Università degli Studi di Milano

FACOLTÀ DI SCIENZE E TECNOLOGIE

Corso di Laurea Magistrale in Matematica per le Applicazioni

Analisi ed implementazione parallela del metodo Parareal per sistemi di equazioni differenziali ordinarie

Relatore:
Prof. Luca Franco Pavarino

Correlatore:
Prof. Simone Scacchi

Tesi di Laurea di:
Kircheis Federico Paolo
Matricola N. 826035

Anno Accademico 2012-2013

Indice

Sommario

L'algoritmo Parareal, introdotto in [Lions et al. [2001]], è un metodo numerico parallelo in tempo per la risoluzione di sistemi di equazioni differenziali ordinarie e alle derivate parziali evolutive.
La risoluzione parallela di tali equazioni differenziali permette di ridurre i tempi di esecuzione e affrontare problemi di dimensioni più grandi, nonché approssimare la soluzione su un intervallo di tempo più esteso.
Negli ultimi anni sono stati pubblicati diversi risultati sulla stabilità e convergenza del metodo (si veda ad esempio [Bal [2005]]), ma pochi lavori sono disponibili sul confronto e rispettivi vantaggi del metodo Parareal rispetto a metodi classici seriali in tempo (ad esempio [Bal [2003]]).
Il metodo Parareal, essendo un metodo di tipo iterativo, necessita di un test di arresto, e anche questo argomento è stato raramente trattato nella letteratura del settore (si veda [Lepsa and Sandu [2010]]).

Lo scopo principale di questa tesi è lo studio del comportamento teorico e pratico dell'algoritmo Parareal, al fine di verificare che esso possa effettivamente essere usato con successo per risolvere equazioni differenziali alle derivate parziali e ordinarie. Il lavoro di tesi è stato suddiviso in diverse parti, ognuna delle quali pone l'attenzione su una diversa proprietà dell'algoritmo Parareal.

Dopo alcuni brevi cenni di calcolo parallelo presentati nel capitolo 1, nel secondo capitolo viene fornita l'idea di base del metodo Parareal e ne viene illustrato il suo funzionamento. Inoltre viene brevemente studiata una formulazione alternativa e fornita una scrittura seriale in pseudocodice.
Nel terzo capitolo viene effettuato uno studio teorico sul comportamento del metodo dal punto di vista della stabilità e convergenza, in modo da avere dei risultati che garantiscano la buona positura.
Nel quarto capitolo sono state studiate l'efficienza e lo speedup dell'algoritmo. In questo modo vengono forniti gli strumenti per capire se e sotto quali ipotesi risulta più conveniente usare un metodo parallelo al posto di uno seriale per la risoluzione di una ODE o PDE in tempo.
Nel quinto capitolo viene fatta una breve analisi sui diversi criteri di arresto che si possono implementare.
Nel capitolo 6 viene descritto il funzionamento del codice Python che e stato scritto appositamente per implementare e testare il metodo Parareal su diverse equazioni differenziali ordinarie.
Nella parte finale della tesi, il capitolo 7, vengono infine riportati e discussi i risultati di diversi test numerici sul metodo Parareal. Poichè l'algoritmo e stato implementato in parallelo e testato su cluster Linux a memoria distribuita, è stato possibile non solo analizzare lo studio dell'errore e della convergenza, come viene fatto spesso in letteratura per la versione seriale del metodo, ma anche di valutare e confrontare i tempi di esecuzione al variare del numero di processori impiegati, considerando diversi test di scalabilità sia forte che debole.

1 Cenni di Calcolo Parallelo

Tradizionalmente un algoritmo viene rappresentato come una sequenza di istruzioni, un procedimento sequenziale, che risolve un determinato problema. Le istruzioni vengono eseguite una alla volta dal processore di un computer, appena finisce l'esecuzione di una istruzione è possibile eseguire quella successiva. La velocità di esecuzione dell'algoritmo dipende unicamente da quanto è veloce il processore ad elaborare le istruzioni, la legge di Moore prevede che

> "Le prestazioni dei processori, e il numero di transistor ad esso relativo, raddoppiano ogni 18 mesi." (Gordon Moore, 1965)

Le prestazioni dei computer sono aumentato dal 1980 al 2004 come previsto dalla legge di Moore, grazie all'aumento della frequenza di clock, maggiore è la frequenza, più sono le operazioni che possono essere eseguite nella stessa unità di tempo.
Tuttavia il consumo di energia di un processore è legato in maniera proporzionale alla frequenza di clock, i processori moderni hanno raggiunto il limite massimo superiore delle frequenze di clock al di sopra delle quali la maggior produzione di calore durante l'elaborazione non può essere smaltita efficacemente senza causare danni al processore stesso.
Un altro modo per aumentare la frequenza di clock è quella ridurre la dimensione del processore, al di sotto di certe soglie si hanno non solo problemi di riscaldamento, ma anche di rumore dei campi elettromagnetici e incertezze quantistiche, che rendono tali dispositivi meno efficienti e più costosi.
Il 2004 è spesso riportata come la data che segna la fine del *frequency scaling*, in quanto l'azienda Intel[1] produttrice di processori cancellò lo sviluppo di alcune sue unità, causa i problemi elencati precedentemente.

Il calcolo parallelo sfrutta più processori in contemporanea per risolvere un singolo problema. Per fare questo è necessario spezzare il problema originale in sottoproblemi il più indipendenti possibili. Ogni unità di calcolo deve risolvere un sottoproblema e infine, per ottenere la soluzione del problema generale, è necessario raggruppare le sottosoluzioni.
In questo modo, senza aumentare la frequenza di clock, si eseguono più istruzioni alla volta, con il risultato di diminuire i tempi di calcolo.
Idealmente lo speedup (si veda la sezione (4.1)) di un algoritmo è lineare, nel senso che raddoppiando il numero di processori si dovrebbe dimezzare il tempo di calcolo. In realtà sono pochi gli algoritmi che hanno uno speedup quasi ottimale, la maggior parte ha uno speedup quasi lineare per un numero limitato di processori, dopodiché aumentare il numero di processori non diminuisce più i tempi di esecuzione.
Questo dipende dal fatto che alcune operazioni sono facilmente parallelizzabili, ma se le unità di calcolo sono troppe rispetto ai dati del problema, alcune unità non eseguiranno nessun lavoro

> "If a man can dig a hole of 1 m 3 in 1 hour, can 60 men dig the same hole in 1 minute (!) ? Can 3600 men do it in 1 second (!!) ? "

Inoltre alcune operazioni non sono del tutto indipendenti

> "I know how to make 4 horses pull a cart, but I do not know how to make 1024 chickens do it." (Enrico Clementi)

e risulta difficile se non impossibile spezzare alcuni parti del problema originale in sottoproblemi.
Oltre ai tempi di calcolo, che idealmente diminuiscono al crescere del numero di processori, vanno aggiunti quelli di comunicazione, che crescono all'aumentare dei processori.

Una grossa difficoltà del calcolo parallelo è quella di ripensare gli algoritmi seriali, esplicitando le operazioni indipendenti, oltre a dover gestire le comunicazioni tra le diverse unità di calcolo e a come distribuire il carico di lavoro, dal momento che la velocità totale di esecuzione dell'algoritmo è determinato dall'unità di calcolo più lenta.

[1] http://www.intel.it

2 Introduzione al metodo Parareal

Le risoluzione di un problema di Cauchy, tramite i metodi classici, come ad esempio Eulero esplicito ed implicito, metodi Runge-Kutta, Predictor-Corrector e Adams, sono formulati in maniera del tutto sequenziale, in quanto si ottiene in modo iterativo la soluzione ad un dato tempo, data la soluzione stessa ad un tempo precedente, o più tempi precedenti. Di conseguenza per ottenere la soluzione su tutti l'intervallo di tempo desiderato è necessario calcolare l'approssimata in modo sequenziale.
I metodi Parareal sono una famiglia di metodi che permettono di risolvere un problema di Cauchy in modo parallelo in tempo.

L'idea che sta dietro ai metodi Parareal è, dato il problema di Cauchy,

$$\begin{cases} \dfrac{\partial u}{\partial t} = f(u) & \text{su } [T_0, T_{\max}] \\ u(0) = u_0 \end{cases} \tag{1}$$

di spezzare l'intervallo $[T_0, T_{\max}]$ in N sottointervalli della forma $[t_n, t_{n+1}]$, con $n = 0, \dots, N-1$, e di risolvere in parallelo su ogni processore un sottoproblema del tipo

$$\begin{cases} \dfrac{\partial u_n}{\partial t} = f(u_n) & \text{su } [t_n, t_{n+1}] \\ u_n(t_n) = u_{0,n} \end{cases} \qquad n = 0, 1, \dots, N-1 \tag{2}$$

È possibile usare dei metodi *classici*, ovvero sequenziali, su ogni processore distinto, rimane da risolvere il problema su come ottenere i dati iniziali $u_{0,n}$ per ogni sottoproblema.
Per ottenere tali valori è possibile considerare il seguente schema iterativo;
Sia G un operatore che risolve in maniera *grossolana* il problema ai valori iniziali su un dato intervallo, e sia F un operatore che risolve in maniera *fine* (ovvero in modo più preciso di G) un problema ai valori iniziali su un dato intervallo

1. Dati u_0, f e $[T_0, T_{\max}]$, si calcolano tramite G i valori $u_{0,n}$, ovvero una prima approssimazione della soluzione di (1).
2. Per ogni n, dati $u_{0,n}$, f e $[t_n, t_{n+1}]$, si calcola su ogni processore tramite F la sottosoluzione di (1), ovvero la soluzione di (2).
3. L'accuratezza della sottosoluzione dipende, anche se F dovesse risolvere in maniera esatta, dalla bontà del dato iniziale $u_{0,n}$, pertanto serve un criterio di aggiornamento per tali valori.
4. Scelto il criterio e applicato l'aggiornamento di tali valori, si itera il calcolo delle sottosoluzioni e l'aggiornamento dei dati iniziali.
5. La soluzione finale è data da un *merge* delle sottosoluzioni, in quanto ognuna approssima su un intervallo distinto il problema originale.

Il metodo di aggiornamento proposto per il metodo Parareal è il seguente

$$\begin{aligned} U_n^k &= \mathcal{G}([t_{n-1}, t_n], U_{n-1}^k) + \mathcal{F}([t_{n-1}, t_n], U_{n-1}^{k-1}) - \mathcal{G}([t_{n-1}, t_n], U_{n-1}^{k-1}) \\ &= \mathcal{H}([t_{n-1}, t_n], U_{n-1}^k, U_{n-1}^{k-1}) \end{aligned}$$

dove

1. La quantità $\mathcal{G}([t_{n-1}, t_n], U_{n-1}^k)$ è il valore di u in t_n approssimato con il metodo G su $[t_{n-1}, t_n]$ con dato iniziale U_{n-1}^k, ovvero il valore aggiornato alla stessa iterata sull'intervallo precedente per $n > 0$.
2. La quantità $\mathcal{F}([t_{n-1}, t_n], U_{n-1}^{k-1})$ è il valore di u in t_n approssimato con il metodo F su $[t_{n-1}, t_n]$ con dato iniziale U_{n-1}^{k-1}, ovvero il valore ottenuto alla iterata precedente sull'intervallo precedente per $n > 0$.

3. La quantità $\mathcal{G}([t_{n-1}, t_n], U_{n-1}^{k-1})$ è il valore di u in t_n approssimato con il metodo G su $[t_{n-1}, t_n]$ con dato iniziale U_{n-1}^{k-1}, ovvero il valore ottenuto alla iterata precedente sull'intervallo precedente per $n > 0$.

Si noti che il metodo di aggiornamento è un metodo esplicito da eseguire in sequenziale, poiché, fissato k per definire U_n^k serve conoscere U_{n-1}^{k-1} e U_{n-1}^k a ritroso, fino ai dati iniziali $U_0^k = U_0^{k-1} = \cdots = u_0$. Inoltre, per $k \to \infty$, supponendo che G converga, si ha che i valori Y_n ottenuti come limite soddisfano l'equazione

$$U_n^k = \mathcal{F}([t_{n-1}, t_n], U_{n-1}^{k-1})$$

ovvero l'approssimazione puntuale grossolana raggiunge l'accuratezza che si ottiene con il metodo F.

Esempio 2.0.1.
Per fissare le idee, si consideri l'esempio preso dall'articolo di [Lions et al. [2001]], si consideri ovvero il problema

$$\begin{cases} u'(t) = \lambda u(t) & \operatorname{Re}\lambda < 0 \text{ su } [0, T] \\ u(0) = u_0 \end{cases}$$

e il metodo di Eulero Implicito come metodo G, per ottenere la soluzione grossolana del problema, ovvero

$$\begin{cases} \dfrac{U_{n+1} - U_n}{\Delta T} = \lambda U_{n+1} & n = 0, 1, \dots N-1 \\ U_0 = u_0 \end{cases}$$

dove si è spezzato l'intervallo $[0, T]$ in N intervalli di ampiezza costante ΔT e si è sfruttata la approssimazione

$$u'(t_n) \simeq \frac{u(t_{n+1}) - u(t_n)}{\Delta T} = \frac{U_{n+1} - U_n}{\Delta T}$$

A questo punto si suppone di usare come F un metodo che risolve in modo esatto i problemi

$$\begin{cases} u_n' = f(u_n) & \text{su } [t_n, t_{n+1}] \\ u_n(t_n) = U_n \end{cases} \qquad n = 0, 1, \dots, N-1$$

Data la soluzione dei sottoproblemi, si pone $U_n^1 = U_n$ e $u_n^1(t) = u_n(t)$, e si definiscono i salti, ovvero le quantità

$$S_n^k = u_{n-1}^k(t_n) - U_n^k$$

gli si fanno propagare su tutto $[0, T]$ nel seguente modo

$$\begin{cases} \dfrac{\delta_{n+1}^k - \delta_n^k}{\Delta T} = \lambda \delta_{n+1}^k + \dfrac{S_n^k}{\Delta T} \\ \delta_0^k = 0 \end{cases}$$

Infine si aggiornano i dati iniziali

$$U_n^k = u_{n-1}^{k-1}(t_n) + \delta_n^{k-1}$$

e si itera su k il procedimento appena presentato fino a raggiungere la precisione desiderata.

Nota.
L'algoritmo presentato da J.Lions per un caso particolare, sembra essere differente da quello presentato precedentemente, per il quale è stata data anche una scrittura in pseudocodice. In questa nota si vuole mostrare l'equivalenza tra questo secondo algoritmo, prendendo come metodo G il metodo di Eulero implicito e come F un risolutore esatto. Per semplicità riscriviamo l'algoritmo proposto da Lions con una notazione differente.
Date le quantità U_n^1, ottenute tramite G, si pone $u_{n-1}^k(t_n) = \mathcal{F}([t_{n-1}, t_n], U_{n-1}^k)$, in quanto $\mathcal{F}$ è un risolutore esatto, e si ridefiniscono le quantità

$$\begin{cases} S_n^k = \mathcal{F}([t_{n-1}, t_n], U_{n-1}^k) - U_n^k & n > 0 \\ \delta_n^k = \dfrac{1}{(1 - \lambda \Delta T)} \left[S_{n-1}^k + \delta_{n-1}^k \right] & \delta_0^k = 0 \end{cases}$$

e l'aggiornamento dei dati iniziali è dato da

$$U_n^k = \mathcal{F}([t_{n-1}, t_n], U_{n-1}^{k-1}) + \delta_n^{k-1}$$

e vogliamo mostrare che esso è equivalente a

$$\mathcal{G}([t_{n-1}, t_n], U_{n-1}^k) + \mathcal{F}([t_{n-1}, t_n], U_{n-1}^{k-1}) - \mathcal{G}([t_{n-1}, t_n], U_{n-1}^{k-1})$$

per $n \geq 0$ e $k \geq 1$.
Poichè ad ambo i termini compaiono la quantità $\mathcal{F}([t_{n-1}, t_n], U_{n-1}^{k-1})$, è sufficiente verificare che valga

$$\delta_n^{k-1} =^? \mathcal{G}([t_{n-1}, t_n], U_{n-1}^k) - \mathcal{G}([t_{n-1}, t_n], U_{n-1}^{k-1})$$

esplicitando il metodo di eulero implicito e il valore a sinistra, si ottiene

$$\frac{1}{(1 - \lambda \Delta T)} \left[S_{n-1}^{k-1} + \delta_{n-1}^{k-1}\right] =^? \frac{1}{1 - \lambda \Delta T} \left[U_{n-1}^k - U_{n-1}^{k-1}\right]$$

supponendo $1 - \lambda \Delta T \neq 0$ e semplificando si ottiene

$$U_{n-1}^k =^? S_{n-1}^{k-1} + \delta_{n-1}^{k-1} + U_{n-1}^{k-1}$$

ed esplicitando S_{n-1}^{k-1}

$$U_{n-1}^k =^? \mathcal{F}([t_{n-2}, t_{n-1}], U_{n-2}^{k-1}) + \delta_{n-1}^{k-1}$$

ma proprio in questo modo è stato definito l'aggiornamento dei valori, pertanto l'uguaglianza è valida, escluso per $n = 0$ in quanto non sono definiti i valori δ_{-1}^{k-1} e $\mathcal{F}([t_{-2}, t_{-1}], U_{-2}^{k-1})$. Però si verifica banalmente che vale

$$\mathcal{G}([t_{-1}, t_0], U_{-1}^k) - \mathcal{G}([t_{-1}, t_0], U_{-1}^{k-1}) = 0$$

pertanto si ha l'equivalenza dei due metodi.

2.1 Prima implementazione dell'algoritmo

In questa sezione diamo una visione in pseudocodice, con una sintassi simile a quella di Matlab©/Gnu Octave, per la implementazione del metodo Parareal in seriale.

Prima implementazione

```
T=[T0:delta_T:Tmax];  % definisco griglia
U = G(f,T,u0);        % prima predizione dei valori U_n
UG_old = U;
UG_new = U;
for k=1:kmax
    TF = [T(j): delta_t: T(j+1)];
        [TF,UF] = F(f,TF,UG(j));   % approssimo in modo fine
        UF_end(j+1) = UF(end);     % salvo valore per il raffinamento
    end
    UF_end(1) = u0;

    U(1)=u0;
    for j=2:length(T)              % aggiornamento dati iniziali
        [T_tmp, UG_new_tmp] = G(f,[T(j), T(j+1)],UG(j-1));
        UG_new(j) = UG_new_tmp(end);
        U(j) = UG_new(j) + UF_end(j) - UG_old(j);
    end
    UG_old = UG_new;              % definisco YG_old per iterata successiva
end
```

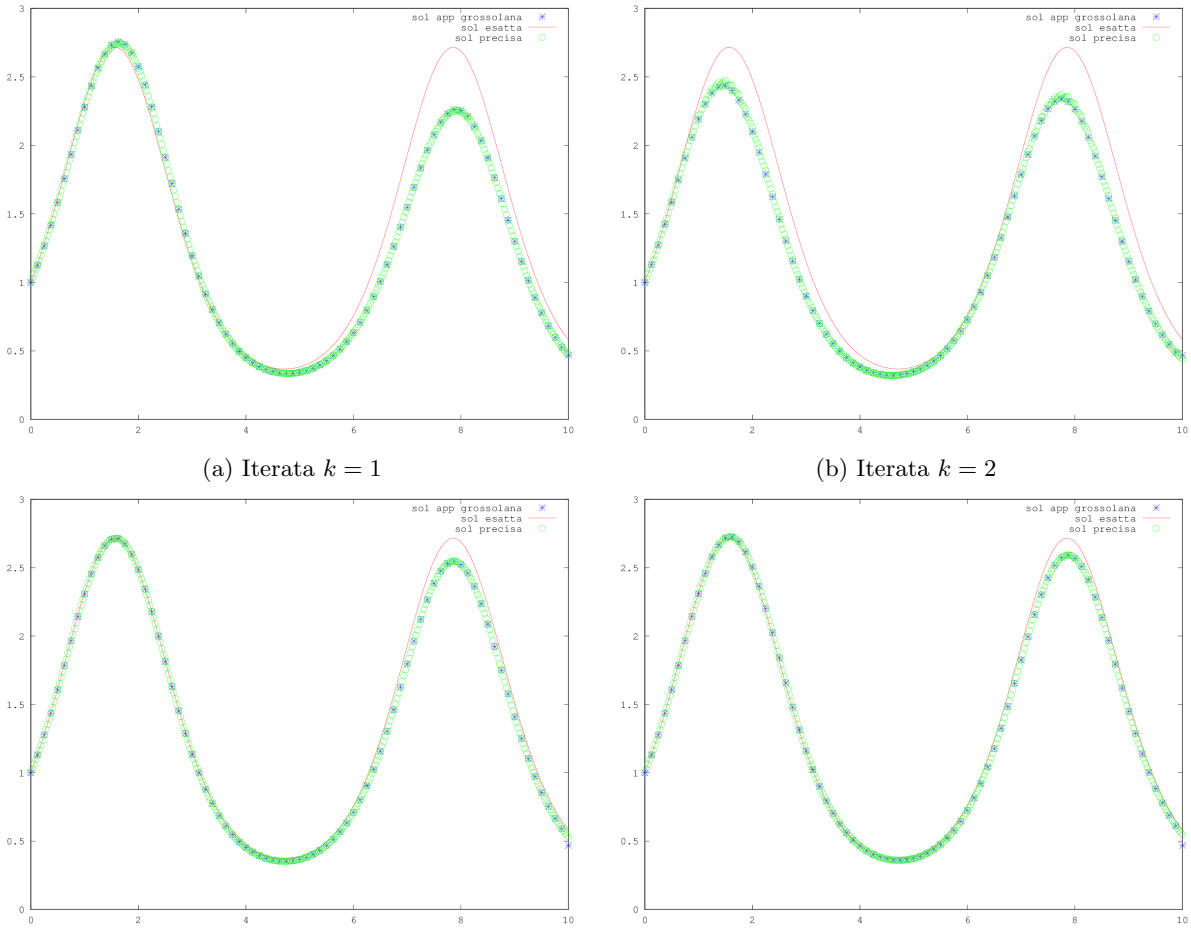
finita l'esecuzione di questo codice si avrà che U è un vettore contenente in N punti l'approssimazione della soluzione.
Le seguenti immagini sono frutto del codice presentato sopra eseguito con il software Gnu Octave, per la risoluzione del problema

$$\begin{cases} \dfrac{\mathrm{d}u}{\mathrm{d}t} = u \cdot cos(t) & \text{su } [0, 10] \\ u(0) = 1 \end{cases}$$

la cui soluzione esatta è

$$u(t) = e^{sin(t)};$$

Sono state fatte 4 iterate, usando il metodo di Eulero esplicito, per ambedue i propagatori F e G.



(a) Iterata $k = 1$

(b) Iterata $k = 2$

(c) Iterata $k = 3$

(d) Iterata $k = 4$

Figura 1: Risultati di una prima implementazione

2.2 Implementazione alternativa

È possibile ottenere il metodo Parareal anche tramite una costruzione più algebrica, dato nuovamente il problema (1)

$$\begin{cases} u' = f(u) & \text{su } [T_0, T_{\max}] \\ u(0) = u_0 \end{cases}$$

si spezza l'intervallo $[T_0, T_{\max}]$ in N sottointervalli $[T_n, T_{n+1}]$, con $n = 0, \dots, N-1$, e si definisce il sistema

$$\begin{cases} u_0' = f(u_0) & u_0(T_0) = U_0 \\ u_1' = f(u_1) & u_0^1(T_1) = U_1 \\ u_2' = f(u_2) & u_0^2(T_2) = U_2 \\ \quad \vdots & \quad \vdots \\ u_{N-1}' = f(u_{N-1}) & u_0^{N-1}(T_{N-1}) = U_{N-1} \end{cases} \tag{3}$$

oltre alle condizioni di incollamento

$$\begin{cases} U_0 = u_0 \\ U_1 = u_0(t_1, U_0) \\ \quad \vdots \\ U_j = u_{j-1}(t_j, U_{j-1}) \\ \quad \vdots \\ U_N = u_{N-1}(t_N, U_{N-1}) \end{cases} \tag{4}$$

queste condizioni si possono riscrivere come $F(\vec{U}) = 0$ con $\vec{U} = (U_0, \dots, U_N)$, e per risolvere tale sistema è possibile considerare il metodo delle tangenti di Newton

$$\vec{U}^k = \vec{U}^{k-1} - J_F^{-1}(\vec{U}^{k-1}) F(\vec{U}^{k-1})$$

dove J_F denota lo jacobiano di F. È possibile esplicitare il termine di aggiornamento $J_F^{-1}(\vec{U}^{k-1})F(\vec{U}^{k-1})$, si ha

$$\begin{bmatrix} I & & & \\ -\frac{\partial u_0}{\partial U_0}(t_1, U_0^{k-1}) & I & & \\ & \ddots & \ddots & \\ & & -\frac{\partial u_{N-1}}{\partial U_{N-1}}(t_{N-1}, U_{N-1}^{k-1}) & I \end{bmatrix}^{-1} \begin{bmatrix} U_0^{k-1} - u_0 \\ U_1^{k-1} - u_0(t_1, U_0^{k-1}) \\ U_2^{k-1} - u_1(t_2, U_1^{k-1}) \\ \vdots \\ U_N^{k-1} - u_{N-1}(T, U_{N-1}^{k-1}) \end{bmatrix}$$

È possibile anche esplicitare la risoluzione di tale sistema, ottenendo il metodo esplicito

$$\begin{cases} U_0^k = u_0 \\ U_{n+1}^k = u_n(t_{n+1}, U_n^{k-1}) + \dfrac{\partial u_n}{\partial U_n}(t_{n+1}, U_n^{k-1})(U_n^k - U_n^{k-1}) \end{cases}$$

Si noti che fino a questo punto è stata data una descrizione a livello continuo del problema, è necessario scegliere un metodo numerico per l'approssimazione di $u_n(t_{n+1}, U_n^{k-1})$ e dell'azione di $\frac{\partial u_n}{\partial U_n}(t_{n+1}, U_n^{k-1})$ sul termine $U_n^k - U_n^{k-1}$. In particolare se

1. Si usa l'operatore fine $\mathcal{F}$ per approssimare $u_n(t_{n+1}, U_n^{k-1})$
2. Si usa l'operatore grossolano $\mathcal{G}$ per dare la seguente approssimazione

$$\frac{\partial u_n}{\partial U_N}(t_{n+1}, U_n^{k-1})(U_n^k - U_n^{k-1}) = \mathcal{G}(t_{n+1}, t_n, U_n^k) - \mathcal{G}(t_{n+1}, t_n, U_n^{k-1})$$

ovvero

$$\frac{\partial u_n}{\partial U_N}(t_{n+1}, U_n^{k-1}) = \frac{\mathcal{G}(t_{n+1}, t_n, U_n^k) - \mathcal{G}(t_{n+1}, t_n, U_n^{k-1})}{(U_n^k - U_n^{k-1})}$$

allora il metodo proposto in questa sezione coincide con quello precedente.

3 Convergenza e stabilità del metodo Parareal

Prima di analizzare l'efficienza e i costi dei metodi Parareal, mostriamo qualche risultato di stabilità e convergenza, in quanto altrimenti non avremmo interesse ad implementare un algoritmo di questo tipo. Solitamente l'analisi della convergenza e stabilità di un metodo classico viene effettuata sul problema test, anche chiamato problema modello

$$\begin{cases} u' = \lambda u, \quad \mathrm{Re}(\lambda) < 0 \\ u(0) = 1 \end{cases}$$

la cui soluzione esatta è $u(t) = \exp(\lambda t)$. Pertanto l'analisi per il metodo Parareal verrà eseguita allo stesso modo.
La scelta di testare l'algoritmo con un problema lineare unidimensionale non è troppo limitativa, in quanto un problema non lineare può essere localmente linearizzato, se sufficientemente regolare, mentre nel caso di un sistema a d dimensioni, se esso è diagonalizzabile, l'analisi si può ridurre a d sistemi autonomi, tramite il calcolo degli autovalori.

Prima di mostrare risultati relativi al metodo Parareal introduciamo le definizioni e proprietà basilari per analizzare un metodo numerico per risolvere ODE, in modo da poter confrontare il metodo Parareal con i metodi classici.

3.1 Convergenza

Definizione 3.1.1 (Errore locale di troncamento)**.**
Sia $u(t)$ la soluzione del problema di Cauchy, si definisce come errore locale di troncamento l'errore che si ottiene forzando la soluzione nel metodo numerico, riscalato al passo h. Denotiamo tale quantità con $\delta_n(h)$, dove h è l'ampiezza del passo, mentre n denota la posizione t_n.

Definizione 3.1.2 (Consistenza)**.**
Diciamo che un metodo numerico è consistente di ordine α, se per $h \to 0$ si ha

$$\delta_n(h) = ch^\alpha = O(h^\alpha)$$

Lemma 3.1.3.
Il metodo di eulero esplicito è consistente di ordine 1.

Dimostrazione.
Il metodo di eulero esplicito è definito come

$$\begin{cases} \dfrac{U_{n+1} - U_n}{h} = f(t_n, U_n) \\ U_0 = u_0 \end{cases}$$

o in modo equivalente

$$\begin{cases} U_{n+1} = U_n + hf(t_n, U_n) \\ U_0 = u_0 \end{cases}$$

si definisce quindi l'errore di troncamento locale

$$h\delta_n(h) = u(t_{n+1}) - u(t_n) - hf(t_n, u(t_n))$$

applicando uno sviluppo in serie di Taylor, e sfruttando l'uguaglianza

$$f(t_n, u(t_n)) = u'(t_n)$$

si ha

$$h\delta_n(h) = u(t_n) + hu'(t_n) + O(h^2) - u(t_n) - hu'(t_n) = O(h^2)$$

□

Definizione 3.1.4 (Errore globale di troncamento)**.**
Sia U_n l'approssimante di $u(t_n)$ ottenuta tramite un metodo numerico, si definisce la quantità *errore globale di troncamento*

$$e_n = u(t_n) - U_n$$

Definizione 3.1.5 (Convergenza in h)**.**
Diciamo che un metodo è convergente di ordine α se $e_n = O(h^\alpha)$ per ogni n per $h \to 0$.

Lemma 3.1.6.
Se f è lipschitziana, allora il metodo di eulero esplicito è convergente di ordine 1.

Dimostrazione.
Si definisce l'errore globale di troncamento

$$\begin{aligned} e_{n+1} &= u(t_{n+1}) - U_{n+1} \\ &= e_n + h\left[f(t_n, u(t_n)) - f(t_n, U_n)\right] + h\delta_n(h) \end{aligned}$$

e quindi

$$\begin{aligned} |e_{n+1}| &\le |e_n| + h\left|f(t_n, u(t_n)) - f(t_n, U_n)\right| + h|\delta_n(h)| \\ &\le |e_n| + hL|u(t_n) - U_n| + h\delta_n(h) \end{aligned}$$

Risolvendo la ricorsione si ha

$$\begin{aligned} |e_{n+1}| &\le (1+hL)|e_n| + h\delta_n(h) \le \dots \\ &\le (1+hL)^{n+1}|e_0| + h\sum_{j=0}^{n}(1+hL)^j|\delta_{n-j}(h)| \end{aligned}$$

poichè $e_0 = 0$ e ponendo $\delta(h) = \max_j |\delta_j(h)|$, si ottiene

$$\begin{aligned} |e_{n+1}| &\le h\delta(h)\sum_{j=0}^{n}(1+hL)^j \\ &= h\delta(h)\frac{(1+hL)^{n-1} - 1}{1+hL-1} \\ &\le \delta(h)\frac{(1+hL)^{n-1} - 1}{L} \\ &\le \frac{\delta(h)}{L}\left(e^{hL}\right)^{n+1} \\ &= \frac{\delta(h)}{L}e^{hL(n+1)} \\ &\le \frac{\delta(h)}{L}e^{TL} \end{aligned}$$

Poiché

$$\delta(h) \le \frac{h}{2}\max u''$$

si ha che il metodo di eulero esplicito è convergente di ordine 1. □

A questo punto mostriamo qualche risultato specifico per il metodo Parareal.

Teorema 3.1.7.
Dato un metodo ad un passo di ordine p, e sia il dato iniziale del problema (1) di ordine q. Allora la soluzione numerica è di ordine $\min(p, q)$.

Nota.
Il teorema precedente mostra l'importanza di aggiornare i dati U_n^k per usare il propagatore $\mathcal{F}$, in modo da ottenere un risultato consistente, in quanto il metodo F avrà un ordine di convergenza più alto del metodo G.

Lemma 3.1.8 (Lemma di esattezza)**.**
Dato il metodo iterativo

$$\begin{cases} U_0^k = U_0 & k \geq 0,\ n = 0 \\ U_n^0 = G(U_{n-1}^0) & k = 0,\ n \geq 1 \\ U_n^k = G(U_{n-1}^k) + F(U_{n-1}^{k-1}) - G(U_{n-1}^{k-1}) & k \geq 1,\ n \geq 1 \end{cases}$$

allora si ha che il metodo è esatto per $k \geq n$, nel senso che vale

$$U_n^k = F(U_{n-1}^{k-1}) = F^n(U_0)$$

dove con F^n si intende F applicato n volte.

Dimostrazione.
La dimostrazione procede per induzione, mostriamo in modo esplicito i casi $k = 1, 2, 3$.
Per il caso $k = 1$ si ha

n =1)
$$U_1^1 = G(U_0^1) + F(U_0^0) - G(U_0^0) = F(U_0) = F^1(U_0)$$

Per il caso $k = 2$ si ha

n =1)
$$U_1^2 = G(U_0^2) + F(U_0^1) - G(U_0^1) = F(U_0) = F^1(U_0)$$

n =2)
$$U_2^2 = G(U_1^2) + F(U_1^1) - G(U_1^1) = F(U_1^1) = F^2(U_0)$$

Per il caso $k = 3$ si ha

n =1)
$$U_1^3 = G(U_0^3) + F(U_0^2) - G(U_0^2) = F(U_0) = F^1(U_0)$$

n =2)
$$U_2^3 = G(U_1^3) + F(U_1^2) - G(U_1^2) = F(U_1^2) = F^2(U_0)$$

n =3)
$$U_3^3 = G(U_2^3) + F(U_2^2) - G(U_2^2) = F(U_2^2) = F^3(U_0)$$

A questo punto supponiamo che l'iptesi sia vera per $k \geq n$, fino ad un certo k fissato, vogliamo vedere cosa succede nel caso $k + 1$; si ottiene

$$\begin{aligned} U_n^{k+1} &= G(U_{n-1}^{k+1}) + F(U_{n-1}^k) - G(U_{n-1}^k) \\ &= G(U_{n-1}^{k+1}) + F^n(U_0) - G(F^{n-1}(U_0)) \end{aligned}$$

quindi la tesi risulta vera se

$$U_{n-1}^{k+1} = F^{n-1}(U_0) \ \Rightarrow\ U_{n-2}^{k+1} = F^{n-2}(U_0) \ \Rightarrow\ \dots \ \Rightarrow\ U_0^{k+1} = F^0(U_0) = U_0$$

ma l'ultima relazione è vera, da cui la tesi. □

Nota.
Si mostra facilmente che per $k < n$, la tesi del lemma (3.1.8) non è in generale vera, ad esempio scegliendo $k = 1$ e $n = 2$, ovvero un caso particolare di $k = n - 1 < n$, si ha

$$U_2^1 = G(U_1^1) + F(U_1^0) - G(U_1^0)$$

e si ha che vale $G(U_1^1) = G(U_1^0)$ se la quantità $U_1^0 = G(U_0^0) = G(U_0)$ è uguale a $U_1^1 = F(U_0)$.
Anche scegliendo, nel metodo Parareal, per F e G lo stesso propagatore, non si avrà in generale che vale $F(U_0) = G(U_0)$, a meno che ambedue i propagatori non operino sulla stessa griglia, ma il metodo Parareal, per funzionare, ha bisogno di due griglie distinte.

Nota.
Tramite il lemma (3.1.8), si ottengono le seguenti informazioni per il metodo Parareal:

1) Il metodo Parareal è un metodo esatto in $\mathcal{F}$ per un numero di iterazioni sufficientemente alto. Per questo motivo è importante usare propagatori tali da dover necessitare poche iterate in k per fornire buone approssimazioni, altrimenti risulterebbe più conveniente usare un metodo classico, ovvero sequenziale.
   I teoremi successivi sulla convergenza forniscono dei risultati che assicurano che è possibile creare metodi che convergano in poche iterate.

2) L'accuratezza della approssimazione ottenuta con il metodo Parareal è al più buona come quella del propagatore fine.

3) La convergenza del metodo Parareal è data dalla convergenza dei propagatori $\mathcal{F}$ e $\mathcal{G}$.

4) È possibile evitare di aggiornare la soluzione sugli intervalli per i quali vale $n \leq k$.

Nota (Ampiezza intervallo)**.**
I concetti di convergenza si basano sul fatto che per h che tende a 0, si ha che l'errore tra la soluzione esatta e quella approssimata tenda a 0.
Poiché nel metodo Parareal, per come è stato definito, si fa uso di due metodi con due partizionamenti distinti, è necessario definire cosa si intende con ampiezza dell'intervallo.
Definiamo quindi le seguenti quantità

1) $\Delta_0 T = T_{\max} - T_0$, ovvero l'ampiezza dell'intervallo su cui è definito il problema.

2) $\Delta_1 T = \max_{n=0,\dots,N-1} \{t_{n+1} - t_n\}$, ovvero il passo massimo del propagatore G, mentre con $\Delta_{1,n} T = t_n - t_{n-1}$ un passo specifico. Per brevità poniamo inoltre $h = \Delta T = \Delta_1 T$

3) $\Delta_2 T$ il passo massimo su tutti gli intervallo $[t_{n-1} - t_n]$ con $n = 0, \dots, N-1$ del propagatore F, mentre con $\Delta_{1,n} T$ si identifica il passo massimo sull'intervallo $[t_{n-1} - t_n]$ con n fissato.

In particolare valgono le relazioni

1) $\frac{\Delta_1 T}{\Delta_0 T}$ è pari al numero di intervalli su cui il propagatore G deve approssimare il problema e il metodo di aggiornamento trasmettere l'informazione.

2) $\frac{\Delta_2 T}{\Delta_1 T}$ è pari al numero di intervallini su cui il propagatore F deve approssimare i sottoproblemi sugli intervalli della forma $[t_n, t_{n+1}]$.

3) $\frac{\Delta_2 T}{\Delta_0 T}$ è la somma di tutti gli intervallini su cui il propagatore F deve approssimare i sottoproblemi sugli intervalli della forma $[t_n, t_{n+1}]$. Esso è anche il numero di intervalli su cui deve operare un operatore $\mathcal{F}$ in seriale per approssimare allo stesso modo la soluzione.

Teorema 3.1.9 (Di convergenza)**.**
Si supponga che

1. *F è un metodo esatto, nel senso che vale $\mathcal{F}([t_{n-1}, t_n], u_{n-1}) = u_{n-1}(t_n)$*

2. *G è un operatore lipschitziano, nel senso che vale*

$$\sup_{n=0,\dots,N-1} \|G(t_n, u) - G(t_n, v)\| \leq (1 + C\Delta T)\|u - v\|$$

3. *G è un operatore consistente di ordine m, nel senso che vale*

$$\|u(t_n) - G(t_n, u)\| \leq C(\Delta T)^{m+1}\|u_0\|$$

4. *$F - G$ è un operatore consistente di ordine m e lipschitziano, nel senso che vale*

$$\|(F - G)(t_n, u) - (F - G)(t_n, v)\| \leq C(\Delta T)^{m+1}\|u - v\|$$

allora si ha che l'algoritmo Parareal è di ordine mk, nel senso che vale

$$\|u(t_n) - U_n^k\| \leq C(\Delta T)^{m(k+1)}\|u_0\|$$

Dimostrazione. (Per induzione)
Per $k = 0$ il teorema è vero per le ipotesi iniziali. Per $k \geq 1$ si ha

$$\begin{aligned} u(t_n) - U_n^k = {} & G(t_{n-1}, u(t_{n-1})) - G(t_{n-1}, U_{n-1}^k) \\ & + (F - G)(t_{n-1}, u(t_{n-1})) - (F - G)(t_{n-1}, U_{n-1}^{k-1}) \end{aligned}$$

e quindi, sfruttando le disuguaglianza scritte precedentemente, e l'ipotesi induttiva, si ottiene

$$\begin{aligned} \|u(t_n) - U_n^k\| &\leq \|G(t_{n-1}, u(t_{n-1})) - G(t_{n-1}, U_{n-1}^k)\| \\ &\quad + \|(F - G)(t_{n-1}, u(t_{n-1})) - (F - G)(t_{n-1}, U_{n-1}^{k-1})\| \\ &\leq (1 + c\Delta T)\|u(t_{n-1}) - U_{n-1}^k\| + c(\Delta T)^{m+1}\|u(t_{n-1}) - U_{n-1}^{k-1}\| \\ &\leq (1 + c\Delta T)\|u(t_{n-1}) - U_{n-1}^k\| + c(\Delta T)^{m(k+1)+1} \\ &\leq (1 + c\Delta T)^2\|u(t_{n-2}) - U_{n-2}^k\| + c(1 + c\Delta T)(\Delta T)^{m(k+1)+1} \\ &\quad + c(\Delta T)^{m(k+1)+1} \\ &\leq c(\Delta T)^{m(k+1)+1}\sum_{j=0}^{n}(1 + C\Delta T)^j \\ &\leq c(\Delta T)^{m(k+1)+1}\frac{(1 + C\Delta T)^{n+1}}{(1 + c\Delta T) - 1} \\ &\leq c(\Delta T)^{m(k+1)}(1 + C\Delta T)^{n+1} \\ &\leq c(\Delta T)^{m(k+1)}\left(e^{C\Delta T}\right)^{n+1} \\ &\leq c(\Delta T)^{m(k+1)}e^{C\Delta T(n+1)} \\ &\leq c(\Delta T)^{m(k+1)} \end{aligned}$$

□

Nota.
Nella pratica, F non sarà un metodo esatto, bensì un metodo di ordine fissato. Il teorema precedente può essere quindi usato per calcolare $\|U_n^k - \mathcal{F}(U_{n-1}^k)\|$. Tale quantità può dare delle informiazioni qualitative sulla soluzione dell'iterata attuale, e quindi sul numero di iterate in k da effettuare.

I teoremi precedenti mostrano la convergenza in funzione di $h \to 0$, ma non danno informazioni in funzione di h fissato e k crescente.
In letteratura [Gander and Vandewalle [2007]] è presente un risultato del seguente tipo, anche se prima è necessario porre qualche lemma e definizione preparatoria.

Definizione 3.1.10 (Matrice $M(\beta)$)**.**
Diciamo che una matrice è di Toeplitz, se è una matrice in cui ogni diagonale discendente da sinistra a destra è costante, ad esempio

$$\begin{pmatrix} a & b & c & d \\ e & a & b & c \\ f & e & a & b \\ g & f & e & a \end{pmatrix}$$

è una matrice di Toeplitz.
Siamo interessati alla parte triangolare inferiore di una matrice quadrata $N \times N$ di Toeplitz di parametro β, ovvero una matrice i cui termini sulle diagonali sono definiti dalla prima colonna nel seguente modo

$$M_{i,1}(\beta) = \begin{cases} 0 & \text{se } i = 1 \\ \beta^{i-2} & \text{se } 2 \leq i \leq N \end{cases}$$

Lemma 3.1.11 (Potenze di $M(\beta)$)**.**
L'i-esimo elemento della prima colonna delle k-esima potenza (con $k \geq 1$) della matrice $M(\beta)$ è

$$M_{i,1}(\beta)^k = \begin{cases} 0 & \text{se } 1 \leq i \leq k \\ \binom{i-2}{k-1}\beta^{i-1-k} & \text{se } k+1 \leq i \leq N \end{cases}$$

Dimostrazione. (Per induzione)
La matrice $M(\beta)$ è triangolare inferiore stretta. Quindi $M(\beta)^k$ ha almeno le prime k diagonali, a partire da quella centrale, nulle.
Si noti che per $k = 1$ la tesi è vera, rimane da verificare per $k > 1$. Esplicitando il prodotto fra matrici, sapendo che le prime k diagonali sono nulle e il fatto che si sta trattando la parte inferiore di una matrice di Toeplitz, si ottiene

$$M^k_{i,1} = \sum_{j=1}^{N} M^{k-1}_{i,j} \cdot M_{i,1} = \sum_{j=2}^{i-k+1} M^{k-1}_{i+1-j,1} \cdot M_{j,1} = \sum_{j=0}^{i-k-1} M^{k-1}_{i-1-j,1} \cdot M_{j+2,1}$$

Quindi si ha

$$M^k_{i,1} = \sum_{j=0}^{i-k-1} \binom{i-3-j}{k-2}\beta^{i-2-j-(k-1)} \cdot \beta^j = \sum_{j=0}^{i-k-1} \binom{i-3-j}{k-2}\beta^{i-1-k}$$

scegliendo J tale che $J - i + k + 1 = -j$ si inverte l'ordine della sommatoria e si ottiene

$$M^k_{i,1} = \beta^{i-1-k} \sum_{j=0}^{i-k-1} \binom{J+k-2}{k-2} = \beta^{i-1-k}\binom{i-2}{k-1}$$

□

Lemma 3.1.12 (Norma delle potenze di $M(\beta)$)**.**
Si ha che vale

$$\|M(\beta)^k\|_\infty = \begin{cases} \binom{N-1}{k} & \text{se } |\beta| = 1 \\ \dfrac{1}{(k-1)!}\dfrac{\mathrm{d}^{k+1}}{\mathrm{d}z^{k+1}}\left(\dfrac{z^{N-1}-1}{z-1}\right), \quad z = |\beta| & \text{se } |\beta| \neq 1 \end{cases}$$

Dimostrazione.
Si ha $\|M(\beta)^k\|_\infty = \|M(\beta)^k\|_1$, poichè l'ultima riga contiene più elementi, e in modulo maggiori, di delle altre. Si ha quindi

$$\|M^k\|_\infty = \sum_{i=k+1}^{N} |M^k_{i,1}| = \sum_{i=k+1}^{N} \binom{i-2}{k-1}|\beta|^{i-1-k} = \sum_{i=0}^{N-k-1} \binom{i+k-1}{k-1}|\beta|^i \tag{5}$$

Per $|\beta| = 1$ si ha

$$\|M^k\|_\infty = \sum_{i=0}^{N-k-1} \binom{i+k-1}{k-1} = \binom{N-1}{k}$$

Per $|\beta| \neq 1$ sia $z = |\beta|$, allora si ha

$$\begin{aligned}
\|M^k\|_\infty &= \frac{1}{(k-1)!} \sum_{i=0}^{N-k-1} (i+k-1)(i+k-2)\dots(i+1)|\beta|^i \\
&= \frac{1}{(k-1)!} \sum_{i=0}^{N-k-1} \frac{\mathrm{d}^{k-1}}{\mathrm{d}z^{k-1}} z^{i+k-1} \\
&= \frac{1}{(k-1)!} \frac{\mathrm{d}^{k-1}}{\mathrm{d}z^{k-1}} \sum_{i=0}^{N-k-1} z^{i+k-1} \\
&= \frac{1}{(k-1)!} \frac{\mathrm{d}^{k-1}}{\mathrm{d}z^{k-1}} \sum_{I=0}^{N-2} z^I && I = i+k-1 \\
&= \frac{1}{(k-1)!} \frac{\mathrm{d}^{k+1}}{\mathrm{d}z^{k+1}} \left(\frac{z^{N-1}-1}{z-1} \right)
\end{aligned}$$

□

Lemma 3.1.13 (Norma delle potenze di $M(\beta)$)**.**
Si ha la norma delle potenze di $M(\beta)$ è maggiorata nel seguente modo

$$\|M(\beta)^k\|_\infty \leq \begin{cases} \min\left\{ \left(\frac{1-|\beta|^{N-1}}{1-|\beta|} \right)^k, \binom{N-1}{k} \right\} & \text{se } |\beta| < 1 \\ |\beta|^{N-k-1} \binom{N-1}{k} & \text{se } |\beta| \geq 1 \end{cases}$$

Dimostrazione.
Per $|\beta| \geq 1$ è sufficiente prendere dal lemma precedente l'equazione (5), e si ottiene

$$\begin{aligned}
\|M^k\|_\infty &= \sum_{i=0}^{N-k-1} \binom{i+k-1}{k-1} |\beta|^i \\
&\leq |\beta|^{N-k-1} \sum_{i=0}^{N-k-1} \binom{i+k-1}{k-1} \\
&= |\beta|^{N-k-1} \binom{N-1}{k}
\end{aligned}$$

Per $|\beta| < 1$, si può sostituire a $|\beta|$ il suo sup, ovvero 1, e ottenere

$$\|M^k\|_\infty = \sum_{i=0}^{N-k-1} \binom{i+k-1}{k-1} |\beta|^i \leq \sum_{i=0}^{N-k-1} \binom{i+k-1}{k-1} = \binom{N-1}{k}$$

oppure, sfruttando una proprietà delle norme indotte, si ottiene l'altra maggiorazione

$$\|M^k\|_\infty \leq \|M\|_\infty^k = \left(\sum_{i=0}^{N-2} |\beta|^i \right)^k = \left(\frac{1-|\beta|^{N-1}}{1-|\beta|} \right)^k$$

□

Teorema 3.1.14 (Convergenza superlineare in k)**.**
Si supponga che

1. *F è un metodo esatto, nel senso che vale $\mathcal{F}([t_{n-1}, t_n], u_{n-1}) = u_{n-1}(t_n)$*

2. G è un operatore lipschitziano, nel senso che vale

$$\sup_{n=0,\dots,N-1} \|G(t_n,u) - G(t_n,v)\| \leq (1 + C\Delta T)\|u - v\|$$

3. G è un operatore consistente di ordine m, nel senso che vale

$$\|u(t_n) - G(t_n,u)\| \leq C(\Delta T)^{m+1}\|u_0\|$$

4. $F - G$ è un operatore consistente di ordine m e lipschitziano, nel senso che vale

$$\|(F-G)(t_n,u) - (F-G)(t_n,v)\| \leq C(\Delta T)^{m+1}\|u - v\|$$

allora

$$\max \|u(t_n) - u_n^k\| \leq c(\Delta T)^{(m+1)k} \frac{T_{\max}^k}{k!}$$

ovvero la convergenza in k è superlineare.

Dimostrazione.
Si definisce

$$\begin{aligned} e_n^k &= \|U_n^k - u(t_n)\| \\ &\leq \left\|G(U_{n-1}^k) - G(u(t_{n-1})\right\| + \left\|(F-G)(U_{n-1}^{k-1}) - (F-G)(u(t_{n-1}))\right\| \end{aligned}$$

si ha, applicando la ricorsione, e ponendo $c_1 = (1 + c\Delta T)$,

$$\begin{aligned} e_n^k &\leq c_1 e_{n-1}^k + c(\Delta T)^{m+1} e_{n-1}^{k-1} \\ &\leq c_1 \left[c_1 e_{n-2}^k + c(\Delta T)^{m+1} e_{n-2}^{k-1}\right] + c(\Delta T)^{m+1} e_{n-1}^{k-1} \\ &= c_1^2 e_{n-2}^k + c_1 c(\Delta T)^{m+1} e_{n-2}^{k-1} + c(\Delta T)^{m+1} e_{n-1}^{k-1} \\ &\leq c(\Delta T)^{m+1} \sum_{j=1}^{n-1} c_1^{n-j-1} e_j^{k-1} \end{aligned}$$

Ponendo $\vec{e}^k = (e_1^k, \dots, e_N^k)$ e $M(c_1)$ la matrice di argomento c_1, si ottiene

$$\vec{e}^k \leq c(\Delta T)^{m+1} M(c_1) \vec{e}^{k-1} \leq \dots \leq [c(\Delta T)^{m+1} M(c_1)]^k \vec{e}^0$$

$\vec{e}^0$ è un vettore che contiene le differenze tra U_n^0 e $u(t_n)$, quindi è un errore dato unicamente da G, per cui avrà il suo ordine di convergenza, ovvero m, quindi si ha

$$\|\vec{e}^k\|_\infty \leq [c(\Delta T)^{m+1} M(c_1)]^k \|\vec{e}^0\|_\infty \leq c[(\Delta T)^{m+1} M(c_1)]^k (\Delta T)^m$$

Ricordando che vale $c_1 = (1 + c\Delta T) > 1$, si ha

$$\|M(c_1)\|_\infty^k \leq |c_1|^{N-1-k} \binom{N-1}{k} \leq |c_1|^{N-k-1} \frac{N^k}{k!} \leq |c_1|^N \frac{N^k}{k!} \tag{6}$$

quindi si può scrivere

$$\|\vec{e}^k\|_\infty \leq c[(\Delta T)^{m+1} M(c_1)]^k (\Delta T)^m \leq c[(\Delta T)^{m+1}]^k |c_1|^N \frac{N^k}{k!} (\Delta T)^m$$

Inoltre vale

$$c_1^N = (1 + c\Delta T)^N \leq e^{c\delta T N} \leq e^{cT_{\max}}$$

e tale quantità è limitata, e quindi si ottiene

$$\begin{aligned} \|\vec{e}^k\|_\infty &\leq c[(\Delta T)^{m+1}]^k |c_1|^N \frac{N^k}{k!} (\Delta T)^m \\ &\leq c[(\Delta T)^{m+1}]^k \frac{N^k}{k!} (\Delta T)^m \\ &\leq c(\Delta T)^{(m+1)k} \frac{T_{\max}^k}{k!} \end{aligned}$$

□

Nota.
Nell'equazione (6), si può notare che se $k = N$, allora si ottiene $\|\vec{e}^k\|_\infty = 0$.
Questo vale poiché si sta considerando F un risolutore esatto, e poichè il metodo Parareal è un metodo esatto per $k \geq n$, si ha $e_n^k = 0$, con $n \leq N$, quindi quando $k = N$ si ha $e_n^k = 0$ per ogni n.

3.2 Stabilità

Dal punto di vista teorico l'ampiezza degli intervalli della discretizzazione tende a 0, ma dal punto di vista pratico, tale valore è limitato, ad esempio dalla precisione macchina che è finita e quindi non è possibile scendere sotto tale valore. Si analizza quindi un metodo numerico per verificare che, per una ampiezza fissata e non troppo piccola, la soluzione approssimante non differisca troppo da quella reale.

Definizione 3.2.1 (Stabilità)**.**
Dato il problema modello, in particolare un problema dissipativo lineare, si ha che $u(t) \to 0$ per t crescente. Diciamo che un metodo numerico è stabile in una certa regione se esiste un insieme significativo, chiamato regione di stabilità, per il quale fissato il passo h all'interno di tale regione, si ha $U_n \to 0$ per $n \to \infty$.

Lemma 3.2.2.
Il metodo di eulero esplicito è stabile se vale

$$h\lambda > -2$$

Dimostrazione.
Si ha

$$U_{n+1} = (1 + h\lambda)U_n = \cdots = (1 + h\lambda)^n U_0 = (1 + h\lambda)^n u(0) = S(z)^n u_0$$

Dove $S(z)$ è anche chiamata funzione di stabilità, con $z = h\lambda$. Se si fissa $h > 0$, e $|1 + h\lambda| \geq 1$, allora non si ha $U_n \to 0$, nonostante il metodo converga con ordine 1. Questo perché il passo risulta troppo lungo per dare una buona approssimazione del problema. Se si prende invece h tale che $|1 + h\lambda| < 1$, allora $U_n \to 0$ per n crescente. Le condizioni su h, nel caso di λ reale, sono date da

$$-1 < 1 + h\lambda < 1 \quad \to \quad -2 < h\lambda < 0$$

mentre la condizione sul lato destro è automaticamente soddisfatta, rimane da verificare quella sul lato sinistro, pertanto la regione di stabilità è definita dalla condizione da soddisfare, ovvero

$$h\lambda > -2$$

□

Lemma 3.2.3.
Il metodo di eulero implicito è stabile

Dimostrazione.
Si ha

$$U_{n+1} = \frac{1}{(1 - h\lambda)} U_n = \cdots = \frac{1}{(1 - h\lambda)^n} U_0 = \frac{1}{(1 - h\lambda)^n} u_0 = S(z)^n u_0$$

Si deve quindi verificare quali condizioni debba soddisfare h per avere

$$|1 - h\lambda| > 1$$

Nel caso λ reale, poiché $\lambda < 0$ si ha che tale condizioni è sempre verificata, pertanto non vi sono ulteriori condizioni su h. □

Nota.
I due lemmi precedenti suggeriscono di usare il metodo di Eulero implicito come metodo G, in quanto deve propagare l'informazioni su grossi intervalli. Il metodo di Newton esplicito potrebbe invece avere appunto problemi di stabilità e propagare l'informazione in maniera meno affidabile.

Prima di analizzare direttamente la stabilità del metodo Parareal, facciamo uso del seguente teorema per risolvere le ricorsioni

Lemma 3.2.4 (Lemma di ricorsione)**.**
Date le seguenti leggi ricorsive in k e n

$$\begin{cases} U_0^k = U_0 & k \geq 0,\ n = 0 \\ U_n^0 = R^1 U_{n-1}^0 = R^n U_0 & k = 0,\ n \geq 1 \\ U_n^k = R U_{n-1}^k + S U_{n-1}^{k-1} & k \geq 1,\ n \geq 1 \end{cases}$$

si ha che vale, per $n, k \geq 0$

$$U_n^k = \sum_{i=0}^{k} \binom{n}{i} S^i R^{n-i} U_0$$

Dimostrazione. (per induzione)
Tale relazione è ovvia per $k = 0$ oppure $n = 0$, mostriamo che la legge formulata vale per $k = 1$ e per ogni n, si ha

$$\begin{aligned} U_n^1 &= R U_{n-1}^1 + S U_{n-1}^0 \\ &= R U_{n-1}^1 + S R^{n-1} U_0 \\ &= R\left(R U_{n-2}^1 + S R^{n-2} U_0\right) + S R^{n-1} U_0 \\ &= R^2 U_{n-2}^1 + 2 S R^{n-1} U_0 \\ &\vdots \\ &= R^n U_0 + n S R^{n-1} U_0 \\ &= \left[R^n + n S R^{n-1}\right] U_0 \\ &= \sum_{i=0}^{1} \binom{n}{i} S^i R^{n-i} U_0 \end{aligned}$$

Supponendo quindi vera tale formula fino a k, verifichiamo che essa rimanga valida anche per $k+1$. Per semplificare la notazione non applichiamo la ricorsione in U_n^{k+1}, ma in U_{n+1}^{k+1}.
Inoltre poniamo

$$\alpha_n^k = \sum_{i=0}^{k} \binom{n}{i} S^i R^{n-i}$$

Si ottiene quindi

$$\begin{aligned}
U_{n+1}^{k+1} &= RU_n^{k+1} + SU_n^k \\
&= RU_n^{k+1} + S\sum_{i=0}^{k}\binom{n}{i}S^iR^{n-i}U_0 \\
&= RU_n^{k+1} + S\alpha_n^kU_0 \\
&= R\left(RU_{n-1}^{k+1} + S\alpha_{n-1}^kU_0\right) + S\alpha_n^kU_0 \\
&= R^2U_{n-1}^{k+1} + RS\alpha_{n-1}^kU_0 + S\alpha_n^kU_0 \\
&\quad\vdots \\
&= R^{n+1}U_0 + \sum_{j=0}^{n}R^jS\alpha_{n-j}^kU_0 \\
&= R^{n+1}U_0 + \sum_{j=0}^{n}R^jS\sum_{i=0}^{k}\binom{n-j}{i}S^iR^{n-i-j}U_0 \\
&= R^{n+1}U_0 + \sum_{j=0}^{n}\sum_{i=0}^{k}\binom{n-j}{i}S^{i+1}R^{n-i}U_0 \\
&= R^{n+1}U_0 + \sum_{j=0}^{n}\sum_{i=1}^{k+1}\binom{n-j}{i-1}S^iR^{n+1-i}U_0 \\
&= R^{n+1}U_0 + \sum_{i=1}^{k+1}\sum_{j=0}^{n}\binom{n-j}{i-1}S^iR^{n+1-i}U_0 \\
&= \sum_{i=0}^{k+1}\sum_{j=0}^{n}\binom{n-j}{i-1}S^iR^{n+1-i}U_0
\end{aligned}$$

Per finire la dimostrazione rimane quindi da verificare se vale

$$\sum_{j=0}^{n}\binom{n-j}{i+1} = \binom{n+1}{i}$$

Sfruttando il principio di addizione, ovvero

$$\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$$

si ottiene

$$\begin{aligned}
\binom{n+1}{i} &= \binom{n}{i-1} + \binom{n}{i} \\
&= \binom{n}{i-1} + \binom{n-1}{i-1} + \binom{n-1}{i} \\
&\quad\vdots \\
&= \sum_{j=0}^{n}\binom{n-j}{i-1}
\end{aligned}$$

e quindi la tesi. □

Teorema 3.2.5 (Stabilità)**.**
Dato il problema modello

$$\begin{cases} u' = \lambda u, & \mathrm{Re}(\lambda) < 0 \\ u(0) = 1 \end{cases}$$

con λ reale, siano S_G e S_F le funzioni di stabilità, rispettivamente, dei metodi G ed F con parametro $(h\lambda)$. Allora si ha che la funzione di stabilità dell'algoritmo Parareal è data da

$$S_H(h\lambda) = \sum_{i=0}^{k} \binom{n}{i} \left[\left(\tilde{S}_F - S_G\right)(h\lambda)\right]^i S_G^{n-i}(h\lambda)$$

con

$$\tilde{S}_F(h\lambda) = [S_F(\Delta_2 T\lambda)]^{\frac{\Delta_1 T}{\Delta_2 T}}$$

Dimostrazione.
Il passo di aggiornamento del metodo Parareal è dato da

$$\begin{aligned} U_n^k &= \mathcal{G}([t_{n-1}, t_n], U_{n-1}^k) + \mathcal{F}([t_{n-1}, t_n], U_{n-1}^{k-1}) - \mathcal{G}([t_{n-1}, t_n], U_{n-1}^{k-1}) \\ &= \mathcal{H}([t_{n-1}, t_n], U_{n-1}^k, U_{n-1}^{k-1}) \end{aligned}$$

Per definire la funzione di stabilità si osserva che vale

$$\begin{aligned} U_n^k &= S_G(h\lambda)U_{n-1}^k + \tilde{S}_F(h\lambda)U_{n-1}^{k-1} - S_G(h\lambda)U_{n-1}^{k-1} \\ &= S_G(h\lambda)U_{n-1}^k + \left[\left(\tilde{S}_F - S_G\right)(h\lambda)\right] U_{n-1}^{k-1} \end{aligned}$$

con

$$\tilde{S}_F(h\lambda) = [S_F(\Delta_2 T\lambda)]^{\frac{\Delta_1 T}{\Delta_2 T}}$$

Per semplicità scriviamo

$$U_n^k = S_1 U_{n-1}^k + S_2 U_{n-1}^{k-1}$$

applicando il lemma di ricorsione [lemma (3.2.4)], si ottiene che la funzione di stabilità è quindi data da

$$S_H(h\lambda) = \sum_{i=0}^{k} \binom{n}{i} \left[\left(\tilde{S}_F - S_G\right)(h\lambda)\right]^i S_G^{n-i}(h\lambda)$$

□

Teorema 3.2.6 (Condizioni di stabilità)**.**

Dato il problema modello

$$\begin{cases} u' = \lambda u, \quad \mathrm{Re}(\lambda) < 0 \\ u(0) = 1 \end{cases}$$

con λ reale, siano S_G e S_F le funzioni di stabilità, rispettivamente, dei metodi G ed F con parametro $(h\lambda)$. Allora si ha che l'algoritmo Parareal risulta stabile per ogni iterata se

$$\tilde{S}_F - 1 < 2S_G < \tilde{S}_F + 1$$

Dimostrazione.
Data la funzione di stabilità

$$S_H(h\lambda) = \sum_{i=0}^{k} \binom{n}{i} \left[\left(\tilde{S}_F - S_G\right)(h\lambda)\right]^i S_G^{n-i}(h\lambda)$$

Poichè vale, ponendo $S_1 = \left(\tilde{S}_F - S_G\right)(h\lambda)$ e $S_2 = S_G(h\lambda)$

$$\begin{aligned} \left|\sum_{i=0}^{k} \binom{n}{i} S_2^i S_1^{n-i}\right| &\leq \sum_{i=0}^{k} \binom{n}{i} |S_2|^i |S_1|^{n-i} \\ &\leq \sum_{i=0}^{n} \binom{n}{i} |S_2|^i |S_1|^{n-i} \\ &= (|S_2| + |S_1|)^n \\ &= \left(|S_G| + \left|\tilde{S}_F - S_G\right|\right)^n \end{aligned}$$

Nel caso valga

$$|S_G| + \left|\tilde{S}_F - S_G\right| = \left|S_G + \tilde{S}_F - S_G\right| = |\tilde{S}_F|$$

non vi sono ulteriori condizioni da aggiungere, se invece vale

$$|S_G| + \left|\tilde{S}_F - S_G\right| = \left|S_G + -\tilde{S}_F + S_G\right| = |2S_G - \tilde{S}_F|$$

è necessario porre

$$|2S_G - \tilde{S}_F| < 1 \quad \Rightarrow \quad \tilde{S}_F - 1 \leq 2S_G \leq \tilde{S}_F + 1$$

□

Nota.
Nel caso particolare $k = n$, si ha

$$\begin{aligned}\left|\sum_{i=0}^{n} \binom{n}{i} S_2^i S_1^{n-i}\right| &= |S_2 + S_1|^n \\ &= \left|S_G + \tilde{S}_F - S_G\right|^n \\ &= \left|\tilde{S}_F\right|^n\end{aligned}$$

ma questo si sapeva già, in quanto sotto queste condizioni si ha che vale, dal lemma (3.1.8)

$$U_n^k = F(U_{n-1}^{k-1}) = F^n(U_0)$$

pertanto anche la stabilità dipende unicamente dal propagatore $\mathcal{F}$.

Lemma 3.2.7.
Fissato $n \in \mathbb{N}$, si ha che vale

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

Dimostrazione.
La dimostrazione procede per ambedue le espressioni per induzione, nel caso $n = 0$ la prima proprietà è ovvia, in quanto vale $\binom{0}{0} = 1$. Supponendo che la formula sia corretta fino al passo n, verifichiamo cosa succede al passo $n + 1$, sfruttando il principio di addizione

$$\sum_{k=0}^{n+1} \binom{n+1}{k} = \sum_{k=1}^{n+1} \binom{n}{k-1} + \sum_{k=0}^{n} \binom{n}{k} = 2^n + 2^n = 2^{n+1}$$

□

Teorema 3.2.8 (Condizioni di stabilità)**.**

Dato il problema modello

$$\begin{cases} u' = \lambda u, & \mathrm{Re}(\lambda) < 0 \\ u(0) = 1 \end{cases}$$

con λ reale, siano S_G e S_F le funzioni di stabilità, rispettivamente, dei metodi G ed F con parametro $(h\lambda)$. Allora si ha che l'algoritmo Parareal risulta stabile per ogni iterata se

$$\max\left(|S_G|, \ \left|\tilde{S}_F - S_G\right|\right) = \alpha < \frac{1}{2}$$

Dimostrazione.
Data la funzione di stabilità

$$S_H(h\lambda) = \sum_{i=0}^{k} \binom{n}{i} \left[\left(\tilde{S}_F - S_G \right) (h\lambda) \right]^i S_G^{n-i}(h\lambda)$$

Poichè vale, sfruttando il lemma precedente (lemma (3.2.7))

$$\left| \sum_{i=0}^{k} \binom{n}{i} S_2^i S_1^{n-i} \right| \leq \sum_{i=0}^{n} \binom{n}{i} \alpha^n \leq (2\alpha)^n \to 0$$

si ha la tesi. □

Teorema 3.2.9 (Condizioni di stabilità)**.**

Dato il problema modello

$$\begin{cases} u' = \lambda u, & \mathrm{Re}(\lambda) < 0 \\ u(0) = 1 \end{cases}$$

con λ reale, siano S_G e S_F le funzioni di stabilità, rispettivamente, dei metodi G ed F con parametro $(h\lambda)$. Allora si ha che l'algoritmo Parareal risulta stabile per ogni iterata se

$$\max\left(|S_G|, \left| \tilde{S}_F \right| \right) = \alpha < \frac{1}{3}$$

Dimostrazione.
Si ha

$$\begin{aligned} S_H(h\lambda) &= \sum_{i=0}^{k} \binom{n}{i} \left[\left(\tilde{S}_F - S_G \right) (h\lambda) \right]^i S_G^{n-i}(h\lambda) \\ &= \sum_{i=0}^{k} \binom{n}{i} \left[\sum_{j=0}^{i} \binom{i}{j} \tilde{S}_F^{i-j} (-S_G)^j (h\lambda) \right] S_G^{n-i}(h\lambda) \end{aligned}$$

e quindi

$$\begin{aligned} |S_H(h\lambda)| &\leq \sum_{i=0}^{k} \binom{n}{i} \left[\sum_{j=0}^{i} \binom{i}{j} |\tilde{S}_F|^{i-j} |S_G|^j (h\lambda) \right] |S_G|^{n-i}(h\lambda) \\ &\leq \sum_{i=0}^{k} \binom{n}{i} \left[\sum_{j=0}^{i} \binom{i}{j} \alpha^i \right] \alpha^{n-i} \\ &= \alpha^n \sum_{i=0}^{k} \binom{n}{i} \left[\sum_{j=0}^{i} \binom{i}{j} \right] \\ &= \alpha^n \sum_{i=0}^{k} \binom{n}{i} 2^i \\ &\leq \alpha^n \sum_{i=0}^{n} \binom{n}{i} 2^i \\ &= (3\alpha)^n \to 0 \end{aligned}$$

□

4 Scalabilità ed Efficienza del metodo Parareal

In questa sezione vogliamo presentare alcuni risultati sul metodo Parareal relativi al calcolo parallelo.
Si noti che i risultati di convergenza e stabilità presentati nelle sezioni precedenti non dipendono dal fatto che l'algoritmo sia stato implementato in modo sequenziale o parallelo.
Le proprietà qui descritte mostrano quanto possa risultare vantaggioso implementare l'algoritmo in parallelo, piuttosto che in sequenziale.
Si definiscono le seguenti quantità di base usate per valutare l'efficienza di un algoritmo

1. Memoria (M): la quantità di spazio necessaria per salvare i dati.
2. Lavoro (W): il numero di operazioni (in flops), necessari per un dato problema.
3. Velocità (V): il numero di operazioni (in flops) al secondo.
4. Tempo (T): tempo di esecuzione.
5. Costo (C): prodotto tra il numero di processori ed il tempo impiegato.
6. Numero di processori (p).
7. Quantità di dati da elaborare (n).

Indichiamo con gli indici il numero di processori usato, pertanto T_p è il tempo impiegato per risolvere un problema con p processori. Si noti che

1. Solitamente vale $M_p \geq M_1$ in quanto è necessario scambiare informazioni tra i processori, se non vi sono dati replicati si può assumere $M_p = M_1$.
2. Se l'algoritmo seriale è ottimale, ci si aspetta che valga $W_p > W_1$ per $p > 1$.
3. Poiché ogni dato verrà usato almeno una volta, ci si aspetta che valga $W \simeq M$.
4. Supponendo che i processori che lavorano in parallelo siano identici, vale $V_p = V_1 = V$.
5. La velocità dipende inversamente dalla quantità di memoria necessaria, si ha quindi $V = V(M)$ e $V\left(\frac{M}{p}\right) > V(M)$.

Al contrario del caso seriale, dove il tempo di esecuzione è composto unicamente dal tempo di computazione, o tempo di calcolo, il tempo di esecuzione per un algoritmo parallelo si divide in tempo di computazione, di idle, o di attesa, e di comunicazione. Scriviamo

$$T_p = T_{comp} + T_{idle} + T_{com} \qquad p \geq 2$$

Il tempo di comunicazione rappresenta il tempo necessario perché un messaggio viaggi da un processore all'altro, mentre il tempo di idle è il tempo in cui un processore non lavora, ad esempio perché è stato più veloce degli altri processori in quanto aveva un compito più breve, o perché sta ad esempio aspettando di ricevere dei dati da elaborare. Nel caso di un solo processore questi tempi sono nulli.

4.1 Efficienza e Scalabilità

Si definisce come efficienza la quantità

$$E_p = \frac{C_1}{C_p} = \frac{T_1}{pT_p} = \frac{W_1}{W_p}\frac{V\left(\frac{M}{p}\right)}{V(M)}$$

e come speedup la quantità

$$S_p = \frac{T_1}{T_p} = pE_p$$

Lo speedup e l'efficienza misurano quanto risulti conveniente risolvere lo stesso problema in parallelo piuttosto che in seriale, nel caso ideale si ha $E_p = 1$ e $S_p = p$.
Si ha

1. Se V è costante, allora, poiché $W_p > W_1$, si ha $E_p < 1$ e $S_p < p$.

2. Se $V\left(\frac{M}{p}\right) > V(M)$, allora si ha $E_p > 1$ e $S_p > p$.
   Questo accade sopratutto nei problemi dove è richiesta molta memoria M, non disponibile in un singolo processore, mentre $\frac{M}{p}$ si.

3. Per semplicità, per l'analisi del algoritmo si assumerà $V = 1$ indipendente da M e $\frac{M}{p}$.

4. Si ha $n \geq p$, dove n rappresenta il numero di partizioni del partizionamento fine su tutto l'intervallo, ovvero $\frac{\Delta_0 T}{\Delta_2 T}$.

Per calcolare l'efficienza del metodo Parareal, come per qualsiasi altro algoritmo parallelo si devono tenere conto di molte variabili, pertanto per semplificare i conti, si assume che

1. Le comunicazioni tra i processori siano istantanee.
   Di conseguenza non verrà tenuto conto nemmeno del numero di comunicazioni necessarie. Nella sezione successiva verrà approfondito questo aspetto del metodo Parareal.

2. Il costo del calcolo di un passo di un metodo classico sia costante.
   Per gli algoritmi espliciti invece l'ipotesi è ben posta, si veda l'esempio (4.1.1).
   Nel caso di metodi impliciti spesso si fa uso di metodi iterativi, come ad esempio punto fisso o Newton, che non hanno un tempo di esecuzione costante in quanto vengono eseguiti in modo iterativo fino al raggiungimento di una certa soglia o ad un numero massimo di iterazioni.

Esempio 4.1.1.
Si consideri, ad esempio, il caso di Eulero esplicito

$$\begin{cases} \dfrac{U_{n+1} - U_n}{\Delta T} = f(U_n) \\ U_0 = u_0 \end{cases} \Rightarrow \begin{cases} U_{n+1} = G(f, \Delta T, U_n) = U_n + \Delta T f(U_n) \\ U_0 = u_0 \end{cases}$$

l'algoritmo è dato da

```
U(0) = u(0)                    % O(1)
for i=1:N-1
   U(i+1) = G(U(i))            % O(1) per (N-1) volte
end
```

e quindi il costo totale è dato da $O(1) + (N-1)O(1) = O(N)$, e poiché si ha $N\Delta_1 T = \Delta_0 T$, scriviamo che il costo d'esecuzione è $O\left(\frac{\Delta_0 T}{\Delta_1 T}\right)$, oppure, per essere più precisi, il costo è $C_G \frac{\Delta_0 T}{\Delta_1 T}$, dove $C_G > 0$ è una qualche costante che dipende da G.

Riscriviamo l'algoritmo del metodo Parareal in *modo parallelo*, supponendo di avere $p = N$ processori. I relativi costi sono stati commentati

Prima implementazione

```
T=[T0:delta_T:Tmax]
UG = G(f,T,u0);                  % O(Delta0 T/Delta1 T)
UG_old = UG;                     % O(1)
UG_new = UG;                     % O(1)
for k=1:K
   foreach I in [0,T]        % ogni proc valuta un solo I
      TF = [T(j): delta_t: T(j+1)];  % O(1)
      UF = F(f, TF, UG(j));          % O(Delta1 T/ Delta2 T)
      UF_end(j+1) = UF(end);         % O(1)
   end
```

```
    for j=1:N
        [T_tmp, UG_new_tmp] = G(f,[T(j), T(j+1)],UG(j-1));   % O(1)
        UG_new(j) = UG_new_tmp(end);                          % O(1)
        UG(j) = UG_new(j) + UF_end(j) - UG_old(j);            % O(1)
    end
    UG_old = UG_new;                % O(1)
end
```

Il tempo complessivo è quindi dato da

$$T_p = O\left(\frac{\Delta_0 T}{\Delta_1 T}\right) + O\left(K\frac{\Delta_1 T}{\Delta_2 T}\right) + O\left(K\frac{\Delta_0 T}{\Delta_1 T}\right) = O\left((K+1)\frac{\Delta_0 T}{\Delta_1 T} + K\frac{\Delta_1 T}{\Delta_2 T}\right)$$

oppure, tenendo conto delle costanti dei singoli propagatori,

$$T_p = (K+1)C_G\frac{\Delta_0 T}{\Delta_1 T} + KC_F\frac{\Delta_1 T}{\Delta_2 T}$$

Mentre per T_1, in seriale, si ha

$$T_1 = C_F\frac{\Delta_0 T}{\Delta_2 T} = pC_F\frac{\Delta_1 T}{\Delta_2 T}$$

Calcoliamo quindi lo speedup

$$S = \frac{T_1}{T_p} = \frac{C_F\frac{\Delta_0 T}{\Delta_2 T}}{(K+1)C_G\frac{\Delta_0 T}{\Delta_1 T} + KC_F\frac{\Delta_1 T}{\Delta_2 T}} = \frac{1}{(K+1)\frac{C_G}{C_F}\frac{\Delta_2 T}{\Delta_1 T} + K\frac{\Delta_1 T}{\Delta_0 T}}$$

e la efficienza, avendo $p = N = \frac{\Delta_0 T}{\Delta_1 T}$ processori, è data da

$$E = \frac{S}{p} = \frac{1}{(K+1)\frac{C_G}{C_F}\frac{\Delta_2 T}{\Delta_1 T} + K\frac{\Delta_1 T}{\Delta_0 T}} \cdot \frac{\Delta_1 T}{\Delta_0 T} = \frac{1}{(K+1)\frac{C_G}{C_F}\frac{\Delta_0 T \Delta_2 T}{(\Delta_1 T)^2} + K}$$

in particolare si ha quindi

$$E \leq \frac{1}{K}$$

Si ha che E, e S, in funzione di K, sono monotone decrescenti, questo sottolinea l'importanza di diminuire il più possibile il numero di iterazioni nonostante il metodo sia esatto a N passi. Al numero massimo di iterazioni si avrebbe

$$\begin{cases} E \simeq \frac{1}{N} = \frac{\Delta_1 T}{\Delta_0 T} \\ S \simeq 1 \end{cases}$$

Denotando con $n = \frac{\Delta_0 T}{\Delta_2 T}$ la quantità di dati e con $p = \frac{\Delta_0 T}{\Delta_1 T}$ il numero di processori, si può scrivere

$$\begin{cases} S = \frac{T_1}{T_p} = \frac{C_F n}{(K+1)C_G p + KC_F\frac{n}{p}} \\ E = \frac{T_1}{pT_p} = \frac{C_F n}{(K+1)C_G p^2 + KC_F n} \end{cases} \tag{7}$$

Per studiare lo scaling dell'algoritmo, supponendo di avere K costante, si vuole che per $p \to \infty$ si abbia $E = O(1)$. Si ottiene quindi

$$n = O(p^2)$$

da cui segue

$$p\frac{\Delta_1 T}{\Delta_2 T} = \frac{\Delta_0 T}{\Delta_2 T} = O(p^2) \qquad \Rightarrow \qquad \frac{\Delta_1 T}{\Delta_2 T} = O(p)$$

Con questa relazione tra i dati e il numero dei processori, l'algoritmo risulta efficiente, ma poco scalabile, si ha infatti

$$W_1(n(p)) \simeq T_1 = C_F n = O\left(p^2\right)$$

4.2 Tempi di comunicazione

In questa sezione si vuole fare una breve analisi sui tempi di comunicazione, si deve però tenere conto che questo dipendono fortemente dalla tipologia di network che si sta usando.
Innanzitutto è necessario notare che durante l'esecuzione dell'algoritmo vengono effettuate tre tipi di comunicazioni differenti.

1. Alla iterata $k = 0$, il primo processore applica il propagatore G per determinare i dati iniziali delle future iterate. Oltre a comunicare i dati iniziali, esso dovrà anche determinare i sottodomini su cui deve lavorare ogni processore. In totale vanno inviati quindi 3 dati, che compongono gli estremi dell'intervalli e il dato iniziale dei sottoproblemi. È possibile distribuire tali dati con una scatter di dimensione $2 + d$, dove $d \geq 1$ è la dimensione del sistema della ODE da risolvere.

2. Durante le iterate successive tutti i processori calcolano la soluzione con il propagatore F, dopodiché è necessario inviare la soluzione al primo processore. Non serve invece inviare informazioni sul dominio, quindi è sufficiente usare una gather di dimensione d.

3. Dopo aver calcolato, in modo sequenziale, gli aggiornamenti per i dati iniziali, essi vanno nuovamente distribuiti ai processori, si usa nuovamente con una scatter, stavolta di dimensione d, in quanto le informazioni sugli intervalli rimangono invariate, a meno di non essere giunti alla iterata finale

Il costo di una scatter e di una gather di dimensioni L è, nel caso di un ipercubo

$$T_{com} = t_s \log_2(p) + t_w L(p-1)$$

In totale è necessario effettuare K scatter e K gather, di dimensioni, rispettivamente $2 + d$, d e d, quindi si ha

$$\begin{aligned} T_{com} &= [t_s \log_2(p) + t_w(2+d)(p-1)] + (2K-1)[t_s \log_2(p) + t_w d(p-1)] \\ &= 2Kt_s \log_2(p) + 2Kt_w d(p-1) + 2t_w(p-1) \end{aligned}$$

Poiché, nel caso dell'ipercubo, si ha $T_{com} = O(pK)$, i tempi di comunicazione non influiscono sull'analisi asintotica dell'efficienza e scalabilità del algoritmo.

4.3 Tempi di idle

Il metodo Parareal, nella sua versione più semplice, non è ben bilanciato, in quanto vi è un tempo di idle, ad ogni iterata, almeno pari a $(K+1)C_G \frac{\Delta_0 T}{\Delta_1 T} = (K+1)C_G p$.
Questo perché mentre il primo processore calcola i dati iniziali per la risoluzione dei sottoproblemi, gli altri processori sono costretti ad attendere.
La seguente immagine mostra un esempio di esecuzione dell'algoritmo, dove ogni riga rappresenta un processore

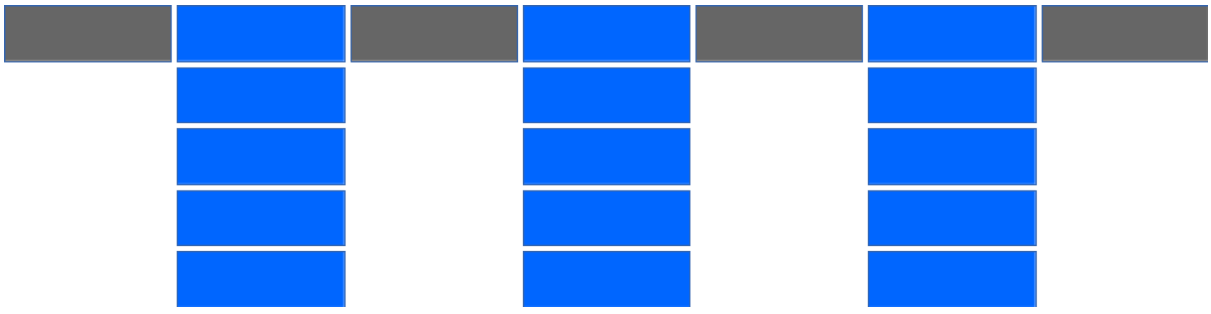


Figura 2: Tempi di lavoro dell'algoritmo Parareal

Mischiando invece il calcolo dei valori iniziali, l'invio e il ricevimento dei dati, si possono ridurre i tempi

di attesa.
Assegnando al primo processore il compito di aggiornare i dati iniziali, applicare solamente il propagatore G e inviare i dati per i sottoproblemi non appena questi siano pronti, il tempo di idle si riduce a KC_G. Agli altri processori, che applicano solamente il propagatore F, il tempo di attesa si riduce a $C_G(K+p)$ per tutti i processori escluso il primo, per il quale i tempi di idle sono pari a KC_G, supponendo che si abbia $C_G \frac{\Delta_0}{\Delta_1} \simeq C_F \frac{\Delta_1}{\Delta_2}$.

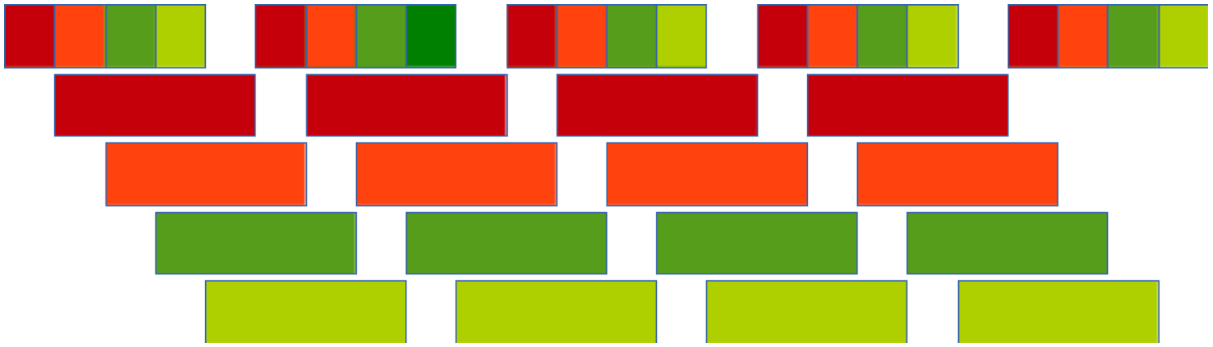


Figura 3: Tempi di lavoro dell'algoritmo Parareal con pipeline

Un altro metodo per ridurre T_{idle} è quello di sfruttare un eventuale parallelismo dell'operatore G. Se si tratta ad esempio di un metodo Runge-Kutta a più stadi è possibile dividere il lavoro su più processori. Altrimenti se si utilizza un metodo implicito per evitare i problemi di stabilità, spesso si farà ricorso ad un metodo iterativo, come ad esempio Newton, che può essere applicato su più processori con punti di partenza differenti per raggiungere più velocemente la soluzione.
Un terzo approccio per ridurre i tempi di idle è quello di far applicare il propagatore G ad ogni processore, riducendo i tempi di comunicazione, in quanto ogni processore ha calcolato anche il proprio dato iniziale. L'approccio è tuttavia sconsigliato in quanto si creerebbe una quantità di dati ridondanti pari a $O(p)$ su ogni processore, e le gather necessarie per notificare i valori ottenuto con il propagatore G fanno si che i thread si debbano comunque sincronizzare. Si ridurrebbe T_{com}, ma il comportamento asintotico rimane invariato.
Generalmente i tempi di idle peggiorano il tempo di esecuzione, applicando una pipeline o creando dei dati ridondanti si ottiene una diminuzione dei tempi di esecuzione, ma questo non influenza il comportamento asintotico.

5 Criterio di arresto

L'algoritmo Parareal è un algoritmo di tipo iterativo, per questo motivo necessita di uno o più criteri di arresto.
Nella sezione precedente si è visto che l'efficienza è inversamente proporzionale al numero di iterate, pertanto è di vitale importanza definire un criterio di arresto che rilevi quando si è giunti alla convergenza e che allo stesso tempo minimizzi il numero di iterate necessarie.

Metodo esatto

È già stato sottolineato più volte che l'algoritmo Parareal è un algoritmo iterativo esatto, ovvero per $k \geq N$ si ha

$$U_n^{k+1} = \mathcal{F}([t_{n-1}, t_n], U_{n-1}^k)$$

Di conseguenza quando k raggiunge N si può fermare l'algoritmo.
In questo modo però l'efficienza sarebbe dell'ordine di $\frac{1}{N} = \frac{1}{p}$, dove N è il numero di intervalli e quindi di processori. Non è necessario aspettare che vengano raggiunte N iterate, infatti anche il propagatore fine F commette un errore rispetto alla soluzione esatta, quindi oltre all'errore commesso dall'algoritmo Parareal rispetto al propagatore fine, vi è anche l'errore della soluzione fine rispetto alla soluzione esatta. Pertanto diciamo che l'algoritmo Parareal ha raggiunto la convergenza se l'errore tra l'algoritmo Parareal e la soluzione fine, ha lo stesso ordine di grandezza tra la soluzione fine e la soluzione esatta, oppure se l'errore tra la soluzione dell'algoritmo Parareal e la soluzione fine è minore dell'errore tra la soluzione ottenuta con il propagatore fine e il metodo Parareal.

Iterate consecutive

Uno dei metodi più usati per verificare se un algoritmo ha raggiunto la convergenza, è quello di valutare la differenza tra la soluzione approssimata a due iterate differenti. Se la differenza è sotto una certa soglia, si può assumere che la soluzione esatta non sia molto distante.
Queste considerazioni rimangono vere anche per l'algoritmo Parareal; è possibile verificare se la soluzione a due iterate differenti si trovi sotto una certa soglia per decidere se arrestare o meno l'algoritmo.
Il problema sta nel definire la soglia adeguata, in quanto si vuole minimizzare il numero di iterate per avere un algoritmo efficiente. Una soglia troppo bassa potrebbe richiedere troppe iterate, mentre una soglia troppo alta rischia di terminare l'algoritmo prima che questo abbia approssimato in maniera buona la soluzione.

Dal punto di vista pratico, grazie ai risultati di convergenza presentati nei capitoli precedenti, impostare una soglia dell'ordine di grandezza dell'accuratezza del propagatore F si è rilevato un buon metodo, anche se non si ha un buon criterio per impostare a priori una buona soglia.

Salto massimo

Un criterio simile a quello delle *iterate consecutive*, è quello di valutare i salti massimi tra la nuova predizione e la soluzione fine della iterata precedente. Questo metodo è stato proposto in [Trindade and Pereira [2006]], ma ha lo stesso difetto di quello precedente, ovvero non esiste un buon criterio per impostare a priori la soglia.

Error controll mechanism

La proposizione successiva, presentata in [Lepsa and Sandu [2010]], è attualmente il miglior criterio di arresto presente in letteratura, anche se presenta un potenziale problemi.

Proposizione 5.0.1.
Si definisca la quantità

$$I_n^k = \mathcal{F}([t_{n-1}, t_n], U_{n-1}^k) - U_n^k$$

ovvero la differenza tra la soluzione data da $\mathcal{F}$ al tempo t_n alla k-esima iterata e la soluzione del metodo Parareal allo stesso tempo e alla stessa iterata.
Sia invece

$$e_n^k = u(t_n) - U_n^k$$

ovvero l'errore tra il metodo Parareal alla k-esima iterata nel punto t_n e la soluzione esatta.
L'algoritmo Parareal raggiunge la sua accuratezza massima se $e_n^k = O((\Delta_2 T)^p)$, ovvero se si comporta asintoticamente come $\mathcal{F}$. Questo accade quando

$$\|I_n^k\| \leq \theta \|E_{F,n}^k\|, \qquad \theta \leq 1$$

dove $E_{F,n}^k$ è l'errore globale del propagatore fine sul sottointervallo $[t_{n-1}, t_n]$.

Dimostrazione.
Si definiscano le quantità

$$\begin{aligned}
e_n^k &= \gamma_n^k + E_{F,n}^k + I_n^k \\
\gamma_n^k &= u(t_n) - \phi([t_{n-1}, t_n], U_{n-1}^k) \\
&= \phi([t_{n-1}, t_n], u(t_n)) - \phi([t_{n-1}, t_n], U_{n-1}^k) \\
E_{F,n}^k &= \phi([t_{n-1}, t_n], U_{n-1}^k) - F([t_{n-1}, t_n], U_{n-1}^k) \\
I_n^k &= \mathcal{F}([t_{n-1}, t_n], U_{n-1}^k) - U_n^k
\end{aligned}$$

con ϕ risolutore esatto. Applicando uno sviluppo di Taylor si ottiene

$$\begin{aligned}
\phi([t_{n-1}, t_n], u(t_{n-1})) &= \phi([t_{n-1}, t_n], U_{n-1}^k) + \frac{\partial}{\partial u}\phi([t_{n-1}, t_n], \hat{U}_{n-1}^k) + [u(t_{n-1}) - U_{n-1}^k)] \\
&= \phi([t_{n-1}, t_n], U_{n-1}^k) + \frac{\partial}{\partial u}\phi([t_{n-1}, t_n], \hat{U}_{n-1}^k) + e_{n-1}^k
\end{aligned}$$

da cui segue

$$\|\gamma_n^k\| \leq C \|e_{n-1}^k\|$$

dove C maggiora in modo uniforme la derivata di ϕ.
Quindi si ha

$$\begin{aligned}
\|e_n^k\| &\leq C\|e_{n-1}^k\| + \|E_{F,n}^k\| + \|I_n^k\| \\
&\leq C\left(C\|e_{n-2}^k\| + \|E_{F,n-1}^k\| + \|I_{n-1}^k\|\right) + \|E_{F,n}^k\| + \|I_n^k\| \\
&= C^2\|e_{n-2}^k\| + C\left(\|E_{F,n-1}^k\| + \|I_{n-1}^k\|\right) + \|E_{F,n}^k\| + \|I_n^k\| \\
&\vdots \\
&\leq C^n\|e_0^k\| + \sum_{j=0}^{n-1} C^j \left(\|E_{F,n-j}^k\| + \|I_{n-j}^k\|\right)
\end{aligned}$$

e poichè vale $\|e_0^k\| = 0$ per ogni k, si ottiene

$$\|e_n^k\| \leq n \cdot \max\{1, C^{n-1}\} \cdot \sup_{j=0,\ldots,n-1} \left(\|E_{F,n-j}^k\| + \|I_{n-j}^k\|\right)$$

Quando $k \geq n$, si ha $I_n^k = 0$, per l'esattezza del metodo, e quindi si ha

$$e_n^k = \gamma_n^k + E_{F,n}^k \quad \Rightarrow \quad e_n^k = O((\Delta_2 T)^p)$$

dove p è l'ordine del propagatore F.
Per ottenere un comportamento di questo tipo, è sufficiente iterare su k fino a che

$$\|I_{n-j}^k\| \leq \theta \|E_{F,n-j}^k\|, \quad \theta \leq 1$$

□

Nota.
Il potenziale problema di questo criterio è che per sapere se l'algoritmo ha raggiunto la convergenza al passo k, è necessario calcolare la soluzione fine, che viene usata al passo $k+1$, di fatto è quindi necessario fare una *mezza iterata* in più, il che rovina l'efficienza visto che il comportamento dell'algoritmo sarebbe maggiorato da $\frac{1}{k+1}$ e non da $\frac{1}{k}$.
Dall'altra parte se non si è interessati solamente alla suluzione al tempo finale, ma su tutto l'intervallo, sarebbe comunque necessario calcolare una volta in più la soluzione fine, pertanto in questo caso non si avrebbe un degrado della efficienza.

Nota.
Non sono noti risultati che permettono di conoscere a priori il parametro θ, ma è possibile implementare il test con la disuguaglianza

$$\|I_n^k\| \leq \|e_{F,n}^k\|$$

anche se in questo caso si potrebbe raggiungere la convergenza ben prima che la disuguaglianza sia verificata.

6 Implementazione del metodo Parareal

Le prime implementazioni dell'algoritmo Parareal sono state scritte come codice seriale sfruttando il linguaggio ad alto livello Matlab©/Gnu Octave per verificare la validità del metodo.
Il software Matlab©[2] è un ambiente per il calcolo numerico e l'analisi statistica prodotto dalla MathWorks, mentre Gnu Octave[3], spesso abbreviato in Octave è un'applicazione software per l'analisi numerica in gran parte compatibile con Matlab©.
Successivamente, per implementare il codice in parallelo, si è fatto uso del linguaggio di programmazione Python™[4], in modo da avere a disposizione alcuni strumenti di alto livello, come ad esempio funzioni simili a quelle presenti nel linguaggio Matlab©/Octave, sopratutto grazie ai pacchetti numpy[5], assenti nei linguaggi di programmazione a basso livello, come ad esempio nei linguaggi C o C++. Tramite il pacchetto mpi4py[6] si è sfruttata la possibilità di usare le librerie MPI del cluster nemo[7] del dipartimento di matematica, per poter comunicare tra diversi processori.
Non sono state effettuate ottimizzazioni sul codice per diminuire i tempi di utilizzo, come ad esempio la compilazione da codice interpretato ad eseguibile, in quanto si è voluto concentrare lo sforzo sul tenere il codice pulito, mantenibile e facilmente ampliabile, anche per poterlo adattare a nuovi problemi.
Ad esempio le prime versioni del codice non permettevano di risolvere sistemi di ODE, mentre successivamente è stata aggiunta la possibilità di risolvere ODE a due o più dimensioni, in modo da poter trattare sistemi a livello più alto.

In questa sezione si vuole mostrare le parti principali del codice usato, anche se abbastanza autoesplicativo, ma non in maniera troppo approfondita.

6.1 Richiamare metodo Parareal

Il seguente frammento di codice mostra le librerie che sono state importate per eseguire correttamente l'algoritmo Parareal sfruttando operazioni di alto livello simili a quelle di Matlab©/Octave e le librerie MPI, oltre a delle librerie scritte al momento per testare l'algoritmo con diversi metodi numerici e problemi

```
import math
import numpy
from copy    import copy, deepcopy
from mpi4py import MPI

import prop
import testf
import parareal
```

1. Il pacchetto *math* contiene gli operatori matematici basilari, come ad esempio la possibilità di elevare a potenza con base e.

2. Il pacchetto *numpy*, che sta per numerical python, è quella che contiene tutte le operazioni ad alto livello spesso usate in ambito numerico, come ad esempio la utility linspace, risolutori di sistemi lineari e via dicendo.

3. Il pacchetto *mpi4py* è necessario per poter usare le librerie MPI.

I seguenti pacchetti sono stati scritti esplicitamente per il metodo Parareal

4. Il pacchetto *prop* contiene diversi propagatori, come ad esempio i metodi Runge-Kutta.

[2] http://www.mathworks.it/
[3] https://www.gnu.org/software/octave/
[4] http://www.python.org/
[5] http://www.numpy.org/
[6] http://mpi4py.scipy.org/
[7] http://cluster.mat.unimi.it/

5. Il pacchetto *testf* contiene diversi problemi definiti su specifici intervalli e corredati di soluzione, in modo da avere una serie di casi su cui testare il metodo Parareal.

6. Il pacchetto *parareal* contiene l'implementazione del metodo Parareal. Si è scelto di scrivere l'algoritmo all'interno di un pacchetto a parte in modo da semplificare la manutenzione e per essere in grado di poterlo richiamare facilmente anche con altri programmi.

Il codice è distribuito in diversi pacchetti, che vengono richiamati dal file *pypararеal.py*, il file principale per impostare il problema da approssimare, i propagatori per il metodo Parareal, oltre ad altri parametri. La prima operazione da fare è quella di assegnare ad ogni processore la dimensione del problema (ovvero il numero di processori), il proprio rango e il proprio nome

```
size = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
name = MPI.Get_processor_name()
```

Successivamente si definisce il numero di intervalli (si noti che *num_int* è sempre pari al numero di processori, mentre il numero di intervalli per il propagatore fine non è dipendente dal numero di processori) e il numero massimo di iterazioni che si intende effettuare.
A questo punto si sceglie un problema da risolvere, in questo caso *PROBMOD* identifica il problema modello di parametro *par*, che è stato posto uguale a -5, mentre successivamente vengono assegnati la funzione f, la sua derivata in y (viene utilizzata nel caso si usi un metodo impliciti associato ad un metodo iterativo di tipo Newton), il dato iniziale *y0*, la soluzione esatta, per poter analizzare successivamente l'errore del metodo, e l'intervallo su cui è definito il problema tramiti gli estremi *t0* e *tmax*.

```
num_intF = 100
kmax     = size

numfun   = 'PROBMOD'
par      = -5
f,dfy,t0,tmax,y0,sol = testf.fun(numfun,par)
```

A questo punto è necessario scegliere il metodo con cui risolvere il problema, vale a dire un propagatore G ed un propagatore F. Nel codice è presente anche un propagatore I, nel caso si volesse usare un propagatore adattivo per la iterata $k = 0$, in caso contrario basta, come nel codice mostrato, porlo uguale a G.
La funzione *RKcomp*, presente nel pacchetto *prop* fornisce le componenti necessarie per applicare i metodi di Runge-Kutta, come ad esempio la matrice di Butcher, mentre *RK*, presente anche esso nel pacchetto *prop*, applica il metodo Runga-Kutta.
Infine il parametro *toll* è la tolleranza che viene usata come criterio di arresto tra due iterate. Se non si vuole usare nessuna tolleranza è sufficiente assegnare un valore negativo, in questo caso gli si è assegnato un valore pari all'ordine del metodo di F.

```
RKG = 'BUTCHERLOBATTO'
RKF = 'MERSON'

BG,aG,cG,ordineG,esplicitoG = prop.RKcomp(RKG)
G = lambda f,T,y0: metodi.RK(f,T,y0,BG,aG,cG,esplicitoG,dfy)
I = G

BF,aF,cF,ordineF,esplicitoF = prop.RKcomp(RKF)
F = lambda f,T,y0: metodi.RK(f,T,y0,BF,aF,cF,esplicitoF,dfy)

toll = (((tmax-t0+0.0)/size)/num_intF)**(ordineF)      # ~ h_F^ordine
```

A questo punto è possibile richiamare il metodo Parareal, che si occupa di salvare la soluzione U su tutto l'intervallo ad ogni iterata nell'array *Y_glob*, mentre *YF_glob* contiene la soluzione del propagatore F su ogni intervallo e ad ogni iterata. *T_glob* e *TF_glob* contengono rispettivamente i punti sui quali è stato approssimato il problema

```
T_glob,Y_glob,TF_glob,YF_glob =
    parareal.kernel(f,t0,tmax,y0,num_int,num_intF,kmax,I,G,F,toll)
```

Il motivo principale per il quale questa implementazione del metodo Parareal restituisce così tante informazioni è per poter, in un secondo momento, studiare l'andamento del metodo e lo studio dell'errore. Tali informazioni servono se si vuole avere una approssimata della soluzione su tutto l'intervallo, mentre se si è interessati solamente alla soluzione al tempo finale è possibile modificare il codice del programma.

6.2 Casi test

Mostriamo, a titolo d'esempio, l'implementazione di un paio di casi test. Il pacchetto *testf* consiste in una struttura *if-else* atta a cercare quale problema si è selezionato, le librerie usate sono *numpy* e *math*, che vengono anche richiamate dal metodo Parareal.
Il seguente è il problema modello, ovvero

$$\begin{cases} u' = \lambda u, & \mathrm{Re}(\lambda) < 0 \\ u(0) = 1 \end{cases}$$

```
if (es == 'PROBMOD'):
    f    = lambda t,y: par*y
    dfy  = lambda t,y: par
    t0   = 0
    tmax = 7/9.0
    y0   = [1]
    sol  = lambda t:   numpy.power(math.e,par*t)
```

Il seguente problema a due dimensioni e periodico

$$\begin{cases} u' = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \omega y \\ u_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \omega \end{cases}$$

scritto per risolvere principalmente il seguente problema di secondo ordine

$$\begin{cases} u'' + \omega^2 u = 0 \\ u_0 = 0 \\ u_0' = \omega \end{cases}$$

è implementato nel seguente modo

```
elif (es== 'PERIODICO'):
    f    = lambda t,y: numpy.dot([[0,1], [-1,0]], y)*par
    dfy  = lambda t,y: [[0,par], [-par,0]]
    t0   = 0
    tmax = 7/9.0
    y0   = [0, 1]
    sol  = lambda t:   ([numpy.sin(t*par), numpy.cos(t*par)])
```

6.3 Propagatore

Il pacchetto *prop* contiene diversi metodi numerici che possono essere usati come propagatori. Principalmente sono stati usati i metodi Runge-Kutta, sia espliciti che semi-impliciti. Per risolvere i sistemi impliciti si è fatto uso del metodo delle tangenti di Newton.

```
dim      = numpy.size(y0)
Y        = numpy.zeros((dim,numpy.size(T)))
Y[:,0] = y0

ns = numpy.size(a)
K  = numpy.zeros([dim,ns])
bd = numpy.diag(B)
B  = B - numpy.diag(bd)
for n in range(0,numpy.size(T)-1):
   h = T[n+1]-T[n]+0.0
   for r in range(0,ns):
      it=0
      err = toll+1
      u   = K[:,r]
      tnr = T[n]+a[r]*h
      aux = Y[:,n]+h*numpy.dot(K,numpy.transpose(B[r][:]))
      while ( (it<nitmax) & (err>toll) ):
         J     = numpy.eye(dim)-h*bd[r]*numpy.array(dfy(tnr,aux+h*bd[r]*u))
         tn    = f(tnr,aux+h*bd[r]*u)-u
         delta = numpy.linalg.solve(J, tn)
         err   = numpy.linalg.norm(delta,numpy.inf);
         it    = it+1
         u     = u + delta
      #end while
      K[:,r] = u
      if ( (it==nitmax) & (err>toll) ):
         print '#############################################################'
         print '# Metodo RK implicito ha avuto problemi a convergere #'
         print '#############################################################'
         #return
      #end if
   #end for r=1:ns
   Y[:,n+1] = Y[:,n] + h*numpy.dot(K,numpy.transpose(c))
#end for n=1:length(T)-1
```

La funzione *RKcomp* invece consiste in una struttura *if-else* atta a cercare quale metodo si è scelto, ad esempio

```
if(metodo=='RK3'):
   B = [[0, 0, 0], [1.0/2, 0, 0], [-1, 2, 0] ]
   a = [0, 1.0/2, 1]
   c = [1.0/6, 4.0/6, 1.0/6]
   ordine    = 3
   esplicito = True
```

6.4 Codice Parareal

In questa sezione è riportato il codice dell'algoritmo Parareal vero e proprio.
Inanzitutto viene definito

1. Il numero di intervalli, che è pari al numero di processori.
2. La dimensione del problema, che servirà per creare degli array delle dimensioni adatte per salvare le soluzioni ad ogni iterata.
3. *diff_iter*, ovvero la variabile che verifica ad ogni iterata se è stata raggiunta o meno la tolleranza. Per poter entrare nel ciclo in maniera consistente si pone tale costante ad un valore maggiore della tolleranza.

```
num_int   = size
dim       = numpy.size(y0)
diff_iter = toll+1
```

Successivamente il primo processore calcola la soluzione con il metodo G (se I è stato messo uguale a G, oppure si usa un metodo adattivo) e manda agli altri processori, che si preparano a ricevere, i dati che gli interessano, tra cui anche gli elementi dell'intervallo sui quali è definito il sottoproblema. Si creano inoltre gli array dove salvare i valori delle iterate successive sia del propagatore G, sia del propagatore F.

```
if (rank==0):
   # creo dati iniziali
   T     = numpy.linspace(t0, tmax, num_int+1)
   T,YG = I(f,T,y0)
   part = numpy.zeros((num_int,2))
   for j in range(0,num_int):
      part[j,:] = [T[:-1][j], T[1:][j]]

   # creo spazio per prossime iterate
   YG_new  = numpy.zeros((dim,numpy.size(T)))
   YG_old  = deepcopy(YG)

   T_glob  = T
   Y_glob  = numpy.zeros((dim,kmax+1,numpy.size(T)))
   YF_glob = numpy.zeros((dim,kmax,num_intF+1,num_int))

   Y_glob[:,0,:] = deepcopy(YG)
else:
   part = None
   YG   = None
# end if-else
   part = MPI.COMM_WORLD.scatter(part, root=0)
   YG   = MPI.COMM_WORLD.bcast(YG, root=0)
```

A questo punto ci si prepara ad entrare nel ciclo, la variabile *condition* vien impostata a *True*, se durante una iterata un criterio segna di terminare l'algoritmo, questa variabile verrà impostata a *False* per terminare l'esecuzione, mentre k conta il numero di iterate effettuate, che a priori sarà minore di kmax.

```
condition = True
k         = 0
while (condition):
   k += 1
```

All'interno del ciclo ogni processore calcola la soluzione fine e manda il propri dati al primo processore, in modo che questo possa usarli per calcolare i futuri dati iniziali.
In questo caso *YF_end* contiene i dati che servono per le iterate successive, mentre *YFF* contiene la

soluzione su tutto il sottointervallo.
In questo caso, la quantità *YFF* viene inviata al primo processore perché si è interessati ad avere una approssimata del problema su tutto l'intervallo.

```
if (k<=rank+1):
    TF     = numpy.linspace(part[0],part[1],num_intF+1)
    TF,YF = F(f,TF,YG[:,rank])

YF_end = MPI.COMM_WORLD.gather(YF[:,-1], root=0)
YFF    = MPI.COMM_WORLD.gather(YF, root=0)
```

A questo punto non resta che aggiornare i dati iniziali per l'iterata successiva e i dati per l'output. Ambedue i lavori vanno fatti in sequenziale, si ha

```
if (rank==0): # aggiornamento dati iniziali e dati in uscita
    for j in range(0,num_int):
        YF_glob[:,k-1,:,j] = YFF[j][:]
    YF_end = [y0]+YF_end

    YG[:,0]     = y0
    YG_new, YG = seq_update(f,T,YG,y0,YG_old,YF_end,G)
    YG_old      = deepcopy(YG_new)

    Y_glob[:,k,:] = deepcopy(YG)
    diff_iter     = numpy.max(numpy.abs(YG -Y_glob[:,k-1,:]))
#end if (rank==0)
```

La funzione *seq_update* è stata implementata nel seguente modo

```
dim      = numpy.size(y0)
YG[:,0] = y0
YG_new  = numpy.zeros((dim,numpy.size(T)))
for j in range(1,numpy.size(T)):
    T_tmp = numpy.linspace(T[j-1],T[j],1+1) # metodo ad 1 passo
    T_tmp,YG_new_tmp = G(f,T_tmp,YG[:,j-1])
    YG_new[:,j] = YG_new_tmp[:,-1]
    YG[:,j] = YG_new[:,j] + YF_end[:][j] - YG_old[:,j]
return YG_new, YG
```

Prima di passare alla iterata successiva, si verifica che la variabile *condition* non debba diventare falsa, in caso negativo viene inviato il dato iniziale per la iterata successiva

```
if (rank==0):
    if (k>=size):
        condition &= False
    if (k >= kmax):
        condition &= False
    if (diff_iter<toll):
        condition &= False
#end if check condition
condition = MPI.COMM_WORLD.bcast(condition, root=0) # notifico se iterare
if (condition==True):
    YG = MPI.COMM_WORLD.bcast(YG, root=0)
```

Una volta uscito dal ciclo *while*, rimangono da sistemare i dati in uscita, in quanto non è necessario restituire un array contenente *kmax* iterate, ma solamente *k*

```
TFF = MPI.COMM_WORLD.gather(TF, root=0)
if(rank==0): # ridurre Y_glob, YF_glob nel caso non avessi raggiunto kmax
    TF_glob = TFF
    return T_glob,Y_glob[:,0:k+1,:],TF_glob,YF_glob[:,0:k,:]
```

6.5 Tabelle e grafici

Il codice Python™ utilizzato per implementare ed analizzare il comportamento del metodo Parareal è in grado di scrivere tabelle LaTeX tramite il pacchetto matrix2latex[8] e disegnare e salvare grafici grazie al pacchetto matplotlib[9].
Tali pacchetti sono stati usati per riportare i risultati su questo elaborato, ma non sono necessari per la comprensione ed esecuzione del metodo Parareal, pertanto il codice utilizzato non verrà descritto qui, essendo di importanza secondaria.

6.6 Ottenere il codice sorgente

È possibile scaricare il codice sorgente di *pypararеal* dal repository https://code.google.com/p/pypareareal/, insieme anche ad una copia pdf di questo elaborato.

[8] https://code.google.com/p/matrix2latex/
[9] http://matplotlib.org/

7 Risultati numerici su macchine parallele

In questa sezione verranno presentati alcuni risultati pratici sull'uso del metodo Parareal.

I tempi di esecuzione sono stati ottenuti tramite l'esecuzione del codice sul Cluster Nemo del dipartimento di matematica della Università degli Studi di Milano, maggiori informazioni sulla configurazione del cluster sono reperibili alla relativa homepage[10].
Il calcolo della soluzione approssimata, degli errori e dell'ordine di convergenza sono indipendenti dalla macchina e dal numero di processori, pertanto possono essere replicati su un qualsiasi computer sul quale sono disponibili le librerie MPI, Python^{TM} e i pacchetti richiesti. I tempi di esecuzione dipendono invece dall'hardware usato, pertanto tali risultati potrebbero differire su macchine differenti.

Nota (Grafici e Tabelle)**.**
I grafici e le tabelle mostrate nelle successive sezioni rappresentano i tempi di calcolo, gli errori e i tempi di esecuzione che si ottengono eseguendo il codice Parareal implementato.
Per lo studio degli errori si sono definite le quantità

1)
$$\begin{cases} e_{1,k} = \|U_N^k - u(t_N)\| \\ e_{2,k} = \|U_N^k - U_N^N\| \end{cases}$$

Tali quantità rappresentano l'errore sull'ultimo nodo dell'intervallo, ovvero sul punto $t_N = T_{\max}$. La quantità $e_{1,k}$ rappresenta l'errore tra il metodo Parareal e la soluzione esatta, mentre $e_{2,k}$ rappresenta l'errore tra il metodo Parareal e la soluzione ottenuta in seriale con il propagatore F, o equivalentemente con la soluzione ottenuta con il metodo Parareal dopo N iterate.

2)
$$\begin{cases} lh = \log(h) = \log(\Delta_1 T) \\ le_{1,k} = \log(e_{1,k}) \\ le_{2,k} = \log(e_{2,k}) \end{cases}$$

Tali quantità rappresentano rispettivamente i logaritmi degli errori definiti prima e il logaritmo dell'ampiezza dell'intervallo. Siamo interessati a queste quantità poiché permettono di verificare se l'algoritmo Parareal rispetta l'ordine di convergenza previsto dalla teoria, come mostrato nel teorema (3.1.9), si ha
$$e_k \simeq ch^{m(k+1)}e^{Chk}$$
da cui segue
$$\frac{le_k}{lh} \simeq \frac{\log(c)}{lh} + m(k+1) + \frac{Ckh}{lh} \Rightarrow \frac{le_k}{lh * (k+1)} \simeq \frac{\log(c)}{lh * (k+1)} + m + \frac{Ch}{lh}$$
e sul lato destro tutti i termini tendono a 0 per lh crescente, escluso il termine m.

3)
$$\begin{cases} dle_{1,k} = \log(e_{1,k}) - \log(e_{1,k-1}) \\ dle_{2,k} = \log(e_{2,k}) - \log(e_{2,k-1}) \end{cases}$$

Nuovamente siamo interessati a queste quantità poiché permettono di verificare se l'algoritmo Parareal rispetta l'ordine di convergenza previsto dalla teoria (3.1.9), si ha infatti
$$e_k \simeq ch^{m(k+1)}e^{Chk}$$
da cui segue
$$\frac{le_k}{lh} \simeq \frac{\log(c)}{lh} + m(k+1) + \frac{Ckh}{lh} \Rightarrow \frac{le_k}{lh \cdot (k+1)} \simeq \frac{\log(c)}{lh \cdot (k+1)} + m + \frac{Ch}{lh}$$

[10] http://cluster.mat.unimi.it/

e quindi

$$\frac{le_{k+1} - le_k}{lh} \simeq m + \frac{Ch}{lh}$$

I grafici relativi alla soluzione mostrano, fissati i passi $\Delta_1 T$, $\Delta_2 T$ e l'iterata k, la soluzione esatta $u(t)$ sul dominio $[T_0, T_{\max}]$, la soluzione approssimante U_n^k, e le soluzioni dei sottoproblemi ottenuti tramite il propagatore F sugli intervalli $[t_n, t_{n-1}]$.
I grafici relativi agli errori mostrano su ogni intervallo $[t_n, t_{n+1}]$ l'andamento dell'errore in scala logaritmica. Anche in questo caso si sono considerati l'errore $\|U_n^k - u(t_n)\|$ e $\|U_n^k - U_n^N\|$. Nel caso degli errori tra l'algoritmo Parareal e il propagatore F, si ha che l'errore è nullo quando $k \geq n$. Avendo mostrato i risultati in scala logaritmica i grafici possono sembrare incompleti,anche se a volte, causa errori di arrotondamento, tali quantità non sono esattamente nulle, ma valore vicini all'epsilon macchina, nel casi testati tale valore era intorno a 1×10^{-17}.

Nota (Scaled speedup e Strong scaling)**.**
Solitamente, quando si vuole analizzare la bontà di un algoritmo al variare del numero p di processori, si eseguono due tipi di test indipendenti.

1. Quando si fissa la dimensione del problema da risolvere, e si fa variare il numero dei processori, si sta testando lo *strong scaling*. Il comportamento atteso è una diminuzione dei tempi finché il problema risulta *grosso* in confronto al numero dei processori, quando invece si stanno usando talmente tanti processori che ognuno di essi ha un carico di lavoro molto esiguo, allora i tempi tendono ad aumentare, in quanto i tempi di comunicazione battono quelli di calcolo.

2. Quando si fissa, per ogni processori, la dimensione del sottoproblema, e si fa variare il numero dei processori, allora si sta testando lo *scaled speedup*. In questo modo la dimensione del problema originale non è fissa, ma aumenta in maniera lineare con il numero dei processori.
   Si dice che un algoritmo scala bene quando i tempi di esecuzione non aumentino in modo lineare all'aumentare dei processori, ma più lentamente.

7.1 Convergenza del metodo

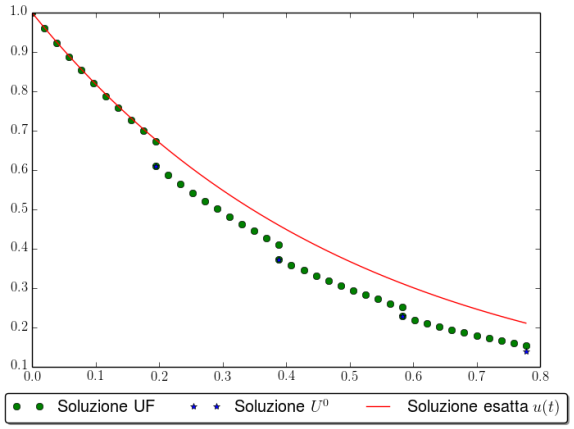
Come primi risultati mostriamo come si comporta l'algoritmo Parareal con il problema modello

$$\begin{cases} u' = \lambda u, \quad \mathrm{Re}(\lambda) < 0 \\ u(0) = 1 \end{cases}$$

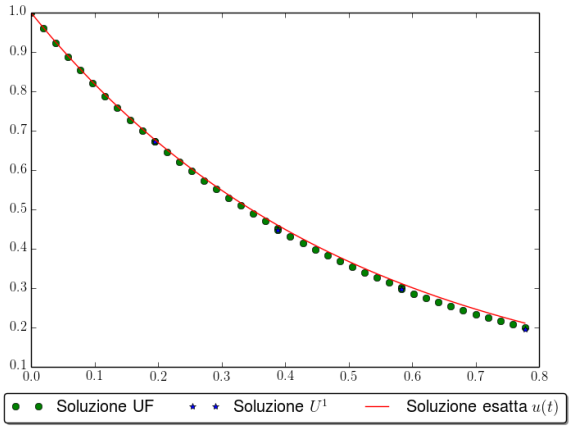
definito sull'intervallo $I = \left[0, \frac{7}{9}\right]$ con parametro $\lambda = -2$ e $n = 4$ processori. Come propagatori F e G usiamo eulero esplicito, e l'ampiezza degli intervalli è data da $\Delta_1 T = \frac{7}{36}$ e imponiamo $\Delta_2 T = \frac{7}{360}$.
Verifichiamo inanzitutto che i ambedue i propagatori non hanno problemi di stabilità, si ha

$$\begin{cases} S_G(\Delta_1 T\lambda) = |1 + \Delta_1 T\lambda| = \frac{11}{18} < 1 \\ S_G(\Delta_2 T\lambda) = |1 + \Delta_2 T\lambda| = \frac{173}{180} < 1 \end{cases}$$

Con questi parametri quindi ambedue i propagatori risultano stabili. Eseguendo l'algoritmo, si ottengono i seguenti risultati numerici



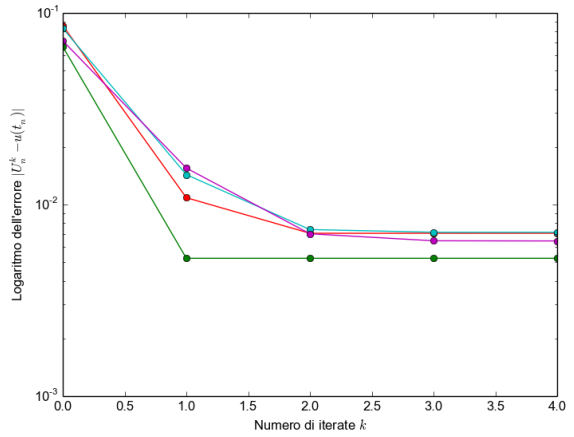
(a) Iterata $k = 0$



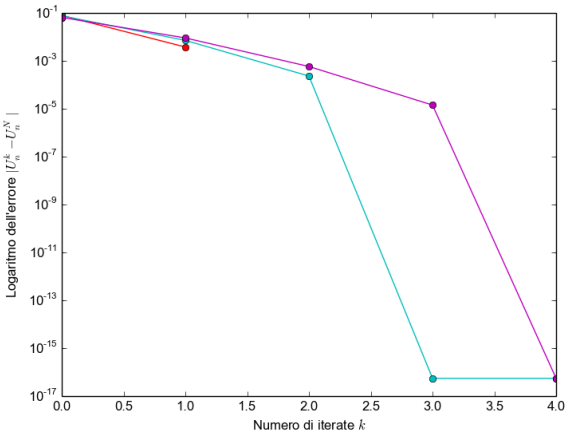
(b) Iterata $k = 1$

Figura 4: Metodo EE-EE su problema modello

Il seguente grafico mostra per ogni valore U_n^k (escluso il dato iniziale, ovvero U_0^k), l'andamento dell'errore con la soluzione esatta in scala logaritmica, mentre il secondo grafico è l'andamento dell'errore in confronto alla soluzione ottenuta con il metodo sequenziale



(a) $\|U_n^k - u(t_n)\|$



(b) $\|U_n^k - U_n^N\|$

Figura 5: Andamento errore in scala logaritmica

La seguente tabella inoltre riporta l'andamento dell'errore del metodo al variare dell'iterata k sull'ultimo intervallo (identifichiamo con e_1 l'errore tra ciò che abbiamo ottenuto con il metodo e la soluzione esatta, con e_2 l'errore tra ciò che si è ottenuto con il metodo Parareal e il metodo seriale, le_1 e le_2 sono i rispettvi logaritmi, lh è il logaritmo di h e k la k-esima iterata a partire da 0)

Tabella 1: Errore con $p = 4$ processori

k	0	1	2	3	4
$e_{1,k}$	0.07160	0.01549	0.00703	0.00646	0.00645
$e_{2,k}$	0.06514	0.00904	0.00058	1.4×10^{-05}	5.5×10^{-17}
$le_{1,k}/lh$	1.61004	2.54456	3.02682	3.07817	3.07951
$le_{2,k}/lh$	1.66773	2.87344	4.54909	6.81341	22.85650
$le_{1,k}/(lh(k+1))$	1.61005	1.27228	1.00894	0.76954	0.61590
$le_{2,k}/(lh(k+1))$	1.66773	1.43672	1.51636	1.70335	4.57129
$dle_{1,k+1}/lh$	0.93451	0.48226	0.05134	0.00134	-
$dle_{2,k+1}/lh$	1.20571	1.67565	2.26432	16.0431	-

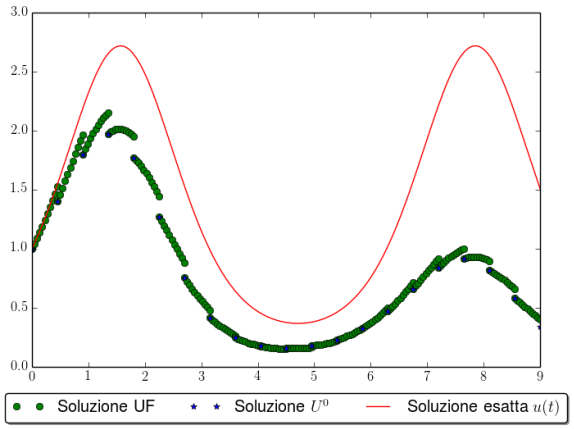
Si noti che la teoria prevede la convergenza per $h \to 0$, pertanto con una sola esecuzione dell'algoritmo è difficile capire quale sia il comportamento asintotico. Essendo inoltre l'ordine di convergenza $m(k+1)$, sono sufficienti poche iterate alla convergenza se il dato iniziale è *buono*.

7.2 Test su problema non autonomo

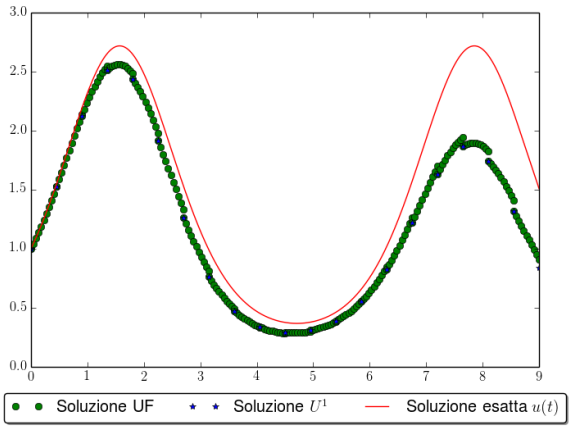
Nell'esempio precedente è stato considerato il problema modello, in particolare si tratta di un problema di Cauchy autonomo.
Mostriamo qui un esempio di esecuzione per un problema non autonomo, che presenta inoltre molte variazioni, consideriamo quindi

$$\begin{cases} u' = \lambda \cos(t) u \\ u_0 = 1 \end{cases}$$

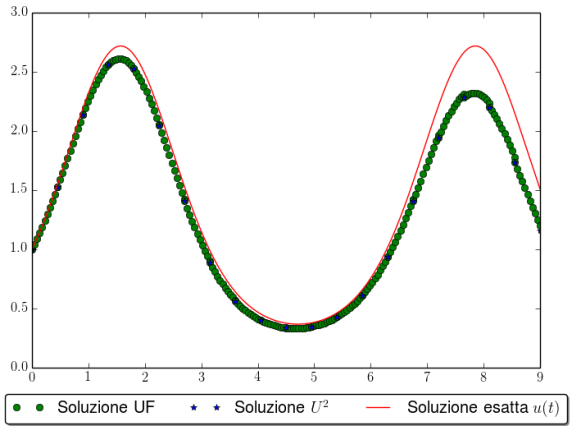
definito su $I = [0, 9]$. La soluzione esatta del problema è data da $u = \exp(\lambda \sin(t))$, e non presenta un andamento monotono su tutto l'intervallo sul quale è stato definito



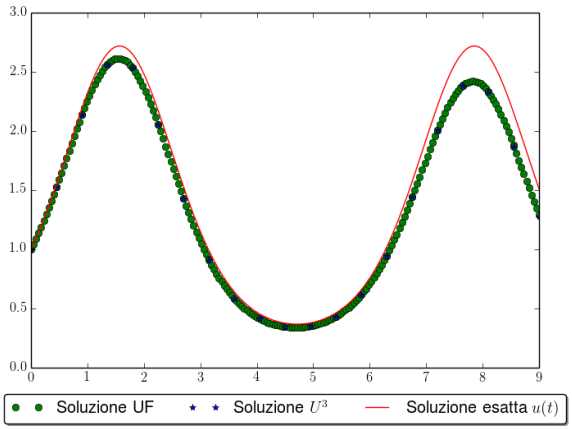
(a) Iterata $k = 0$



(b) Iterata $k = 1$



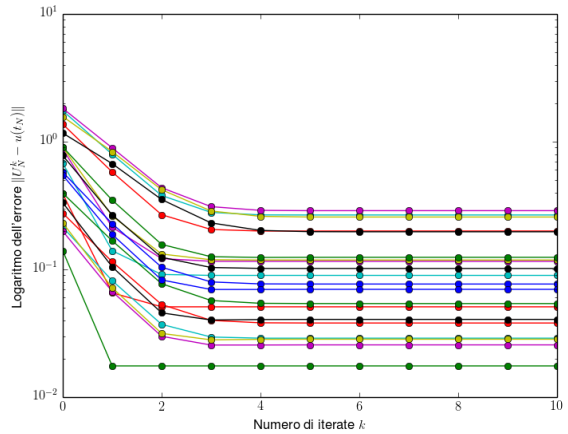
(c) Iterata $k = 2$



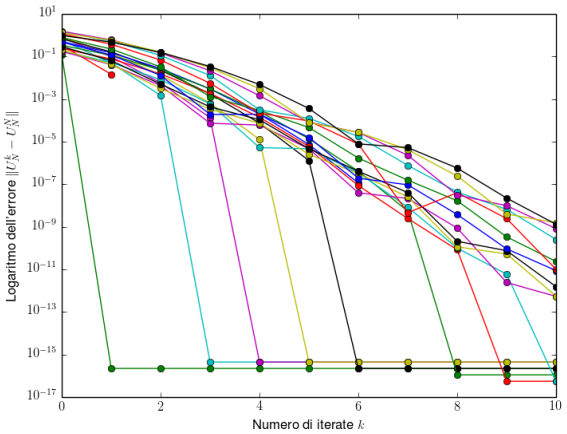
(d) Iterata $k = 3$

Figura 6: Metodo EE-EE

Come propagatori sono stati usati nuovamente eulero esplicito sia per G, sia per F, con $p = 20$ processori.



(a) $\|U_n^k - u(t_n)\|$



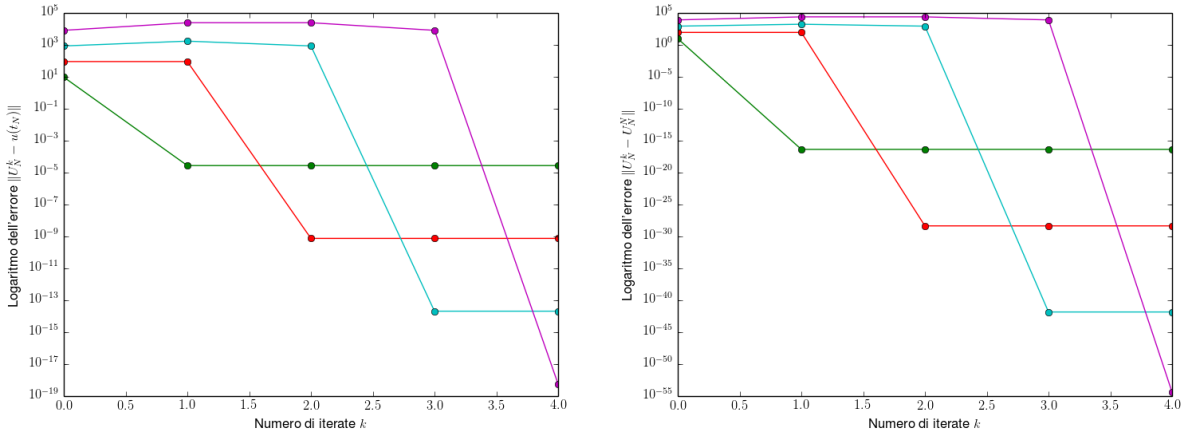
(b) $\|U_n^k - U_n^N\|$

Figura 7: Andamento errore in scala logaritmica

L'algoritmo Parareal sembra quindi comportarsi bene, in quanto si ottiene comunque la convergenza. Inoltre si può notare ancora meglio che nell'esempio precedente, che l'algoritmo Parareal converge alla soluzione esatta ben prima delle 20 iterate.

7.3 Problemi di stabilità

Si consideri nuovamente il problema modello, con $p = 5$ processori e si si aumenti la lunghezza dell'intervallo, fino ad arrivare a $I = [0, 21]$. Si sta quindi assegnando ad ogni processore un intervallo molto ampio e che il propagatore G tratti un passo h più grosso, si ottiene il seguente comportamento.



(a) $\|U_n^k - u(t_n)\|$ (b) $\|U_n^k - U_n^N\|$

Figura 8: Andamento errore in scala logaritmica

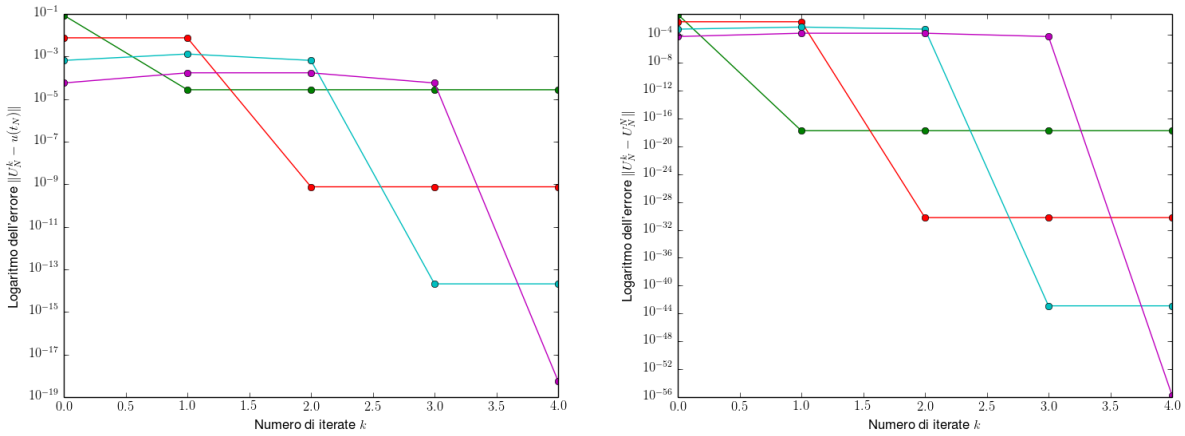
L'errore aumenta ad ogni iterata poiché eulero esplicito risulta, in questo caso, instabile sulla griglia di G, ma non quella di F, infatti si ha

$$\begin{cases} S_G(\Delta_1 T\lambda) = |1 + \Delta_1 T\lambda| = \frac{19}{2} > 1 \\ S_F(\Delta_2 T\lambda) = |1 + \Delta_2 T\lambda| = \frac{1}{20} < 1 \end{cases}$$

Poiché il metodo Parareal è esatto, nel senso che su ogni intervallo, quando il numero di iterate k raggiunge n, l'approssimazione del metodo Parareal eguaglia quella ottenuta solamente con il metodo fine su tutto l'intervallo in sequenziale, si ha comunque una convergenza, anche se è la peggiore possibile, infatti in questo caso il metodo Parareal converge alla soluzione esatta solo quando raggiunge il numero massimo di iterate, ovvero per $k = N$.

7.4 Uso di metodi impliciti

Nei test precedenti sono state usati solamente metodi espliciti. In realtà per il propagatore G risulta più conveniente usare un metodo implicito, come ad esempio eulero implicito, che non ha problemi di stabilità, e in questo modo permette di propagare l'informazione su intervalli molto ampi. Consideriamo quindi nuovamente il problema modello sull'intervallo $I = [0, 21]$ con $n = 4$ processori e $\lambda = -2$ e eulero implicito come propagatore G e eulero esplicito per il propagatore F. Si ha quindi che ambedue i propagatori sono stabili, infatti i risultati numerici di convergenza sono migliori in confronto al caso precedente.



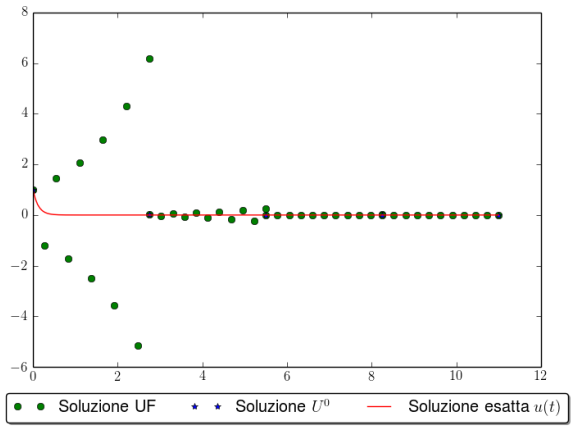
(a) $\|U_n^k - u(t_n)\|$ (b) $\|U_n^k - U_n^N\|$

Figura 9: Andamento errore in scala logaritmica

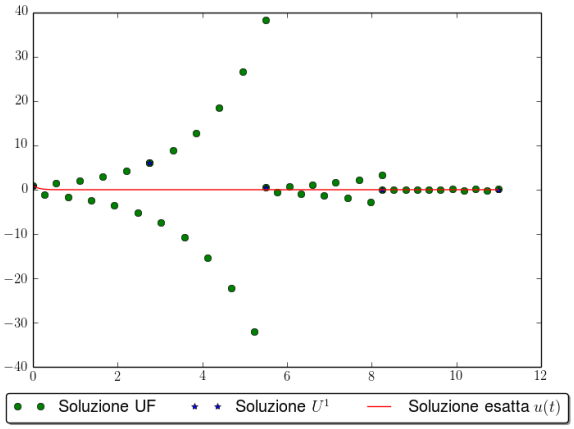
Si noti come l'errore abbia un valore finale (quando si ha $k = N$) uguale al caso precedente, ma i valori assoluti dell'errore su ogni intervallo si comportano decisamente meglio.

7.5 Problema molto stiff

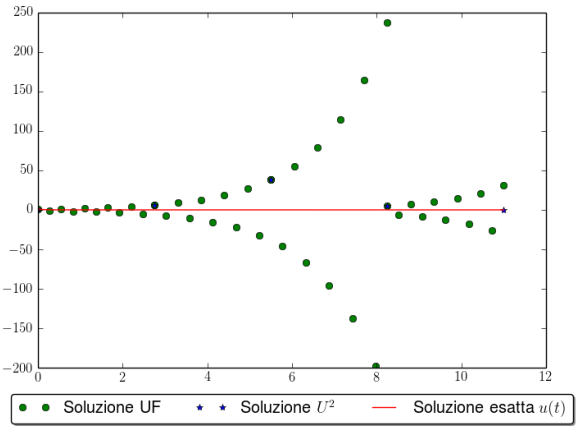
La scelta del propagatore G implicito potrebbe non essere sufficiente a garantire una buona convergenza del metodo. Ad esempio se il propagatore F ha problemi di stabilità, poiché il metodo è esatto si avrà per $k = N$ una pessima approssimazione della soluzione.
In questo esempio abbiamo considerato il problema modello con $\lambda = -8$ sull'intervallo $[0, 11]$ con $p = 4$ processori e la coppia di metodi Eulero Implicito ed Eulero Esplicito.
In questo caso il propagatore G non soffre di problemi di stabilità, mentre il propagatore F si, come si può notare dai grafici ad ogni iterata la soluzione approssimata diverge, poiché l'instabilità di F peggiora i dati iniziali per le iterate successive.



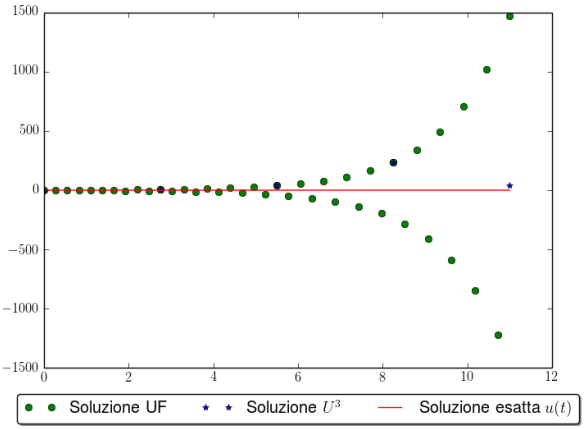
(a) Iterata $k = 0$



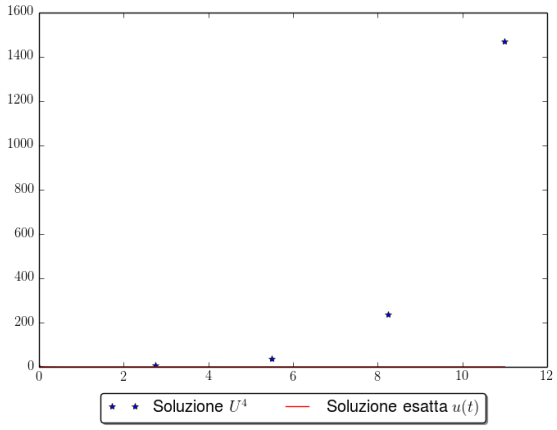
(b) Iterata $k = 1$



(c) Iterata $k = 2$



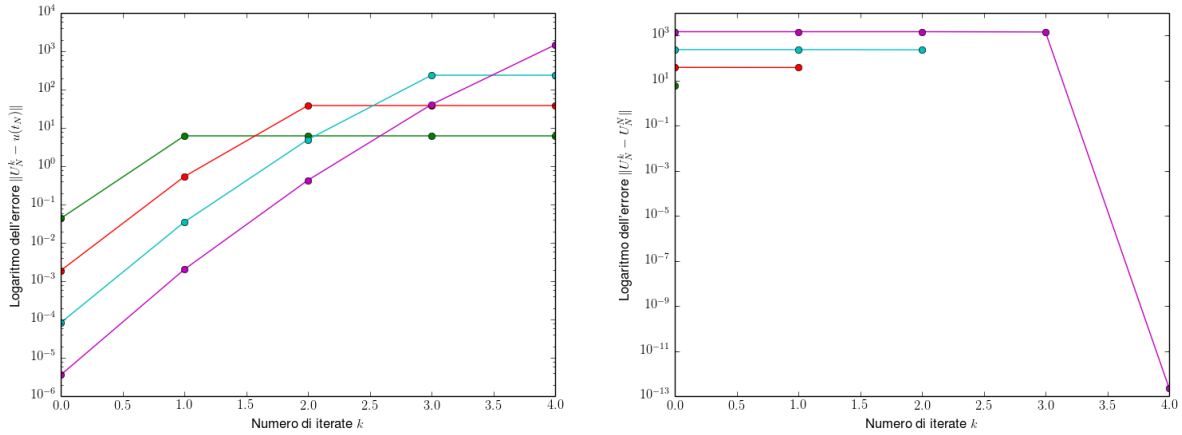
(d) Iterata $k = 3$



(e) Iterata $k = 4$

Figura 10: Metodo EI-EE su problema molto stiff

Anche dal grafico degli errori si può notare come il metodo Parareal diverga dalla soluzione esatta, ma converge, quando k raggiunge N, alla soluzione ottenuta con il propagatore F su tutto l'intervallo.



(a) $\|U_n^k - u(t_n)\|$ (b) $\|U_n^k - U_n^N\|$

Figura 11: Andamento errore in scala logaritmica

7.6 Equazioni di ordine maggiore

L'algoritmo Parareal è stato concepito per equazioni di primo ordine, pertanto dato un problema di Cauchy di ordine maggiore di 1, è necessario riscriverlo come un sistema di ordine 1 per potervi applicare l'algoritmo. Si consideri ad esempio il problema di Cauchy

$$\begin{cases} u'' + \omega^2 u = 0 \\ u_0 = 0 \\ u_0' = \omega \end{cases}$$

la cui soluzione esatta è

$$u(t) = \sin(\lambda t)$$

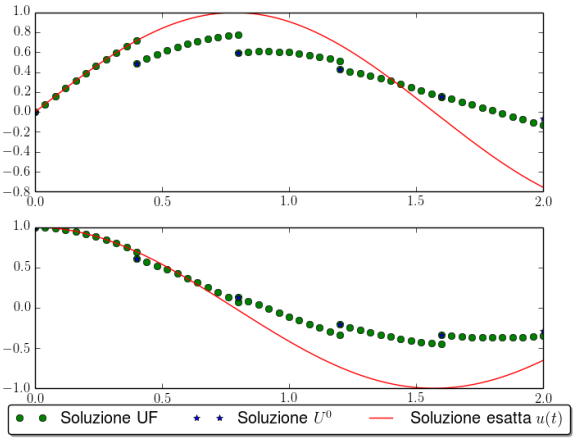
Esso è riscrivibile come

$$\begin{cases} u' = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \omega y \\ u_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \omega \end{cases}$$

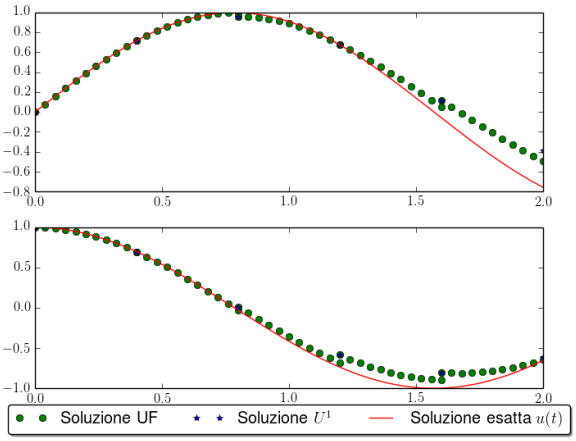
con soluzione

$$u = \begin{pmatrix} \sin(\lambda t) \\ \cos(\lambda t) \end{pmatrix}$$

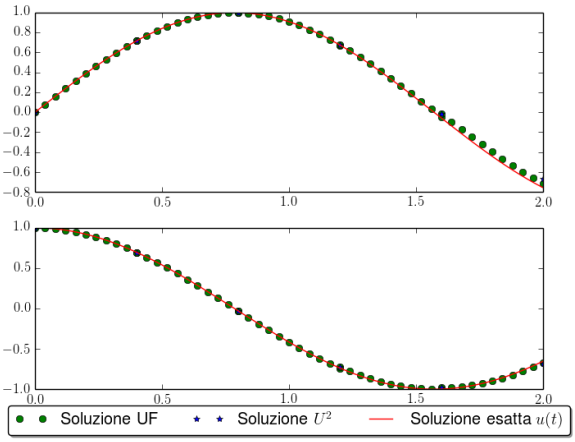
in realtà a noi interesserebbe solo la prima componente, ma in linea del tutto generale supponiamo sempre di considerare il sistema come problema iniziale, e quindi teniamo in considerazione anche la seconda componente. Ponendo ad esempio $\lambda = 2$, con $p = 5$ processori sull'intervallo $I = [0, 2]$, si ottiene, accoppiando i metodi eulero implicito con il metodo Runge-Kutta *Ottima* come propagatore fine, i seguenti risultati



(a) Iterata $k = 0$

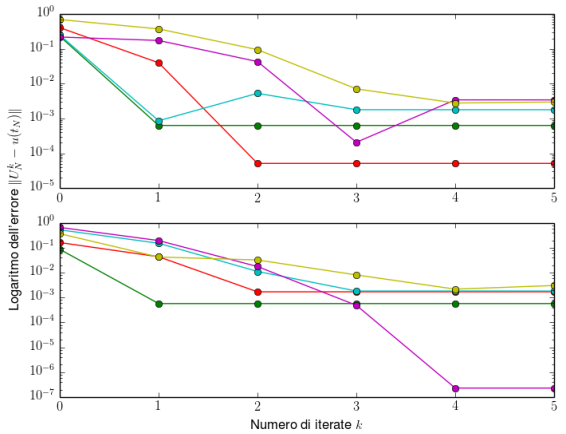


(b) Iterata $k = 1$

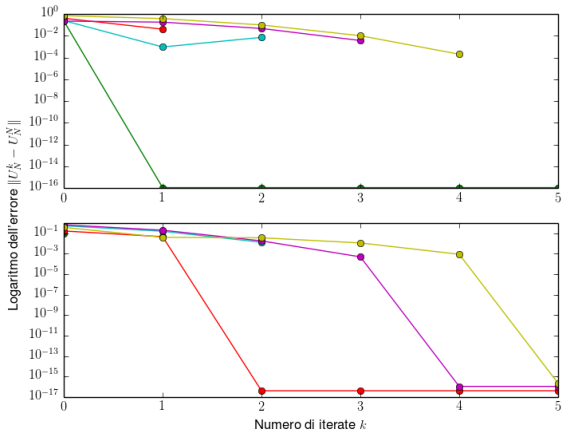


(c) Iterata $k = 2$

Figura 12: Metodo EE-OTTIMA



(a) $\|U_n^k - u(t_n)\|$



(b) $\|U_n^k - U_n^N\|$

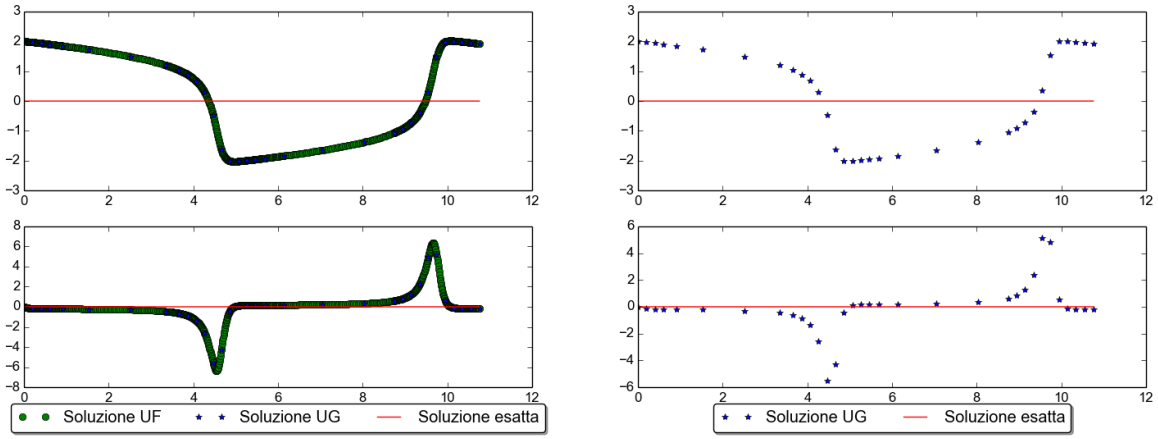
Figura 13: Andamento errore in scala logaritmica

Tabella 2: Errore con $p = 5$ processori

k	0	1	2	3	4	5
$e_{1,k}$	0.69001	0.37212	0.09433	0.00706	0.00277	0.00297
$e_{2,k}$	0.69298	0.37509	0.09730	0.01003	0.00019	0.00000
$le_{1,k}/lh$	0.40493	1.07884	2.57657	5.40457	6.42632	6.35040
$le_{2,k}/lh$	0.40024	1.07017	2.54273	5.02161	9.29721	∞
$le_{1,k}/(lh(k+1))$	0.40493	0.53942	0.85885	1.35114	1.28526	1.05840
$le_{2,k}/(lh(k+1))$	0.40024	0.53508	0.84757	1.25540	1.85944	∞
$dle_{1,k+1}/lh$	0.67390	1.49773	2.82800	1.02175	0.07591	-
$dle_{2,k+1}/lh$	0.66991	1.47257	2.47888	4.27560	∞	-

7.7 Uso di un passo adattivo

Si è voluto sperimentare con la possibilità di usare un propagatore G con passo adattivo. Per questa occasione è stato analizzato l'oscillatore di Van der Pool, un sistema non lineare, con un parametro di smorzamento non lineare. Per approssimare bene tale problem è necessario usare un passo molto piccolo, oppure un passo adattivo, in quanto sono presenti sia zone con variazioni molto alte, sia con variazioni piccole.
Si è usato G con passo adattivo solamente al primo passo per fissare le ampiezze degli intervalli, successivamente sono stati usati i propagatori F e G non adattivi sulla griglia definita prima.
Si noti che al contrario del caso seriale, nel quale si definisce un punto di partenza ed uno di arrivo, e, tramite un passo minimo e massimo si può controllare parzialmente il numero di intervalli che si creeranno, nel caso parallelo è essenziale avere il controllo completo sul numero di intervalli che si verranno a creare, pertanto si definisce il punto di partenza t_0,il numero di intervalli desiderati, e tramite il passo minimo e massimo si ha un controllo parziale sul punto di partenza.
Come propagatore G è stato usato HAMMER, di ordine 3, con passo h, $\frac{h}{2}$ per definire la griglia non uniforme, e come propagatore F è stato usato RK4, con 10 sottointervalli.
Ovviamente anche nel caso del propagatore F si sarebbe potuto usare un propagatore con passo adattivo, e con le iterate successive dei propagatori che usassero la griglia predefinita, ma questa possibilità non è stata ulteriormente esplorata.
Poiché il dato iniziale varia, non è, come nel caso del propagatore G, consigliato imporre la griglia su cui risolvere il problema. Poiché non è necessario comunicare il proprio partizionamento agli altri processori, non c'è nessun problema ad usare una griglia differente ad ogni iterata.



(a) Iterata $k = 0$

(b) Iterata $k = 1$

Figura 14: Metodo HAMMER adattivo-RK4 su problema VDP

7.8 Scaled speedup a intervallo fissato e ordine di convergenza

Siamo interessati a studiare lo scaled scaling del metodo parareal. Per analizzare il suo comportamento, fissiamo un problema e aumentiamo il numero di processori, calcolando alla fine l'errore della soluzione discreta e il tempo di esecuzione.
I seguenti comportamenti rendono difficoltoso capire la bontà del metodo, in quanto

1. Per $h \to 0$ possono cambiare le condizioni di stabilità.

2. I metodi impliciti sono stati implementati con il metodo delle tangenti di Newton con un criterio di arresto basato sulla tolleranza. Per $h \to 0$ tale tolleranza, sotto ipotesi di regolarità, verrà raggiunta sempre più velocemente, quindi per un alto numero di processori si avrà che i metodi impliciti verranno eseguiti più velocemente.

3. Anche il metodo parareal fa uso di un criterio di arresto basato sulla tolleranza per decidere se è necessario effettuare ancora una iterata o meno, per $h \to 0$ il metodo è più preciso, pertanto una tolleranza fissata viene raggiunta più velocemente.

Si noti che in linea del tutto generale non è possibile fissare a priori un numero massimo di iterate, sia per i metodi Newton, sia per i metodi parareal, in quanto è difficile stabilire un numero di iterate tale che il metodo abbia raggiunto una buona accuratezza. Essendo poi essenziale per il metodo parareal eseguire meno iterate possibili per ogni numero di processori prefissato, il compito è particolarmente arduo.
È possibile arginare il problema del cambiamento delle condizioni di stabilità facendo in modo che per il numero minimo di processori considerato il metodo parareal risulti stabile, per quanto concerne la verifica dello speedup, si è scelto di riportare nelle rispettive tabelle il tempo necessario per risolvere lo stesso problema con un solo processore.

7.8.1 Test 1

Consideriamo il problema modello

$$\begin{cases} u' = \lambda u, \quad \mathrm{Re}(\lambda) < 0 \\ u(0) = 1 \end{cases}$$

di parametro $\lambda = -2$, sull'intervallo $I = \left[0, \frac{7}{9}\right]$ con $p = 2, 4, 8, 16, 32, 64, 128$ processori e $\frac{\Delta_2 T}{\Delta_1 T} = 100$. In questo caso si è fissato a 3 il numero di iterate, indipendentemente dal numero di processori, questo vuol dire che l'accuratezza della soluzione non è fissata.

Tabella 3: Errore metodo con $p = 2, 4, 8, 16, 32, 64, 128$ processori

k	0	1	2	3	p
$le_{1,k}/(lh(k+1))$	1.929 22	1.517 65	2.350 97		2
$le_{2,k}/(lh(k+1))$	1.937 64	1.529 69	∞		
$le_{1,k}/(lh(k+1))$	1.610 05	1.371 94	1.342 43	1.118 37	4
$le_{2,k}/(lh(k+1))$	1.615 52	1.389 91	1.470 72	1.658 09	
$le_{1,k}/(lh(k+1))$	1.453 94	1.261 74	1.110 13	0.862 40	8
$le_{2,k}/(lh(k+1))$	1.458 02	1.287 82	1.308 77	1.368 95	
$le_{1,k}/(lh(k+1))$	1.359 24	1.186 96	0.954 42	0.722 667	16
$le_{2,k}/(lh(k+1))$	1.362 47	1.225 79	1.231 74	1.264 390	
$le_{1,k}/(lh(k+1))$	1.295 96	1.127 44	0.844 17	0.634 617	32
$le_{2,k}/(lh(k+1))$	1.298 63	1.185 07	1.186 21	1.208 330	
$le_{1,k}/(lh(k+1))$	1.251 00	1.073 78	0.765 14	0.574 18	64
$le_{2,k}/(lh(k+1))$	1.253 26	1.156 52	1.156 01	1.172 950	
$le_{1,k}/(lh(k+1))$	1.217 57	1.022 94	0.706 77	0.530 152	128
$le_{2,k}/(lh(k+1))$	1.219 53	1.135 49	1.134 42	1.148 350	

Si noti che nonostante $h \to 0$, utilizzando 128 processori si ha $h = \frac{7}{9 \cdot 128} \simeq 0.006076$, quindi non un valore particolarmente piccolo.
La seguente tabella riporta l'andamento dei tempi

Tabella 4: Tempi di esecuzione con $p = 2, 4, 8, 16, 32$ processori

p	tempo minimo	tempo massimo	tempo medio	tempo seriale
2	0.017	0.021	0.019	0.015
4	0.020	0.030	0.025	0.029
8	0.022	0.030	0.027	0.057
16	0.030	0.041	0.033	0.113
32	0.041	0.053	0.043	0.227

Si tenga conto del fatto che aumenta anche l'ordine di accuratezza della soluzione, come si può notare osservando la tabella degli errori per 2 processori e quella per 32:

Tabella 5: Errore con $p = 2$ processori

k	0	1	2
$e_{1,k}$	0.161 68	0.056 88	0.001 27
$e_{2,k}$	0.160 41	0.055 60	0.000 00
$le_{1,k}/lh$	1.929 22	3.035 29	7.052 90
$le_{2,k}/lh$	1.937 64	3.059 38	∞
$le_{1,k}/(lh(k+1))$	1.929 22	1.517 65	2.350 97
$le_{2,k}/(lh(k+1))$	1.937 64	1.529 69	∞
$dle_{1,k+1}/lh$	1.106 07	4.017 61	-
$dle_{2,k+1}/lh$	1.121 74	∞	-

Tabella 6: Errore con $p = 32$ processori

k	0	1	2	3
$e_{1,k}$	0.00808	0.00022	8.1×10^{-05}	7.9×10^{-05}
$e_{2,k}$	0.00800	0.00014	1.8×10^{-06}	1.5×10^{-08}
$le_{1,k}/lh$	1.29596	2.25488	2.53252	2.53847
$le_{2,k}/lh$	1.29863	2.37013	3.55864	4.83331
$le_{1,k}/(lh(k+1))$	1.29596	1.12744	0.84417	0.63461
$le_{2,k}/(lh(k+1))$	1.29863	1.18507	1.18621	1.20833
$dle_{1,k+1}/lh$	0.95892	0.27763	0.00594	-
$dle_{2,k+1}/lh$	1.0715	1.18851	1.27466	-

Per quanto concerne lo scaling notiamo che per $p < 4$ si ha che T_p è maggiore di T_1, questo deriva anche dal fatto che per raggiungere la convergenza è necessario effettuare $O(p)$ iterate, mentre per $p > 4$ si ha che il tempo continua ad aumentare in maniera lineare, mentre T_p aumenta più lentamente. La seguente tabella mostra la relazione tra il numero di processori e le iterate necessarie per raggiungere la convergenza

Tabella 7: Relazione p-k.

p	2	4	8	16	32	64	128
k	2	3	3	3	3	3	3

7.8.2 Test 2

In questo caso studiamo il problema periodico

$$\begin{cases} u' = \begin{pmatrix} 0 & \omega_1 \\ -\omega_1 & 0 \end{pmatrix} u - \begin{pmatrix} 0 \\ (\omega_2^2 - \omega_1^2)\frac{\sin(\omega_2 t)}{\omega_1} \end{pmatrix} \\ u_0 = \begin{pmatrix} 0 \\ 1 + \frac{\omega_2}{\omega_1} \end{pmatrix} \end{cases}$$

definito su $I = \left[0, \frac{70}{9}\right]$, con soluzione esatta

$$u(t) = \begin{pmatrix} \sin(\omega_1 t) + \sin(\omega_2 t) \\ \cos(\omega_2 t) + \frac{\omega_2}{\omega_1}\cos(\omega_2 t) \end{pmatrix}$$

tale problema deriva dalla equazione di secondo ordine

$$u'' + \omega_1 u + (\omega_2^2 - \omega_1^2)\sin(\omega_2 t) = 0$$

con relativi dati iniziali.
Come propagatori sono stati usati i metodi Runge Kutta *Butcher-Lobatto* e *Merson*, e si è fatto variare il numero di processori per $p = 2, 4, 8, 16, 32, 64, 128$, si è fissato $\frac{\Delta_2 T}{\Delta_1 T} = 100$ e i tempi sono stati registrati solamente per $p \leq 32$, causa la limitazioni hardware date dal cluster.
Al contrario del caso precedente è stato impostato un parametro di tolleranza tra le iterate del metodo, del valore proporzionale all'ordine di convergenza del propagatore F. La seguente tabella mostra l'andamento dell'errore, e il numero massimo di iterate effetuate

Tabella 8: Errore metodo con $p = 2, 4, 8, 16, 32, 64, 128$ processori

k	0	1	2	3	4	n
$le_{1,k}/(lh(k+1))$	2.313 23	0.791 06	−4.193 09			2
$le_{2,k}/(lh(k+1))$	2.313 23	0.791 06	−8.406 67			
$le_{1,k}/(lh(k+1))$	3.117 42	−1.4013	−1.756 29	−2.868 60	−5.964 61	4
$le_{2,k}/(lh(k+1))$	3.117 42	−1.4013	−1.756 29	−2.868 60	−10.840 60	
$le_{1,k}/(lh(k+1))$	33.753 20	106.808 00	133.218 00	155.369 00	160.788 00	8
$le_{2,k}/(lh(k+1))$	33.753 20	106.808 00	133.218 00	155.314 00	181.039 00	
$le_{1,k}/(lh(k+1))$	2.545 39	6.086 16	7.690 98	8.744 41	7.031 16	16
$le_{2,k}/(lh(k+1))$	2.545 39	6.086 16	7.691 06	9.478 18	9.355 39	
$le_{1,k}/(lh(k+1))$	1.795 01	4.254 89	5.404 64	4.971 54		32
$le_{2,k}/(lh(k+1))$	1.795 01	4.254 89	5.405 98	5.659 08		
$le_{1,k}/(lh(k+1))$	1.539 05	3.663 68	4.611 88	3.678 71		64
$le_{2,k}/(lh(k+1))$	1.539 05	3.663 68	4.640 59	4.014 78		
$le_{1,k}/(lh(k+1))$	1.409 28	3.372 36	3.897 49	2.902 54		128
$le_{2,k}/(lh(k+1))$	1.409 28	3.372 36	4.028 24	3.155 44		

Si noti che nonostante $h \to 0$, utilizzando 128 processori si ha $h = \frac{70}{9 \cdot 128} \simeq 0.06076$, quindi non un valore particolarmente piccolo.
La seguente tabella riporta l'andamento dei tempi, per i valori $p \leq 32$, essendo il numero di processori realmente disponibili.

Tabella 9: Tempi d'esecuzione con $p = 2, 4, 8, 16, 32$ processori

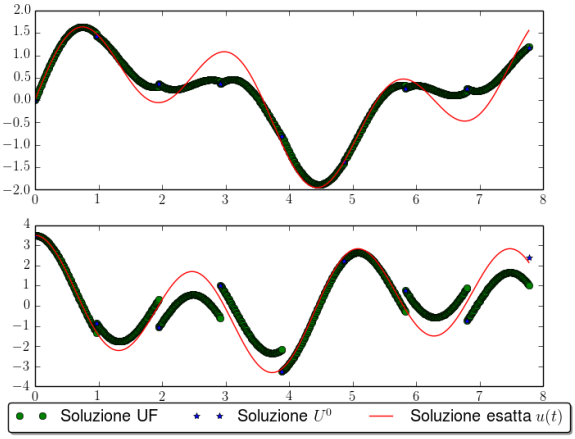
p	tempo minimo	tempo massimo	tempo medio	tempo seriale
2	0.108	0.111	0.109	0.093
4	0.210	0.219	0.214	0.189
8	0.235	0.242	0.239	0.372
16	0.288	0.295	0.290	0.747
32	0.299	0.310	0.302	1.496

Per le stesse motivazioni del caso precedente notiamo che per p piccoli si ha che $T_1 \leq T_p$, mentre per p grande aumenta lo speedup. La seguente tabella mostra la relazione tra il numero di processori e le iterate necessarie per raggiungere la convergenza

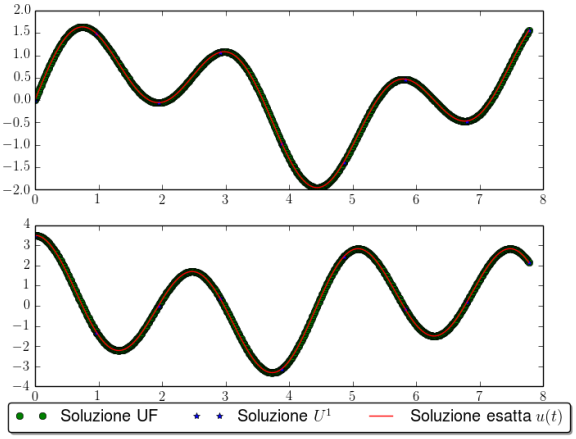
Tabella 10: Relazione p-k.

p	2	4	8	16	32	64	128
k	2	4	4	4	3	3	3

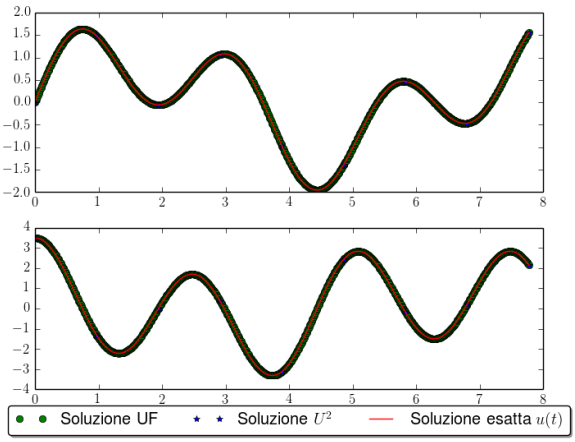
A titolo dimostrativo mostriamo i grafici dell'andamento dell'algoritmo per $p = 8$



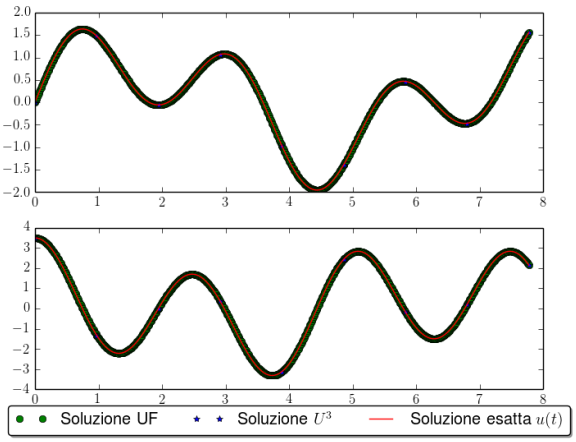
(a) $k = 0$



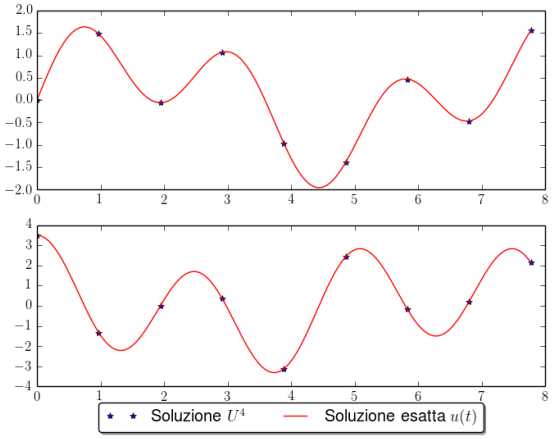
(b) $k = 1$



(c) $k = 2$



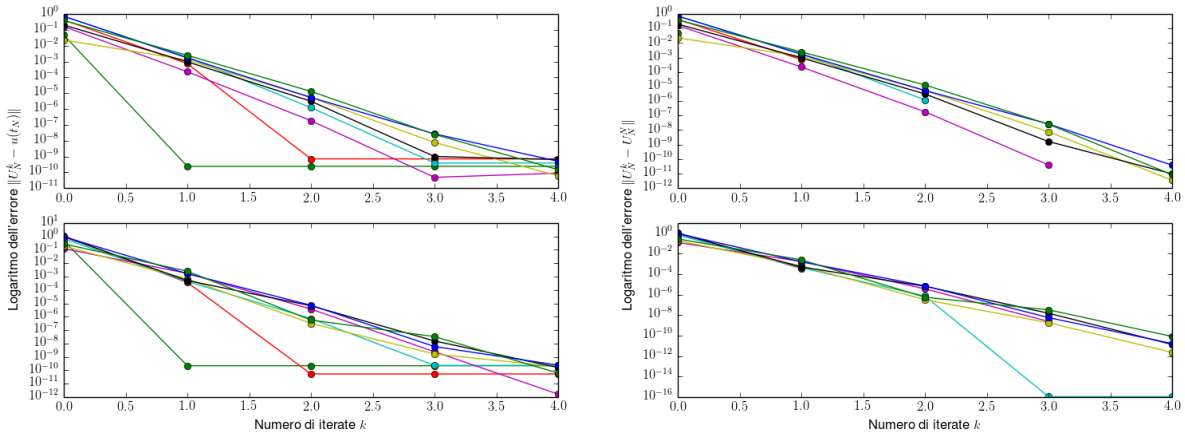
(d) $k = 3$



(e) Soluzione finale

Figura 15: Metodo *Butcher-Lobatto - Merson*

E degli errori in scala logaritmica



(a) $p = 8$, $\|U_N^k - u(t_N)\|$ (b) $p = 8$, $\|U_N^k - U_N^N\|$

Figura 16: Andamento degli errori in scala logaritmica

7.8.3 Test 3

In questa sezione riprendiamo lo stesso caso test precedente, ma imponiamo $\frac{\Delta_2 T}{\Delta_1 T} = 1000$. Otteniamo i seguenti risultati relativi ai tempi di esecuzione

Tabella 11: Tempi d'esecuzione con $p = 2, 4, 8, 16, 32$ processori

p	tempo minimo	tempo massimo	tempo medio	tempo seriale
2	0.108	0.111	0.109	0.093
4	0.210	0.219	0.214	0.189
8	0.235	0.242	0.239	0.372
16	0.288	0.295	0.290	0.747
32	0.299	0.310	0.302	1.496

La seguente tabella mostra la relazione tra il numero di processori e le iterate necessarie per raggiungere la convergenza

Tabella 12: Relazione p-k.

p	2	4	8	16	32	64	128
k	2	4	4	4	3	3	3

7.8.4 Nota finale

Nei test effettuati precedentemente, escluso Test 3 descritto in (7.8.3), si è posto $\frac{\Delta_2 T}{\Delta_1 T} = 100$. Se si eseguono gli stessi test, ponendo $\frac{\Delta_2 T}{\Delta_1 T} = 10$, allora si ha che vale sempre $T_1 < T_p$, indipendentemente da p e k.
Questo accade perché se $\frac{\Delta_2 T}{\Delta_1 T}$ è *piccolo*, allora può succedere che il tempo di esecuzione di k volte del propagatore G abbia un costo maggiore che eseguire una volta il propagatore F in modo seriale su tutto l'intervallo.
Se invece $\frac{\Delta_2 T}{\Delta_1 T}$ è molto grande, allora eseguire in seriale il propagatore F ha in generale un costo maggiore che eseguire k volte il propagatore G.
Questa proprietà si può evincere anche da (7), in quando se si ha che $K = O(1)$ e $KC_G p$ è trascurabile

in confronto a $C_G n$, allora vale, per p sufficientemente grande

$$S = \frac{T_1}{T_p} \simeq \frac{C_F n}{K C_F \frac{n}{p}} = \frac{p}{K} > 1$$

Il fatto che convenga avere un propagatore F più costoso del propagatore G non è una proprietà inaspettata, in quant è il propagatore F ad essere eseguito in parallelo, ed è quindi la parte dell'algoritmo della quale si possono ridurre i tempi di esecuzioni. Se la maggior parte del tempo viene usata per applicare il propagatore G, e tale parte viene eseguita in seriale, allora il guadagno in termini di prestazioni non potrà essere sostanziale aumentando il numero di processori.
A maggior ragione, per sottolineare questa proprietà, se si confrontano i tempi di esecuzione riportati nella sezioni (7.8.2) e (7.8.3), si noterà che nel seconda caso si ha uno speedup maggiore.

7.9 Scaled speedup con intervallo variabile

Nei casi precedenti abbiamo osservato come si comporta l'algoritmo all'aumentare dei processori con $h \to 0$, supponendo di voler risolvere il problema su un intervallo fissato.
In questo caso vogliamo studiare come si comporta l'algoritmo all'aumentare del numero dei processori tenendo h costante, ovvero aumentando in modo lineare l'ampiezza dell'intervallo.
I seguenti fattori rendono difficoltosa l'analisi dello scaling dei tempi

1. Sul problema modello e altri problemi di Cauchy che per tempi grossi *si stabilizzano*, in generale quindi i problemi dissipativi, i metodi impliciti, realizzati ad esempio tramite Newton, acquistano velocità, in quanto diminuisce la variazione della soluzione e quindi viene raggiunta più velocemente l'accuratezza desiderata.

Per in parte di affrontare tale problema analizziamo l'algoritmo sopratutto su problemi periodici, in questo modo il carico di lavoro risulterà più distribuito. Inoltre, poichè in generale $\Delta_1 T$ e grosso, si usano propagatori impliciti come propagatore G. Non verrà effettuato uno studio sull'ordine di convergenza per h, in quanto il comportamento previsto dalla teoria è asintotico per h piccolo.
Infine, per semplificare l'analisi dello scaling, verrà riportato anche il tempo seriale.

7.9.1 Test 1

Si consideri il problema

$$\begin{cases} u' = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \omega y \\ u_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \omega \end{cases}$$

con soluzione

$$u = \begin{pmatrix} \sin(\omega t) \\ \cos(\omega t) \end{pmatrix}$$

derivante dal problema

$$\begin{cases} u'' + \omega^2 u = 0 \\ u_0 = 0 \\ u_0' = \omega \end{cases}$$

quindi da una equazione periodica, con ω fissato.
Come propagatori sono stati usati i metodi Runge-Kutta, in particolare per F il metodo *Ottima*, di ordine 2, mentre per G il metodo Eulero Implicito. L'algoritmo Parareal è stato eseguito con un criterio di arresto fissato sulla tolleranza (dipendente dall'ordine del metodo F), per $p = 2, 4, 8, 16, 32, 64, 128$ processori e con $\frac{\Delta_2 T}{\Delta_1 T} = 100$ fissato, anche se i tempi sono riportati solamente per $p \leq 32$ processori causa i limiti hardware del cluster.

Tabella 13: Tempi d'esecuzione con $p = 2, 4, 8, 16, 32$ processori

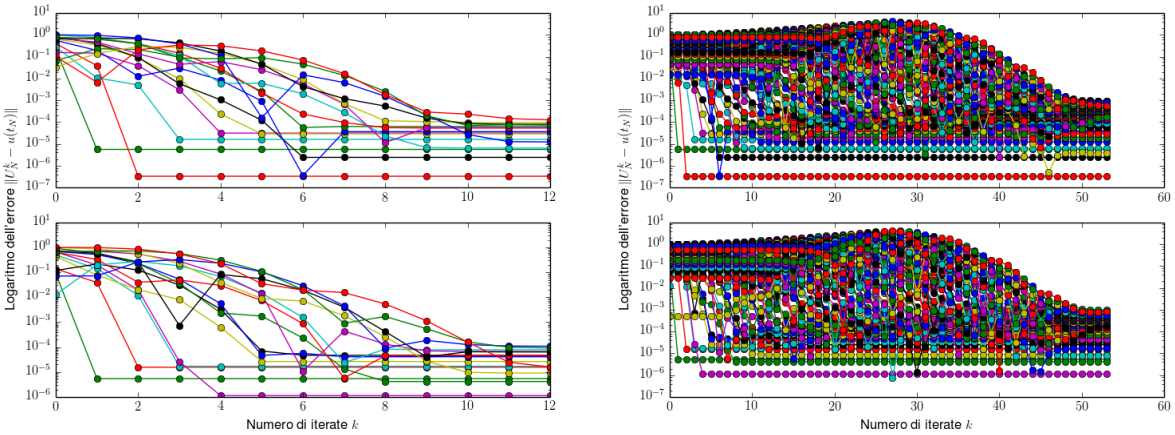
p	tempo minimo	tempo massimo	tempo medio	tempo seriale
2	0.038	0.043	0.040	0.034
4	0.073	0.080	0.078	0.067
8	0.170	0.176	0.170	0.133
16	0.299	0.312	0.306	0.268
32	0.630	0.641	0.632	0.533

La seguente tabella mostra la relazione tra il numero di processori e le iterate necessarie per raggiungere la convergenza

Tabella 14: Relazione p-k.

p	2	4	8	16	32	64	128
k	2	4	8	12	19	31	53

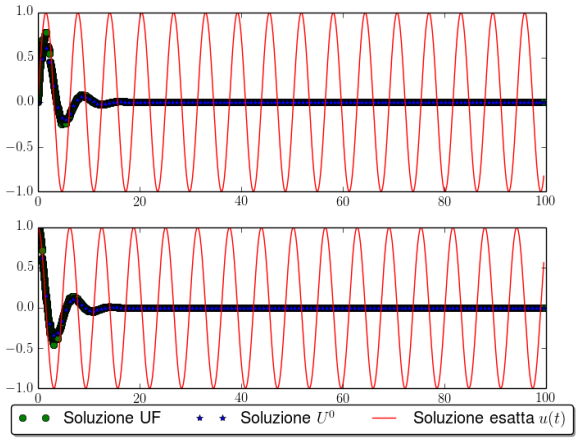
In via del tutto indicativa mostriamo anche l'andamento degli errori



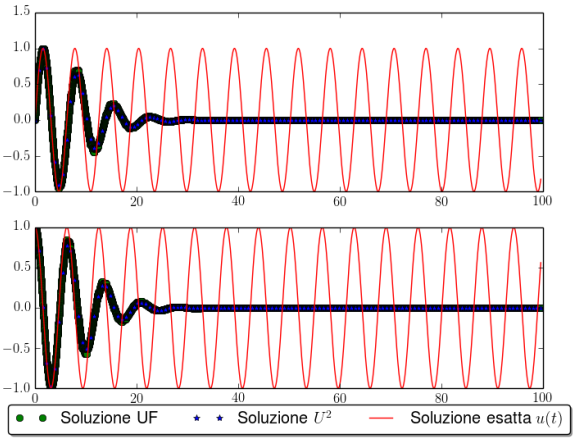
(a) $p = 16$, $\|U_N^k - u(t_N)\|$ (b) $p = 128$, $\|U_N^k - U_N^N\|$

Figura 17: Andamento errore in scala logaritmica

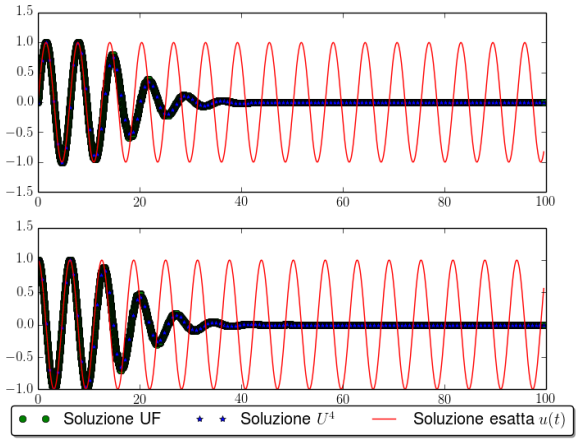
Si può notare che, nonostante la tolleranza sia fissata, indipendentemente dal numero di processori, il numero di iterate necessarie per raggiungere la convergenza, cresce; questo è dato dal fatto che le continue variazioni, tipiche in un problema periodico, come si può vedere nelle seguenti immagini, rallentano il propagarsi delle informazioni.



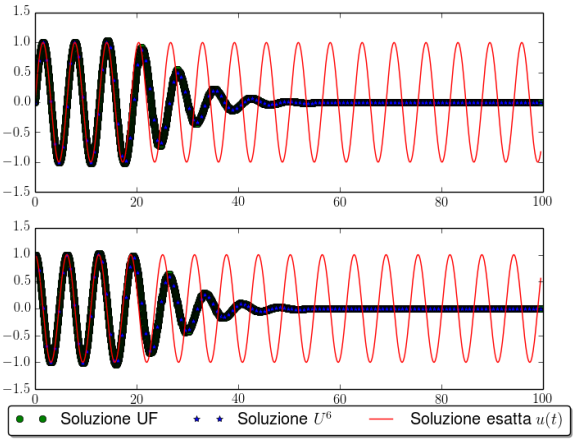
(a) $k = 0$



(b) $k = 2$



(c) $k = 4$



(d) $k = 6$

Figura 18

Il fatto che il numero di iterate aumenti rovina la scalabilità dell'algoritmo, infatti al contrario dei casi precedenti, in questo caso non conviene applicare l'algoritmo Parareal, in quanto i tempi di esecuzione sono maggiori in confronto ai tempi di esecuzione dell'algoritmo classico.

7.9.2 Test 2

Si consideri il problema

$$\begin{cases} u' = \begin{pmatrix} 0 & \omega_1 \\ -\omega_1 & 0 \end{pmatrix} u - \begin{pmatrix} 0 \\ (\omega_2^2 - \omega_1^2)\frac{\sin(\omega_2 t)}{\omega_1} \end{pmatrix} \\ u_0 = \begin{pmatrix} 0 \\ 1 + \frac{\omega_2}{\omega_1} \end{pmatrix} \end{cases}$$

con soluzione

$$u(t) = \begin{pmatrix} \sin(\omega_1 t) + \sin(\omega_2 t) \\ \cos(\omega_2 t) + \frac{\omega_2}{\omega_1}\cos(\omega_2 t) \end{pmatrix}$$

derivante dal problema

$$u'' + \omega_1 u + (\omega_2^2 - \omega_1^2)\sin(\omega_2 t) = 0$$

con $\omega_2 = \frac{5}{2}\omega_1$ fissato, quindi da un problema con due periodi distinti. Nuovamente si è fatto uso di Eulero implicito per il propagatore G e *Ottima* per il propagatore F e si sono considerati $p = 2, 4, 8, 16, 32, 64, 128$ processori e con $\frac{\Delta_2 T}{\Delta_1 T} = 100$ fissato.

In modo analogo al caso precedente il numero di iterate aumenta all'aumentare della dimensione dell'intervallo, e quindi all'aumentare dei processori, rovinando la scalabilità, i tempi sono riportati nella seguente tabella

Tabella 15: Tempi d'esecuzione con $p = 2, 4, 8, 16, 32$ processori

p	tempo minimo	tempo massimo	tempo medio	tempo seriale
2	0.051	0.056	0.053	0.041
4	0.095	0.096	0.095	0.083
8	0.200	0.210	0.206	0.166
16	0.359	0.373	0.365	0.333
32	0.690	0.710	0.702	0.667

La seguente tabella mostra la relazione tra il numero di processori e le iterate necessarie per raggiungere la convergenza

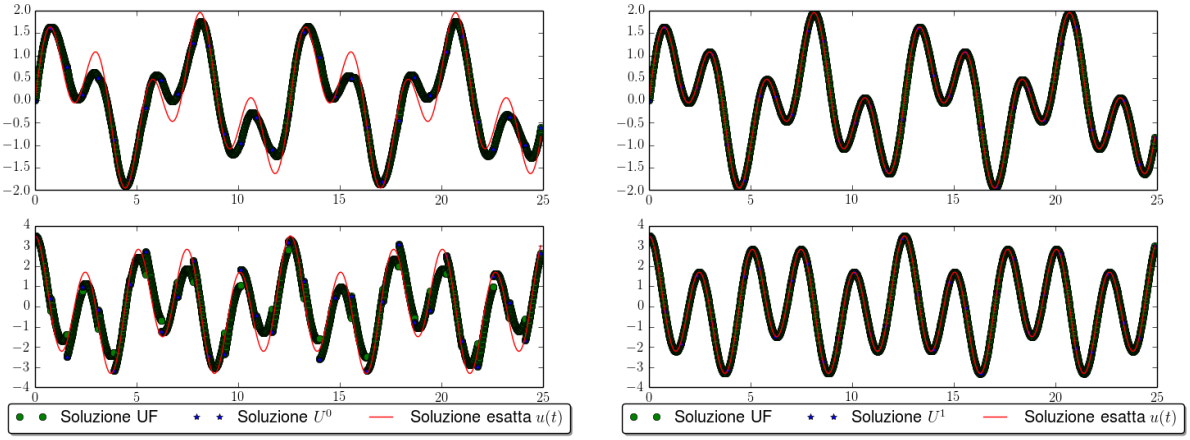
Tabella 16: Relazione p-k.

p	2	4	8	16	32	64	128
k	2	4	8	12	18	29	51

Le conclusioni sono le stesse del caso precedente, l'algoritmo seriale risulta più conveniente per la risoluzione di questo problema, in quanto i tempi di esecuzione sono minori.

7.9.3 Test 3

In questo caso si è considerato nuovamente il problema con 2 periodi del caso precedente, ma si è fatto uso dei propagatori *Butcher-Lobatto* per G e *Merson* per F, che sono di ordine più alto rispetto a quello precedente.
In questo caso la tolleranza desiderata viene raggiunta in poche iterate, una o due, indipendentemente dal numero di processori.
Come si può evincere anche dai grafici, l'algoritmo approssima molto velocemente la soluzione

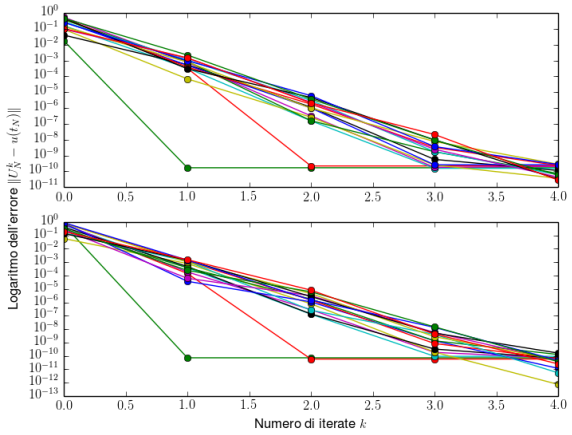


(a) $k = 0$, $p = 32$

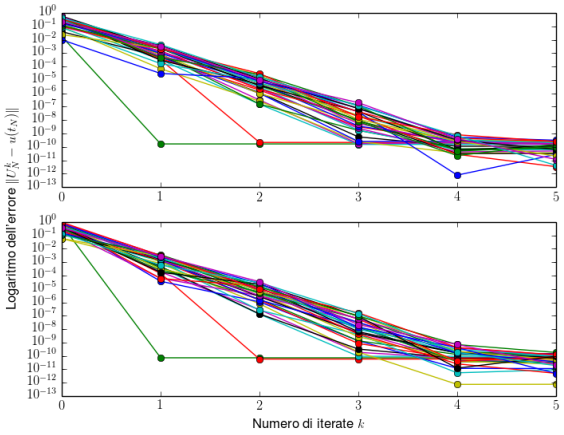
(b) $k = 1$, $p = 32$

Figura 19

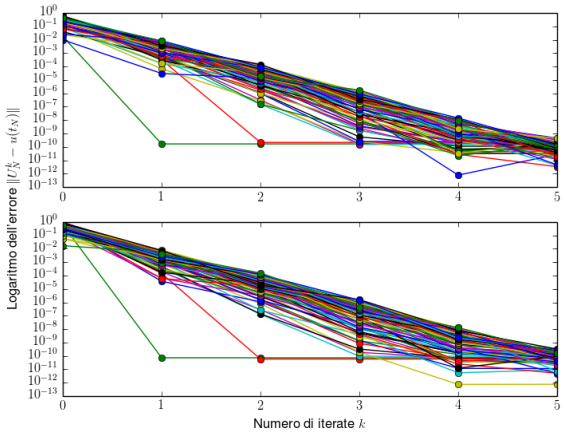
Mostriamo nuovamente l'andamento degli errori per qualche p particolare



(a) $p = 16$, $\|U_N^k - u(t_N)\|$



(b) $p = 32$, $\|U_N^k - u(t_N)\|$



(c) $p = 64$, $\|U_N^k - u(t_N)\|$

Figura 20: Andamento degli errori in scala logaritmica

L'andamento dei tempi è riportato nella seguente tabella

Tabella 17: Tempi d'esecuzione con $p = 2, 4, 8, 16, 32$ processori

p	tempo minimo	tempo massimo	tempo medio	tempo seriale
2	0.103	0.107	0.105	0.093
4	0.210	0.218	0.213	0.188
8	0.232	0.240	0.237	0.375
16	0.281	0.292	0.289	0.745
32	0.466	0.482	0.473	1.492

La seguente tabella mostra la relazione tra il numero di processori e le iterate necessarie per raggiungere la convergenza

Tabella 18: Relazione p-k.

p	2	4	8	16	32	64	128
k	2	4	4	4	5	5	6

Al contrario dei casi precedente, il propagatore *Butcher-Lobatto* riesce sin dalla prima iterata a propagare *bene* l'informazione, questo permette di raggiungere la convergenza in poche iterazioni, e quindi ottenere uno speedup positivo usando più processori.

7.9.4 Nota finale

Anche nei test descritti in questa sezione si è posto $\frac{\Delta_2 T}{\Delta_1 T} = 100$. Nuovamente eseguendo gli stessi test, ponendo $\frac{\Delta_2 T}{\Delta_1 T} = 10$, si ha che vale sempre $T_1 < T_p$, indipendentemente da p e k.
Le motivazioni sono le stesse descritte in (7.8.4)

7.10 Strong scaling

Nei test precedenti non si era fissata la dimensione del problema, questa cresceva linearmente con il numero dei processori poiché si era fissato ad ogni processori una quantità fissata di dati da elaborare.
In questa sezione si vuole invece analizzare come si comporta l'algoritmo Parareal dato un problema con una dimensione fissata al variare del numero di processori.
Fissato quindi un intervallo $[T_0, T_{\max}]$ e n dati si avrà che la quantità $\Delta_1 T = \frac{\Delta_0 T}{p}$ non sarà costante, mentre la quantità $\Delta_2 T = \frac{\Delta_0 T}{n}$ si. Questo significa inoltre che la tolleranza fissata per il test di arresto rimane costante al variare del numero dei processori, poiché non varia la accuratezza finale del propagatore F.
Inoltre si ha che al crescere di p si raffina il partizionamento sul quale lavora il propagatore G, e quindi diminuiscono gli intervalli sui quali deve lavorare il propagatore F.
Si noti che il potrebbero cambiare le condizioni di stabilità per il propagatore G, ma non per il propagatore F.

7.10.1 Test 1

Si consideri il problema

$$\begin{cases} u' = \lambda u, & \mathrm{Re}(\lambda) < 0 \\ u(0) = 1 \end{cases}$$

con parametro λ pari a -5, sull'intervallo $I = [0, \frac{7}{9}]$ e $n = 2^{10}$. Il metodo Parareal è stato eseguito simulando $p = 2, 4, 8, 16, 32, 64, 128$ processori usando, sia per F, sia per G, il metodo di eulero implicito, anche se i tempi sono riportati solo per $p \leq 32$ essendo il numero di processori fisicamente disponibili. Si ha

Tabella 19: Tempi d'esecuzione con $p = 2, 4, 8, 16, 32$ processori

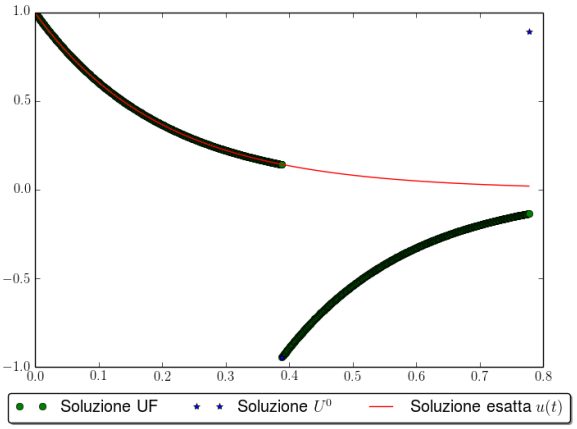
p	tempo minimo	tempo massimo	tempo medio	tempo seriale
2	0.080	0.083	0.081	0.073
4	0.073	0.081	0.078	0.073
8	0.030	0.039	0.033	0.073
16	0.010	0.021	0.017	0.073
32	0.011	0.022	0.019	0.073

Nella seguente tabella sono invece riportate il numero di iterate k per raggiungere la convergenza, in funzione di p.

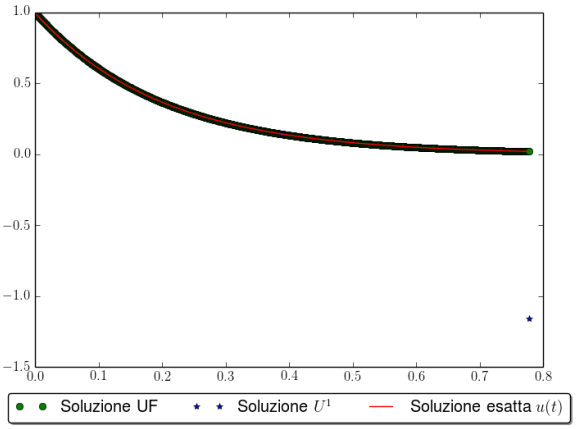
Tabella 20: Relazione p-k.

p	2	4	8	16	32	64	128
k	2	4	3	2	2	2	1

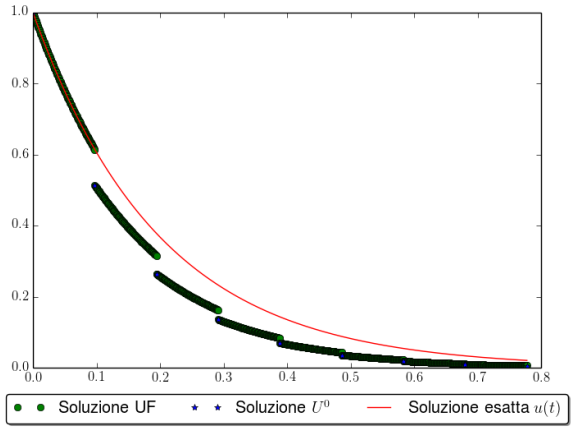
A titolo di esempio vengono riportate i grafici che mostrano l'evoluzione della soluzione approssimata durante le prime iterate.



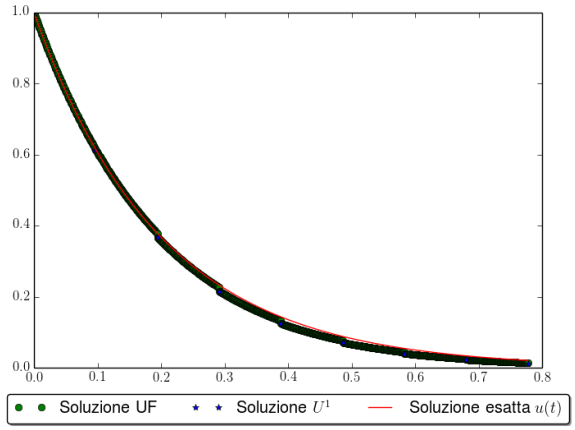
(a) $k = 0$, $p = 2$



(b) $k = 1$, $p = 2$



(c) $k = 0$, $p = 8$



(d) $k = 1$, $p = 8$

Figura 21: Metodo *EE* - *EE*

7.10.2 Test 2

Si consideri il problema

$$\begin{cases} u' = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \omega y \\ u_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \omega \end{cases}$$

con parametro $\omega = -5$, definito sull'intervallo $I = [0, \frac{7}{9}]$ e con $n = 2^{10}$. Come propagatore sono stati scelti eulero implicito per G e eulero esplicito per F. In modo analogo al caso precedente sono stati eseguiti test per $p = 2, 4, 8, 16, 32, 64, 128$ processori mentre i tempi sono riportati solo per $p \leq 32$.

Tabella 21: Tempi d'esecuzione con $p = 2, 4, 8, 16, 32$ processori

p	tempo minimo	tempo massimo	tempo medio	tempo seriale
2	0.115	0.115	0.115	0.105
4	0.115	0.125	0.122	0.105
8	0.122	0.133	0.128	0.105
16	0.090	0.101	0.094	0.105
32	0.090	0.107	0.103	0.105

Tabella 22: Relazione p-k.

p	2	4	8	16	32	64	128
k	2	4	7	6	4	3	3

7.10.3 Test 3

Si consideri il problema

$$\begin{cases} u' = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \omega y \\ u_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \omega \end{cases}$$

con parametro $\omega = -5$, definito sull'intervallo $I = [0, \frac{7}{9}]$ e con $n = 2^{10}$. Come propagatore sono stati scelti eulero implicito per G e *Ottima* per F. In modo analogo al caso precedente sono stati eseguiti test per $p = 2, 4, 8, 16, 32, 64, 128$ processori mentre i tempi sono riportati solo per $p \leq 32$.

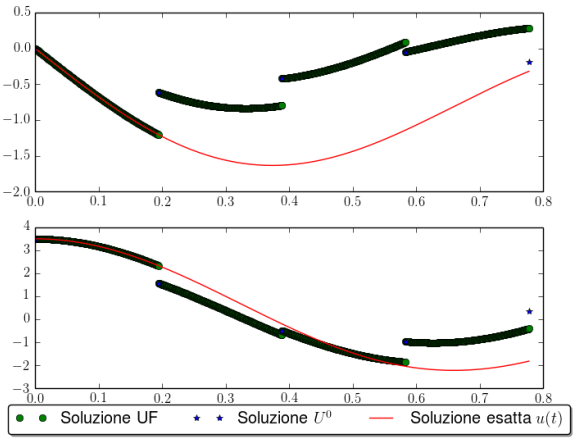
Tabella 23: Tempi d'esecuzione con $p = 2, 4, 8, 16, 32$ processori

p	tempo minimo	tempo massimo	tempo medio	tempo seriale
2	0.219	0.219	0.219	0.213
4	0.221	0.230	0.226	0.213
8	0.160	0.162	0.160	0.213
16	0.110	0.121	0.118	0.213
32	0.102	0.116	0.109	0.213

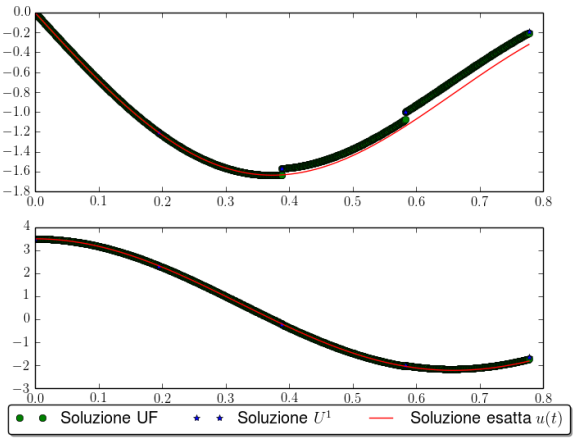
Tabella 24: Relazione p-k.

p	2	4	8	16	32	64	128
k	2	4	5	5	4	4	3

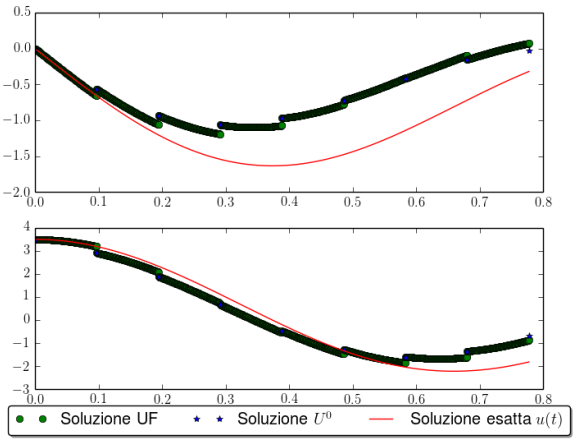
Riportiamo inoltre, a titolo di esempio, i grafici per qualche iterata dell'approssimazione della soluzione esatta con $p = 4$ e $p = 8$ processori



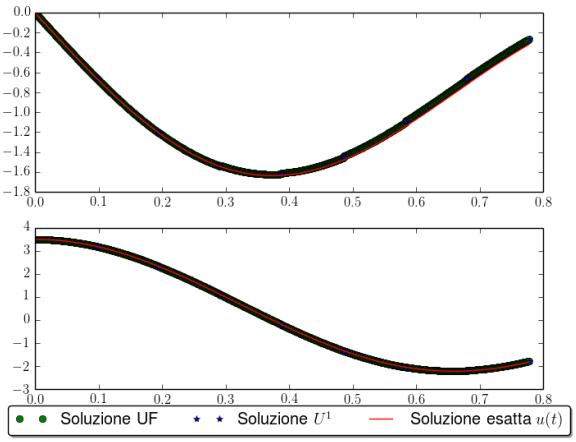
(a) $k = 0$, $p = 4$



(b) $k = 1$, $p = 4$



(c) $k = 0$, $p = 8$



(d) $k = 1$, $p = 8$

Figura 22: Metodo *EI* - *Ottima*

7.10.4 Nota finale

Il passaggio da 16 a 32 processori si è rilevato cruciale per i tempi di esecuzione. Non si ha infatti un netto miglioramento come nei casi precedenti, 2 volte su 3 si ha un peggioramento dei tempi. Questo dipende dal fatto che il propagatore G lavora su $32 = 2^5$ intervalli, come anche il propagatore F. In questo caso quindi non si ha che $\frac{\Delta_2 T}{\Delta_1 T}$ è grande rispetto a $\frac{\Delta_1 T}{\Delta_0 T}$, quindi, come viene descritto in (7.8.4), ci si aspetta un degrado nello speedup.

8 Conclusione

Come ci si era prefissati all'inizio della stesura di questo lavoro, è stata fatta una analisi dell'algoritmo Parareal sia teorica sia pratica.

Per lo studio della convergenza si è partiti dai concetti classici per lo studio di risolutori di ODE, per ottenere un risultato sulla convergenza, che però non torna utile se si vuole utilizzare una griglia rada per il propagatore G. Infatti il teorema (3.1.9) mostra un comportamento asintotico per $h \to 0$. Il teorema (3.1.14) mostra invece che l'algoritmo converge in maniera più che lineare in k, e questo lascia sperare che si possa raggiungere la convergenza in poche iterate.
Per lo studio della stabilità sono stati usati inizialmente gli strumenti classici per l'analisi dei metodi seriali, e tramite l'analisi della funzione di stabilità si sono ottenuti diversi criteri per capire se e sotto quali condizioni l'algoritmo è stabile.

Lo studio della efficienza e speedup purtroppo dimostra che tali quantità sono inversamente proporzionali al numero di iterate eseguito dall'algoritmo Parareal. Questa proprietà purtroppo mina le motivazioni per l'uso dell'algoritmo Parareal, vi è infatti il rischio che convenga, dal punto di vista operativo, utilizzare l'algoritmo definito dal propagatore $\mathcal{F}$ in seriale per impiegarvi meno tempo. È stata anche fatta una breve analisi sui tempi di idle e comunicazione.

Poichè la efficienza è inversamente proporzionale al numero di iterate è stato riportato qualche risultato in merito ai test di arresto, in quanto è importante saper riconoscere quando l'algoritmo ha raggiunto la convergenza per evitare di sprecare risorse inutilmente.

Per implementare l'algoritmo è stato usato il linguaggio di programmazione Python$^{\text{TM}}$, che si è rilevato più che sufficiente per poter effettuare una analisi numerica del comportamento dell'algoritmo, e ha permesso di concentrarsi maggiormente sullo studio dell'algoritmo, che delle problematiche implementative legate ai linguaggi di programmazione. I risultati confermano quanto mostrato nella teoria, si è posto particolare attenzione sul numero di iterate ed i tempi di esecuzione, mostrando che vale effettivamente la relazione di proporzionalità inversa.
Lo studio pratico dell'algoritmo ha permesso di osservare alcuni risultati della teoria che non sono saltati subito all'occhio, come ad esempio convenga avere $\frac{\Delta_1 T}{\Delta_0 T}$ grande rispetto a $\frac{\Delta_2 T}{\Delta_1 T}$, come notato in (7.8.4). Questo perchè nella teoria è difficile dare una stima delle costanti C_G e C_F, oltre al fatto che si è fatta una analisi asintotica, ovver per il numero di processori p che cresce in modo indefinito, mentre dal lato pratico il numero di processori a disposizione è limitato.

Per rendere disponibile il codice che è stato implementato ed utilizzato per lo studio del metodo Parareal, è stato creata creata una pagina web all'indirizzo `https://code.google.com/p/pyparareal/`, dal quale è possibile scaricare anche questo elaborato in formato PDF.

Ci sono alcuni aspetti che sono stati tralasciati e che meriterebbero di essere trattati in uno studio futuro. Inanzitutto l'algoritmo Parareal non è stato concepito inizialmente per le ODE, ma per le PDE. Sebbene dal punto di vista teorico, una volta introdotto uno spazio agli elementi finiti, ci si può ridurre ad un sistema di ODE, dal lato pratico una implementazione parallela è sicuramente più impegnativa, in quanto si possono sfruttare due diversi parallelismi, sia quello temporale, tramite il metodo Parareal, sia quello spaziale, tramite i metodi di *Domain Decomposition*.
Un altra tematica che non è stata approfondita sono come migliorare la efficienza e lo speedup, utilizzando ad esempio un eventuale parallelismo dei propagatori, e ridurre allo stesso tempo i tempi di idle.
Un altro argomento, per il quale non è stato trovato nessun risultato in letteratura, è l'uso di un propagatore a passo adattivo per la prima iterata di G. Sebbene tale possibilità sia già stata implementata e testata in Python$^{\text{TM}}$, esso presenta alcune difficoltà, sia implementative, come ad esempio definire l'intervallo su cui si sta risolvendo il problema, sia analitiche, ad esempio calcolare la convergenza dell'errore dell'algoritmo. L'argomento non è stato ulteriormente approfondito, anche se è sicuramente rilevante per risolvere ODE per tempi lunghi.

Riferimenti bibliografici

Guillaume Bal. Parallelization in time of (stochastic) ordinary differential equations. *Math. Meth. Anal. Num.(submitted)*, 2003.

Guillaume Bal. On the convergence and the stability of the parareal algorithm to solve partial differential equations. In *Domain decomposition methods in science and engineering, , R. Kornhuber et al. Eds, Lect. Notes Comput. Sci. Eng., 40*, pages 425–432. Springer, 2005.

Wael R Elwasif, Samantha S Foley, David E Bernholdt, Lee A Berry, Debasmita Samaddar, David E Newman, and Raul Sanchez. A dependency-driven formulation of parareal: parallel-in-time solution of pdes as a many-task application. In *Proceedings of the 2011 ACM international workshop on Many task computing on grids and supercomputers*, pages 15–24. ACM, 2011.

Paul F Fischer, Frédéric Hecht, and Yvon Maday. A parareal in time semi-implicit approximation of the navier-stokes equations. In *Domain decomposition methods in science and engineering*, pages 433–440. Springer, 2005.

Martin J Gander and Stefan Vandewalle. Analysis of the parareal time-parallel time-integration method. *SIAM Journal on Scientific Computing*, 29(2):556–578, 2007.

Jürgen Geiser and Stefan Güttel. Coupling methods for heat transfer and heat flow: Operator splitting and the parareal algorithm. *Journal of mathematical analysis and applications*, 388(2):873–887, 2012.

Bianca Lepsa and Adrian Sandu. An efficient error control mechanism for the adaptive'parareal'time discretization algorithm. In *Proceedings of the 2010 Spring Simulation Multiconference*, page 87. Society for Computer Simulation International, 2010.

Jacques-Louis Lions, Yvon Maday, and Gabriel Turinici. Résolution d'edp par un schéma en temps «pararéel». *Comptes Rendus de l'Académie des Sciences-Series I-Mathematics*, 332(7):661–668, 2001.

Yvon Maday and Gabriel Turinici. The parareal in time iterative solver: a further direction to parallel implementation. In *Domain decomposition methods in science and engineering*, pages 441–448. Springer, 2005.

Daniel Ruprecht and Rolf Krause. Explicit parallel-in-time integration of a linear acoustic-advection system. *Computers & Fluids*, 59:72–83, 2012.

Debasmita Samaddar, David E Newman, and Raúl Sánchez. Parallelization in time of numerical simulations of fully-developed plasma turbulence using the parareal algorithm. *Journal of Computational Physics*, 229(18):6558–6573, 2010.

Gunnar A Staff. The parareal algorithm, Tech. Rep., Norwegian University of Scienze and Technology, 2003a.

Gunnar A Staff. Convergence and stability of the parareal algorithm: A numerical and theoretical investigation. Tech. Rep., Norwegian University of Scienze and Technology, 2003b.

JMF Trindade and JCF Pereira. Parallel-in-time simulation of two-dimensional, unsteady, incompressible laminar flows. *Numerical Heat Transfer, Part B: Fundamentals*, 50(1):25–40, 2006.