PROGETTO CORSO ALGORITMI PER IL CALCOLO PARALLELO
A.A. 2023-2024

Progetto: Equazione del calore su griglie strutturate, parallelizzazione con direttive MPI.

Marco Perrotta **A.A: 2023-2024**

Obiettivo

Scrittura, applicazione e parallelizzazione (con direttive MPI) di un programma riguardante l'equazione del calore su griglie strutturate.
Considerando la seguente equazione che descrive la diffusione del calore:

$$\frac{\partial T}{\partial t} - k\nabla^2 T = 0, \quad k = \frac{\lambda}{\rho c},$$

con la seguente discretizzazione alle differenze finite:

$$T_{i,j}^{n+1} = T_{i,j}^{n} + k\frac{\Delta t}{\Delta x^2}\left(T^{n}_{i+1,j} - 2T^{n}_{i,j} + T^{n}_{i-1,j} \right) + k\frac{\Delta t}{\Delta y^2}\left(T^{n}_{i,j+1} - 2T^{n}_{i,j} + T^{n}_{i,j-1} \right)$$

dato Il dominio di calcolo (computational domain) **Ω = [0; 2] × [0; 2]** e il coefficiente di conduzione termica equivalente a **k = 1** valutando condizione iniziale data da:

$$T(0) = \begin{cases} TL & \text{if} \quad x \leq 1 \\ TR & \text{if} \quad x > 1 \end{cases}, \qquad TL = 100, \quad TR = 50.$$

evolvendo la soluzione fino al tempo finale $t_f = 0.05$.
Controllando poi i risultati con la soluzione esatta data da:

$$T_e = \frac{1}{2}(TR + TL) + \frac{1}{2} erf\left(\frac{x-1}{2\sqrt{k\,t}}\right)(TR - TL)$$

Breve descrizione strumenti (Linguaggi) utilizzati

Per l'attuazione del progetto sono stati utilizzati i seguenti linguaggi:

- **Fortran**, acronimo di Formula Translation, è un linguaggio di programmazione ad alto livello sviluppato per scopi scientifici e di calcolo numerico. Utile per la sua efficienza nel manipolare matrici e array e nel gestire calcoli numerici complessi. Fortran ha subito diverse evoluzioni nel corso degli anni, con le versioni più recenti che offrono caratteristiche moderne come la programmazione ad oggetti e il supporto per la parallelizzazione. Utilizzato in ambito scientifico e ingegneristico, specialmente per applicazioni che richiedono prestazioni elevate.
Eseguito per il progetto su ambiente "Visual Studio 2019" e utilizzato per la parallelizzazione con direttive/librerie **MPI**, acronimo di Message Passing Interface, è una specifica per lo sviluppo di programmi paralleli e distribuiti. Consente a processi separati di comunicare tra loro scambiandosi messaggi. Questi messaggi possono includere dati, istruzioni o altri tipi di informazioni necessarie per il coordinamento tra i processi. Offre un'interfaccia standard per la scrittura di programmi paralleli, facilitando lo sviluppo di applicazioni che sfruttano la potenza di calcolo di più nodi o computer.

- **Matlab**, ambiente di sviluppo e linguaggio di programmazione progettato principalmente per il calcolo numerico, la visualizzazione e l'analisi dati.

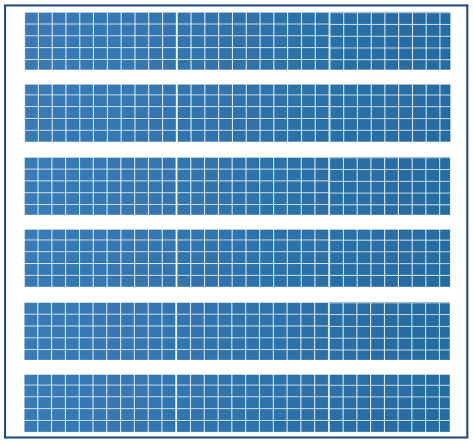
Procedimento

Inizialmente, si è diviso il progetto in diversi punti:

I.Progettazione della risoluzione, del dominio e della comunicazione tra processi

Dopo una prima analisi dell'obiettivo richiesto si è da subito notato che i valori $T^{n+1}_{i,j}$ sono dipendenti dalle temperature T^n precedenti temporalmente e adiacenti nella matrice $(T^n_{i,j}, T^n_{i,j+1}, T^n_{i+1,j}, T^n_{i,j-1}, T^n_{i-1,j})$.
Si è quindi progettato di svolgere una scomposizione in 2 dimensioni con MPI ("*2D domain decomposition with MPI")*, valutando e/o discretizzando con un numero totale di $IMAX * JMAX$ (dove IMAX è il numero totale di celle verticali e JMAX di celle orizzontali) il "Computational Domain" $\Omega = [0; 2] \times [0; 2]$ il quale viene discretizzato quindi per un numero IMAX per JMAX di celle, in modo che ogni cella diventa rappresentativa di una diversa $T^n_{i,j}$ che si trova in posizione (i, j) al tempo n; Successivamente si è suddiviso/partizionato i "gruppi di celle" , create in precedenza, come matrici rettangolari di dimensione: $[\frac{IMAX}{nCpu}; JMAX]$ $con\ IMAX = JMAX\ e\ (IMAX * JMAX)\ multiplo\ di\ nCpu$, dove:

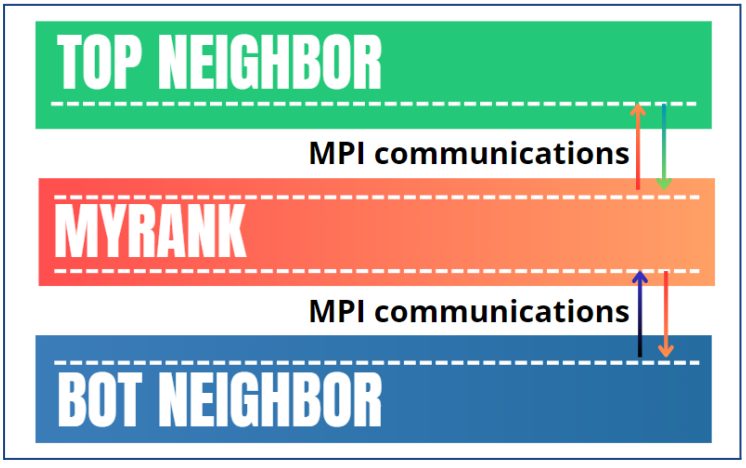
$nCpu$ = *numero di processi (CPU) che si vogliono usare per eseguire in parallelo .*



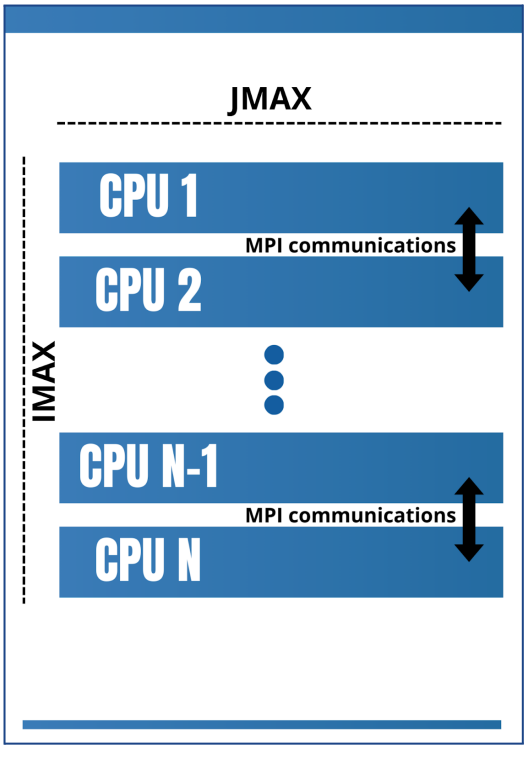
2D domain decomposition e partizione

Si è poi assegnato ad ogni processo/CPU un'area così delineata determinando in questa maniera i processi vicini ("*neighbors of each process*").

E' in seguito stato deciso di utilizzare le comunicazioni MPI per scambiare dati tra processi attraverso i MPI boundaries (confini superiori ed inferiori).



Corrispondenza e comunicazione tra Cpu e neighbors



Assegnazione delle partizioni alle Cpu

L'implementazione su Fortran può essere suddivisa nei seguenti passi fondamentali:

- ⇨ **"Inizializzazione" di base e modifica delle proprietà del progetto per il testing**

```
PROGRAM MAIN
    ! ------------------------------------------------------------------------ !
    IMPLICIT NONE
    INCLUDE 'mpif.h'
```

Dove dichiariamo la tipologia del seguente script come programma di nome MAIN (se fosse richiesta una subroutine x avremmo dovuto dichiarare SUBROUTINE X).
Attraverso la linea di codice "*IMPLICIT NONE*" si afferma che tutte le variabili sono dichiarate, mentre attraverso "*INCLUDE 'mpif.h'* " si include la libreria mpif.

Per quanto riguarda la modifica delle proprietà del progetto si procede a "set-uppare" l'ambiente (Visual studio 2019) procedendo come segue:

1) Si raggiunge l'interfaccia "Proprietà" attraverso l "Esplora soluzioni".***(Figura 1)***
2) Si modificano, accedendo alla sezione "Debugging" delle "Proprietà di configurazione", le voci "Command" e "Command Argument" aggiungendo rispettivamente le seguenti stringhe: " $(I_MPI_ONEAPI_ROOT)\bin\mpiexec.exe" , " -n NUMERO_A_SCELTA $(TargetPath)" dove NUMERO_A_SCELTA (es. 5) rappresenta il numero di CPU che si intende mandare parallelamente in esecuzione. ***(Figura 2)***
3) Si procede poi al inserimento della stringa "$(I_MPI_ONEAPI_ROOT)\include" nella voce "Additional include Directories" della sottosezione "General" della sezione "Fortran", e nella sottosezione "Preprocessor" della stessa macro sezione "Fortran". ***(Figura 3)***
4) Infine si aggiunge la stringa "impi.lib" alla voce "Additional Dependencies" della Sotto-sezione "Input" della sezione "Linker". ***(Figura 4)***

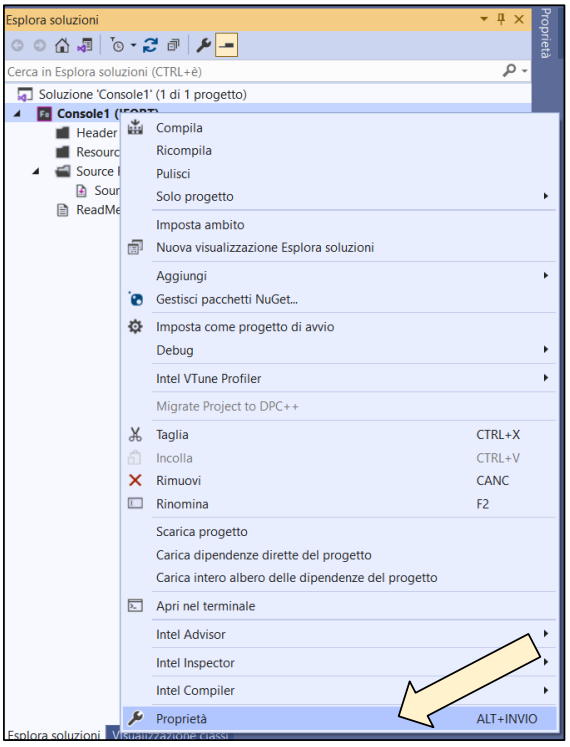


Figura (1)

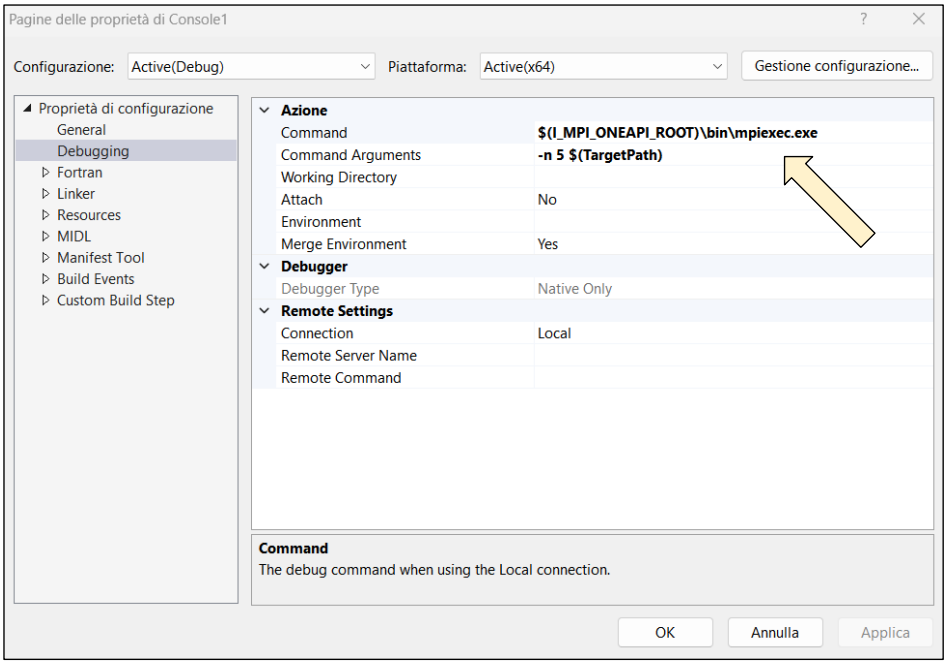


Figura (2)

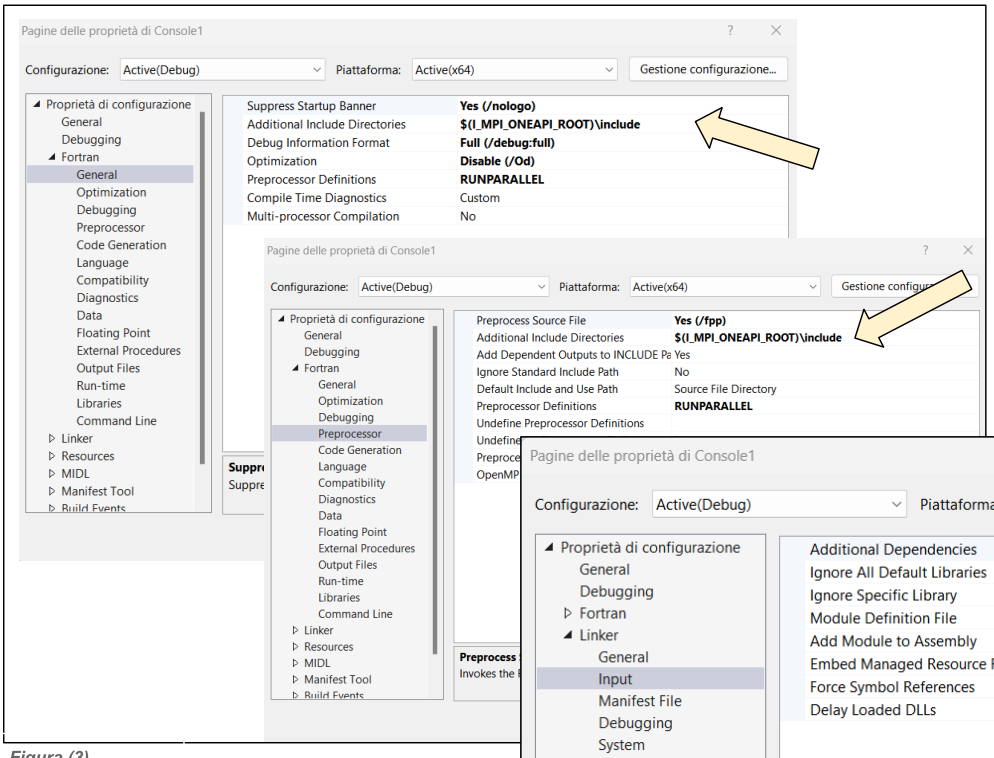


Figura (3)

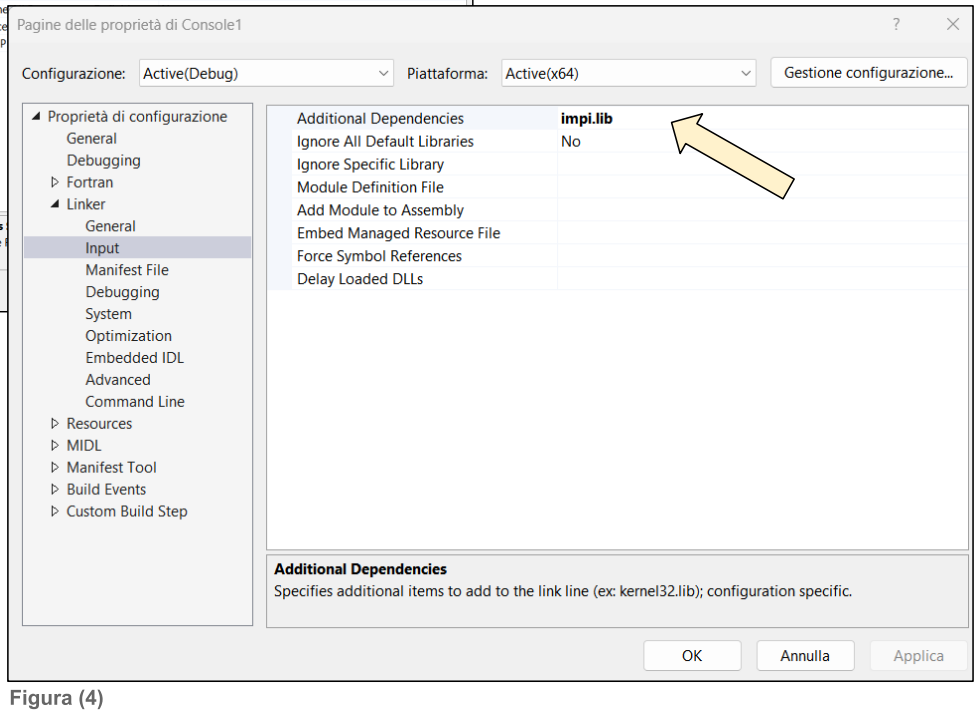


Figura (4)

⇨ Dichiarazione delle variabili utili

```
INTEGER :: i,j, n, timestep = 0, idummy ! utili
INTEGER, PARAMETER :: IMAX=145    ! Numero di celle verticali/orizzontali
INTEGER, PARAMETER :: NMAX=1e6    ! Numero max di time steps
REAL    :: xL, xR  ! left and right coordinate del dominio
REAL    :: yT, yB  ! top and bottom coordinate del dominio

REAL    :: dx, dx2 ! mesh space (and its squared value)
REAL    :: dy, dy2 ! mesh space (and its squared value)

REAL    :: CFL     ! CFL number (<=1 for stability of EXPLICIT SCHEMES)
REAL    :: time    ! Tempo corrente
REAL    :: dt      ! time step
REAL    :: tend    ! Tempo finale
REAL    :: TL, TR  ! "Condizioni di confine"
REAL    :: kappa   ! coefficiente di conduzione del calore = k
INTEGER :: NDIM = 2 ! “quante dimensioni va diviso”

REAL, ALLOCATABLE    :: T(:,:), Tnew(:,:)   ! soluzioni correnti e nuove
REAL, ALLOCATABLE    :: x(:),y(:)           ! coordinate verticali e orizzontali
LOGICAL, ALLOCATABLE :: periods(:)          ! identificare se il dominio è periodico o meno
INTEGER, ALLOCATABLE :: dims(:)             ! dimensioni in cui dividere la griglia
                              !! sezione MPI
INTEGER            :: TCPU, BCPU, MsgLength, nMsg
REAL, ALLOCATABLE  :: send_messageB(:), send_messageT(:), recv_messageB(:), recv_messageT(:)
INTEGER       :: send_request(2), recv_request(2)  ! tag dei messaggi che sono stati mandati
INTEGER       :: send_status_list(MPI_STATUS_SIZE,2), recv_status_list(MPI_STATUS_SIZE,2)
INTEGER       :: source
INTEGER,DIMENSION(8)   :: timeI,timeF

 TYPE tMPI                    ! STRUTTURA/CLASSE (attributi di ogni cpu)
      INTEGER :: myrank, nCPU, iErr
      INTEGER :: iMMax
      INTEGER :: AUTO_REAL            ! variazione della precisione dei numeri reali
      INTEGER :: nElem, iStart, iEnd  ! numero di celle che ha ogni cpu,
      !
      INTEGER, ALLOCATABLE :: myCoords (:) ! coordinate cella in cui si trova la cpu
 END TYPE tMPI

  TYPE(tMPI) :: MPI
  REAL       :: realtest
  INTEGER    :: COMM_CART !! CANALE DI COMUNICAZIONE PER IL PIANO CARTESIANO

  ! ------------------------------------------------------------------ !
```

⇨ Classical MPI initialization

```
!!
! CLASSICAL MPI INITIALIZATION
!!

CALL MPI_INIT(MPI%iErr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MPI%myrank,MPI%iErr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,MPI%nCPU,MPI%iErr)

!------------------------------------------------------------!
```

Dove la chiamata a funzione " CALL **MPI_INIT**(MPI%iErr) " ha l'obiettivo di inizializzare l'ambiente di esecuzione MPI, mentre la chiamata a funzione **MPI_COMM_RANK**(MPI_COMM_WORLD,MPI%myrank,MPI%iErr) determina il rango del processo di chiamata nel comunicatore, ed infine attraverso la chiamata a funzione "CALL **MPI_COMM_SIZE**(MPI_COMM_WORLD,MPI%nCPU,MPI%iErr)" si determina la dimensione del gruppo associato al comunicatore.

⇨ Domain decomposition / Building the computational domain

```
! per capire il "TIPO" DI messaggi da inviare
! per rendere generale la compilazione del codice
!_____________________________________________________
SELECT CASE(KIND(realtest))
CASE(4)
    MPI%AUTO_REAL = MPI_REAL
CASE(8)
    MPI%AUTO_REAL = MPI_DOUBLE_PRECISION
END SELECT
! _____________________________________________________

!!
!  Vero e proprio inizio DOMAIN DECOMPOSITION
!!

IF(MOD(IMAX,MPI%nCPU).NE.0) THEN  ! vogliamo che il NumeroCelle sia multiplo del NumeroCpu
    PRINT *, ' ERROR. Number of cells (IMAX) must be a multiple of number of CPU (nCPU)!'
    CALL MPI_FINALIZE(MPI%iErr)
    STOP
ELSE
    IF(MPI%myrank.EQ.0) WRITE(*,*) ' Parallel simulation with MPI. '
ENDIF
```

Innanzitutto vengono svolti due semplici controlli, il primo per rendere "dinamico" il programma alla variazione della precisione dei numeri reali, il secondo per assicurare che il numero di celle sia multiplo del numero di CPU, seguendo quanto detto nel punto della progettazione.

Dopodichè si procede all' allocamento e assegnazione di valori dei 3 vettori "dims" , "periods", "myCoords":

```
!!ALLOCO
ALLOCATE(dims(NDIM),periods(NDIM),MPI%myCoords(NDIM))
!!ASSEGNO VALORI
dims    = (/1,MPI%nCpu/)
periods = (/.FALSE.,.TRUE./)
```

dove *dims* sono le dimensioni in cui dividere la griglia, *periods* è un array logico di dimensione *dims* che specifica se la griglia è periodica (TRUE) o no (FALSE) in ciascuna dimensione, mentre *mycords* sono le coordinate nella griglia della singola Cpu.

Si procede poi con le seguenti chiamate a funzione:

```
!!
CALL MPI_CART_CREATE(MPI_COMM_WORLD,NDIM,dims,periods,.TRUE.,COMM_CART,MPI%iErr)
CALL MPI_COMM_RANK(COMM_CART,MPI%myrank,MPI%iErr)
```

Attraverso **MPI_CART_CREATE** si crea un nuovo comunicatore a cui sono state allegate le informazioni sulla topologia, mentre attraverso **MPI_COMM_RANK** si determina il rango del processo di chiamata nel comunicatore.

A seguire si è poi fatto identificare ai vari processi i loro "*neighbors*" :

```
! TROVA I VICINI
  CALL MPI_CART_SHIFT(COMM_CART,1,1,source,TCPU,MPI%iErr)
  CALL MPI_CART_SHIFT(COMM_CART,1,-1,source,BCPU,MPI%iErr)
!
```

sia superiori (*MPI_CART_SHIFT(COMM_CART,**1,1**,source,TCPU,MPI%iErr)*) che inferiori (`CALL MPI_CART_SHIFT(COMM_CART,1,-1,source,BCPU,MPI%iErr)`). Da notare nelle due chiamate a funzione i due interi: dove il primo (l' 1, in entrambe le funzioni) identifica la direzione, mentre il secondo (1 nella prima e -1 nella seconda) identifica lo spostamento (> 0: spostamento verso l'alto, < 0: spostamento verso il basso).

Si determinano poi le coordinate dei processi nella topologia cartesiana in base ai ranghi nel gruppo:

```
CALL MPI_CART_COORDS(COMM_CART,MPI%myrank,NDIM,MPI%myCoords,MPI%iErr)
```

Si assegnano le seguenti variabili

```
MPI%nElem  = IMAX                    ! orizzontalmente va dall' inizio alla fine
MPI%iMMax  = IMAX/MPI%nCpu           ! verticalmente  numero di celle diviso per ogni cpu

MPI%iStart = 1 + MPI%myCoords(1)*MPI%iMMax  ! sempre verticalmente
MPI%iEnd   = MPI%iStart + MPI%iMMax - 1     !
```

Si assegna, poi, il valore della lunghezza del messaggio che andremo ad inviare attraverso MPI, messaggio che sarà un vettore identificante riga di inizio o di fine del dominio/partizione del processo, a seconda che debba inviarlo al processo superiore o inferiore; in entrambi i casi la lunghezza del messaggio sarà lunga quanto la larghezza del nostro dominio, ovvero ciò che in fase di progettazione è stato definito JMAX e, essendo JMAX=IMAX, nel codice (per questioni di ottimizzazione) è stato assimilato all'IMAX.

```
MsgLength = IMAX !! NEL CASO MPI%NUMELEM
```

A seguire si procede con l'assegnazione dell'CLF (number <=1 for stability of EXPLICIT SCHEMES) impostato a 0.9:

```
CFL = 0.9 ! CFL number
```

e alla assegnazione dei valori per la domain definition, boundary conditions e alla assegnazione del tempo iniziale t_0, tempo finale t_{end} e k dati dal problema/obiettivo:

```
! Domain definition
xL    = 0.
xR    = 2.

yT = 0.
yB = 2.
! Boundary conditions
TL    = 100.
TR    = 50.

time = 0.0
tend = 0.05

kappa= 1.0
```

Si allocano poi tutte le rimanenti strutture (vettori / matrici) definite in precedenza:

```
ALLOCATE( x(IMAX) , y(IMAX) )                          ! GRIGLIA
ALLOCATE( Tnew(IMAX,IMAX) )                            ! SOL nuova (n+1)
ALLOCATE( T(MPI%iStart-1:MPI%iEnd+1, 1:MPI%nElem) ) ! SOL tempo n
ALLOCATE(send_messageT(MPI%nElem), send_messageB(MPI%nElem))
ALLOCATE(recv_messageT(MPI%nElem), recv_messageB(MPI%nElem))
```

infine, prima di poter procedere all'implementazione della computazione, si procede con l'assegnazione dei valori di Δx e Δy, con il *"Building the computational domain"* (assegnazione sia per le x che per le y, generazione della vera e propria griglia, utile allo studio del grafico e per l'effettiva discretizzazione della propagazione del calore) e con l' *"Assigning the initial condition"* che, come descritto dal problema, prevede una situazione iniziale:

$$T(0) = \begin{cases} TL & \text{if} \quad x \leq 1 \\ TR & \text{if} \quad x > 1 \end{cases}, \qquad TL = 100, \quad TR = 50.$$

come da figura:

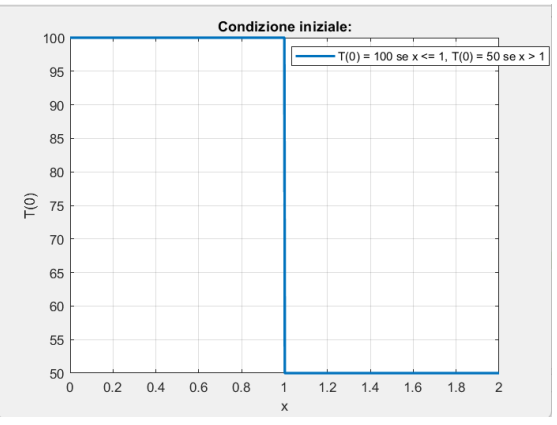


Grafico esemplificativo condizione in T(0)

```
IF(MPI%myrank.EQ.0) WRITE(*,*) ' Building the computational domain... ' !debug!
CONTINUE
! Gestiamo i Δ
dx = (xR-xL)/REAL(IMAX-1)
dx2= dx**2

dy = (yB-yT)/REAL(IMAX-1)
dy2= dy**2

x(1) = xL
DO i = 1, IMAX-1
    x(i+1) = x(i) + dx
ENDDO


y(1) = yT
DO i = 1, IMAX-1
    y(i+1) = y(i) + dy
ENDDO

IF(MPI%myrank.EQ.0) WRITE(*,*) ' Assigning the initial condition... '

DO j = 1, MPI%nElem
    IF(x(j).LE.1) THEN
        T(:,j) = TL
    ELSE
        T(:,j) = TR
    ENDIF
ENDDO

DO j = 1, IMAX
    CALL MPI_ALLGATHER( T(:,j), MPI%iMMax, MPI%AUTO_REAL, Tnew(:,j), MPI%iMMax,
                        MPI%AUTO_REAL, MPI_COMM_WORLD, MPI%iErr )
ENDDO
IF(MPI%myrank.EQ.0) CALL PlotOutputMPI(IMAX,x,y,Tnew,timestep,MPI%myrank,1,IMAX,MPI%nCpu)
IF(MPI%myrank.EQ.0) WRITE(*,*) ' ready FOR THE COMPUTATION '
```

Da notare che attraverso il commando ***MPI_ALLGATHER*** si raccolgono i dati da tutte le cpu e si distribuiscono i dati combinati a tutte le cpu, utilizzato questo comando è stata evocata/chiamata poi la funzione/subroutine prodotta PlotOutputMPI (funzione che permette di conservare le informazioni utili all'analisi dei dati all'interno di un file) dalla sola CPU con my rank = 0.
Si è ora pronti alla fase di computazione effettiva del codice.

⇨ Computazione

Si prevede, per la risoluzione, un "main loop in time" il quale itererà, aggiornando il valore n, (che parte da 1 al passo dei time step) fino a NMAX e fino a quando il tempo attuale non risulta > del tempo previsto per la terminazione (*fintanto che* $t_{att} < t_{end}$) :

```
DO n = 1, NMAX   ! main loop in time

      IF(time.GE.tend) EXIT

      dt = 0.5*CFL*dx2/kappa
      IF(time+dt.GT.tend) THEN
          dt = tend-time  ! adjust the last time step in order to exactly match tend
      ENDIF
```

L'idea principale si basa sul mandare tutti i messaggi e le istruzioni per riceverli, intanto che i seguenti messaggi vengono recapitati, si svolgono parti di computazione dove non è necessario l'arrivo dei "messaggi"; a seguire vi sarà un update delle celle di bordo "Update of the MPI boundary cell" che hanno bisogno della recapitazione dei messaggi per essere computate.

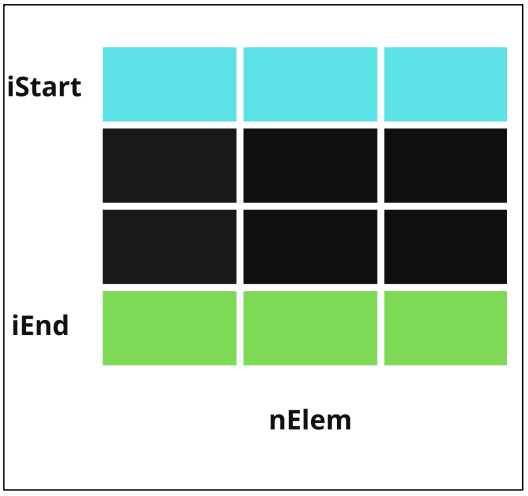
L'effettiva computazione può essere,quindi, divisa in 3 passi fondamentali:

- ❖ Creazione della struttura per inviare e ricevere i messaggi ("Setup all communication")

```
      !!
      ! Message exchange
      !!
      ! TCPU E BCPU SETTATI IN PRECEDENZA CON MPI_CART_SHIFT
      ! 1) SETUP ALL COMMUNICATIONS
      ! CREIAMO PRIMA I MESSAGGI
      send_messageT = T(MPI%iStart, 1 : MPI%nElem)
      send_messageB = T(MPI%iEnd,   1 : MPI%nElem)

      ! N.B no blocking comunication
      CALL MPI_ISEND(send_messageT,MsgLength,MPI%AUTO_REAL,TCPU,1,COMM_CART,
                     send_request(1),MPI%iErr) ! Bot al Top
      CALL MPI_ISEND(send_messageB,MsgLength,MPI%AUTO_REAL,BCPU,2,COMM_CART,
                     send_request(2),MPI%iErr) ! Top al Bot

      CALL MPI_IRECV(recv_messageT,MsgLength,MPI%AUTO_REAL,TCPU,2,COMM_CART,
                    recv_request(1),MPI%iErr)
      CALL MPI_IRECV(recv_messageB,MsgLength,MPI%AUTO_REAL,BCPU,1,COMM_CART,
                    recv_request(2),MPI%iErr)
```

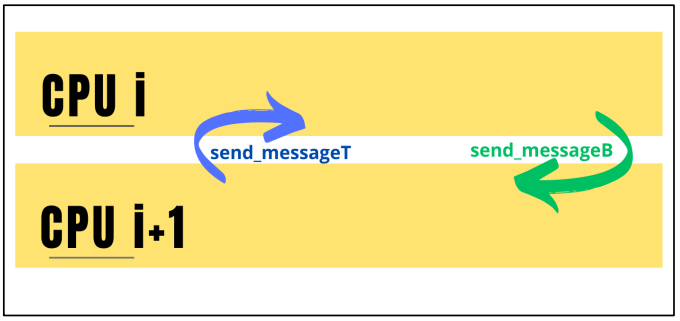
Per la creazione dei messaggi è stata utilizzata la sintassi vettoriale, send_messaggeT rappresenta il "vettore di celle" superiori della partizione che vanno dunque dalla riga di partenza di tale partizione ("iStart") per tutta la lunghezza orizzontale (numero di elementi per riga "nElem"); send_messageB rappresenta, invece, il "vettore di celle" inferiori della partizione che vanno dunque dalla ultima riga della partizione per tutta la lunghezza orizzontale (da iEnd per nElem).

Come da esempio grafico:



Esempio per singola partizione:
send_messageT
send_messageB

Il send_messageT verrà inviato dalla cpu sottostante a quella superiore, mentre il send_messageB verrà inviato dalla cpu sovrastante a quella sottostante (come da immagine).



Esempio comunicazione tra due cpu

I messaggi vengono inviati attraverso le CALL della funzione ***MPI_ISEND,*** la quale inizia un invio non bloccante (*no blocking communication*) che garantisce la possibilità di continuare la computazione interna ai blocchi nel mentre delle comunicazioni.

Successivamente i messaggi vengono ricevuti attraverso le Call della funzione ***MPI_IRECV***, la quale inizia una ricezione non bloccante (*no blocking communication*).
Si noti che i tag da inserire in caso di MPI_IRECV sono opposti a quelli dell MPI_ISEND.

- Aggiornamento di tutti gli elementi interni ("*update all internal (non MPI) elements*")

```
! 2) USEFUL TIME! In the meantime, we update all internal (non MPI) elements
CONTINUE
DO i = MPI%iStart+1, MPI%iEnd-1
    DO j = 2, MPI%nElem-1
        Tnew(i,j) = T(i,j) + kappa*dt/dx2 * (T(i+1,j) - 2.*T(i,j) + T(i-1,j)) +
                    kappa*dt/dy2 * (T(i,j+1) - 2.*T(i,j) + T(i,j-1))
    ENDDO
ENDDO
```

❖ Aggiornamento di tutti gli elementi “sui bordi” *(“update MPI boundary elements”)*

```
! 3) UPDATE MPI boundary elements
 !    aspettiamo che la comunicazione sia finita

 nMsg = 2
 CALL MPI_WAITALL(nMsg,send_request,send_status_list,MPI%iErr)
 CALL MPI_WAITALL(nMsg,recv_request,recv_status_list,MPI%iErr)

 ! AGGIORNAMENTO BORDI

 T(MPI%iStart-1,:) = recv_messageT
 T(MPI%iEnd+1,:)   = recv_messageB

 ! MANCANO SOLO I BORDI
     ! fisici (colonne)
 Tnew(:,1)         = TL
 Tnew(:,MPI%nElem) = TR
     ! non
 DO i = MPI%iStart, MPI%iEnd,(MPI%iEnd - MPI%iStart)
     DO j = 2, MPI%nElem-1
         Tnew(i,j) = T(i,j) + kappa*dt/dx2 * (T(i+1,j) - 2.*T(i,j) + T(i-1,j)) +
                     kappa*dt/dy2 * (T(i,j+1) - 2.*T(i,j) + T(i,j-1))
     ENDDO
 ENDDO
```

Attraverso le call alla funzione ***MPI_WAITALL,*** che si occupa di attendere il completamento di tutte le richieste MPI specificate, ci si assicura che tutti i messaggi siano stati inviati e ricevuti (per questo nel primo vi è stato inserito *send_status_list* e nel secondo *recv_status_list*) .
Quando si è sicuri di aver inviato e ricevuto tutte le risorse necessarie, si passa al completamento dell’ aggiornamento della nuova matrice:
Attraverso `T(MPI%iStart-1,:) = recv_messageT` e `T(MPI%iEnd+1,:) = recv_messageB` si aggiorna la percezione delle singole cpu riguardante gli aggiornamenti sui bordi effettuati dalle altre/adiacenti (sono gli effettivi messaggi ricevuti dalla cpu superiore ed inferiore);
mentre, per quanto affermato secondo la teoria, si possono già inserire i valori nella prima e ultima colonna sapendo che sono rispettivamente TL e TR, questo attraverso le due righe seguenti:

```
    Tnew(:,1)         = TL
    Tnew(:,MPI%nElem) = TR
```

Dopo tutto ciò si può quindi procedere alla conclusione dell’update delle celle rimanenti (vettori/righe iStart e di iEnd) attraverso il doppio ciclo annidato:

```
    DO i = MPI%iStart, MPI%iEnd,(MPI%iEnd - MPI%iStart)
        DO j = 2, MPI%nElem-1
            Tnew(i,j) = T(i,j) + kappa*dt/dx2 * (T(i+1,j) - 2.*T(i,j) + T(i-1,j)) +
                        kappa*dt/dy2 * (T(i,j+1) - 2.*T(i,j) + T(i,j-1))
        ENDDO
    ENDDO
```

A conclusione della "*business logic*" della computazione vi sono poi gli aggiornamenti per la gestione del tempo e della corrente soluzione.

```
        ! Update time and current solution
        time = time + dt                                          ! avanzo il tempo
        T(MPI%istart:MPI%iend,:) = Tnew(MPI%istart:MPI%iend,:) ! sovrascrivo la
                                                                  ! nuova soluzione
        timestep = n
        PRINT *,n," tempo: ",time !! per vedere se va avanti
    ENDDO ! CONCLUSIONE DEL CICLO INIZIATO AD INIZIO COMPUTAZIONE (circa)
```

⇨ **Raccoglimento dati e scrittura su file**

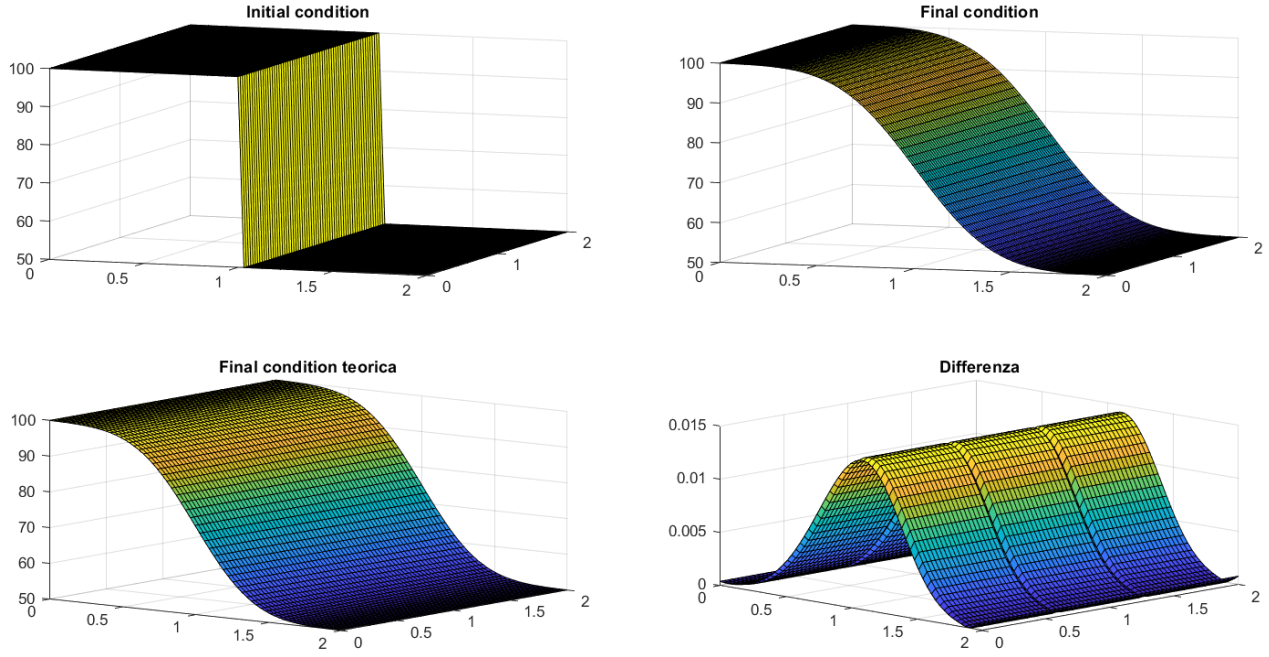
Per concludere e per poter analizzare poi i dati si collezionano i dati dai vari domini / dalle varie cpu, attraverso la CALL del comando sopra-spiegato ***MPI_ALLGATHER***, e si evoca, sulla sola Cpu con rank 0, la funzione/subroutine ***PlotOutputMPI*** per la conservazione dei dati all'interno di un apposito file che verrà poi analizzato attraverso uno script Matlab.
Si noti inoltre l'utilizzo sia in questo punto del codice che in precedenza (prima dell'inizio dalla computazione) della CALL ***DATE_AND_TIME***, la quale ottiene le informazioni su data e ora corrispondenti dall'orologio di sistema in tempo reale, utilizzato per poter svolgere poi ulteriori riflessioni in merito all'ottimizzazione di tale parallelizzazione.

```
 IF(MPI%myrank.EQ.0) WRITE(*,*) ' END OF THE COMPUTATION '

    ! colleziono i dati dai vari domini
    DO j = 1, IMAX
        CALL MPI_ALLGATHER( T(:,j), MPI%iMMax, MPI%AUTO_REAL, Tnew(:,j), MPI%iMMax,
                            MPI%AUTO_REAL, MPI_COMM_WORLD, MPI%iErr )
    ENDDO

    CALL DATE_AND_TIME(VALUES=timeF)

    timestep = ( timeF(6)*60000 + timeF(7)*1000 + timeF(8) ) - ( timeI(6)*60000 +
                 timeI(7)*1000 + timeI(8) )

    IF(MPI%myrank.EQ.0) CALL PlotOutputMPI(IMAX,x,y,Tnew,timestep,MPI%myrank,1,IMAX,MPI%nCpu)

    ! Empty memory
    DEALLOCATE( T, Tnew, x , y ) ! de-alloco tutto ciò che è stato precedentemente allocato

    CALL MPI_FINALIZE(MPI%iErr)
END PROGRAM  MAIN
```

Analisi Dati

Analisi dei dati del problema

Per il controllo della correttezza della soluzione trovata attraverso discretizzazione alle differenze finite della formula della diffusione del calore, è stata svolta un analisi utilizzando Matlab, attraverso il quale è possibile visionare i seguenti grafici (rappresentati del/nel dominio Ω = [0; 2] × [0; 2]):



Dove il primo, rappresentativo della condizione iniziale (titolato “Initial condition”), corrisponde alle richieste di progettazione a tempo $t = 0$ ($T(0)$), mentre il secondo (titolato come “Final condition”) rappresenta la condizione della matrice delle temperature evoluta fino al tempo finale $t_f = 0.05$.

Quest’ultimo può essere confrontato con quello della “Final Condition teorica” generato con i valori calcolati attraverso la seguente formula della temperatura esatta:

$$T_e = \frac{1}{2}(TR + TL) + \frac{1}{2} erf(\frac{x-1}{2\sqrt{kt}})(TR - Tl)$$

Oltre ad un confronto visivo, dove nell'esempio presentato risulta essere verosimile la correttezza, si è poi proceduti al controllo numerico attraverso il calcolo dell’errore relativo

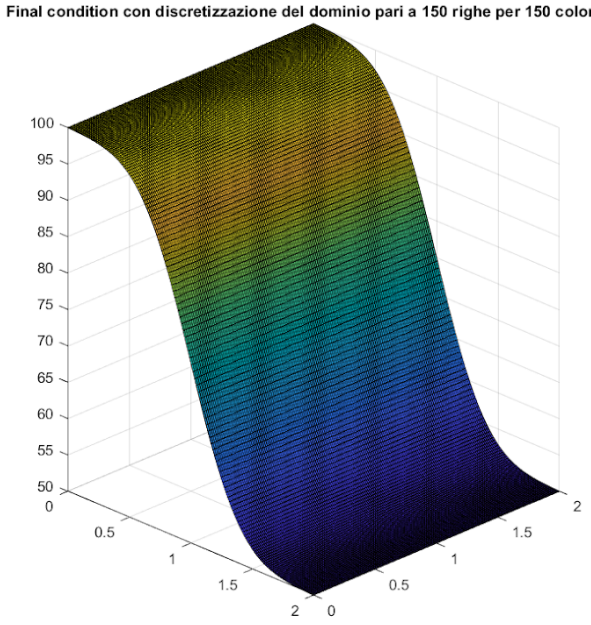
$$Er = \frac{|T_e - T_{calc}|}{|T_e|}$$

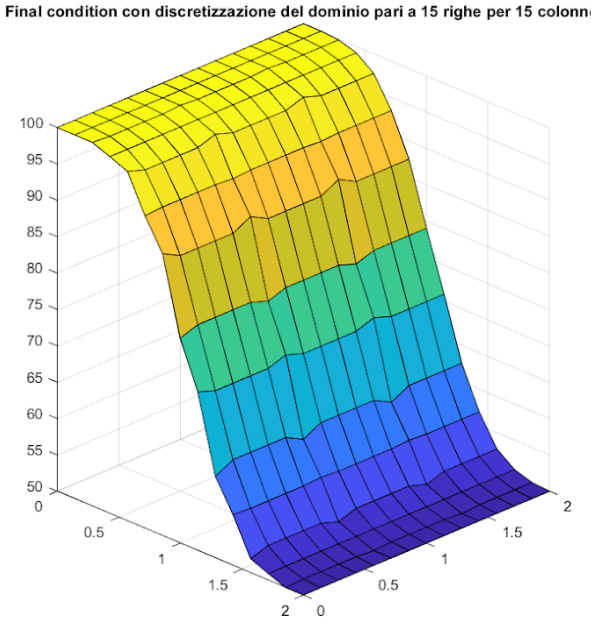
implementata nel seguente modo:

```
Tt = repmat(Te',N,1);
Tdiff = abs(Tt - T1)./abs(Tt);
```

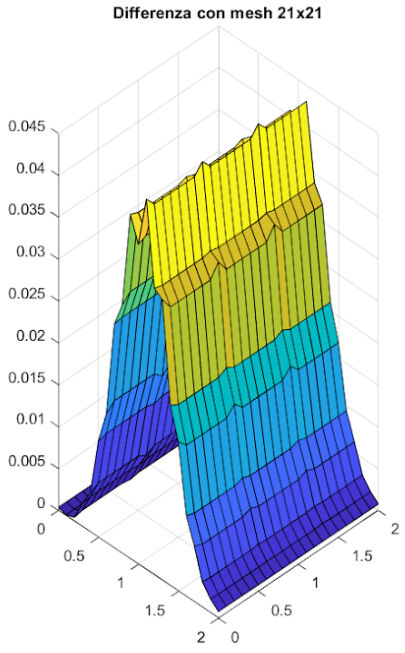
e si è calcolato/trovato su tali soluzioni il valore assoluto massimo (*Tdiff risulterà essere una matrice di valori*) trovando un risultato che varia al variare della discretizzazione (della “*mesh” (maglia)*). Tali valori di Tdiff sono stati poi rappresentati attraverso il grafico titolato “Differenze”.

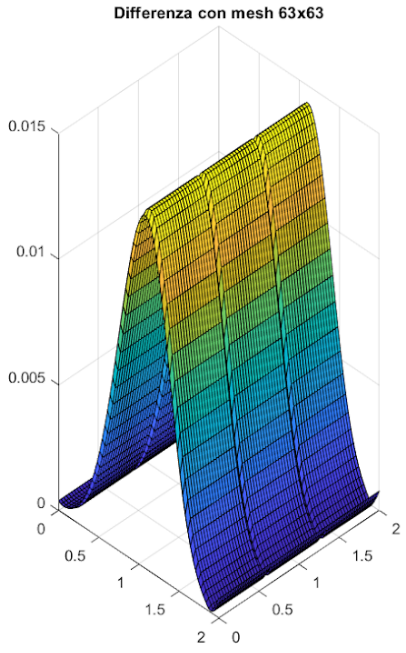
Altra considerazione interessante si svolge confrontando le soluzioni finali di più esecuzioni con valori di numero di celle verticali/orizzontali (*IMAX*) differenti, ovvero con valori di discretizzazione del dominio diversi (mesh diverse):

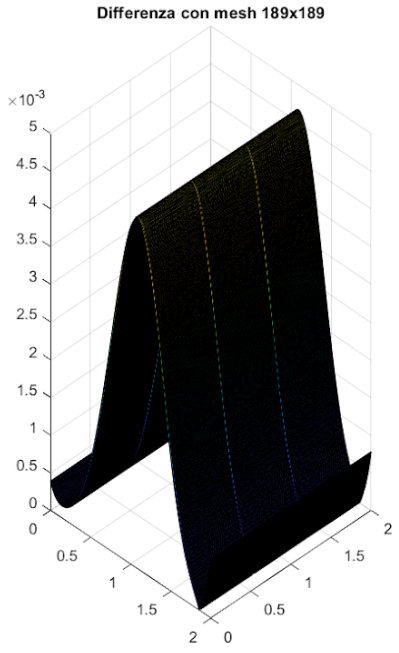




Come visibile dai due grafici, si può evincere un diretta proporzionalità tra accuratezza della soluzione e valori per la discretizzazione: maggiore è il numero di celle utilizzate per discretizzare il dominio, maggiore sarà l'accuratezza della soluzione (raffinando la griglia il valore dell'errore diminuisce). Questo è particolarmente visibile anche osservando i diversi grafici "Differenza" con mesh diverse, come di esempio:







Numericamente, infatti, con 3 esecuzioni diverse: la prima con valore per la discretizzazione di *IMAX* = 21 (mesh/discretizzazione = 21 x 21), la seconda con IMAX = 63 (mesh/discretizzazione = 63 x 63) e infine, una terza con valore per la caratterizzazione di discretizzazione quindi di *IMAX* = 189 (mesh/discretizzazione = 189 x 189) possiamo analizzare i 3 valori massimi assoluti di ognuno:

valore massimo assoluto con mesh 21 x 21 = $max(abs(E_{r\,21x21}))$ = 4.4217e-02 ≃ 0.0442

valore massimo assoluto con mesh 63 x 63 = $max(abs(E_{r\,63x63}))$ = 1.4300e-02 ≃ 0.0143

valore massimo assoluto con mesh 189 x 189 = $max(abs(E_{r\,189x189}))$ = 4.6674e-03 ≃ 0.0047

Il tutto conferma quindi l'ipotesi da teoria secondo cui "raffinando la griglia" il valore dell'errore diminuisce.

Analisi dei dati della Parallelizzazione del progetto

Dati utili per l'analisi della bontà di parallelizzazione sono le diverse misure di performance ("*Performance measure"*):

- **Speed-up**

  Misura la riduzione del tempo di calcolo (computational time t_p) che è stato ottenuto utilizzando un numero totale di p processori mantenendo fissa la dimensione del problema.

  Vi sono in generale due principali tipologie di speed-up: Absolute speed-up anche detto performance measure, e Relative speed-up anche detto scalability measure.
  Nel primo lo speed-up è misurata rispetto al miglior codice seriale con computational time t_{best}:

$$S(p) = \frac{t_{best}}{t(p)}$$

  Nel secondo invece (Relative speed-up) lo speed-up è misurata rispetto allo stesso codice seriale con p = 1:

$$S(p) = \frac{t_{(p=1)}}{t(p)}$$

- **Efficiency**

  L'efficienza è definita come il rapporto:

$$E(p) = \frac{S(p)}{p}$$

  Fondamentale il concetto secondo cui si deve ottimizzare il numero di processori ("Optimize the number of processors"):
  Secondo il punto di vista dello speed-up il numero ottimale di processori è quello che ci permette di raggiungere il punto di saturazione, mentre secondo il punto di vista dell'efficienza il numero ottimale di processori è quello con E(p) = 1 : p = 1.
  Nel punto di saturazione lo speed-up è massimo ma l'efficienza è bassa.
  Per trovare un equilibrio si introduce dunque la **Funzione di Kuck**:
  la funzione di Kuck K(p) viene utilizzata per misurare l'efficienza della parallelizzazione in termini di numero di processori p:

$$K(p) = S(p)\,E(p)$$

Da 5 diverse osservazioni con mesh costante (valore di IMAX = 400) e con valori di processori variabile si è giunti alle seguenti rilevazioni:

Num cpu: (p)	**Tempo di esecuzione:** ($t(p)$) *(millies)*	**Speed-up:** ($S(p) = \frac{t_{(p=1)}}{t(p)}$)	**Efficiency:** ($E(p) = \frac{S(p)}{p}$)	**Funzione di Kuck:** ($K(p) = S(p)\,E(p)$)
1	*55891*	/	/	/
2	25819	2.1647	1.0824	2.3430
4	13606	4.1078	1.0270	4.2185
5	12205	4.5794	0.9159	4.1941
8	10084	5.5425	0.6928	3.8400
10	14931	3.7433	0.3743	1.4012

Si deduce quindi quanto ipotizzabile a priori: all'aumentare del numero di cpu (per la parallelizzazione) aumenta il valore di speed-up ma si abbassa l'efficienza.
Importante considerare poi il valore/punto di saturazione; nelle rilevazioni è visibile come utilizzando 10 cpu il valore di speed-up non aumenta ulteriormente ma diminuisce rispetto ai precedenti, come detto da teoria sulla funzione di kuck (***Figura(5)***), che, in caso delle rilevazioni osservate, risulta essere verosimile a quanto studiato.
Di interesse vi sono anche le rilevazioni di efficienza in caso di 2 e 4 cpu nelle quali il valore risulta essere > 1 è questo il caso di superlinear speedup, raggiungibile, come studiato dalla teoria in due possibili circostanze:

- La dimensione del problema che appartiene ad un processore potrebbe essere ridotta fino al punto che può essere interamente memorizzato e gestito in/nella cache.
- In un sistema di memoria distribuita, se il numero di processori aumenta, aumenta anche la quantità totale di memoria. Pertanto, i dati e i risultati intermedi possono essere memorizzati, evitando quindi la necessità di elaborarli nuovamente. In tal modo, il numero di FLOP (FLoating point OPeration), cioè il numero di calcoli, può essere ridotto rispetto ad un'esecuzione su meno processori;

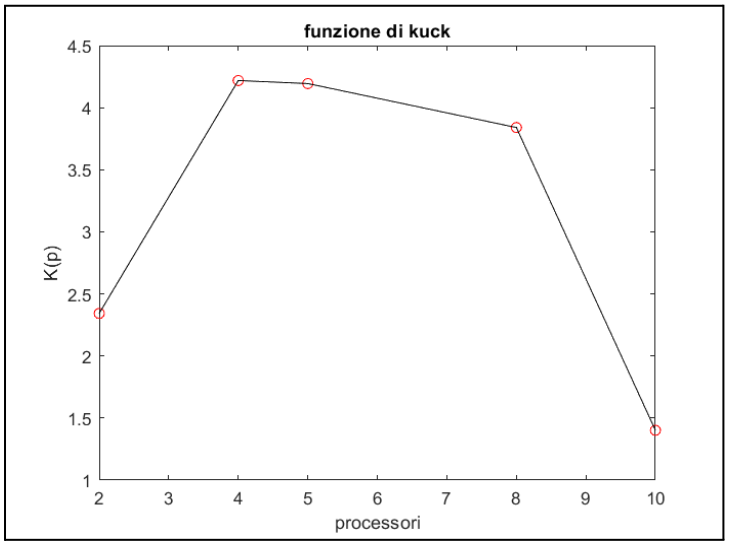


Figura (5)

Conclusioni

Dopo un'analisi dei dati approfondita si può giungere alle seguenti conclusioni:

La scrittura e l'applicazione del programma per l'equazione del calore su griglie strutturate sono state realizzate con successo dimostrato attraverso il confronto con la formula della soluzione esatta, e, dimostrando un aumento della precisione e accuratezza all’aumentare della mesh, questa validazione conferma quanto ipotizzato al livello teorico.
L'impiego delle direttive MPI ha consentito una parallelizzazione efficiente del programma, permettendo una distribuzione equa ed efficace del carico di lavoro. Questo approccio ha portato a una significativa riduzione dei tempi di esecuzione rispetto alla versione sequenziale e ha reso possibile quanto visto dal punto di vista teorico per quanto riguarda gli equilibri tra le misure di speed-up e efficiency, e per quanto riguarda il punto di saturazione e la funzione di kuck.

In conclusione, il progetto ha raggiunto con successo gli obiettivi prefissati.

Codici utilizzati oltre al Program main in fortran sono:

- Per la scrittura su file sia della situazione a tempo iniziale che a tempo finale t_0 t_{end}

```
SUBROUTINE PlotOutputMPI(IMAX,x,y,T,n,myrank,istart,iend,ncpu)
    IMPLICIT NONE
    INTEGER             :: IMAX, i, j, n, myrank, istart, iend, OutUnit,ncpu
    REAL                :: x(IMAX),y(IMAX), T(istart:iend,IMAX)
    CHARACTER(LEN=10)   :: citer, cmyrank
    CHARACTER(LEN=200)  :: IOFileName

    WRITE(citer,'(I4.4)') n
    WRITE(cmyrank,'(I4.4)') myrank
    IOFileName = 'Heat2D_output-'//TRIM(citer)//'-'//TRIM(cmyrank)//'.dat'

    OutUnit = 100+myrank
    OPEN(UNIT=OutUnit, FILE=IOFileName, STATUS='unknown', ACTION='write')

    WRITE(OutUnit,*) iend-istart+1
    WRITE(OutUnit,*) n
    WRITE(OutUnit,*) ncpu

    DO i = 1, IMAX
        WRITE(OutUnit,*) x(i)
    ENDDO

    DO i = 1, IMAX
        WRITE(OutUnit,*) y(i)
    ENDDO

    DO i = 1, IMAX
        DO j = 1, IMAX
            WRITE(OutUnit,*) T(i,j)
        ENDDO
    ENDDO

END SUBROUTINE PlotOutputMPI
```

- Per l’analisi dei dati e stampa dei diversi grafici utilizzati a scopo dimostrativo.

```
% Read Heat2D.f90 output file

clear all
close all
clc

% INITIAL CONDITION

% Open file
FileID = fopen('Heat2D_output-0000-0000.dat');
% Read array dimension
% Read data
N = fscanf(FileID,'%d \n',1);
TempoI = fscanf(FileID,'%d \n',1);
nCPU = fscanf(FileID,'%d \n',1);
x  = fscanf(FileID,'%f \n',N);
y  = fscanf(FileID,'%f \n',N);
T0 = zeros(N,N);

TR = 50
TL = 100
t = 0.05 % tempo finale

for k = 1:N
    T0(k,:) = fscanf(FileID,'%f \n',N)';
end



% FINAL SOLUTION
% Open file
FileID = fopen('Heat2D_output-0003-0000.dat'); %inserire nome file a t(end)
% Read array dimension
N  = fscanf(FileID,'%d \n',1);
TempoF = fscanf(FileID,'%d \n',1);
nCPU = fscanf(FileID,'%d \n',1);
x  = fscanf(FileID,'%f \n',N);
y  = fscanf(FileID,'%f \n',N);
T1 = zeros(N,N);

Te = 0.5 .* (TR+TL)+ 0.5 .* erf((x-1)/(2.*sqrt(t))) .* (TR-TL);
Tt = repmat(Te',N,1);



for k = 1:N
```

```
    T1(k,:) = fscanf(FileID,'%f \n',N)';
end

Tdiff = abs(Tt - T1)./abs(Tt);

fprintf("Numero di Cpu: %g  \n" ,nCPU)
fprintf("Tempo di esecuzione: %g \n" ,TempoF)
fprintf("Valore max abs: %g \n" ,max(max(abs(Tdiff))))

% Plot data
fg = figure(1);
fg.Position = [100 100 1300 650];


tiledlayout(2, 2)
nexttile
surf(x,y,T0)
view(45,22)
ylim([0 2])
xlim([0 2])
title('Initial condition')

nexttile
surf(x,y,T1)
ylim([0 2])
xlim([0 2])
view(45,22)

title('Final condition ')

nexttile
surf(x,y,Tt)
ylim([0 2])
xlim([0 2])
view(45,22)

title('Final condition teorica')

nexttile
surf(x,y,Tdiff)
ylim([0 2])
xlim([0 2])
view(45,22)

title('Differenza')
```