettivamente a residui dell'ordine di  $10^{-10}$  e  $10^{-12}$ , quindi all'inizio e alla fine del procedimento iterativo:

Tabella 27: Confronto tra i tempi per iterazione all'inizio e alla fine della procedura risolutiva

# Reattori	$t_0$ [s]	$t_{fin}$ [s]
590	26.49	38.00
2360	370.832	548.26
9440	2189.83	5381.92
24575	20561.05	57505.8

A parità di numero di reattori si rilevano differenze tra i tempi davvero significative: l'efficienza del metodo decresce quindi progressivamente al diminuire dei residui. Inoltre, mentre il peso computazionale della risoluzione della sequenza di CSTR si abbassa, il tempo di comunicazione rimane costante, perché la quantità di informazioni da trasmettere per ogni iterazione è sempre la stessa, quindi si rischia che il tempo di calcolo non sia più il fattore controllante.

Fermo restando il fatto che la risoluzione in sequenza dei CSTR risulta un metodo che ben si presta ad un approccio parallelo, si può inferire che esso non sia tuttavia l'unico approccio tramite il quale giungere alla convergenza completa. Nel Capitolo 1 si sono descritti metodi di risoluzione globali che garantiscono velocità di convergenza molto superiori, quando convergono, come ad esempio il metodo di Newton, che ha una velocità di convergenza quadratica. Implementando un metodo di Newton a valle della risoluzione in sequenza della rete di reattori e di un ODE globale, per un caso di dimensioni ridotte (2360 celle, schema cinetico *GRI30*) l'andamento del valore medio della norma dei residui è mostrato in Figura 30:



**Figura 30: Andamento della media della norma 1 dei residui con soluzione in sequenza, ODE globale e Newton globale**

Dopo l'innesco del metodo globale si ha un crollo dei residui con raggiungimento della convergenza quasi in modo immediato. L'introduzione di metodi globali porterebbe perciò ad un'accelerazione nel calcolo della soluzione molto notevole; tuttavia non è stato possibile implementare tali metodi su problemi di dimensioni troppo elevate per il limite tecnologico imposto dalle dimensioni della griglia e dello schema cinetico coinvolto, che richiederebbero una eccessiva richiesta di allocazione di memoria. Un approccio che può considerarsi la naturale conseguenza dei risultati ottenuti nel presente lavoro di Tesi può essere quello di implementare metodi diversi in successione: la risoluzione in sequenza di CSTR, caratterizzandosi per la sua robustezza, è applicabile vantaggiosamente in una prima fase della risoluzione in cui i residui sono ancora alti e si è lontani dalla soluzione. Però, come si è verificato, dopo un certo numero di iterazioni, la curva dei residui rispetto alle iterazioni compiute tende ad appiattirsi, segnalando una diminuzione della velocità di convergenza: sarebbe opportuno in tale momento abbandonare la sequenza e attivare una seconda metodologia. In questa fase si può utilizzare un metodo iterativo per la risoluzione di sistemi lineari, come i metodi di Jacobi o Gauss-Seidel. Tali metodi sono basati sulla riscrittura del sistema:

$$Ax = b \quad (6.6)$$

con un'opportuna matrice di servizio, portando alla forma equivalente:

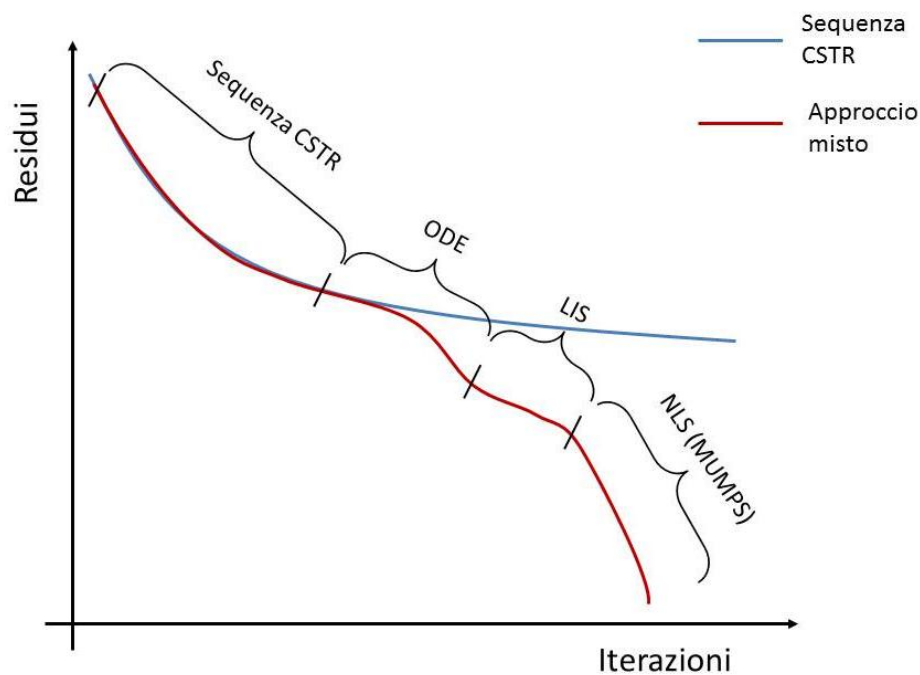
$$Gx = (G - A)x + b \quad (6.7)$$

da cui si ricava la procedura iterativa:

$$Gx^{(i+1)} = (G - A)x^{(i)} + b = c \quad (6.8)$$

Se la matrice di servizio  $G$  corrisponde alla diagonale principale della matrice  $A$ , allora si sta utilizzando il metodo di Jacobi per matrici diagonali. Nel caso in cui si abbia una matrice triangolare sinistra o destra, si può applicare il metodo di Gauss-Seidel *forward* o *backward*, nel quale la matrice di servizio  $G$  è la parte triangolare sinistra o destra della matrice di partenza. Il metodo di Jacobi in particolare, per come è definito è perfettamente parallelizzabile.

Dopo un numero sufficiente di iterazioni, per incrementare ulteriormente la velocità di convergenza si può innescare una procedura risolutiva tramite librerie di calcolo avanzato come LIS (*Library of Iterative Solvers*), una libreria di solutori lineari iterativi che può funzionare in seriale o in parallelo secondo lo *standard* OpenMP, oppure MPI, a seconda dell'architettura della macchina. Infine come ultimo passaggio si può lanciare la soluzione come NLS (*Non Linear System*) per mezzo di altre librerie, tra cui MUMPS (*Multifrontal Massively Parallel Sparse Direct Solver*). L'andamento dei residui in funzione delle iterazioni svolte avrebbe qualitativamente l'andamento rappresentato in Figura 31:



**Figura 31: Confronto qualitativo tra la velocità di convergenza con il solo metodo della sequenza di CSTR e l'approccio misto**

Dal grafico mostrato risulta evidente come la soluzione in sequenza di CSTR, per quanto importante nella procedura, costituisca l'inizio di un lavoro con grandi possibilità di sviluppo. Seguendo la metodologia adottata e consolidata durante il presente lavoro di Tesi, si può estendere la tecnica di calcolo parallelo ai successivi metodi, anche quelli globali, individuando la *concurrency* tra operazioni o utilizzando apposite librerie come LIS o MUMPS, che sono state pensate per implementare la computazione su sistemi a memoria distribuita o condivisa. La sequenza di solutori proposta e visualizzata in Figura 31 è dovuta alla complessità ed alle dimensioni del problema affrontato: non è infatti possibile utilizzare direttamente MUMPS, che implementa metodi diretti di risoluzione per sistemi lineari. I metodi diretti, se convergono, hanno velocità di convergenza molto maggiore dei metodi iterativi, quindi ci si attende un crollo dei residui nella parte finale della procedura per giungere alla soluzione cercata; tuttavia se non si è ancora sufficientemente vicini alla soluzione il metodo può presentare maggiori problemi di convergenza. Per tale motivo è conveniente implementare prima dei metodi diretti una libreria come LIS, che ha la caratteristica di utilizzare metodi iterativi i quali, sebbene siano generalmente più lenti di quelli diretti, risultano più

veloci quando si debba sgrezzare il problema. È opportuno tenere comunque presente che nell'ottica futura di una generalizzazione completa del codice, sarà necessario orientare la scelta degli algoritmi risolutivi verso quelli iterativi. Infatti, per sistemi di grandi dimensioni questi ultimi consentono di risolvere i problemi di occupazione di memoria già evidenziati nel Capitolo 1 ed inoltre rendono meno pesante la comunicazione all'interno della griglia di calcolo, dal momento che il trasporto dell'informazione viene dilazionato nelle diverse iterazioni. Questa è una caratteristica molto importante nelle reti di reattori, come è il caso studiato nel presente lavoro di Tesi, nel quale, se non gestita opportunamente, la comunicazione può arrivare ad essere lo stadio determinante sul tempo di risoluzione.

### **6.3. Confronto con altre librerie parallele**

Come si è descritto nel Paragrafo 6.2, esistono diverse librerie che implementano computazioni in parallelo per la risoluzione di sistemi lineari, come le già nominate LIS, MUMPS o le implementazioni parallele di BLAS. La ragione dello sviluppo di questo nuovo tipo di ambienti di programmazione è dovuto all'esigenza di sfruttare nuove tecnologie *hardware* che si stanno sviluppando, tra cui la crescente diffusione di *cluster* e l'introduzione di modalità di comunicazione *point-to-point* tra processori a sempre maggiore velocità, come il protocollo *Infiniband*. Inoltre vi è interesse a progettare *software* scalabili, portabili tra sistemi operativi e adattabili ad intere famiglie di problemi, al contrario di programmi *ad hoc* pensati per risolvere uno specifico sistema, quali spesso vengono scritti e commercializzati. La parallelizzazione del KPP si inserisce all'interno di tale panorama, sfruttando come valore aggiunto rispetto alla maggior parte degli strumenti per calcolo parallelo oggi disponibili la recente tendenza ad implementare computazioni parallele con un linguaggio di programmazione orientato agli oggetti, il che presenta notevoli vantaggi rispetto ad un linguaggio procedurale (Buzzi-Ferraris, 2010), come ad esempio il fatto che lanciare simultaneamente più oggetti generati dalla stessa classe su diversi nodi di calcolo risulta più semplice in termini di scrittura di codice, che non la chiamata contemporanea alle funzioni desiderate.

Dal confronto della *performance* del KPP parallelo rispetto ad altri codici attualmente disponibili per svolgere computazioni intensive su macchine a memoria distribuita emergono problematiche e risultati confrontabili, almeno qualitativamente, vista la varietà di problemi descritti, con quelli trovati durante il presente lavoro di Tesi. Si è sottolineata nel Capitolo 5 l'importanza del rapporto tra il tempo di *overhead* e l'effettivo tempo di calcolo, e associato a ciò anche il corretto bilanciamento dei processi paralleli in corso. Analisi presenti in letteratura (Amestoy et al., 2008; Gupta, 2002; Duff e van der Vorst, 1999) confermano gli aspetti riscontrati. A titolo di esempio è stata studiata la *performance* in parallelo di una delle funzioni che fanno parte della libreria per la soluzione di sistemi lineari MUMPS; l'analisi è stata limitata ad una funzione per poter studiare il comportamento parallelo del singolo algoritmo. Nel caso descritto da Amestoy et al. (2008), viene presentato l'algoritmo per la scalatura di una matrice in modo da migliorare il numero di condizionamento; questa è una delle operazioni che vengono compiute da un programma completo per la soluzione di sistemi lineari. L'approccio alla parallelizzazione su una macchina a memoria distribuita prevede innanzitutto la suddivisione della matrice di partenza sui processi a disposizione, dopodiché vengono eseguite le iterazioni previste dall'algoritmo. Nonostante la differenza nelle operazioni svolte dall'algoritmo rispetto al KPP, i risultati portano alle stesse conclusioni: scalabilità migliori si riscontrano per le matrici con il più elevato numero di elementi non nulli, quindi quelle per cui le computazioni risultano più pesanti. Anche per tale algoritmo la scalabilità diminuisce all'aumentare del numero dei processori, sebbene il crollo di efficienza sia in alcuni casi molto maggiore rispetto alla diminuzione rilevata con il KPP parallelo: mentre con 7 processi *slave* paralleli il KPP ha un'efficienza di circa 85% in tutti i casi testati, la *routine* di MUMPS oscilla tra il 95% e il 30% a seconda delle dimensioni della matrice utilizzata. Lungi dal voler confrontare i risultati di algoritmi creati con lo scopo di svolgere operazioni differenti, è però utile notare come il rapporto tra tempo di comunicazione e tempo di computazione possa avere un'influenza enorme sulla *performance*: infatti anche nell'algoritmo per la scalatura di matrici, se l'*overhead* diviene significativo l'efficienza necessariamente si degrada. Anche Duff e van der Vorst (1999) hanno

studiato l'effetto della scalabilità sulla fattorizzazione di matrici per la soluzione di sistemi lineari sottolineando l'effetto della comunicazione su problemi di dimensione diversa. In questo caso è stata utilizzata un'implementazione parallela delle librerie di algebra numerica BLAS con risultati paragonabili ai precedenti: un'efficienza compresa tra 80% - 90% per i primi processori, ma un crollo fino al di sotto del 50% quando il numero di processori supera il grado di parallelizzabilità dell'algoritmo, secondo quanto dettato dalla legge di Amdahl.

Sia nel caso del KPP che dell'algoritmo per la scalatura di matrici è stato studiato il modello di comunicazione per valutare il bilanciamento del carico del processo. Anche in questo caso si sono riscontrate tendenze simili per i due algoritmi diversi. Misurando lo sbilanciamento come:

$$\frac{w_{\max} - w_{\text{medio}}}{w_{\text{medio}}} \quad (6.9)$$

in cui compaiono i carichi, ovvero i tempi di calcolo richiesti, massimo  $w_{\max}$  e quello medio  $w_{\text{medio}}$  ottenuto dividendo il carico totale per il numero dei processi, si riottiene quella percentuale equivalente di parallelizzazione che si era introdotta nel Paragrafo 6.1. Nuovamente si riscontra il fatto che è sufficiente uno sbilanciamento anche relativamente piccolo, di pochi punti percentuali, per dare adito ad un notevole calo dello *speedup*.

Studiando qualitativamente il comportamento di altri algoritmi paralleli affermati nell'uso e nella letteratura scientifica si sono riscontrate le stesse criticità rilevate per il KPP parallelo. Dunque la parallelizzazione del Post Processore Cinetico, sebbene possa essere ulteriormente migliorata introducendo nuovi metodi numerici globali in parallelo, oltre a quello iterativo implementato durante il lavoro di Tesi, ha tuttavia raggiunto livelli qualitativi e prestazionali paragonabili con lo stato dell'arte dei programmi MPI attualmente sviluppati nell'ambito del calcolo numerico ad alte prestazioni.

#### **6.4. Conclusioni**

I programmi di calcolo sono ormai parte essenziale di molte aree scientifiche, anche grazie alla diffusione di calcolatori sempre più potenti ed economicamente accessibili. Mentre sono ormai consolidate numerose *routine* generalizzate per la risoluzione di problemi sotto forma di algoritmi procedurali, negli ultimi anni due aspetti hanno cambiato radicalmente il modo di concepire tali algoritmi: l'uso della programmazione orientata agli oggetti ed il calcolo parallelo; il primo consente la scrittura di algoritmi robusti e meno soggetti ad errori di programmazione, grazie all'uso di oggetti invece di funzioni (Buzzi-Ferraris, 2011), mentre il secondo permette di affrontare e risolvere in tempi ragionevoli problemi che non sarebbe stato altresì possibile affrontare mantenendo una praticità di utilizzo dello strumento. Inoltre il parallelismo permette di risolvere il problema della scrittura su memoria RAM dei dati suddividendoli sui banchi dei diversi nodi a disposizione, poiché a tutt'oggi le limitazioni tecnologiche non rendono disponibile un singolo banco di memoria di dimensioni sufficienti per problemi enormi, come quelli che ci si prefigge di affrontare con il KPP.

In aggiunta, la programmazione orientata agli oggetti ben si presta al calcolo parallelo, dal momento che gli oggetti costituiscono delle entità autonome con dati e funzioni proprie, che possono interagire mediante interfacce predefinite e che quindi possono essere suddivisi su processi differenti a seconda delle informazioni che condividono o che processano. Tali caratteristiche sono state sfruttate nella parallelizzazione del KPP, determinando la stesura di un codice robusto e che ha dimostrato una buona scalabilità sulle macchine su cui è stato testato. Il programma è concepito per essere portabile su *cluster* con diverse architetture e con qualsiasi numero di nodi, adattandosi automaticamente alle richieste dell'utente in termini di utilizzo di processori e suddivisione dei dati. In questo modo si è realizzato uno strumento per la valutazione delle emissioni adatto a fiamme i cui schemi cinetici sono di grande complessità e caratterizzati da griglie con un elevato numero di celle, grazie all'implementazione di metodi di calcolo in parallelo particolarmente robusti. Sono inoltre stati ridotti significativamente i tempi di calcolo rispetto ad un equivalente metodo realizzato in seriale, aprendo in questo modo la via all'introduzione successiva di metodi globali.



# Glossario

## Capitolo 1

$A$	Fattore preesponenziale
$a_i$	Accelerazione agente lungo la direzione $i$
$E_{att}$	Energia di attivazione [ $J$ ]
$\dot{E}$	Flusso di energia netta [ $J/s$ ]
$F_i$	Forza agente lungo la direzione $i$
$h$	Entalpia [ $J/kg$ ]
$J$	Matrice Jacobiana
$k$	Costante di reazione
$m$	Massa [ $kg$ ]
$N_{celle}$	Numero di celle di calcolo, o reattori, della griglia
$N_{specie}$	Numero di specie considerate nello schema cinetico
$\dot{Q}$	Flusso di calore netto [ $J/s$ ]
$R$	Costante dei gas [ $J/(mol \times K)$ ]
$T$	Temperatura [ $K$ ]
$V$	Volume [ $m^3$ ]
$\dot{W}$	Calore netto dovuto al lavoro [ $J/s$ ]
$\beta$	Esponente della temperatura
$\phi$	Generica proprietà di trasporto
$\rho$	Densità [ $kg/m^3$ ]
$\mu$	Coefficiente di viscosità [ $Pa \times s$ ]

## Capitolo 2

<i>Buffer</i>	Memoria tampone per il trasferimento di dati tra processi o dispositivi
Effetto <i>cache</i>	Distribuzione di dati sulla <i>cache</i> di più processori che porta un miglioramento delle prestazioni
<i>Flop</i>	<i>F</i> Lloating- <i>p</i> oint <i>O</i> Peration: operazione singola in virgola mobile
<i>Fork-join</i>	Modello concettuale per la creazione di processi paralleli
Frequenza di <i>clock</i>	Numero di operazioni logiche eseguibili da un processore al secondo
OpenMP	<i>O</i> pen <i>M</i> ulti- <i>P</i> rocessing; Standard di programmazione su sistemi a memoria condivisa
OpenMPI	Implementazione dello standard MPI
MPI	<i>M</i> essage <i>P</i> assing <i>I</i> nterface; Protocollo standardizzato di comunicazione tra processi
Mpich2	Implementazione dello standard MPI
<i>Multi-Core</i>	Architettura <i>hardware</i> che unisce diversi processori e le rispettive cache in un unico componente
<i>Speedup</i>	Misura della scalabilità di un algoritmo parallelo
<i>Thread</i>	Singolo flusso di operazioni sequenziali lanciato contemporaneamente ad altri in un programma parallelo (in sistemi a memoria condivisa)
<i>Transistor</i>	Dispositivo semiconduttore per l'amplificazione di segnali elettrici

## Capitolo 3

$A$	Area di passaggio [ $m^2$ ]
$F_i$	Flusso della specie $i$ -esima
$\bar{f}$	Vettore delle condizioni di bordo
$f_j^{in}$	Condizioni di bordo per la $j$ -esima specie
$C$	Matrice dei termini convettivi

$c$	Numero di Courant
<i>Concurrency</i>	Proprietà di un sistema in cui diverse operazioni possono essere eseguite contemporaneamente
$c_i^{in}$	Portata della $i$ -esima corrente in ingresso [ $kg/s$ ]
$c_i^{out}$	Portata della $i$ -esima corrente in uscita [ $kg/s$ ]
$h$	Entalpia specifica [ $J/kg$ ]
$\mathbf{M}$	Matrice di massa
$m$	Massa [ $kg$ ]
$\dot{m}_j^{in}$	Portata massiva della $j$ -esima specie in ingresso [ $kg/s$ ]
$\dot{m}_j^{out}$	Portata massiva della $j$ -esima specie in uscita [ $kg/s$ ]
$n$	Vettore normale uscente dal frontiera in ogni punto
$N_{nodi}$	Numero di nodi di calcolo
$N_{specie}$	Numero di specie
<i>Nested parallelism</i>	Tecnica che sfrutta il parallelismo MPI tra nodi a memoria distribuita formati da <i>multi-core</i> , e su ciascuno di essi innesta un parallelismo OpenMP a memoria condivisa
<i>Overhead</i>	Tempo dovuto alla comunicazione tra nodi
$Q$	Coefficiente di scambio termico [ $W/(m^2 K)$ ]
$R_j$	Matrice di reazione della $j$ -esima specie
$T_a$	Temperatura ambiente [ $K$ ]
$T$	Temperatura [ $K$ ]
$t_n$	Tempo all' $n$ -esima iterazione [ $s$ ]
$t_{res}$	Tempo di residenza [ $s$ ]
$u$	Velocità del flusso [ $m/s$ ]
$x$	Dimensione caratteristica della cella [ $m$ ]
$\bar{z}$	Vettore delle frazioni molari

$\varepsilon$	Valore dei residui alla fine della risoluzione in sequenza dei CSTR
$\omega_j$	Frazione massiva della specie $j$ -esima
$\bar{\omega}$	Vettore delle frazioni massive

#### Capitolo 4

<i>Concurrency</i>	Proprietà di un sistema in cui diverse operazioni possono essere eseguite contemporaneamente
$c_{ij}$	Elemento della $i$ -esima riga e $j$ -esima colonna
$E$	Efficienza di un algoritmo parallelo
<i>Infiniband</i>	Protocollo di comunicazione ad alta velocità tramite <i>switch point-to-point</i> bidirezionali
<i>Overhead</i>	Tempo accessorio dovuto alla comunicazione tra processi
$p$	Numero di processi paralleli
<i>Switch</i>	Dispositivo di rete per la connessione tra nodi
$T_p$	Tempo totale di esecuzione di un algoritmo parallelo su $p$ processi
$T_0$	Tempo di <i>overhead</i> [ $s$ ]
$t_s$	Tempo di preparazione di un messaggio MPI [ $s$ ]
$t_w$	Tempo di comunicazione per “parola” di un messaggio MPI [ $s$ ]
$W$	Tempo impiegato dal miglior algoritmo seriale [ $s$ ]
$\beta$	Campo di moto convettivo

#### Capitolo 5

$\bar{\omega}$	Vettore delle frazioni massive
$C$	Termine convettivo
$R$	Termine reattivo
$\bar{f}$	Vettore delle condizioni iniziali

<i>Bluff Body</i>	Corpo interposto dopo il bruciatore che stabilizza la fiamma formando a valle una zona di bassa pressione con conseguenti ricircolazioni
BzzMath	Librerie di calcolo numerico sviluppate in C++
<i>Data dependency</i>	Dipendenza delle operazioni dai dati
Fluent	<i>Software</i> per simulazioni di fluidodinamica computazionale
<i>Fork-join</i>	Modello concettuale per la creazione di processi paralleli
LIS	<i>Library of Iterative Solvers</i> ; Libreria per la risoluzione iterativa di sistemi lineari, compatibile anche con architetture parallele
MUMPS	<i>Multifrontal Massively Parallel sparse direct Solver</i> ; libreria per la risoluzione di sistemi lineari caratterizzati da matrice sparsa compatibile anche con architetture parallele
$n$	Numero di equazioni
NS	Numero di specie
<i>Object-oriented</i>	Paradigma di programmazione che utilizza strutture di dati, operazioni e metodi che interagiscono tra loro
ODE	<i>Ordinary Differential Equation</i>
$p$	Numero di processori
<i>Stack Overflow</i>	Sbancamento di memoria

## Capitolo 6

$w_{medio}$	Tempo di calcolo medio per iterazione
$w_{max}$	Tempo di calcolo massimo per iterazione



## Bibliografia

Amestoy P.R. et al., “*A parallel matrix scaling algorithm*”, Lecture Notes in Computer Science, Springer, (2008)

Atkinson K.E., An Introduction to Numerical Analysis, 2<sup>nd</sup> ed., John Wiley & Sons, New York, (1989)

Buzzi-Ferraris G., “BzzMath: Numerical libraries in C++”, Politecnico di Milano, [www.chem.polimi.it/homes/gbuzzi](http://www.chem.polimi.it/homes/gbuzzi), (2011)

Buzzi-Ferraris G., “*New trends in building numerical programs*”, Computers and Chemical Engineering, 35, (2011)

Buzzi-Ferraris G., Fundamentals of Linear Algebra for the Chemical Engineer, Wiley-VCH, (2010)

Buzzi-Ferraris G., Scientific C++, Addison-Wesley, Cambridge University Press, (1994)

Chapman, B. et al., Using OpenMP: Portable Shared Memory Parallel Programming, The MIT Press, (2008)

Cuoci A., “*Pollutant Formation in Turbulent Reactive Flows: interactions between chemistry and turbulence*”, Tesi di Dottorato Politecnico di Milano, (2008)

Cuoci A., Frassoldati A. Buzzi-Ferraris G., Faravelli T., Ranzi E., “*The ignition, combustion and flame structure of carbon monoxide/hydrogen mixtures. Note 2: Fluid dynamics and kinetics aspects of syngas combustion*”, Intl. Journal of Hydrogen Energy, 32, (2007)

De Falco C., Culpo M., “Extra Packages for GNU Octave: *bim, fpl, msh*”, Octave-Forge, <http://octave.sourceforge.net/packages.php>, (2010)

Deuffhard P., Newton Methods for Nonlinear Problems, Springer (2000)

Duff I.S., Van der Vorst H.A., “*Developments and trends in the parallel solution of linear systems*”, Parallel Computing, (1999)

Ferziger J.H., Milovan P., Computational Methods for Fluid Dynamics, Springer, (1999)

Gropp W. et al., Using MPI: Portable Parallel Programming with the Message-Passing Interface, The MIT Press, (1999)

Gupta A., “*Recent advances in direct methods for solving unsymmetric sparse systems of linear equations*”, ACM Transactions on Mathematical Software, (2002)

Gupta A., Kumar V., “*Scalability of Parallel Algorithms for Matrix Multiplication*”, ICPP, vol. 3, (1993)

ISS Technology Update, Vol.7 No.8, HP, (2008).

Manca D., Buzzi-Ferraris G., Cuoci A., Frassoldati A., “*The solution of very large non-linear algebraic systems*”, Comp. Chem. Eng., 33, (2009)

Ranzi E., Faravelli T., Frassoldati A., Cuoci A., “Kinetic Mechanism Version 1101 (January, 2011)”, Politecnico di Milano, [creckmodeling.chem.polimi.it/kinetic.html](http://creckmodeling.chem.polimi.it/kinetic.html)

Rauber T., Rüniger G., Parallel Programming for Multicore and Cluster Systems, Springer, (2010)

Ren Z., Pope S.B., “*Second-order splitting schemes for a class of reactive systems*”, Journal of Computational Physics, 227, (2008)

Snir M. et al., MPI: The Complete Reference, The MIT Press, (1996)

Yeoh G. H., Yuen K. K., Computational Fluid Dynamics in Fire Engineering, Butterworth-Heinemann, Oxford, (2009)

## Ringraziamenti

Al termine del percorso intrapreso in questo Lavoro di Tesi, è giusto ringraziare tutti coloro che hanno contribuito alla sua stesura o sono comunque stati presenti durante il periodo trascorso al primo piano del Dipartimento di Chimica.

*In primis* desideriamo ringraziare il Prof. Tiziano Faravelli, dal quale è scaturita l'idea di questa Tesi e che ha saputo assicurarci il necessario sostegno in corso d'opera, oltre ad aver supervisionato la sua stesura finale. La nostra riconoscenza va anche all'Ing. Alberto Cuoci, che ci è stato di grandissimo aiuto in fase di apprendimento del C++, oltre che nella successiva opera di programmazione; la sua esperienza pregressa sul KPP è stata il punto di partenza del nostro lavoro. Un ringraziamento particolare va anche al prof. Barresi del Politecnico di Torino, per aver svolto il suo ruolo di correlatore nell'ambito dell'Alta Scuola Politecnica.

È giusto inoltre ringraziare coloro che in questo anno e mezzo hanno condiviso con noi la vita di dipartimento, da Emma a Tiziano, passando per Alessio, Roberto, Flavio, il Prof. Ranzi. Senza dimenticare i nostri colleghi e amici laureandi, con cui abbiamo passato gran parte del nostro tempo in auletta, oltre a pranzi e pause caffè.

La nostra gratitudine va anche alle nostre famiglie che in questo periodo, come nel resto della nostra carriera accademica, ci hanno sempre sostenuto, risultando talvolta decisivi nel superare i momenti difficili che ci sono stati. E sicuramente continueranno ad essere decisivi nei giorni a venire.

E infine un particolare grazie a tutti gli amici che hanno condiviso con noi questi cinque anni.



*Perché tutti i gentiluomini nella sala eran nei  
loro più giovani anni; nessuno più di loro felice  
sotto il cielo; il loro re, il più nobile tra gli uomini.  
Difficile sarebbe trovare oggi in un castello  
una così prode compagnia.*

(Sir Gawain e il Cavaliere Verde)