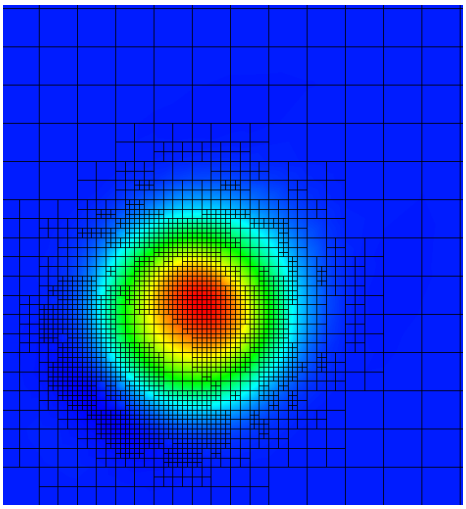
UNIVERSITÀ DEGLI STUDI DI PADOVA
FACOLTÀ DI INGEGNERIA
CORSO DI LAUREA IN INGEGNERIA AEROSPAZIALE

Tecniche di parallel computing

algoritmi paralleli in C++ per la risoluzione
di PDE in ambiente openMP, MPI e DEAL II

Omar Kahol
1162382



Tesi di laurea in Aerodinamica

Indice

a mamma e papà

1 INTRODUZIONE

La risoluzione di problemi ingegneristici è spesso legata ad un insieme di modelli matematici che si presentano in forma di equazioni differenziali alle derivate parziali. Tali modelli devono essere convalidati e risolti per poter ottenere dei risultati utili per le analisi successive. Trascurando il problema della validazione del modello, si può certamente affermare che una soluzione analitica, assegnato un dominio Ω e delle condizioni al contorno sulla sua frontiera $\partial\Omega$, in generale non esiste. Si è quindi costretti a procedere mediante metodi risolutivi numerici. In questo elaborato verrà presentato sia il metodo delle differenze finite che il metodo degli elementi finiti. La filosofia che sta alla base della maggior parte di questi metodi è quella di "riduzione dei gradi di libertà della soluzione cercata". Ciò significa che, la soluzione analitica, in un dominio arbitrario, si esprime mediante una funzione $u(\vec{x}, t)$ che risolve il sistema di equazioni in tutti i punti interni al dominio e rispetta le condizioni al contorno nei punti della frontiera. Tale soluzione analitica può essere vista come un insieme di ∞^n numeri, ovvero infiniti valori che dipendono da n parametri liberi (n è la dimensione del problema). Per un modello 3D, ad esempio, ci sarà richiesto di conoscere il valore in un punto (x, y, z, t) qualsiasi del dominio e ciò corrisponde a conoscere ∞^4 valori totali. La semplificazione attuata dai modelli numerici consiste "nell'accontentarsi" di un insieme non infinito di valori noti. Ad esempio potremmo esprimere la soluzione come un numero $N_x \cdot N_y \cdot N_z \cdot N_t$ di valori, in cui le N rappresentano il numero di gradi di libertà assegnati ad una particolare dimensione. Nel metodo delle differenze finite tali parametri sono i punti in cui si risolve l'equazione e, nel metodo degli elementi finiti, il numero di iterazioni temporali e il numero di gradi libertà della soluzione. È importante notare che la soluzione numerica non è una interpolazione della soluzione analitica nei punti del dominio e questo ci permette di cercarla mediante la risoluzione di problemi diversi e più semplici della PDE.
Se il problema è ben posto allora l'errore commesso diminuisce all'aumentare del numero di punti. Ciò ci spinge, per ottenere risultati sempre più precisi, ad avere un numero molto elevato di parametri liberi e a lasciare che sia l'enorme potenza di calcolo di un computer a risolvere il problema numerico. Un programma viene tipicamente eseguito in maniera sequenziale, ciò significa che la CPU esegue un'unica operazione alla volta. Un codice numerico che implementa questo tipo di approccio è detto seriale. Questa limitazione non è un problema per una buona percentuale di problemi, considerato che il numero di operazioni che possono essere svolte in un secondo è molto elevato; però, in alcuni casi, un codice seriale risulta essere troppo lento e quindi inadatto e poco efficiente. Questo elaborato intende quindi presentare un modo alternativo di risolvere tali problemi che consiste nella suddivisione del carico computazionale a più processori. Due approcci sono dunque possibili:

1) Memoria condivisa, cioè le macchine hanno libero accesso agli stessi dati. L'implementazione sfrutta l'ambiente openMP che consiste in una serie di direttive al preprocessore che permettono di parallelizzare in modo

automatico una serie di istruzioni seriali.

2) Memoria distribuita, cioè ogni macchina dispone della sua memoria non condivisa. La comunicazione avviene tramite uno scambio di informazioni. L'interfaccia utilizzata per implementare questo standard è MPI (Message Passing Interface) ovvero una libreria che implementa una serie di funzioni che permettono lo scambio di dati tra processi differenti.

2 PROCEDIMENTO ADOTTATO

Le tecniche di parallel computing discusse sono state implementate per risolvere il seguente problema:

$$\frac{\partial u}{\partial t} + (\vec{c} \cdot \nabla)u = k\nabla^2 u$$

al quale sono state imposte condizioni al contorno periodiche in modo tale da rendere l'insieme Ω (in cui è definita la soluzione) un anello, per il caso monodimensionale, e un toroide nel caso bidimensionale.
La scelta del problema è giustificata dalla sua semplicità (in alcuni casi esiste anche una soluzione analitica) che ha quindi permesso un maggiore focus alla parallellizzazione del codice (piuttosto che alla codifica) ma anche dalla somiglianza con l'equazione della quantità di moto della meccanica dei fluidi. Sì può notare infatti la presenza di un termine diffusivo e di un termine convettivo (che in questo caso è però lineare).
Il linguaggio utilizzato è il C++; la scelta è guidata dal fatto che tale linguaggio supporta pienamente entrambi gli standard openMP e MPI, è molto veloce, è un linguaggio ad oggetti (per questo motivo si presta molto per l'implementazione di codici scientifici molto strutturati) ed è efficiente nella gestione della memoria.
Tutti i codici prodotti sono disponibili su GitHub:
`https://github.com/omi14098/Parallel-PDE-solvers`
e raggruppati in questo modo

1. DIFFERENZE FINITE
   - 1.1 1D
     - 1.1.1 OPENMP
     - 1.1.2 MPI
     - 1.1.3 OPEMP - adpative meshes
   - 1.2 2D
     - 1.1.1 OPENMP
     - 1.1.2 MPI
2. ELEMENTI FINITI

Il metodo delle differenze finite è semplice da implementare e per questo motivo è stato scelto per illustrare i meccanismi di un codice parallelo. Tutte le funzioni,

tranne quelle della standard template library, sono state codificate in maniera indipendente, senza l'ausilio di librerie esterne. L'implentazione in 2D verrà usata per estendere il concetto di suddivisione del dominio in MPI e le tecniche di allocamento della memoria. Il codice che sfrutta il metodo degli elementi finiti è stato invece concepito per fornire una implementazione robusta e flessibile di un solver parallelo. Per fare ciò si è fatto un uso di librerie esterne. Quella principale è DEAL II che, attraverso dei wrapper per un insieme di altre librerie (prima fra tutte PETSc per l'implementazione di algoritmi paralleli per l'algebra lineare), offre un ambiente comodo per la risoluzione di PDE. Il codice FEM risolve il problema analizzato implementando le seguenti features:

1 **DIMENSION INDIPENDENCE** = l'implementazione è, grazie all'utilizzo dei templates del linguaggio C++, pressochè indipendente dalla dimensione del dominio fisico in cui è definita la soluzione.

2 **ADAPTIVE MESHING** = ogni n iterazioni temporali la mesh viene modificata in modo tale da essere più fitta nelle zone in cui la stima dell'errore è maggiore.

3 **MPI** = parallelizzazione completa utilizzando dei wrapper per le librerie PETSc che implementano degli algoritmi per svolgere operazioni di algebra lineare in parallelo con MPI.

3 PARALLEL DESIGN PATTERNS

In un codice possono esistere due tipi di parallelismo

1 IMPLICITO, cioè la suddivisione del carico computazione non è a carico del programmatore. Il parallelismo può essere il risultato del linguaggio utilizzato (linguaggi di alto livello) o del compilatore scelto. Questo ha il vantaggio di semplificare l'implementazione ma rende i bug più difficili da risolvere e peggiora l'efficienza del programma.

2 ESPLICITO, cioè la parallellizzazione avviene mediante costrutti o annotazioni, che sono al di sopra del linguaggio utilizzato, che devono essere invocati in maniera esplicita dal programmatore. Il programma è potenzialmente molto efficiente.

Una ulteriore classificazione delle tipologie, a livello hardware, del parallelismo è esplicata dalla tassonomia di Flynn. Più precisamente, tale classificazione interessa l'hardware dei calcolatori e influenza il tipo di algoritmo parallelo che deve essere scritto. La tassonomia di Flynn individua come parametri fondamentali i seguenti:

1. INSTRUCTION, cioè il numero di istruzioni differenti che vengono eseguite. Un sistema può essere o single o multiple instruction

2. DATA, cioè il numero di dati diversi elaborati. Un sistema può essere o single o multiple data.

Quindi i tipi di elaboratori considerati sono i seguenti

1. SISD [single instruction single data] = l'hardware non implementa nessun tipo di parallelismo. Questo significa che il processore esegue una istruzione alla volta su un unico stream di dati. I programmi seriali possono essere elaborati da questo tipo di architettura.

2. MISD [multiple instruction single data] = architettura poco comune e che non trova, limitatamente all'interesse di questo elaborato, applicazioni pratiche.

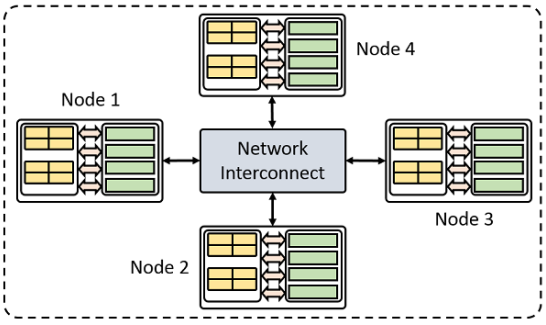
3. SIMD [single instruction multiple data] = l'elaboratore esegue un'unica istruzione su diversi stream di dati. È il tipico modello di GPU computing.

4. MIMD [multiple instruction multiple data] = parallelizzazione totale a livello di dati elaborati ed istruzioni eseguite. Ciò significa che ogni processore esegue una istruzione diversa su flussi di dati diversi e il problema viene quindi diviso in sottoproblemi quasi totalmente indipendenti. A seconda del modo in cui avviene lo scambio di informazioni tra i processori si ha la seguente classificazione:

   (a) SISTEMI A MEMORIA CONDIVISA - I processori lavorano in maniera asincrona condividendo una parte della memoria. Questo è il tipo di architettura che viene sfruttata dai codici in openMP. La comunicazione avviene mediante la sincronizzazione (perdita parziale dell'asincronia) delle operazioni.

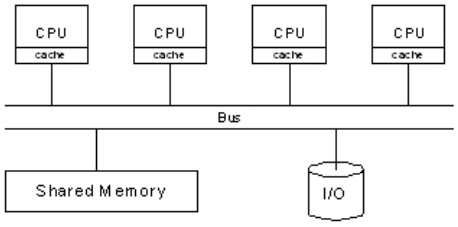
   (b) SISTEMI A MEMORIA DISTRIBUITA - I processori lavorano in maniera asincrona su pool di dati differenti. La comunicazione avviene mediante uno scambio di informazioni sincronizzato. Gli standard MPI (message passing interface) permetto di eseguire codice su queste architetture.

Un cluster di elaboratori ha la seguente struttura.



https://cdn.comsol.com/wordpress/2014/10/Model-of-cluster.png

Ogni nodo consiste in un elaboratore a memoria condivisa.



https://static.usenix.org

Ogni processore dispone di una memoria privata, detta cache, e di una connessione attraverso una interfaccia ad una memoria condivisa. Le operazioni di I/O sono anch'esse sincronizzate da un interfaccia ma non riguardano questo elaborato. È importante notare che le cache possono essere anche condivise da più processi. Questo fatto porta a delle spiacevoli conseguenze che un codice shared memory non può permettersi di ignorare:

1 DATA RACE = Una variabile presente nella memoria condivisa e utilizzata da due processi diversi viene caricata nelle rispettive cache dei due processori. A questo punto ogni modifica effettuata da un processore non è visibile all'altro. Si dice cioè che i processori non hanno una "consistent view" della memoria. È immediato concludere che non è possibile avere più processi che effettuano contemporaneamente operazioni di tipo write sulla stessa variabile.

2 FALSE SHARING = Questo fenomeno si verifica soprattutto quando si opera con array. Se due processori lavorano su due variabili indipendenti che sono però sulla stessa cache line allora ogni modifica di una variabile invalida automaticamente l'altra, nonostante siano indipendenti. Ciò costringe il secondo processore ad effettuare una operazione di read e ciò diminuisce il livello di asincronismo del programma che rischia di diventare seriale.

Si evince che il modo in cui vengono gestite le operazioni di read e write gioca un ruolo fondamentale nella corretta esecuzione del programma. Il programma scritto dal programmatore contiene una serie di queste istruzioni ordinate secondo uno schema logico ben definito. Il compilatore può scegliere di mantenere tale ordine (sequential consistency) oppure di cambiarlo (relaxed consistency). Per ragioni legate alla performance, l'output del compiler (questo vale per i comiplatori C++ usati in questo programma ovvero "mpicxx compiler" e "GNU C compiler") riordina le operazioni di read e write. OpenMP, ad esempio, implementa dei costrutti che permettono di forzare in un punto qualsiasi del programma le operazioni di read e write. Detto ciò, è importante prestare molta attenzione alle operazioni di sincronizzazione (per evitare data race) e alle operazioni di memory allocation (se si lavora su array o su std::vector).

4 IL PROBLEMA DI TRASPORTO DIFFUSIONE

L'equazione di trasporto diffusione modella il trasporto di una sostanza, la cui concentrazione è espressa in termini di una funzione $u(x, y, z, t)$. Il modello completo è il seguente

$$\frac{\partial u}{\partial t} + \nabla \cdot \vec{C}u = \nabla \cdot K \nabla u$$

Il termne $C(x, y, z, t)$ indica la velocità del fluido di trasporto mentre il coefficiente $K(x, y, z, t)$ indica il coefficiente di diffusione. Una semplificazione molto comune è quella di considerare sia la velocità che la diffusività costanti. Si ottiene quindi il modello descritto nei paragrafi precedenti che, in 1D, assume la forma

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = k\frac{\partial^2 u}{\partial x^2}$$

Prima di procedere al calcolo di una soluzione numerica, è importante fare qualche considerazione sulle soluzioni analitiche che è possibile calcolare. Consideriamo infatti il seguente problema

$$\begin{cases} \frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = k\frac{\partial^2 u}{\partial x^2} \\ u(x, 0) = u_0(x) \end{cases}$$

In cui chiediamo che la funzione u sia definita nel seguente dominio

$$u(x, t) \in (-\infty, \infty) \times [0, \infty)$$

La condizione iniziale verrà imposta alla fine e, per ora, è lasciata come una funzione generica. La tecnica utilizzata per il calcolo delle soluzioni analitiche sfrutta la teoria dell'analisi modale. In questo modo si vanno a cercare soluzioni nel dominio della frequenza al posto di quello dello spazio. Per ogni frequenza assegnata esiste quindi una soluzione particolare. Sfruttando le tecniche dell'analisi del segnale, è possibile "ricombinare" tali soluzioni per ottenere una soluzione che rispetta anche le condizioni iniziali. Per procedere è quindi necessario definire la trasformata di Fourier di un segnale e alcune proprietà fondamentali. Si definisce trasformata di Fourier di un segnale $x(t)$ l'integrale

$$X(jw) = F(x(t)) = \int_{-\infty}^{\infty} x(t)e^{-jwt}dt$$

Trascurando i dettagli matematici che governano l'esistenza di tale integrale, possiamo ricordare un'importante proprietà della TdF

$$F(\frac{d}{dt}x(t)) = jw \cdot F(x(t))$$

Questa proprietà è fondamentale perchè ci permette di trasformare una equazione differenziale in una equzione algebrica. L'antitrasformata di Fourier di un segnale è infine definita come

$$x(t) = F^{-1}(X(jw)) = \frac{1}{2\pi}\int_{-\infty}^{\infty} X(jw)e^{jwt}dw$$

Calcolando la trasformata di Fourier (per l'incognita x) dell'equazione di trasporto-diffusione si ottiene

$$\frac{\partial}{\partial t}U(jw,t) + jwcU(jw,t) = k(jw)^2U(jw,t)$$

Avendo definito

$$U(jw,t) = F_x(u(x,t))$$

L'equazione trasformata è una semplice ODE, la cui soluzione analitica è la seguente funzione

$$U(jw,t) = C_1(w)e^{t(-jwc-kw^2)}$$

In cui C_1 è una costante di integrazione, nel nostro caso anche una possibile funzione di ω, che può essere determinata applicando le condizioni iniziali. Infatti, con una semplice sostituzione si verfica che la soluzione analitica è

$$U(jw,t) = U_0(jw,t)e^{t(-jwc-kw^2)}$$

Tale funzione può poi essere ritrasformata nel dominio usuale utilizzando la trasformata di Fourier inversa. Consideriamo ora una concentrazione iniziale pari a

$$u_0(x,t) = \delta(x)$$

In cui la funzione $\delta(x)$ è la delta di Dirac. Secondo la teoria del segnale, tale oggetto matematico è definito dalla seguente proprietà

$$\int_{-\infty}^{\infty} g(x)\delta(x)dx = g(0)$$

Tale funzione può quindi essere vista come un impulso concentrato nell'origine. Essendo la TdF della delta pari alla costante 1, si ha che la TdF della soluzione analitica corrispondente è pari a

$$U(jw,t) = e^{t(-jwc-kw^2)}$$

Sfruttando le tecniche della teoria del segnale, è possibile calcolare la funzione che ammette questa TdF ovvero

$$u(x,t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} e^{t(-jwc-kw^2)}e^{jwt}dw = \frac{1}{2\sqrt{\pi kt}}e^{-\frac{(x-ct)^2}{4at}}$$

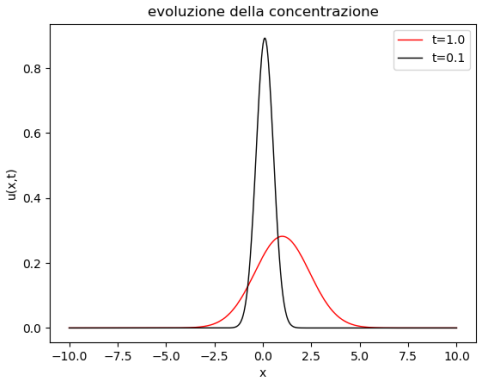
I calcoli che permettono di arrivare a questo valore sono stati omessi. Un fatto molto importante è che la "massa" totale della sostanza, la cui concentrazione è espressa in termini di $u(x,t)$, rimane costante nel tempo e assume il valore pari alla "massa" della condizione iniziale

$$m(t) = \int_{-\infty}^{\infty} \frac{1}{2\sqrt{\pi kt}}e^{-\frac{(x-ct)^2}{4at}}dx = 1$$

e infatti, per definizione della delta

$$m(0) = \int_{-\infty}^{\infty} u_o(x)dx = \int_{-\infty}^{\infty} \delta(x)dx = 1$$

Questo fatto è una diretta conseguenza della legge della conservazione della massa. È interessante osservare visivamente come la soluzione evolva nel tempo. Per farlo imponiamo $k = c = 1$ (per semplicità). La soluzione al tempo $t = 0$ è nota ed è pari ad un impulso nell'origine. Visualizziamo la soluzione agli istanti $t = 0.1s$ e $t = 1.0s$



Si nota che tale soluzione viene contemporaneamente trasportata nella direzione di moto e diffusa in tutte le direzioni. Un'altra importante proprietà di tale equazione è che

$$\lim_{t\to+\infty} u(x,t) = \lim_{t\to+\infty} \frac{1}{2\sqrt{\pi kt}} e^{-\frac{(x-ct)^2}{4at}} = 0$$

Si nota quindi che, nonostante la soluzione venga trasportata lontano da alcuni punti del dominio, la diffusione riesce a uniformare, al limite, la concentrazione della sostanza su tutto il dominio.

Calcolando tre volte la trasformata di Fourier e calcolando altre tre volte l'antitrasformata di Fourier si può ottenere anche la soluione analitica per la stessa PDE in 3D ovvero

$$\frac{\partial u}{\partial t} + c_x\frac{\partial u}{\partial x} + c_y\frac{\partial u}{\partial y} + c_z\frac{\partial u}{\partial z} = k(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2})$$

Definendo quindi

$$U(k_x, k_y, k_z, t) = F_z(F_y(F_x(u(x, y, z, t))))$$

E ripetendo gli stessi identici calcoli precedenti si ottiene la seguente ODE

$$\frac{\partial U}{\partial t} + jc_xk_xU + jc_yk_yU + jc_zk_zU + ak_x^2U + ak_y^2U + ak_z^2U = 0$$

La cui soluzione analitica è

$$U = e^{-t(jc_xk_x+jc_yk_y+jc_zk_z+ak_x^2+ak_y^2+ak_z^2)}$$

Avendo già imposto una condizione iniziale pari a

$$u(x, y, z, 0) = \delta(x)\delta(y)\delta(z)$$

Il calcolo dell'antitrasformata è identico a quello del caso 1D ma necessita di essere ripetuto 3 volte. Ci si accorge però che il risultato deve essere simmetrico e assume quindi la forma

$$u(x, y, z, t) = \frac{1}{(4\pi kt)^{3/2}} e^{-\frac{(x-c_xt)^2+(y-c_yt)^2+(z-c_zt)^2}{4kt}}$$

Si può quindi dedurre che se si aumenta la dimensione del problema, la soluzione tende al suo valore limite più velocemente.

Ci preoccupiamo ora della costruzione di una soluzione analitica per il problema analizzato con il metodo delle differenze finite. Cerchiamo prima di risolvere lo stesso problema ma con condizione iniziale pari a

$$u_0(x) = sin(x)$$

La cui TdF è il segnale

$$U_0(jw) = j\pi(\delta(w+1) - \delta(w-1))$$

La TdF della soluzione analitica sarà quindi

$$U(jw, t) = j\pi(\delta(w+1) - \delta(w-1))e^{t(-jwc-kw^2)}$$

Per le proprietà $x(t)\delta(t-t_0) = x(t_0)\delta(t-t_0)$ prodotto segnale-impulso

$$U(jw, t) = j\pi(\delta(w+1)e^{t(+jc-k)} - \delta(w-1)e^{t(-jc-k)})$$

La cui trasformata inversa è il segnale

$$u(x, t) = sin(x - ct)e^{-kt}$$

Se la condizione iniziale è una sinusoide di frequenza diversa e pari a $sin(mx)$ allora i calcoli sono molto simili e portano al risultato seguente

$$U(jw, t) = j\pi(\delta(w+m)e^{t(+jmc-km^2)} - \delta(w-m)e^{t(-jcm-km^2)})$$

La cui antitrasformata è la funzione

$$u(x, t) = sin(m(x - ct))e^{-km^2t}$$

Il problema che ci poniamo di risolvere è il seguente

$$\begin{cases} \frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = k\frac{\partial^2 u}{\partial x^2}, u(x,t) \in (0,1) \times (0,1) \\ u(0,t) = u(1,t) \\ \frac{\partial u}{\partial x}(0,t) = \frac{\partial u}{\partial x}(1,t) \\ u(x,0) = \sin(2\pi x), x \in [0,1] \end{cases}$$

Notiamo subito che la soluzione del problema è già nota. Infatti, per questioni legate alla simmetria del dominio scelto (un anello), il comportamento della soluzione deve essere simile a quello della soluione del problema definito in $(-\infty, \infty)$. Si verifica che la funzione

$$u(x,t) = sin(2\pi(x - ct)) \cdot e^{-k(2\pi)^2 t}$$

risolve il problema differenziale del dominio e rispetta le codizioni al contorno assegnate ed è pertanto la soluzione del problema assegnato.

Tuttavia, è possibile ottenere tale soluzione anche utilizzando le tecniche dell'analisi modale. Non è possibile usare la trasformata di Fourier nel dominio del tempo nè in quello spaziale perchè nessuno dei sue domini coincide con quello di definizione della TdF (ergo la retta dei reali). Definiamo quindi un'altra trasformata integrale detta trasformata di Laplace unilatera. Sia $x(t)$ un segnale qualsiasi, allora la sua TdL è la funzione

$$X(s) = \mathcal{L}(x(t)) = \int_0^\infty x(t)e^{-st}dt$$

Con $s = \alpha + j\omega$ numero complesso. È possibile definire anche la trasformata di Laplce inversa ma generalmente si preferisce, piuutosto che calcolarla direttamente, scomporre la trasformata di Laplce in funzioni elementari (nel nostro caso fratti semplici) di cui si conosce la funzione generatrice. Questa operazione può essere svolta con l'ausilio di tabelle di trasformate elementari oppure con un software di calcolo simbolico. Una importante proprietà della trasfromata di Laplace unilatera è la seguente

$$\mathcal{L}(\frac{d}{dt}x(t)) = sX(s) - x(0)$$

Con queste conoscenze possiamo applicare la TdL al nostro problema per ottenere

$$sU(x,s) - u_0(x) + cU_x(x,s) - kU_{xx}(x,s) = 0$$

Si ottiene la seguente ODE non omogenea

$$-kU_{xx}(x,s) + cU_x(x,s) + sU(x,s) = sin(2\pi x)$$

le condizioni al contorno sono

$$U(0,s) = U(1,s), U_x(0,s) = U_x(1,s)$$

Per semplicità e per chiarezza espositiva, rappresentiamo questo problema nella forma classica, sostituento $y(x) = U(x, s)$

$$\begin{cases} -ky''(x) + cy'(x) + sy(x) = sin(2\pi x) \\ y(0) = y(1) \\ y'(0) = y'(1) \end{cases}$$

Questo è un problema noto come Boundary Value Problem (BVP) con condizioni al contorno di tipo periodico. Dalla teoria di base sappiamo che possiamo scrivere la soluzione di tale problema come

$$y(x) = Ae^{\lambda_1 x} + Be^{\lambda_2 x} + y_p(x)$$

In cui la parte $Ae^{\lambda_1 x} + Be^{\lambda_2 x}$ è la soluzione del problema omogeneo associato e λ_i sono le radici del suo polinomio caratteristico. Le costanti A, B di integrazione si possono calcolare imponendo le condizioni al contorno e, nel nostro caso risultano identicamente nulle. La soluzione particolare si può calcolare in tre modi.

1 FUNZIONI DI GREEN, cerchiamo una funzione $G(x, y)$ che risolve il problema non omogeneo con forzante pari a $\delta(x-y)$. La soluzione particolare sarà data da $y(x) = \int_0^1 G(x, y)f(y)dy$

2 TRASFORMATA DI FOURIER, sfruttiamo, grazie alle condizioni al contorno periodiche, la simmetria che esiste tra $\mathcal{R}$ e il dominio della soluzione.

3 FUNZIONE PARTICOLARE, la soluzione particolare, per poter risolvere il problema, deve essere del tipo $y_p(x) = Asin(2\pi x) + Bcos(2\pi x)$. Inserendo questa funzione nell'ODE si possono calcolare i due parametri.

Tutti e tre i casi portano alla seguente soluzione

$$y(x) = U(x, s) = \frac{(4\pi^2 k + s)sin(2\pi x) - 2\pi c cos(2\pi x)}{4\pi^2 c + (4\pi^2 k + s)^2}$$

Inserendo questa funzione in un software di calcolo simbolico si ottiene che l'antitrasformata di Laplace, ovvero la soluzione del problema, è

$$u(x, t) = sin(2\pi(x - ct)) \cdot e^{-k(2\pi)^2 t}$$

Come già ipotizzato in precedenza.

5 IL METODO DELLE DIFFERENZE FINITE

La soluzione numerica con il metodo delle differenze finite prevede inizialmente la discretizzazione dell'operatore derivata temporale. Le modalità più elementari sono

1 ESPLICITO

$$\frac{u^{n+1}-u^n}{\Delta t} = \mathcal{L}(u^n)$$

2 IMPLICITO

$$\frac{u^{n+1}-u^n}{\Delta t} = \mathcal{L}(u^{n+1})$$

È naturale chiedersi se e per quali scelte di Δt le soluzioni ottenute saranno stabili. Per rispondere a questa domanda è necessario però fornire una classificazione del modello, in particolare sappiamo che tale equazione ha un carattere iperbolico (dato dal termine convettivo) e un carattere parabolico (dato dal termine diffusivo). Si dimostra che il metodo implicito è sempre stabile mentre quello esplicito necessita di uno step temporale deve essere proporzionale alla finezza della discretizzazione spaziale, cioè Δx, nel caso di problemi iperbolici e al suo quadrato, nel caso di problemi parabolici. Il metodo implicito sembra quindi essere la scelta migliore però il metodo esplicito è più semplice dal punto di vista dell'implementazione. Per questo motivo è stato preferito. Per la scrittura del codice in DEAL.II ho considerato altre alternative:

1 IMEX cioè usare un metodo implicito per trattare il termine parabolico e un metodo esplicito per il termine iperbolico.

$$\frac{u^{n+1}-u^n}{\Delta t} + \vec{c}\cdot\nabla u^n = k\nabla^2 u^{n+1}$$

2 θ-SCHEME ovvero

$$\frac{u^{n+1}-u^n}{\Delta t} = \mathcal{L}(\theta u^{n+1} + (1-\theta)u^n)$$

Dalla scelta della discretizzazione spaziale si ottiene il seguente problema

$$\frac{u_i^{n+1}-u_i^n}{\Delta t} + c\frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x} = k\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}$$

In questa scrittura i e n corrispondono ai gradi di libertà della dimensione spaziale e temporale ovvero $i = 1..N_x$ e $n = 1..N_t$. L'incognita del problema è quindi u_i^{n+1} e può essere calcolata direttamente con due cicli for.

5.1 la struttura del codice

La scelta della gestione della memoria è direttamente influenza dalla scelta della gestione dell'I/O. Infatti è necessario conoscere la soluzione approssimata per ogni istante discreto t_n. Ciò può essere effettuato utilizzando due array $u_{new}[i]$ e $u_{old}[i]$ oppure un'unica matrice $u[t][i]$. La prima scelta ci costringe ad effettuare un operazione di scrittura sul un file ad ogni iterazione temporale e, siccome tali operazioni sono parecchio costose (la memoria utilizzata è infatti più lenta di quella RAM) è stata scartata (l'ho usata invece nel codice FEM). La struttura dati più adatta al problema è un semplice array con un accesso agli elementi circolare ovvero qualcosa del tipo

```
double &access(double *pMem, const int &at) {
  if ((at%MAX) < 0) {
    return pMem[MAX+(at%MAX)];
  else {
    return pMem[at%MAX];
  }
}
```

L'allocamento della memoria lascia invece le seguenti possibilità. È possibile creare un doppio array in cui ogni elemento corrisponde ad un vettore soluzione; oppure un unico array "flatten".

1. MATRICE

```
double **pMemory = new double*[MAX_TIME];
for(int t=0; t<MAX_TIME; t++) {
  pMemory[t] = new double[MAX_X];
}
```

2. FLATTEN ARRAY

```
double *pMemory = new double[MAX_TIME*MAX_X];
```

La prima opzione è la più naturale e si può estendere a più di due dimensioni ma è molto inefficiente perchè rallenta l'accesso ai dati. Infatti ogni elemento contiene un puntatore ad un indirizzo di memoria che contine puntatori ad altri indirizzi di memoria che contengono puntatori all'indirizzo dell'elemento in questione. È stata usata la quindi seconda opzione. Per accedere ad un particolare elemento (t,i) si usa:

```
access(pMemory, t*MAX_X+i);
```

Per conludere, ad ogni iterazione temporale, il codice deve calcolare e salvare l'energia dell' errore

$$E(t) = \int_0^1 ||u_{esatto}(x,t) - u_{approx}(x,t)||_2^2 dx$$

Una routine si dovrà inoltre occupare di salvare su un file .csv la soluzione, stampare alcune informazioni utili sulla consolle (tempo di esecuzione, ..) e chiamare (se richiesto) un post-processore scritto in python che dovrà visualizzare i risultati. I codici sono stati eseguiti su una CPU "Intel(R) Core(TM) i7-4510U" con 4 cores fisici.

5.2 open MP

Il codice seriale prevede essenzialemente due elementi, la matrice della soluzione e un processo. Quest'ultimo lavora, elemento per elemento sulla memoria. L'approccio shared memory permette quindi di creare un insieme di processori

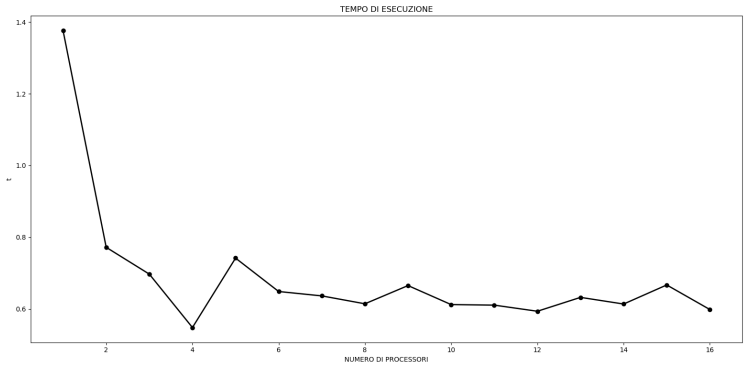
e di spartire le operazioni sulla matrice tra di essi. Una volta scelta le strutture dati adeguate, le modifiche che rendono il codice parallelo sono davvero minime. OpenMP permette infatti di usare delle pragma per automatizzare il procedimento. Una pragma è una direttiva al preprocessore, il quale ha il compito di modificare il codice prima di inviarlo al compilatore e al linker. La parallelizzazione comincia con la creazione di un numero di processi asincroni e l'assegnazione ad ognuno di loro di una id numerica (che va da 0 a $N_{processori} - 1$). L'operazione che viene parallizzata è esclusivamente il ciclo for riguardante la discretizzazione spaziale. OpenMP richiede l'aggiunta di

```
#pragma omp parallel [...]
    for(t=0; t<MAX_TIME-1; t++)
      [...]
#pragma omp for [...]
      for(i=0; i<MAX_X; i++)
      [...]
```

In realtà prima di entrare in un regione parallela è necessario dichiarare quali variabili saranno private e quali condivise e altre operazioni. IL primo comando crea una regione parallela e il secondo modifica il ciclo for per creare una sorta di Round-Robin distribution. Cioè spartisce gli elementi in modo tale da assegnare ogni elemento ad un processore diverso. Se ci sono, ad esempio, 4 processori e 10 elementi si ha che al primo vengono assegnati gli elementi [0,4,8] al secondo [1,5,9], al terzo [2,6] e al quarto [3,7]. Per implementare ciò basterebbe modificare leggeremente il ciclo for.

```
#pragma omp parallel
    for(t=0; t<MAX_TIME-1; t++)
      ...
      for(i=ID; i<MAX_X; i+=N_processori)
      ...
```

Questo codice è efficiente e non presenta problemi legati a data race o al false sharing. Ciò accade perchè alla fine di ogni iterazione temporale $t-1$ openmp garantisce che ogni processo possieda una "consistent view" della memoria condivisa cioè forza il completamento di tutte le operazioni di read e write e aspetta che ogni thread abbia effettuato altrettanto. Quindi è garantito che tutti i processi comincino l'iterazione temporale t contemporaneamente. Durante questa operazione ogni processo deve effettuare delle operazioni di read su pMemory[t] e delle operazioni di write sulla nuova soluzione pMemory[t+1]. Notiamo che è impossibile che ci sia un processo che debba leggere dei valori che un altro processo deve modificare e quindi non ci possono essere problemi di data race. I risultati in termine di tempo di esecuzione, per $N_x = 10001$ e $N_t = 1000$ è



Mentre l'errore in funzione del numero di processori risulta essere identicamente uguale ad un valore dell'ordine di 10^{-3}. Come si può notare, l'errore è indipendente dal numero di processori e lo speedup è evidente fino a 4 processori (e ciò è coerente con l'architettura usata). Oltre tale numero si ha che il tempo di esecuzione rimane pressochè costante e si assesta sul limite teorico.

5.3 MPI

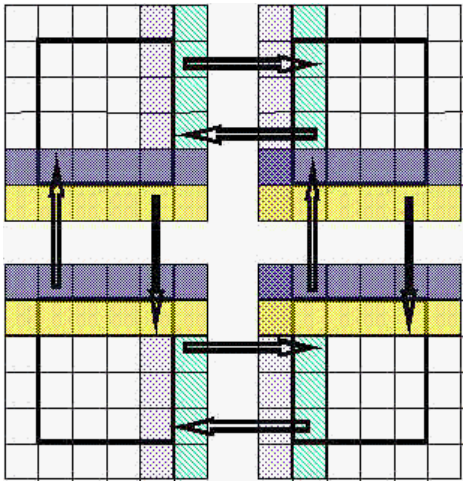
Lo standard MPI implementa delle librerie che permettono la comunicazione di POD (plain old data) tra più processi. I processi possono esistere nello stesso nodo o in nodi diversi e per questo motivo l'approccio si adatta anche a sistemi shared memory; la velocità di comunicazione dipende essenzialmente dalla velocità con cui possono essere trasmessi i dati. Il codice è stato pertanto eseguito sulla stessa macchina. Le strutture dati e le operazioni effettuate sono le stesse, ciò che cambia è la suddivisione del lavoro. Un processo dispone di una pozione del dominio $\Omega_s \subseteq \Omega$ e della funzione condizione iniziale. Ciò significa che ogni processo dispone di

$$u_0(x_i, t), x_i \in \Omega_s$$

ma non può evolvere in maniera asincrona tale soluzione perchè la derivata nei punti della frontiera $x_i \in \partial\Omega_s$ richiede la conoscenza del valore dei punti appartenenti ad un altra porzione. Ovvero

$$u(x_i, t+1) = f(x_{i+1}, x_i, x_{i-1})$$

La condizione $x_i \in \Omega_s$ è sicuramente vera ma non necessariamente per gli altri punti. Il valore del punto sulla frontiera dovrà essere cominicato da un altro processo. Per poterlo calcolare è necessario che questo punto sia nel dominio interno del processore. Ovvero le porzioni del dominio devono avere delle zone in comune, di overlap.



Per dieci punti e 2 processi si ha che

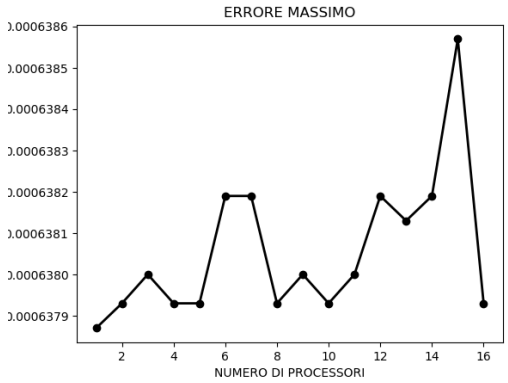
$$\begin{cases} \Omega_1 = [x_0, x_1, x_2, x_3, x_4, x_5] \\ \Omega_2 = [x_4, x_5, x_6, x_7, x_0, x_1] \end{cases}$$

Ovvero ad ogni iterazione temporale il nodo 1 deve calcolare il nuovo valore dei punti interni x_1, x_2, x_3, x_4 e ricevere il valore di x_0, x_5 punti di frontiera (che sono infatti punti interni del dominio del nodo 2). Tale nodo dovrà anche inviare al nodo 2 il valore dei punti x_4, x_1. La soluzione iniziale viene calcolata attraverso questa funzione:

```
const double phase{id*(nPoints-2)*DX};
double x{0.0};
for(int i=0; i<nPoints; i++) {
    x = phase + i*DX;
    solution[i] = std::sin(2*M_PI*x);
}
if (id == numProcs-1) {
    solution[nPoints-2] = std::sin(2*M_PI*0);
    solution[nPoints-1] = std::sin(2*M_PI*DX);
}
```

In cui la variabile phase rappresenta il valore del punto iniziale del dominio. L'ultima partizione si occupa anche dei due punti iniziali della prima partizione (per via delle condizioni al contorno di tipo periodico). Il resto del codice è simile al codice seriale. La comuncazione viene gestita attraverso le due funzioni ISend e IRecv che implementano la comunicazione monodirezionale di tipo non-blocking. Un costrutto alla fine dell'iterazione temporale si assicura che tutte le comunicazioni siano avvenute con successo prima di procedere con la successiva iterazione. Ogni partizione si occupa anche di calcolare l'energia dell'errore nel proprio sottodominio e di inviarla (alla fine) al processo master che le somma e ne calcola il valore massimo. Il tempo di esecuzione in funzione del numero di processi è il seguente

Tale valore è stato calcolato tenendo conto del tempo di esecuzione medio di ogni processo. Come si può facilemente notare la scalabilità è ottima fino a 4 processori. L'errore in funzione del numero di processori è il seguente



Il grafico può sembrare sospetto ma ho calcolato che il valore medio è circa $6 \cdot 10^{-4}$ e la deviazione standard è $1.73 \cdot 10^{-7}$. Ciò significa che tale quantità non dipende dal numero di processori e le oscillazioni sono dovute al fatto che il numero di punti effettivi su cui viene calcolata la soluzione è nominalmente 10001 ma viene modificato leggermente, in funzione del numero di processori, per poter equipartire il dominio in maniera semplice.

6 IL METODO DEGLI ELEMENTI FINITI

L'obiettivo è quello di produrre un codice a elementi finiti che sia il più possibile modulare e parallelo. Una caratteristica molto importante è che non deve dipendere dalla dimensione ed essere facilmente modificabile per risolvere ogni

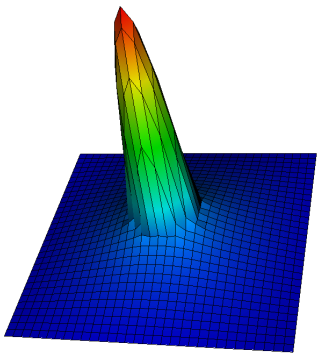
equazione lineare. Il problema è

$$\frac{\partial u}{\partial t} + (\vec{c} \cdot \nabla)u = k\nabla^2 u$$

Le condizioni al contorno scelte sono di tipo periodico. La condizione iniziale $u_0(\vec{x})$ deve a sua volta risolvere il problema di Laplace

$$\begin{cases} -\nabla \cdot a(x)\nabla u(x,t) = f(x), u(x) \in \Omega \\ u(x) = 0, u(x) \in \partial\Omega \end{cases}$$

Per un'opportuna scelta di coefficienti in modo tale che $f(x)$ sia non nullo solo in un intorno dell'origine e che $a(x)$ abbia il valore minimo in tale intorno. In questo modo si ottiene una soluzione iniziale che è il più possibile continua e derivabile e "non nulla" solo in tale intorno dell'origine.



6.1 la teoria alla base del metodo FEM

Si riporta di seguito una breve discussione dell'approccio variazionale, che sta alla base dei metodi FEM, usando come esempio la costruzione della soluzione iniziale. Sia V uno spazio vettoriale costituito dalle funzioni $\phi(x)$ continue e derivabili in Ω e nulle in $\partial\Omega$. Sia $< v(x), w(x) >= \int_\Omega v(x)w(x)dx$ il prodotto scalare tra elementi di V. Si cerca quindi una funzione $u(x) \in V$ per cui vale

$$- < \phi(x), \nabla \cdot a(x)\nabla u(x) >=< \phi(x), f(x) >$$

Questa scrittura deve valere per ogni funzione $\phi(x) \in V$. Integrando per parti (e sfruttando la struttura di V)

$$< \nabla\phi(x), a(x)\nabla u(x) >=< \phi(x), f(x) >$$

L'approssimazione introdotta dal metodo FEM è quindi la ricerca della soluzione non in V (che è uno spazio infinito dimensionale) ma in $V_h \subset V$ a dimensione finita. Se N è la dimensione di V_h e $\phi_i(x), i = (1..N)$ una sua base allora possiamo scrivere la soluzione di tale problema come $u_h(x) = U_j\phi_j(x)$ in cui U_j sono N coefficienti incogniti. Le ϕ_i sono funzioni che hanno la caratteristica di essere non nulle solo in una porzione del dominio. Scrivendo la condizione per tutte le N funzioni base abbiamo un sistema di N equazioni in N incognite che può essere scritto come

$$< \nabla\phi_i(x), a(x)\nabla\phi_j(x) > U_j =< \phi_i(x), f(x) >$$

$$A_{ij}U_j = F_i$$

In cui la matrice A è sparsa. Questo è chiamato metodo di Galerkin.

6.2 formulazione FEM del problema

Per avere una matrice A simmetrica e definita positiva, il time stepping utilizzato è l'IMEX (implicit-explicit time stepping) con un Δt proporzionale alla finezza della mesh. Il problema è quindi il seguente.

$$\frac{u^{n+1} - u^n}{\Delta t} + \vec{c}\cdot\nabla u^n = k\nabla^2 u^{n+1}$$

La ricerca della soluzione avviene quindi in un sottospazio V_h formato da funzioni ad energia finita. Non ho imposto condizioni particolari sul valore delle funzioni in $\partial\Omega$ in quanto il dominio in cui viene risolta è un toroide. Deve quindi valere, integrando per parti

$$\begin{aligned}
&< \phi_i^{n+1}, \phi_j^{n+1} > U_j^{n+1} - < \phi_i^{n+1}, \phi_j^n > U_j^n + \\
&\Delta t < \phi_i^{n+1}, \vec{c}\cdot\nabla\phi_j^n > U_j^n + \\
&k\Delta t < \nabla\phi_i^{n+1}, \nabla\phi_j^{n+1} > U_j^{n+1} = 0
\end{aligned}$$

La presenza dell'indice temporale sulle funzioni base è dovuta al fatto che la mesh al tempo (n+1) non corrisponde esattamente alla mesh al tempo (n). Questa formulazione è però difficile da implementare. Si sostituisce perciò la funzione $U_j^n\phi_j^n$ con $[I^{n+1}U_j^n]\phi_j^{n+1}$, in cui $I^{n+1}U_j^n$ è la funzione che interpola la soluzione definita nella mesh al tempo (n) alla mesh al tempo (n+1). A questo punto si possono omettere gli indici temporali nelle funzioni base e formulare il problema come

$$A_{ij}U_j^{n+1} = F_i$$

$$\begin{cases} A_{ij} =< \phi_i, \phi_j > +k\Delta t < \nabla\phi_i, \nabla\phi_j > \\ F_i = [-\Delta t < \phi_i, \vec{c}\cdot\nabla\phi_j > + < \phi_i, \phi_j >]\cdot U_j^n \end{cases}$$

6.3 la struttura del codice

Trascurando il problema della costruzione della soluzione iniziale, il codice deve creare la porzione della mesh assegnata ad ogni processore, distribuire i gradi libertà del sistema, assemblare una porzione della matrice e termine noto, risolvere tale sistema ed eventualmente cambiare la mesh. La struttura è ispirata dai seguenti codici

1. step-26, un solver completamente seriale che risolve l'equazione del calore su delle adaptive meshes.
2. step-17, un solver parzialmente parallelo shared memory MPI che risolve il problema elastico.
3. step-45, un solver parzialmente parallelo per le equazioni di Stokes su un dominio periodico.
4. step-40, un solver completamente parallelo che risolve il problema di Poisson.

Le librerie utilizzate sono quindi le seguenti

1. DEAL 2, offre un ambiente in cui risolvere facilmente le PDE implementando funzioni per gestire le matrici sparse, gli elementi finiti, le formule di quadratura, ...
2. PETSc, algoritmi di algebra lineare paralleli.
3. p4est, algoritmi per la ripartizione di mesh tra più processori.

Di seguito è riportato il prototipo della classe che risolve il problema proposto.

```
template <int dim>
class solver {
public:
    solver();
    void run();
private:
    void makeGrid(const int&);
    void buildInitialSolution(const int&);
    void setupSystem(const bool&);
    void assembleSystem();
    void assembleLaplaceProblem();
    int solve();
    void refineGrid(const double&, const double&, const int&, const
     int&, const bool&);
    void outputResults(const unsigned int &);
    void printRelevantInfo(const int&);
private:
    MPI_Comm communicator;
    parallel::distributed::Triangulation<dim> mesh;
    FE_Q<dim> fe;
    DoFHandler<dim> dofHandler;

    IndexSet ownedDofs;
    IndexSet relevantDofs;
```

```
24
25      AffineConstraints<double> constraints;
26      SparsityPattern sparsityPattern;
27      LinearAlgebraPETSc::MPI::SparseMatrix systemMatrix;
28      LinearAlgebraPETSc::MPI::Vector localSolution;
29      LinearAlgebraPETSc::MPI::Vector systemRhs;
30
31      ConditionalOStream pcout;
32      TimerOutput timer;
33
34      const double K{0.8};
35      Tensor<1,dim> C;
36      double timeStep{1.0/1000};
37      int timeStepNumber{0};
38      double time{0.0};
39  };
```

6.3.1 le variabili

Siccome il codice deve essere completamente parallelo è necessario assicurarsi che ogni processore disponga esclusivamente delle variabili associate alla propria partizione. La prima variabile è il comunicatore MPI che consiste essenzialmente in un wrapper per tutte le variabili low level di MPI. La mesh è di tipo **parallel::distributed::Triangulation** perchè deve contenere una porzione delle celle. La variabile **FE_Q** implementa la definizione dello spazio vettoriale V_h costituito da particolari polinomi di Lagrange mentre **DoFHandler** gestisce l'enumerazione dei gradi di libertà. I due **IndexSet** rappresentano i gradi di libertà posseduti da una particolare partizione e sono di tipo owned (ovvero quelli del dominio del processo) o relevant (dominio e ghost layers). L'oggetto **AffineContraints** consite in un tensore, in cui vengono salvate tutte le restrizione sui gradi di libertà (condizioni al contorno, hanging nodes, ...), che sarà applicato al sistema (o alla soluzione). Questa classe è un template nella variabile della dimensione fisica in cui si risolve il problema. È quindi possibile creare più istanze con dimensioni diverse. La classe TimerOutput tiene conto delle varie chiamate alle funzioni e al tempo impegato, producendo un output molto utile per le analisi.

6.3.2 makeGrid

La funzione makeGrid crea un dominio ipercubico (in 2D un quadrato, in 3D un cubo, ...) e lo rispartisce ad ogni processo. In DEAL 2 le mesh sono formate da quadrati e vengono salvate in un struttura detta quadtree (o un octree in 3D). Un quadtree è un struttura dati ad albero in cui ogni nodo ha 4 nodi figli. La mesh è quindi creata partendo da un unico nodo che viene successivamente suddiviso n volte. Ciò crea una struttura a livelli il cui ultimo livello fornisce i vertici delle celle attive. Questa funzione si occupa anche di salvare le celle associate a condizioni al contorno periodiche.

6.3.3 setupSystem

In questa routine vengono, per in prima istanza, assegnati i gradi di libertà associati al dominio di ogni processore. La funzione inizializza anche lo **sparsityPattern** della matrice (ovvero gli ingressi non nulli) e grazie a ciò reinizializza la matrice, il termine noto e la soluzione. La presenza di un unico vettore soluzione potrebbe sembrare sospetta ma in realtà la funzione setupSystem viene chiamata solamente quando cambiano i gradi di libertà della soluzione (ciò quando cambia la mesh). In questo caso si utilizza una variabile temporanea in cui salvare la vecchia soluzione che sarà successivamente interpolata nel nuovo dominio. In questa funzione viene anche costruito l'oggetto **AffineConstraints**.

6.3.4 assembleSystem

La funzione assembleSystem riempie gli ingressi della matrice e del termine noto del sistema lineare. Per fare ciò dobbiamo calcolare una quantità della forma.

$$< L_1(\phi_i(x)), L_2(\phi_j(x)) >= \int_\Omega L_1(\phi_i(x)) \cdot L_2(\phi_j(x))dx$$

In cui L_n è un qualsiasi operatore lineare. Trascurando per semplicità l'operatore lineare e calcolando questa quantità sulla singola cella si ottiene

$$\sum_{C=1}^{N} \int_C \phi_i(x) \cdot \phi_j(x)dx$$

Le quantità dentro il segno di integrale sono definite su un'unica "reference cell". Per questo motivo si usa un cambio di variabili attravero la conoscenza di una funzione di mapping ovvero

$$\sum_{C=1}^{N} \int_{C_{ref}} \phi_i(\hat{x}) \cdot \phi_j(\hat{x}) \cdot \det J_C(\hat{x})d\hat{x}$$

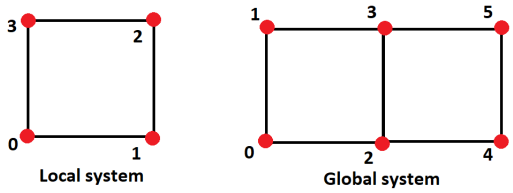
In cui $\hat{x}$ è un punto sulla reference cell e $J_C(\hat{x})$ è la matrice jacobiana del cambio di variabili. Utilizzando una formula di quadratura con pesi w_q e N_q punti x_q di quadratura si ha

$$\sum_{C=1}^{N} \sum_{q=1}^{Nq} w_q \cdot \phi_i(\hat{x}_q) \cdot \phi_j(\hat{x}_q) \cdot \det J_C(\hat{x}_q)$$

La costruizione di tale matrice può essere semplificata enumerando i gradi di libertà globalmente e localmente. Ad ogni grado di libertà della è assegnato un numero che fa riferimento ad una enumerazione locale, per una cella è sempre [0,1,2,3]. Per ognuno di questi numeri è assegnato un secondo numero che indica il numero del grado di libertà globale. Si procede quindi a costruire una matrice, per ogni cella, della forma

$$A_{ij}^C = \sum_{q=1}^{Nq} w_q \cdot \phi_i(\hat{x}_q) \cdot \phi_j(\hat{x}_q) \cdot \det J_C(\hat{x}_q)$$

A questo punto possiamo riempire gli ingressi della matrice globale. Supponiamo ad esempio di avere 2 sole celle



Per la cella numero due gli ingressi locali $[0, 1, 2, 3]$ corrispondono agli ingressi globali $[2, 4, 5, 3]$, l'ingresso locale 02 (prima riga, terza colonna) sarà inserito nella matrice globale nell'ingresso 25, e così via.

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{03} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & a_{00} & a_{03} & a_{01} & a_{02} & 0 & 0 \\ 0 & 0 & a_{30} & a_{33} & a_{31} & a_{32} & 0 & 0 \\ 0 & 0 & a_{10} & a_{13} & a_{11} & a_{12} & 0 & 0 \\ 0 & 0 & a_{20} & a_{23} & a_{21} & a_{22} & 0 & 0 \end{bmatrix}$$

Chiamando b_{ij} gli ingressi della cella enumerata globalmente con $[0, 2, 3, 1]$ e ripetendo lo stesso procedimento otteniamo la matrice globale

$$\begin{bmatrix} b_{00} & b_{03} & b_{01} & b_{02} & 0 & 0 & 0 & 0 \\ b_{30} & b_{33} & b_{31} & b_{32} & 0 & 0 & 0 & 0 \\ b_{10} & b_{13} & a_{00} + b_{11} & a_{03} + b_{12} & a_{01} & a_{02} & 0 & 0 \\ b_{20} & b_{23} & a_{30} + b_{21} & a_{33} + b_{22} & a_{31} & a_{32} & 0 & 0 \\ 0 & 0 & a_{10} & a_{13} & a_{11} & a_{12} & 0 & 0 \\ 0 & 0 & a_{20} & a_{23} & a_{21} & a_{22} & 0 & 0 \end{bmatrix}$$

In questo modo si vede chiaramente la struttura della matrice ottenuta e il tipo di operazione che stiamo facendo. Il dover passare prima per un sistema locale è fondamentale per poter lavorare in parallelo. Infatti non possiamo dimenticare che tale matrice non è completa in quanto non soddisfa i vincoli (ad esempio hanging nodes, boundary values, ...). Tali vincoli, in sintesi, modificano gli ingressi della matrice e del termine noto. Siccome tali quantità sono distribuite, i processori devono sincronizzare le operazioni di riempimento, modificare la matrice sparsa e nel frattempo sincronizzare le operazioni eseguite. Per evitare queste operazioni particolarmente esose si preferisce avere una matrice locale sulla quale saranno eseguite le operazioni necessarie, come ad esempio l'applicazione dei vincoli. Una volta che un processore ha completato le operazioni su una cella può riempiere gli ingressi della matrice globale. Tale processo va svolto anche per il termine noto.

6.3.5 solve

Questa funzione è la più semplice dal punto di vista implementativo, in quanto consiste in poco più di 10 righe di codice, ma certamente la più complessa. Per la risoluzione viene quindi usato un solver parallelo fornito dalla libreria PETSc.

6.3.6 refineGrid

Il refinement della mesh computazionale prevede la conoscenza di una stima dell'errore per ogni cella del dominio. Trascurando il problema della scelta della funzione stima, una volta che tale quantità è disponibile, è necessario sceliere quali vanno ulteriormente suddivise e quali devono essere "riunite", inglobate in una unica cella. Per questo problema ho scelto di effettuare un refine per le cells (alle quali è associato l'errore più grande) che contribuiscono al 60% dell'errore totale e un'operazione di coarsening per quelle alle quali è associato l'errore più piccolo e che contribuiscono al 40% di tale errore. Successivamente è necessario iterare nei livelli massimi e minimi consentiti del quadtree per assicurarsi che non ci siano celle troppo grandi o troppo piccole. Una volta completata tale operazione, la soluzione viene copiata in un oggetto chiamato **SolutionTransfer** che si occupa di interpolarla sul nuovo vettore (nel mentre è stata chiamata la funzione setupSystem).

6.4 risultati

Il problema è stato risolto in una mesh con circa 2000 celle con un refinement ogni 11 iterazioni temporali su un totale di 150. L'output del timer, per 1,2,4,8 processori è

Total wallclock time elapsed since start		39.2s	
Section	no. calls	wall time	% of total
assembly	150	4.6s	12%
initial solution	1	1.36s	3.5%
output	151	31.6s	81%
refine	20	2.17s	5.5%
setup system	21	0.419s	1.1%
solve	158	0.564s	1.4%

1 processore

Total wallclock time elapsed since start		25.6s	
Section	no. calls	wall time	% of total
assembly	150	2.73s	11%
initial solution	1	0.212s	0.83%
output	151	21.3s	83%
refine	20	1.14s	4.5%
setup system	21	0.223s	0.87%
solve	158	0.224s	0.88%

2 processori

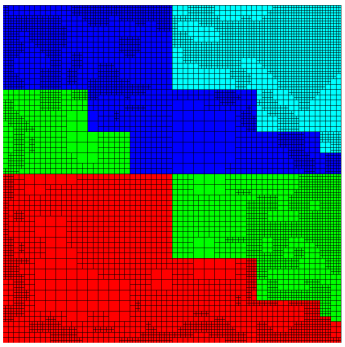
Total wallclock time elapsed since start		20.9s	
Section	no. calls	wall time	% of total
assembly	150	3.06s	15%
initial solution	1	0.292s	1.4%
output	151	15.7s	75%
refine	20	1.31s	6.3%
setup system	21	0.268s	1.3%
solve	158	0.371s	1.8%

4 processori

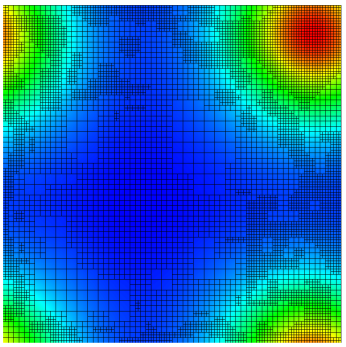
Total wallclock time elapsed since start		21.8s	
Section	no. calls	wall time	% of total
assembly	150	3.82s	18%
initial solution	1	0.517s	2.4%
output	151	12.9s	59%
refine	20	1.93s	8.9%
setup system	21	0.562s	2.6%
solve	158	0.875s	4%

8 processori

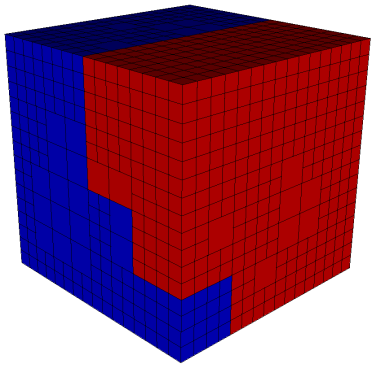
Come possiamo notare, nonostante il basso numero di gradi di libertà, il problema scala abbastanza bene fino a 4 processori. Notiamo che il principale "bottleneck" è l'output dei risultati. Ciò è comprensibile visto che le operazioni di I/O sono piuttosto esose in termini di tempo, soprattutto per memorie non moderne e lente. La partizione del dominio per 4 processori in parallelo è la seguente



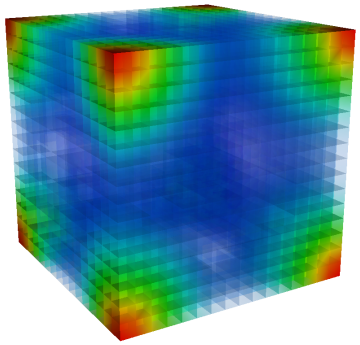
Come possiamo notare la divisione del dominio è funzione del numero di celle. Il risultato dell'integrazione è il seguente



Questo codice, come già specificato è dimension indipendent. Ciò significa che possiamo eseguire lo stesso codice anche in 3D. La mesh è un cubo di lato 2 ed è stata suddivisa, per due processori, nel seguente modo



L'integrazione invece genera il seguente output



7 CONCLUSIONI

Questo lavoro è quindi servito per esplorare le varie tecniche di calcolo parallelo e la loro implementazione in un linguaggio come il C++. Certamente le tecniche MPI e openMP si possono adattare anche alla risoluzione di altre famiglie di PDE. Un solver robusto può mantenere, con le dovute modifiche, una struttura simile a quella proposta in questo elaborato. Una classe di PDE molto importanti e difficili da risolvere sono i problemi di natura non lineare che sono discretizzati in problemi a loro volta non lineari. Nella cartella GitHub del progetto è presente una sezione con altri solver. Le funzioni e, in parte, anche le variabili sono quelle discusse nella parallelizzazione della formulazione FEM. Il calcolo parallelo può essere sfruttato per risolvere anche altri problemi di natura ingegneristica se tali problemi rendono possibile una, anche parziale, suddivisione del carico computazionale.

Quindi il processo di parallelizzazione procede con la scelta del sistema usato per risolvere il problema e la sua riformulazione che ne rende possibile la parallelizzazione. Una volta che una suddivisione del carico computazionale è stata individuata, è necessario identificare tutte le situazioni in cui i processori non possono lavorare in maniera asincrona e realizzare le condizioni per cui il sincronismo si realizzi. Questa, limitatamente all'interesse di questo elaborato, è risultata essere la parte più difficile da implementare e quella che ha generato il maggior numero di bug. Nel caso della formulazione per sistemi shared memory la sincronizzazione è implicita e difficile da identificare. Infatti è quasi sempre possibile scrivere un codice completamente asincrono. Tale codice ha un comportamento ben determinato a compile-time e non produce errori ma un comportamento indeterminato a run-time e, è possibile affermarlo con certezza, produce sempre o errori o risultati non corretti. Nonostante ciò è facile sincronizzare i vari processori. Nel caso dell'approccio MPI, i punti in cui è necessario sincronizzare i processori sono molto evidenti ma tale sicronizzazione è difficile da realizzare e richiede spesso importanti modifiche al codice.

Una ulteriore riflessione è nata nel momento in cui è stato necessario scegliere l'ambiente e i software con cui implementare il solver. Sono infatti disponibili molti linguaggi di programmazione che permettono un qualche livello di parallelismo nel codice. Tra questi si devono citare python, matlab e C++. Quest'ultimo è un linguaggio di basso livello che permette quindi un maggiore controllo delle operazioni e una maggiore velocità di esecuzione pur rimanendo object-oriented e moderno. Per ogni linguaggio sono disponibili diverse librerie per il calcolo scientifico. È chiaro che scrivere un codice, implementando la maggior parte delle funzioni e limitando l'utilizzo di librerie eseterne, è più laborioso ma semplifica il processo di debugging e il tempo impiegato per documentarsi. Tuttavia ho constatato che affidarsi a librerie esterne open source (con la possibilità di visionare ed eventualmente modificare il codice sorgente) permette di scrivere un codice robusto e in grado di garantire un ampio numero di features; pur mantenendo la stessa flessibilità di un codice self-contained.