UNIVERSITÀ DEGLI STUDI DI PARMA

DIPARTIMENTO DI MATEMATICA E INFORMATICA
Corso di Laurea Triennale in Informatica

Tesi di Laurea

ANALISI DELLA RIPARTIZIONE DI CARICO IN UN AMBIENTE MPI

Load distribution analysis in an MPI framework

Candidato:
Francesco Fabiano

Relatore:
Prof. Alessandro Dal Palù

Anno Accademico 2015-2016

Alla mia famiglia

Indice

Elenco delle figure

Elenco degli algoritmi

Introduzione

L'utilizzo di sistemi di calcolo adeguati ha, da sempre, influenzato profondamente la ricerca scientifica. Di conseguenza una miglioria al sistema di calcolo permette di poter eseguire ricerche che prima di questa potevano sembrare impensabili.
Per questo motivo dopo che il computer si è affermato come lo strumento di calcolo moderno, essendo ormai alla base di tutti progetti scientifici, si è sempre cercato di migliorarne le capacità computazionali.
Nei primi anni '60 è stato quindi introdotto il concetto di *supercomputer* ovvero un computer con una capacità computazionale maggiore di qualsiasi altro computer generico del tempo.
Questa grande capacità di calcolo deriva dall'enorme quantità (rispetto ad un singolo computer) di processori associati. Infatti, un elevato numero di processori permette un alto livello di parallelizzazione: permette cioè di eseguire un gran numero di istruzioni contemporaneamente.
Ai giorni nostri i supercomputer sono essenzialmente cluster di singoli computer (spesso dotati di strumenti hardware per aumentarne la capacità di calcolo, come le GPU) che comunicano tra loro e nei quali viene suddiviso il programma che viene quindi risolto parallelamente nelle singole parti.
I sistemi di calcolo distribuito permettono di sviluppare progetti, soprattutto in ambito scientifico, che sarebbero altrimenti irrealizzabili.
Inoltre il *computer-clustering* permette di ottimizzare gli applicativi sviluppati inizialmente per architetture a singolo nodo, ottimizzandone i tempi di esecuzione e mettendo a disposizione un maggiore spazio di memorizzazione.
In particolare queste ottimizzazioni sono necessarie per lo svolgimento di tutta una categoria di applicativi che richiedono l'esecuzione di procedure "computazionalmente pesanti" in tempi brevi.

Il progetto *Shallow Water Equation on GPU*, sviluppato dalla collaborazione di vari dipartimenti dell'Università degli studi di Parma, ricade appunto nella categoria sopracitata.
Questo applicativo infatti, si propone di simulare il comportamento dei corsi d'acqua e di farlo in tempi ragionevoli.
Per il momento l'implementazione di *sweGPU* è basata solo su un'architettura a singolo nodo ma la versione per il calcolo distribuito è in fase di sviluppo.
L'obbiettivo di questa tesi è proprio quello di fornire una prima implementazione delle fasi iniziali per la distribuzione del programma in un ambiente multi-nodo. In particolare si è studiata la distribuzione delle informazioni, su cui lavora il progetto *sweGPU*, tra i vari nodi di calcolo e si è implementato un modello di comunicazione elementare tra i nodi stessi.

Il modello di comunicazione è stato realizzato seguendo lo standard de facto *MPI* (*Message Passing Interface*) che impone le regole da seguire nei modelli di comunicazione dei sistemi di calcolo distribuiti.

1 Shallow Water Equations on GPU

I danni legati ad eventi di inondazioni in Europa sono cresciuti negli ultimi anni e, secondo le stime, continueranno a farlo, arrivando addirittura, a provocare nel 2080 dalle 540.000 alle 950.000 vittime all'anno. Per questo motivo le nuove Direttive Europee hanno fatto sì che molti paesi istituissero ricerche per lo studio del rischio idrogeologico.
Shallow Water Equations on GPU (sweGPU) è un progetto, sviluppato dal dipartimento di Matematica e Informatica in collaborazione col dipartimento di Ingegneria Civile e Ambientale e Architettura dell'Università degli studi di Parma, che permette l'analisi e la valutazione del rischio idrogeologico tramite la simulazione numerica dei flussi d'acqua. Queste analisi vengono effettuate su aree geografiche di grandezze variabili arrivando a ricoprire anche intere province.
L'ampiezza dei territori che vengono considerati, e di conseguenza la grande quantità di informazioni che ne deriva, ha reso necessaria la rappresentazione del territorio tramite una tecnica di *multi risoluzione* in modo da regolare le informazioni ricavate da una zona di terreno, in base all'importanza che queste rivestono nella simulazione.

1.1 La GPU

Il progetto *sweGPU*, come si evince dal nome, viene eseguito su *GPU* (graphics processing units) così da sfruttarne la grande potenza di calcolo per poter eseguire simulazioni in tempi ragionevoli. L'esecuzione su GPU, infatti, permette una forte parallelizzazione delle istruzioni.
In particolare l'applicativo utilizza il linguaggio *CUDA-C* (NVIDIA®) che consente di sfruttare le GPU ad architettura *CUDA* (NVIDIA®).

1.1.1 L'architettura

Le caratteristiche hardware delle schede grafiche CUDA consentono di eseguire molte istruzioni delle stessa natura.
La loro struttura prevede una quantità scalabile di streaming multiprocessors (SM) ognuno dei quali contiene diversi *CUDA core*. I CUDA core sono le singole unità di calcolo che eseguono le istruzioni.
All'interno di un singolo SM i CUDA core, associati ad un *thread* diverso, eseguono concorrentemente lo stesso codice.
Per questo motivo l'architettura è stata definita *SIMT*, ovvero Single Instruction Multiple Thread. In particolare i thread sono divisi in gruppi denominati *blocchi CUDA* che a loro volta sono raggruppati in *griglie*. Sia per quanto

riguarda i *threads* che i *blocchi CUDA*, i raggruppamenti possono avvenire su una, due o tre dimensioni: *un blocco di threads* potrà quindi essere rappresentato come un vettore o come matrice bidimensionale o tridimensionale. Un'illustrazione è data in figura 1).

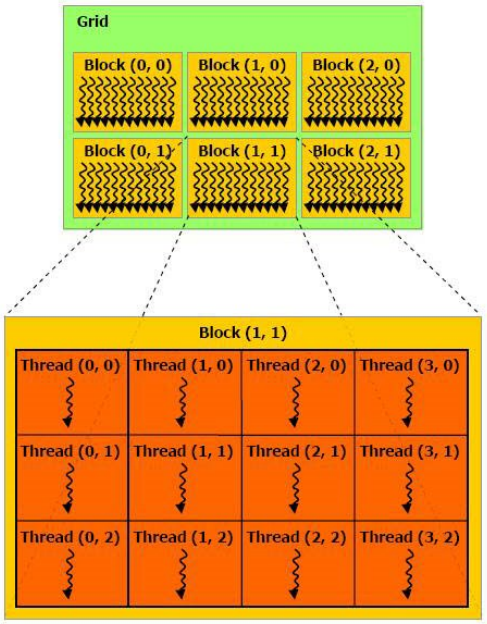


Figura 1: Rappresentazione di una griglia di blocchi CUDA

1.1.2 L'utilizzo della GPU nel progetto

Per eseguire il lavoro, *sweGPU* utilizza informazioni mappate in strutture dati chiamate *celle* e *blocchi di calcolo*, illustrate nel paragrafo (1.2).
In particolare ogni *cella di calcolo* è assegnata ad un CUDA Core ovvero ad un singolo thread e i *blocchi calcolo* sono insiemi di 8×8 o 16×16 *celle* e di conseguenza insiemi di 8×8 o 16×16 *threads*. Per questo ogni *blocco di calcolo* viene generalmente mappato su un singolo *blocco CUDA* [8].
I *blocchi di calcolo* dopo essere stati generati vengono inseriti in un piano cartesiano denominato *griglia di calcolo* (figura 2).

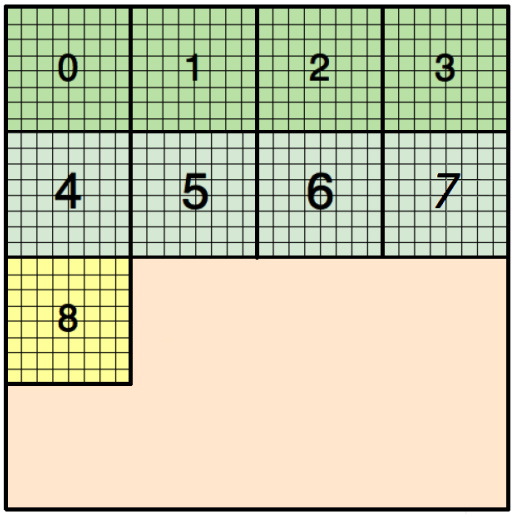


Figura 2: La disposizione dei blocchi di calcolo

Questa disposizione delle informazioni permette all'applicativo di eseguire i calcoli, che devono essere svolti per ogni cella, parallelamente.

1.2 Le Celle e i Blocchi

L'applicativo per la simulazione dei flussi d'acqua, basa la sua esecuzione sulle informazioni che vengono ricavate dalle mappe geografiche della zona interessata. Una volta elaborate, queste informazioni vengono inserite in appropriate strutture dati, mantenendo inalterate le caratteristiche del territorio utili per la simulazione.
La struttura dati elementare è la *cella*, che contiene diversi parametri che vengono utilizzati per descrivere un insieme di punti in coordinate geografiche reali dal punto di vista della conformazione del terreno e del livello d'acqua. Come detto nel paragrafo (1.1.2) ogni cella verrà mappata su un singolo thread della GPU.

Definizione 1.1. Si definisce *Cella* la trasposizione di una porzione di dominio di area quadrata (solitamente di 1 m^2), descritta da quantità scalari [1], in uno spazio di memoria del dominio logico.

A loro volta le celle con stessa risoluzione (che verrà trattata nel paragrafo 1.3) vengono mappate in gruppi da 8×8 o 16×16 su *griglie Cartesiane* chiamate *blocchi*.
La memorizzazione delle celle viene raggruppata in blocchi in modo da poter

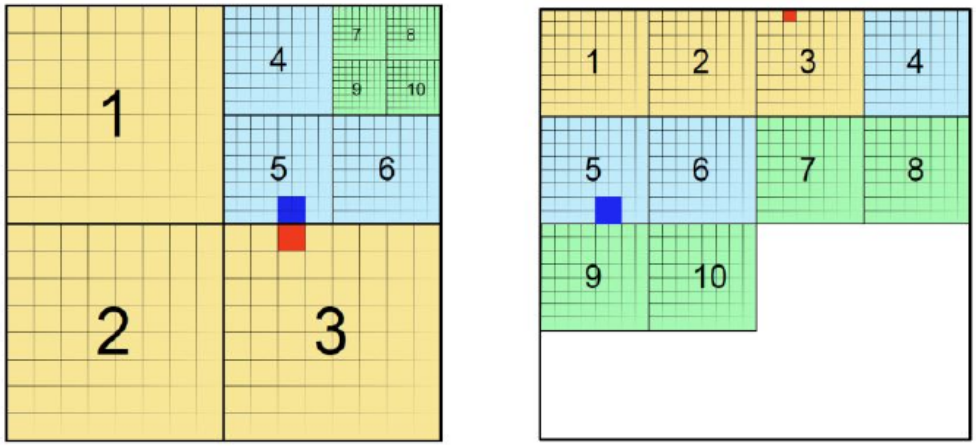
[1] Altezza dell'acqua, velocità dell'acqua e altezza del terreno

adattare le strutture dati dell'algoritmo all'architettura *CUDA*.
Inoltre la memorizzazione logica delle celle al'interno blocchi rispecchia la disposizione di queste all'interno del dominio reale.

Definizione 1.2. Un *Blocco* è una *griglia Cartesiana* di celle che mantiene inalterate le condizioni di adiacenza tra le *celle* in esso contenute.

Si definisce inoltre *BLOCKSIZE* il numero di celle per lato del blocco, che come detto prima può assumere come valore 8 o 16.

La disposizione logica dei blocchi, invece, non rispecchia quella fisica. Infatti i blocchi sono allocati sulla GPU nella griglia di calcolo non rispettando i criteri di adiacenza della disposizione reale (come si vede in figura 3). Il dominio reale viene quindi completamente mappato in celle che contengono tutte le informazioni sulle caratteristiche del territorio e che vengono a loro volta unite in blocchi; all'interno dei blocchi la disposizione logica delle celle rispetta quella fisica.



(a) Disposizione fisica dei blocchi (b) Disposizione logica dei blocchi

Figura 3: Confronto tra disposizione fisica e logica dei blocchi

1.3 La Multi Risoluzione

Dato che il dominio considerato dal programma potrebbe essere molto vasto e quindi le informazioni derivanti potrebbero essere molto pesanti, si è deciso di adottare un approccio a *multi risoluzione* per ridurre il carico di dati derivanti da zone di basso interesse per l'esecuzione dell'applicativo.
La multi risoluzione permette a celle logiche differenti di rappresentare aree di

terreno di ampiezza diversa, anche se, dal punto di vista della memorizzazione le celle hanno tutte lo stesso peso. Pertanto indipendentemente dal numero di punti in coordinate reali mappati in una cella rispetto ad un'altra entrambe vengono allocate nel programma usando la stessa quantità di memoria.
Ponendo il livello 1 come il livello a maggior risoluzione la relazione tra le aree rappresentate dalle celle ai vari livelli sarà:

- Livello 1 con cella di dimensione Δ_1.
- Livello 2 con cella di dimensione $\Delta_2 = 2 * \Delta_1$.

  $\vdots$

- Livello n con cella di dimensione $\Delta_n = 2 * \Delta_{n-1} = 2^{n-1} * \Delta_1$.

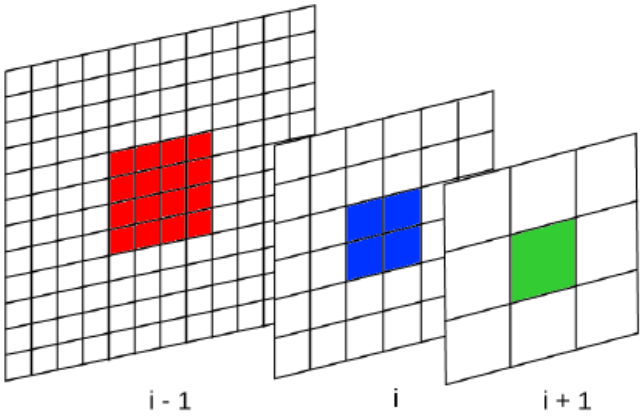


Figura 4: Rappresentazione della multi risoluzione tra celle

Questa differenza sul numero di punti in coordinate geografiche rappresentate da una stessa struttura dati, legata al concetto di multi risoluzione, permette di rappresentare zone di terreno ad alto interesse con una maggiore quantità di informazioni e con più precisione rispetto ad altre meno importanti per la simulazione.
In particolare la multi risoluzione si riflette anche sui blocchi: infatti questi contengono, come già detto, solo celle con stessa risoluzione. Si può quindi parlare di livelli di risoluzione diversa anche per i blocchi. Nonostante i blocchi contengano tutti lo stesso numero di celle e, di conseguenza, occupino la stessa quantità di memoria, l'ampiezza delle zone di terreno associata ad un blocco dipenderà direttamente dalla risoluzione delle celle che questo contiene

(la multi risoluzione tra blocchi rispetta lo schema mostrato in figura 4). È importante notare che il livello di risoluzione può differire al massimo di uno tra blocchi adiacenti.

Una matrice a multi risoluzione, composta da blocchi a differenti livelli di risoluzione, è di dimensioni 10-50 volte ridotte rispetto alle dimensioni che avrebbe se non fosse presente la multi risoluzione.
Si assuma un blocco con $BLOCKSIZE = 8$ contenente celle a livello di *risoluzione* 1. Ciascuna cella rappresenta un insieme di punti a distanza Δ_1 (solitamente un *metro*), si ricava quindi che il blocco stesso (considerato a sua volta di risoluzione 1) rappresenta 64 punti in coordinate geografiche.
Invece, se si assume un livello i di risoluzione più alto, un blocco (con stesso $BLOCKSIZE$) rappresenta un insieme di punti che hanno distanza Δ_i l'uno dal'altro, ovvero un insieme di $8 * 8 * \Delta_i$ punti in coordinate reali. Quindi con la stessa quantità di memoria viene rappresentata una zona di territorio più vasta.
In questo modo la multi risoluzione, cambia la topologia della rappresentazione logica del dominio, concentrando nei punti di interesse un alto numero di celle e di blocchi, e rendendo le zone ad alta risoluzione più ricche di informazioni ma più pesanti dal punto di vista computazionale (figura 5).

Figura 5: Multi risoluzione in un caso reale

1.4 Evoluzione di un passo di calcolo

Una volta che il dominio è completamente mappato sulle celle e sui blocchi, il programma può iniziare a simulare il comportamento dei corsi d'acqua all'interno del dominio.
A seconda delle condizioni al contorno si può infatti simulare: acqua che entra (controllata nel tempo), acqua che esce libera o vincolata, oppure il comportamento dell'acqua in un contenitore chiuso (ad esempio in una diga).
La simulazione avviene eseguendo vari calcoli su ogni cella del dominio logico

per determinare il livello dell'acqua in ogni punto in cui questa è presente. Questi calcoli però necessitano, oltre alle informazioni della cella stessa, delle grandezze conservate nelle celle vicine, che rappresentano cioè, aree di terreno adiacenti nel dominio concreto.
Le celle, perciò, devono sempre essere a conoscenza delle informazioni dei vari vicini per poter calcolare il loro stato durante ogni passo di computazione.
Per questo la disposizione delle celle nei blocchi garantisce a tutte le *celle interne*, ovvero non di bordo, di conoscere i propri vicini essendo, come nel paragrafo 1.3, la disposizione logica all'interno del blocco uguale a quella fisica.
Il problema sorge quando una *cella di bordo* deve comunicare con un suo vicino che non si trova all'interno del blocco come mostrato in figura 6. Per ovviare al problema si memorizzano, per ciascun blocco, le informazioni topologiche relative ai blocchi vicini (che contengono di conseguenza anche i vicini delle celle di bordo) in strutture dati appropriate. Tali informazioni consentono di ricostruire la posizione del blocco vicino e delle relative celle in memoria e quindi di poter accedere alle informazioni utili al calcolo.

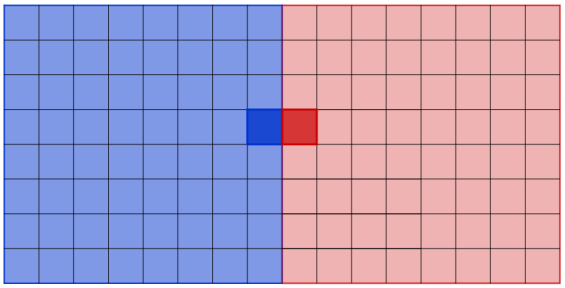


Figura 6: Caso di comunicazione tra celle di blocchi diversi

1.4.1 Le strutture dati

Di seguito si fornisce una breve introduzione alle strutture dati presenti nel programma che verranno poi utilizzate nel corso della tesi come strutture di supporto per eseguire varie operazioni. La struttura dati principale è *maps_*, che verrà analizzata unicamente nelle componenti utili a questa tesi e che contiene tutte le informazioni riguardanti il terreno e i legami tra la disposizione fisica e logica delle celle.

In particolare della struttura *maps_* verranno presi in considerazione tre array: *host_grid_level_multi*, *host_ofs_blocks* e *neigh*.
Il primo è un array di *unsigned char* che contiene la risoluzione di ogni blocco.
Il secondo rappresenta, con due *unsigned short*, le informazioni relative alla posizione del blocco nel dominio reale, tenendo conto anche della multi risoluzione.
L'ultimo invece contiene le informazioni dei vicini per ogni blocco.
Definito *tot_blocks* come il numero totale di blocchi presenti nel dominio la dimensione di ognuno dei primi due array è sempre uguale a *tot_blocks*, rappresentando questi le informazioni necessarie per ogni blocco.
Nel terzo invece, dovendo allocare per ogni blocco una cella dell'array per ognuno degli otto vicini[2], la dimensione non è *tot_blocks* ma $8 * tot\_blocks$.
Le informazioni dei vicini del blocco di indice i, rappresentate nella struttura dati *neigh_t* (definizione in listato 1), saranno salvate nelle celle dell'array che vanno da $(i * 8)$ a $(i * 8 + 7)$, come rappresentato in figura 7.

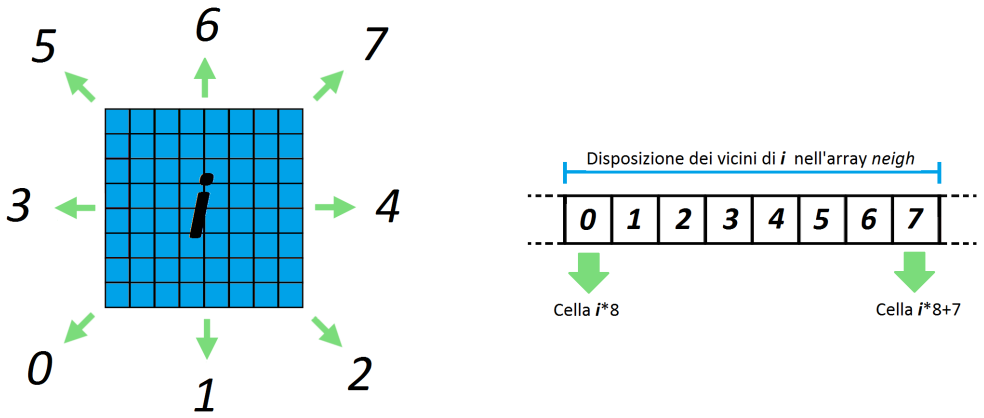


Figura 7: Disposizione dei vicini del blocco i nell'array *neigh*

```
struct {
  char lev;
  int n1;
  int n2;
} neigh_t;
```

Codice 1: Struttura dati *neig_t*

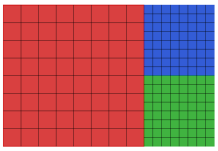
[2]Nord, Sud, Est, Ovest, Nord-Est, Nord-Ovest, Sud-Est, Sud-Ovest

Il campo *lev* indica la differenza di risoluzione tra il vicino e il blocco, questo valore può variare tra:

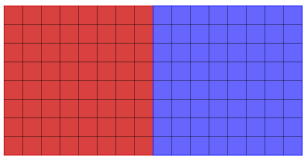
- **-1:** nel caso in cui il vicino abbia una risoluzione maggiore (figura 8a);
- **0:** nel caso in cui il vicino abbia la stessa risoluzione (figura 8b);
- **1:** nel caso in cui il vicino abbia una risoluzione minore (figura 8c).

Il campo $n1$ invece indica l'identificativo del blocco vicino (-1 se non esiste) che permette di ricavare la sua locazione in memoria.
Il campo $n2$ può contenere informazioni diverse a seconda del valore di *lev*, in particolare:

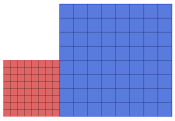
- se lev $= -1$ $n2$ conterrà l'identificativo del secondo vicino;
- se lev $= 0$ $n2$ non conterrà informazioni utili;
- se lev $= 1$ in $n2$ sono contenute informazioni aggiuntive sulla posizione del blocco vicino.

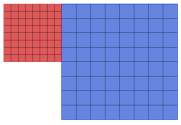


(a) *lev* = -1



(b) *lev* = 0





(c) *lev* = 1

Figura 8: Diversi casi di adiacenza

2 MPI

MPI cioè, Message Passing Interface, è uno *standard per l'interfaccia di una libreria per lo scambio di messaggi* [4].
Questo standard *de facto* è nato grazie al lavoro svolto su *MPI Forum*, un forum nel quale aziende che si occupano di "Parallel Computing", ricercatori nell'ambito informatico, sviluppatori di librerie e di applicazioni hanno collaborato per raggiungere questo risultato.
Come detto *MPI* è una specifica e non una libreria e si propone quindi di imporre regole e di dare istruzioni su quello che una libreria per lo scambio di messaggi deve fare. Inoltre, non essendo *MPI* un linguaggio di programmazione tutte le operazioni sono espresse come funzioni, subroutine, o metodi a seconda del *binding* con i linguaggi che fanno parte dello standard.
L'obbiettivo di *MPI*, quindi, è quello di stabilire uno standard per il *message-passing portabile*, *flessibile*, *pratico* ed *efficiente*.
MPI è rivolto principalmente al modello di *message-passing* nella *programmazione parallela*, in cui i dati vengono spostati dallo spazio di indirizzi di un processo a quello di un altro tramite operazioni cooperative su ciascun processo.
In particolare in un ambiente di comunicazione a memoria distribuita, in cui le astrazioni di livello superiore sono costruite sulla base del livello inferiore, ovvero sulle routine di *message-passing*, i benefici della standardizzazione sono particolarmente evidenti.
Inoltre, la definizione di uno standard *message-passing*, come quella proposta da MPI, fornisce un insieme di funzioni base ben definito, efficientemente implementabili dai vendor interessati al "Parallel Computing", o per le quali sia possibile fornire supporto hardware, rafforzando in tal modo la scalabilità.
L'obiettivo di *MPI* è quindi quello di sviluppare uno standard ampiamente utilizzato per la scrittura di programmi di *message-passing*.

2.1 La struttura di un programma MPI

Un programma *MPI* è un programma che, seguendo le procedure introdotte dallo standard, esegue operazioni di *message-passing* su architetture di calcolo parallelo.
D'ora in avanti si affronteranno programmi *MPI* in *C/C++* che utilizzano la libreria *OpenMPI* [3] per l'implementazione delle funzioni, ma le considerazioni fatte sono comunque valide per le relative funzioni legate al *Fortran* e per implementazioni di altre librerie.
Ogni programma *MPI* deve utilizzare la direttiva del preprocessore: `#include "mpi.h"`

mpi.h infatti contiene le definizioni, le macro e i prototipi delle funzioni necessari per compilare un programma *MPI*.
Prima di ogni altra funzione *MPI* bisogna invocare la funzione *MPI_Init()*; questa funzione deve essere chiamata una sola volta. La *MPI_Init()* prende come argomenti i puntatori ai parametri del *main* e permette al sistema di preparare l'ambiente per l'esecuzione delle varie routine *MPI* che possono essere usate.
Una volta che il programma ha finito di utilizzare le funzioni *MPI* deve chiamare la *MPI_Finalize()* che permette di "pulire" ogni lavoro non terminato dalle varie funzioni *MPI* invocate precedentemente.
Nel listato 2 è mostrata la generica struttura di un programma *MPI* [7].

```
    ⋮

#include "mpi.h"

    ⋮

main(int argc, char** argv){

        ⋮

    //Nessuna chiamata a funzioni MPI prima di questa
    MPI_Init(&argc, &argv);

        ⋮

    MPI_Finalize();
    //Nessuna chiamata a funzioni MPI dopo questa

        ⋮
}
```

Codice 2: Programma MPI generico

2.2 Le principali componenti

In MPI la comunicazione fra i processi si basa sullo scambio di messaggi.
MPI quindi definisce le strutture e le informazioni che servono per realizzare uno scambio di messaggi tra processi.
In particolare lo standard descrive:

- In che modo rappresentare il dato, da inviare nel messaggio, durante la comunicazione;

- Come specificare i processi protagonisti dello scambio del messaggio;
- I vari modi in cui può essere implementata la comunicazione.

2.2.1 Definizione dei dati in MPI

Per poter inviare un dato attraverso un messaggio, che poi verrà scambiato tra i processi, occorre definire alcune informazioni fondamentali legate al dato stesso; in particolare è necessario definire:

- **Il buffer**, ovvero l'indirizzo di memoria che contiene i dati da inviare;
- **Il numero di elementi** da inviare nel messaggio;
- **Il tipo di dato** degli elementi.

Il tipo di dato può rispecchiare uno dei tipi di dato più comuni del linguaggio *C* (figura 9) oppure può essere un tipo di dato definito da utente in quanto MPI fornisce le primitive per costruire strutture dati partendo da altre già esistenti.

tipo MPI	Tipo C	Byte
MPI_CHAR	signed char	1
MPI_SHORT	signed short int	2
MPI_INT	signed int	4
MPI_LONG	signed long int	4
MPI_UNSIGNED_CHAR	unsigned char	1
MPI_UNSIGNED_SHORT	unsigned short	1
MPI_UNSIGNED	unsigned int	4
MPI_UNSIGNED_LONG	unsigned long int	4
MPI_FLOAT	float	4
MPI_DOUBLE	double	8
MPI_LONG_DOUBLE	long double	12
MPI_BYTE	8 binary digit	1
MPI_PACKED	packed with MPI_Pack() unpacked with MPI_Unpack()	

Figura 9: Tipi di dato in C e in MPI

2.2.2 Specificazione dei processi coinvolti nella comunicazione

Durante l'invio di un messaggio è necessario specificare il destinatario e garantire che le informazioni che lo identificano siano univoche per non compromettere la comunicazione.
A questo scopo durante l'invio di un messaggio MPI bisogna sempre indicare:

- **Il communicator** che rappresenta uno specifico contesto a cui appartiene un gruppo di processi (il *communicator* di default si chiama *MPI_COMM_WORLD* e racchiude tutti i processi disponibili).

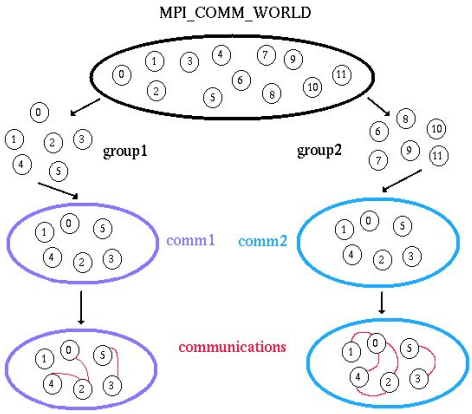


Figura 10: Il *communicator MPI_COMM_WORLD* e due sottogruppi

- **Il rank**, che è un identificativo univoco assegnato ad ogni processo, del processo destinatario;
- **Il tag** che permette, grazie ad un intero definito da utente, di differenziare ulteriormente il messaggio.

2.2.3 Le implementazioni delle comunicazioni punto a punto

MPI prevede diverse modalità per gestire la comunicazione tra i processi.

La comunicazione standard

La modalità standard è *bloccante e asincrona* si effettua utilizzando la funzioni *MPI_Send* per l'invio e *MPI_Recv* per la ricezione.

```
MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm)

MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source
    , int tag, MPI_Comm comm, MPI_Status *status)
```

Codice 3: Intestazioni delle primitive di comunicazione MPI

In entrambe le funzioni i parametri *buf*, *count* e *datatype* servono per fornire le informazioni sul contenuto nel messaggio come spiegato nel paragrafo

2.2.1.
Il parametro *dest* nella funzione di invio e il parametro *source* in quella di ricezione indicano rispettivamente il *rank* del processo che deve ricevere e di quello che deve inviare i dati. Oltre al *rank* dei processi coinvolti, queste funzioni prendono in argomento anche le altre informazioni illustrate nel paragrafo 2.2.2 relative al destinatario; in particolare richiedono: il *tag* del messaggio e il *communicator(comm)* nel quale si svolge la comunicazione.
La funzione *MPI_Recv* grazie a *status* può ottenere informazioni aggiuntive sul messaggio.
Essendo la *MPI_Send* bloccante, ritorna solo quando i dati sono stati copiati nel buffer di sistema del destinatario e quindi, per esempio, se il buffer è pieno, la funzione resta bloccata in attesa che si liberi. Una volta che la funzione è ritornata, non c'è alcuna garanzia che il processo destinatario abbia ricevuto il messaggio.
A sua volta la *MPI_Recv*, essendo bloccante ritorna solamente quando i dati sono stati copiati dal buffer di sistema all'indirizzo di memoria specificato.
Lo schema di una comunicazione bloccante è mostrato in figura 11.

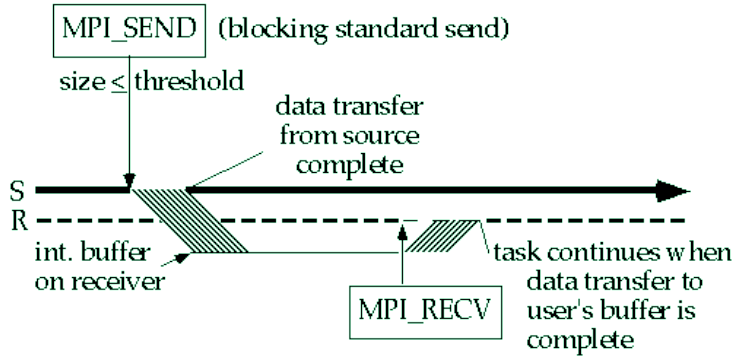


Figura 11: Schema di una comunicazione bloccante

Altri tipi di comunicazione punto a punto

Oltre alla modalità standard (asincrona) MPI definisce altre modalità, *bloccanti e non*, di comunicazione per l'invio di un messaggio.
Le seguenti modalità di invio sono *bloccanti* e richiedono come parametri gli stessi richiesti dalla funzione primitiva MPI_Send:

- MPI_Ssend $(\cdots)$: questa funzione ritorna solo quando il destinatario ha iniziato a ricevere;
- MPI_Bsend $(\cdots)$: questa funzione ritorna quando i dati sono copiati in un buffer specifico;

- MPI_Rsend (· · ·): da utilizzare solo quando si è certi che il destinatario sia in ascolto.

Esistono anche modalità di comunicazione *non bloccanti* (di cui viene mostrato lo schema in figura 12) che vengono descritte dalle seguenti funzioni nel codice 4.

```
MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm, MPI_Request &request)

MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int
   source, int tag, MPI_Comm comm, MPI_Request &request)
```

Codice 4: Intestazioni delle comunicazioni non bloccanti MPI

Le funzioni sopracitate ritornano immediatamente, ma la memoria all'indirizzo *buf* non può essere scritta o letta finché non si è certi della conclusione dell'operazione.
Il campo *request* delle due funzioni è un *handle* che permette di verificare e interagire con lo status della richiesta grazie ad opportune funzioni (MPI_Test e MPI_Wait).

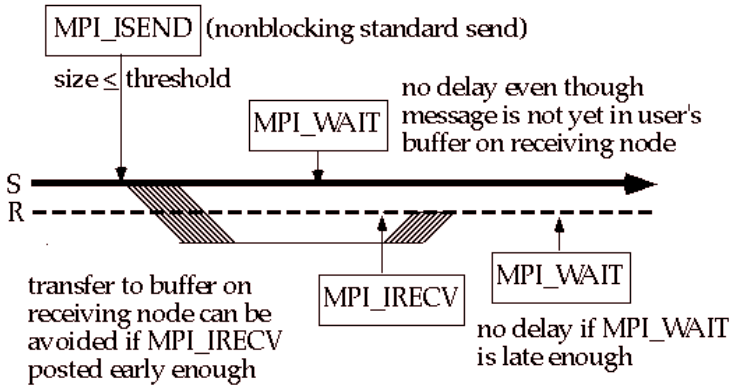


Figura 12: Schema di una comunicazione non bloccante

2.2.4 Le comunicazioni collettive

Un ultimo tipo di comunicazione messo a disposizione da MPI è la comunicazione collettiva. Una comunicazione collettiva è una comunicazione che riguarda tutti i processi presenti all'interno di uno stesso *communicator* che non interferisce con le comunicazioni *punto-punto* e che non può essere bloccante.
MPI mette a disposizione diverse funzioni per gestire vari tipi di comunicazione collettiva (figura 13):

- **MPI_Bcast** per permettere ad un processo di inviare lo stesso messaggio a tutti gli altri;
- **MPI_Reduce** per permettere ad un processo di suddividere un messaggio tra tutti gli altri;
- **MPI_Scatter** per far si che tutti i processi inviino un messaggio ad uno stesso nodo;
- **MPI_Gather** per far si che tutti i processi cooperino per calcolare un risultato da inviare ad uno stesso processo.

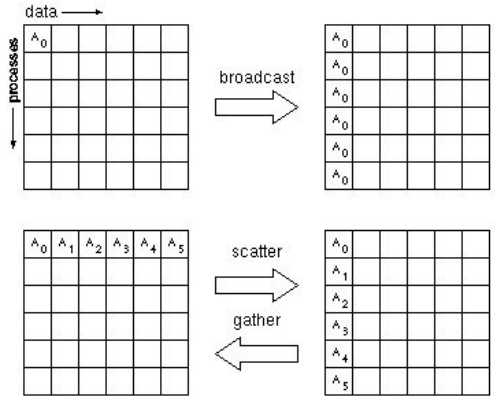


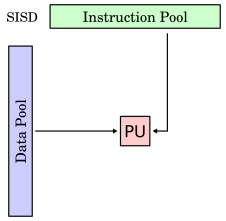
Figura 13: Rappresentazione di alcune comunicazioni collettive

2.3 Il modello hardware per un ambiente MPI

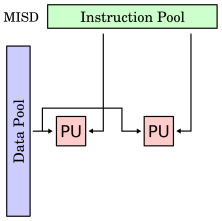
Come detto MPI è un'interfaccia per le librerie che si devono occupare di message-passing.
Questo scambio di informazioni è dunque legato ad un ambiente di calcolo parallelo e a una relativa architettura hardware che consente di eseguire uno stesso programma parallelamente su più processori o addirittura su più nodi.
Le varie configurazioni che può avere un ambiente di calcolo parallelo, secondo la tassonomia di Flynn (1986), derivano dalla configurazione hardware e software di due componenti principali del sistema: il *flusso di istruzioni (processi)* e il *flusso di dati*. Le varie configurazioni previste da questa suddivisione sono:

- **SISD** (*Single Instruction Single Data*): Architetture dei sistemi seriali, senza quindi, nessun grado di parallelismo (schema in figura 14a);

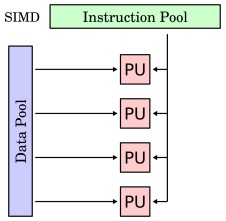
- **SIMD** (*Single Instruction Multiple Data*): Architetture composte da molte unità di elaborazione che eseguono contemporaneamente la stessa istruzione ma lavorano su insiemi di dati diversi (schema in figura 14c);
- **MISD** (*Multiple Instruction Single Data*): Architetture in cui più flussi di istruzioni lavorano contemporaneamente su un unico flusso di dati (schema in figura 14b). Questa tipologia di architettura non è stata praticamente mai adottata;
- **MIMD** (*Multiple Instruction Multiple Data*): Architetture in cui più istruzioni vengono eseguite contemporaneamente su più dati diversi (schema in figura 14d). Sotto questa classificazione ricadono i *cluster* di computer e di conseguenza l'ambiente sul quale è stato basato lo sviluppo di questa tesi.



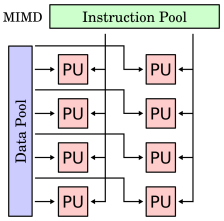
(a) Architettura SISD



(b) Architettura MISD



(c) Architettura SIMD

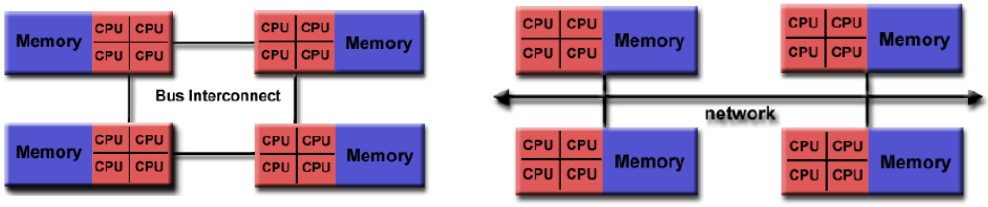


(d) Architettura MIMD

Figura 14: Le varie architetture della Tassonomia di Flynn

In particolare tra i sistemi MIMD possono distinguersi le architetture a *memoria condivisa* (figura 15b) o quelle a *memoria distribuita* (figura 15a). Nella prima configurazione la memoria è condivisa fisicamente (UMA) o solo logicamente assegnando indirizzi globali alle vari memorie locali (NUMA), permettendo ai vari nodi di lavorare sugli stessi spazi di memoria, evitando quindi di dover gestire le comunicazioni.

Nell'architettura a memoria distribuita, ogni nodo possiede una propria memoria locale che non fa parte dello spazio di indirizzamento degli altri processori e, per regolare l'accesso di un nodo alle risorse che non sono locali, bisogna utilizzare protocolli di comunicazione e scambio dati come, per esempio, quelli descritti da MPI.



(a) Memoria condivisa (b) Memoria distribuita

Figura 15: Le due diverse architetture MIMD

3 I Grafi e il loro Partizionamento

Nel capitolo 1 di questa tesi si è discusso di come le informazioni del territorio, su cui si vuole svolgere la simulazione, siano completamente mappate su strutture dati specifiche del progetto (celle e blocchi).
L'unione dei blocchi (composti a loro volta dalle celle), crea il dominio logico sul quale verranno svolte le operazioni di *partizionamento* spiegate nel capitolo 4.
Prima di descrivere le operazioni svolte sul dominio è necessario introdurre il concetto di *grafo* e alcune delle operazioni legate ad esso. Il grafo è la struttura di base su cui si basa la rappresentazione del dominio e la relativa suddivisione.

3.1 Introduzione ai grafi

Il *grafo* è una struttura matematica che ha trovato ampio utilizzo nell'ambito dell'informatica.
Concettualmente, un grafo, è formato da *vertici* e da *archi*, che connettono una coppia di vertici tra loro [5].

Definizione 3.1. Si dice grafo G una coppia *(V,E)* dove dove *V(G)* è l'insieme dei *vertici* (o nodi) ed *E(G)* è l'insieme degli *archi* (che può anche essere vuoto).

Un grafo semplice G consiste quindi in un insieme finito e non-vuoto *V(G)* di vertici e in un insieme finito *E(G)* di coppie distinte e non ordinate di elementi di *V(G)*.

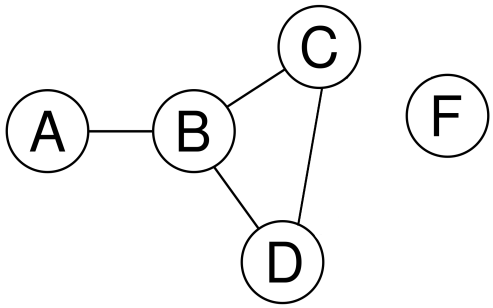


Figura 16: Un semplice esempio di grafo

3.1.1 I vari tipi di grafo

Quando si parla di grafi bisogna però distinguere di quale tipologia si sta trattando.

Esistono infatti vari tipi di grafo ognuno con caratteristiche differenti [9]. La prima distinzione si ha tra grafi *diretti* e *indiretti*:

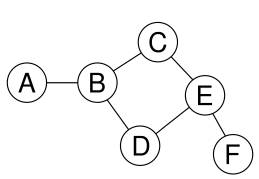
Grafi Indiretti
Nei grafi indiretti la relazione di adiacenza è simmetrica: l'arco che connette il nodo u al nodo v è lo stesso che connette il v al nodo u. Pertanto l'ordine dei vertici nella coppia che compone l'arco non ha importanza (figura 17a).

Grafi Diretti
In un grafo diretto (o orientato) gli elementi di *E(G)* sono coppie *ordinate* e di conseguenza, gli archi sono associati ad una direzione (figura 17b). Per esempio l'arco dal vertice u al nodo v, indicato come (u, v), è diverso dall'arco di direzione opposta (v, u).

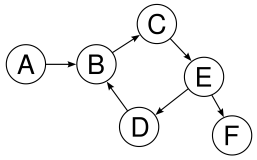
Definizione 3.2. Si definisce *ciclo* un cammino lungo archi orientati che parte da un vertice ed arriva allo stesso.

Un'altra caratterizzazione dei grafi tiene conto della presenza o meno di un ciclo: si distingue tra **Grafi diretti Ciclici** (figura 17b) nel caso in cui il ciclo sia presente e **Grafi diretti Aciclici (DAG)** (figura 17c) qualora non vi sia.

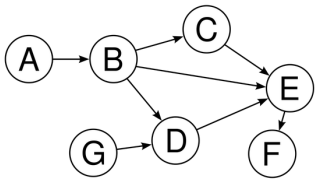
Infine un'ultima caratteristica che possono avere tutti i tipi di grafo sopra elencati, è l'associazione o meno di un valore, detto *peso*, ad ogni arco appartenete al grafo.
Un **Grafo Pesato** (figura 17d) G è quindi un grafo dove ogni elemento dell'insieme *E(G)* ha associato un valore numerico. Questa associazione permette di eseguire operazioni sui vari archi tramite il confronto dei pesi, come ad esempio, individuare un cammino minimo tra due vertici.



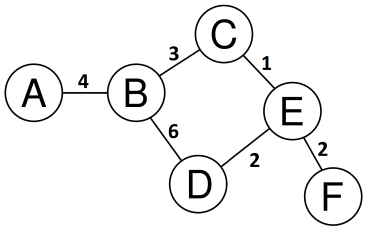
(a) Esempio di grafo indiretto

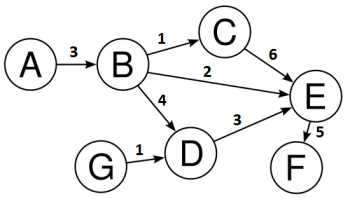


(b) Esempio di grafo diretto che contiene un ciclo



(c) Esempio di DAG (grafo diretto aciclico)





(d) Due esempi di grafi pesati

Figura 17: I diversi tipi di grafo

3.2 La rappresentazione dei grafi

I grafi sono utilizzati spesso per descrivere entità fisiche all'interno della memoria di un computer. Nel caso di questa tesi viene utilizzato un grafo orientato per descrivere il territorio nel quale si svolge la simulazione.
Esistono due metodi standard per rappresentare un grafo $G = (V, E)$: attraverso una *collezione di liste di adiacenza* e attraverso una *matrice di adiacenza*.

3.2.1 Rappresentazione con Matrice di adiacenza

Per questa rappresentazione si suppone che i vertici del grafo $G = (V, E)$ siano numerati da 1 a $|V|$ in modo arbitrario. La rappresentazione tramite matrice di adiacenza del grafo consiste in una matrice di dimensioni $|V| \times |V|$ in cui vale che l'elemento m_{ij} rispetti la seguente condizione:

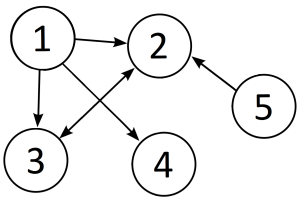
$$m_{ij} = \begin{cases} 1 & \text{se } (i,j) \in E \\ 0 & \text{altrimenti} \end{cases}$$

La matrice di adiacenza, mostrata in figura 18b occupa una quantità di memoria quadratica rispetto al numero di vertici indipendentemente dal numero degli archi del grafo.
La rappresentazione di un grafo pesato tramite matrice di adiacenza, sostituisce il valore 1 che indica presenza dell'arco tra i nodi con il peso dell'arco

stesso e utilizza un valore speciale (come 0, ∞ o *NIL*) per il caso in cui l'arco non sia presente.
La matrice di adiacenza è consigliata quando il grafo è un *grafo denso* ovvero $|E|$ si avvicina a $|V|^2$ o quando c'è bisogno di constatare rapidamente se esiste un arco specifico.

3.2.2 Rappresentazione con Liste di adiacenza

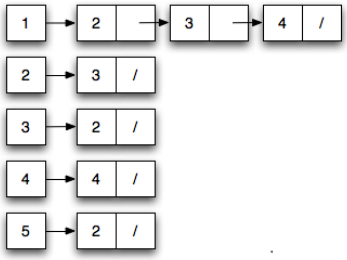
La rappresentazione tramite liste di adiacenza di un grafo $G = (V, E)$ consiste in un array (indicato d'ora in avanti con Adj) di $|V|$ liste, una per ogni vertice. Per ogni nodo quindi si crea una lista che contiene tutti i vertici adiacenti. Un esempio di lista di adiacenza è mostrato in figura 18c.
La memoria richiesta da questa rappresentazione dipende dal tipo di grafo su cui si basa: infatti la somma totale delle lunghezze delle liste è pari a $|E|$ se il grafo è orientato, e $2|E|$ se il grafo non lo è. La rappresentazione in lista di adiacenza conserva comunque, in entrambi i casi, l'interessante proprietà di occupare una quantità di memoria massima $\Theta(E)$.
Questa rappresentazione è consigliata per la maggioranza dei grafi perchè permette di rappresentare in modo compatto i *grafi sparsi*, quelli in cui $|E|$ è molto più piccolo di $|V|^2$. Inoltre, tramite le liste di adiacenza, è possibile associare facilmente ad ogni arco un peso nel caso in cui si voglia rappresentare un grafo pesato.
L'unico svantaggio è rappresentato dall'impossibilità di determinare velocemente la presenza di uno specificato arco, cosa che invece è possibile fare nelle matrici da adiacenza al costo però di un utilizzo maggiore di memoria.



(a) Il grafo da rappresentare

	1	2	3	4	5
1	0	1	1	1	0
2	0	0	1	0	0
3	0	1	0	0	0
4	0	0	0	1	0
5	0	1	0	0	0

(b) Rappresentazione tramite matrice di adiacenza



(c) Rappresentazione tramite lista di adiacenza

Figura 18: Rappresentazione di uno stesso grafo tramite lista e matrice di adiacenza

3.3 Il Partizionamento

Il *Graph-Partitioning* è un processo di pre-computazione fondamentale per la maggior parte delle applicazioni su larga scala che sono implementate da piattaforme di calcolo parallelo. Il problema del partizionamento definito su un grafo G sta nel suddividere G in componenti più piccole con specifiche proprietà. Per esempio, una partizione a k divide l'insieme dei vertici in k componenti più piccole.
Una partizione è considerata buona se il numero di archi che intercorre tra le componenti separate è piccolo.
Costruire *un grafo a partizione uniforme* è un problema di partizionamento che consiste appunto nel dividere il grafo in componenti della stessa grandezza (in termini di nodi o di peso di questi) cercando di minimizzare le comunicazioni tra le varie componenti.

3.4 Alberi di copertura Minimi

Definizione 3.3. Dato un grafo $G = (V, E)$ non orientato e connesso, un *albero di copertura* di G è un sottoinsieme $T \subseteq E$ tale che il sottografo S =

(V, T) è un albero non orientato connesso ed aciclico.

Si osserva quindi che un albero di copertura è un sottografo di G che contiene tutti i suoi vertici ed è sia aciclico che connesso.
Sia definita una funzione peso $\omega : E \to \mathbb{R}$; definiamo il costo di un albero di copertura come la somma dei pesi degli archi che lo compongono:

$$\omega(T) = \sum_{e \in T} \omega(e) \tag{1}$$

Definizione 3.4. Dato un grafo $G = (V, E)$ non orientato, connesso e pesato sugli archi, un *albero di copertura minimo* di G è un albero di copertura di G di costo minimo. Si abbrevierà il nome in *MST*, dall'inglese Minimum Spanning Tree.

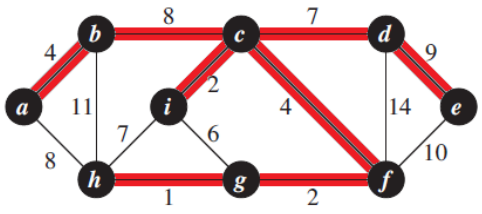


Figura 19: Esempio di un albero di copertura minimo (MST)

3.4.1 Il teorema Fondamentale degli alberi a copertura minimi [5]

Definizione 3.5. Un *taglio* $(S, V \backslash S)$ di un grafo non orientato $G = (V, E)$ è una partizione di V.

Definizione 3.6. Un *arco attraversa* il taglio $(S, V \backslash S)$ se uno dei suoi estremi si trova in S e l'altro in $V \backslash S$.

Definizione 3.7. Un *taglio rispetta* un insieme A di archi se nessun arco di A attraversa il taglio.

Definizione 3.8. Un arco che attraversa un taglio si dice *leggero* se ha peso pari al minimo tra i pesi di tutti gli archi che attraversano tale taglio.

Definizione 3.9. Sia A sottoinsieme di un qualche albero di copertura minimo; un arco si dice *sicuro* per A se può essere aggiunto ad A e quest'ultimo continua ad essere sottoinsieme di un albero di copertura minimo.

Siccome le strategie per la ricerca di un albero a connessione minima sono strategie *greedy*, è importante avere un metodo per verificare che gli archi selezionati passo a passo siano *archi validi*. Il teorema di seguito enunciato permette di verificare questa richiesta.

Teorema 3.1. *(Fondamentale). Sia $G = (V, E, \omega)$ un grafo non orientato, connesso e pesato; sia inoltre:*

- $A \subseteq E$ *contenuto in qualche albero di copertura minimo;*
- $(S, V \backslash S)$ *un taglio che rispetta A;*
- $(u, v) \in E$ *un arco leggero che attraversa il taglio.*

Allora (u, v) è sicuro per A.

3.4.2 L'algoritmo di Prim

Introdotto il teorema fondamentale (paragrafo 3.4.1) si può passare a descrivere velocemente l'*algoritmo di Prim* che è alla base delle operazioni di partizionamento (capitolo 4) svolte in questa tesi.
L'algoritmo di Prim si basa sul teorema fondamentale degli alberi di copertura minimi (paragrafo 3.4.1). Il costo dell'algoritmo varia a seconda della struttura dati di supporto utilizzata: se realizzato con coda di priorità ha un costo di $O(E \cdot log(V))$ mentre se realizzato tramite *heap di Fibonacci* ha un costo di esecuzione di $O(E + V \cdot log(V))$.
Infatti in questo algoritmo gli archi dell'insieme A (ovvero degli archi già selezionati dall'algoritmo) formano sempre un albero singolo. L'algoritmo parte da un nodo scelto $r \in V$, che sarà la radice dell'albero di copertura, e aggiunge ad A un *arco leggero* che collega A con un *vertice isolato* (che non sia estremo di qualche altro arco in A) ad ogni passo, garantendo così che ogni arco aggiunto sia *sicuro*.
Per gestire l'insieme dei vertici non ancora inclusi nell'albero di copertura, viene utilizzata una struttura dati del tipo *coda a min-priorità* basata sul campo *key*.
Il campo *key* è dato dal peso minimo di un arco che collega un vertice v a qualsiasi altro nell'albero; nel caso in cui questo arco non esista nell'insieme A, il valore di *key* sarà ∞.
Inoltre per ogni nodo viene indicato il predecessore tramite il campo π. Grazie proprio ai predecessori è possibile ricostruire l'albero finale in A.

$$A = \{(v, v.\pi) : v \in (V \backslash \{r\})\} \tag{2}$$

Algorithm 1 Prim minimun spanning tree

```
1:  function MST-Prim(grafo G, funzione_peso ω, nodo_radice r)
2:
3:      Q: coda di priorita contenente tutti i vertici in V
4:      for ogni u ∈ V(G) do
5:          u.key ← ∞
6:          u.π ← NIL
7:      r.key ← 0
8:      Q ← V(G)
9:      while Q ≠ 0 do
10:         u ← EXTRACT-MIN(Q)
11:         for ogni v ∈ G.Adj[u] do
12:             if v ∈ Q and ω(u, v) < v.key then
13:                 v.key ← ω(u, v)
14:                 v.π ← v
15:     return
```

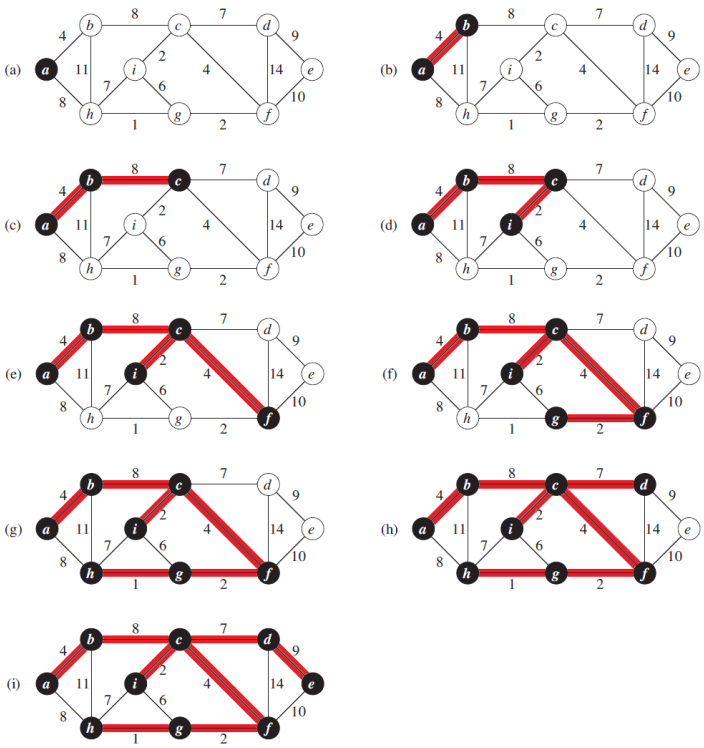


Figura 20: Esempio del funzionamento dell'algoritmo di Prim

4 La ripartizione del Dominio

Come detto nel paragrafo 3.3, la suddivisione del dominio (che avviene essenzialmente tramite *graph-partitioning*) è un processo di pre-computazione fondamentale per la maggior parte delle applicazioni che lavorano su architetture distribuite. Infatti, un partizionamento corretto permette di sfruttare al meglio la potenza di calcolo offerta da un sistema di *supercomputing*.
Vista la necessità di portare il progetto *Shallow Water Equation on GPU* su un'architettura di nodi distribuita, così da poter raggiungere la potenza di calcolo richiesta per eseguire simulazioni in tempi ragionevoli, si è iniziato a modellare l'applicativo per poterlo adattare alla suddetta architettura.
In particolare questa tesi studia la ripartizione del dominio sui vari nodi.
In questo capitolo verrà quindi introdotto il *criterio di suddivisione* studiato spiegando nel dettaglio la sua implementazione.
La suddivisione verrà affrontata in varie versioni che varieranno i risultati della ripartizione stessa dal punto di vista del bilanciamento (numero di elementi contenuti nei vari sotto-domini) e delle comunicazioni tra i vari sotto-domini.
Si procederà infine a fornire un confronto tra le varie versioni della suddivisione.

4.1 La conformazione del dominio

Prima di descrivere le varie operazioni di ripartizione è necessario però analizzare nel dettaglio il dominio stesso.
Come detto nel capitolo 1, il dominio reale del programma per la simulazione numerica dei flussi d'acqua, è una regione di territorio e tutte le informazioni che lo descrivono, vengono mappate nel dominio logico grazie alle celle e ai blocchi.
Ai fini della tesi verranno considerati solo i blocchi, senza tenere conto delle celle che li compongono, in quanto, le informazioni interessanti per la simulazioni vengono fornite dai blocchi stessi e non dalle singole celle che possono pertanto essere tralasciate.
In quest'ottica quando si parla di multi risoluzione ci si riferisce alla differenza di risoluzione dei blocchi e non delle celle che li compongono.

4.1.1 Le caratteristiche del grafo

Per rappresentare la conformazione del dominio reale all'interno del programma si è utilizzato un grafo diretto (paragrafo 3.1.1) in cui l'insieme V dei nodi è uguale all'insieme dei blocchi e l'insieme E degli archi è composto

dalle comunicazioni che ogni blocco ha con i relativi vicini.
La particolare conformazione di questo grafo fa sì che ogni blocco abbia un arco uscente e uno entrante per ogni vicino. Vista questa proprietà si assumerà d'ora in poi che il grafo sia indiretto e che l'adiacenza di un blocco con i vicini sia rappresentata da un solo arco $(blocco, vicino) \equiv (vicino, blocco)$ che sostituirà i due archi, quello entrante e quello uscente, senza alterare la struttura del dominio come si può vedere dalla figura 21.

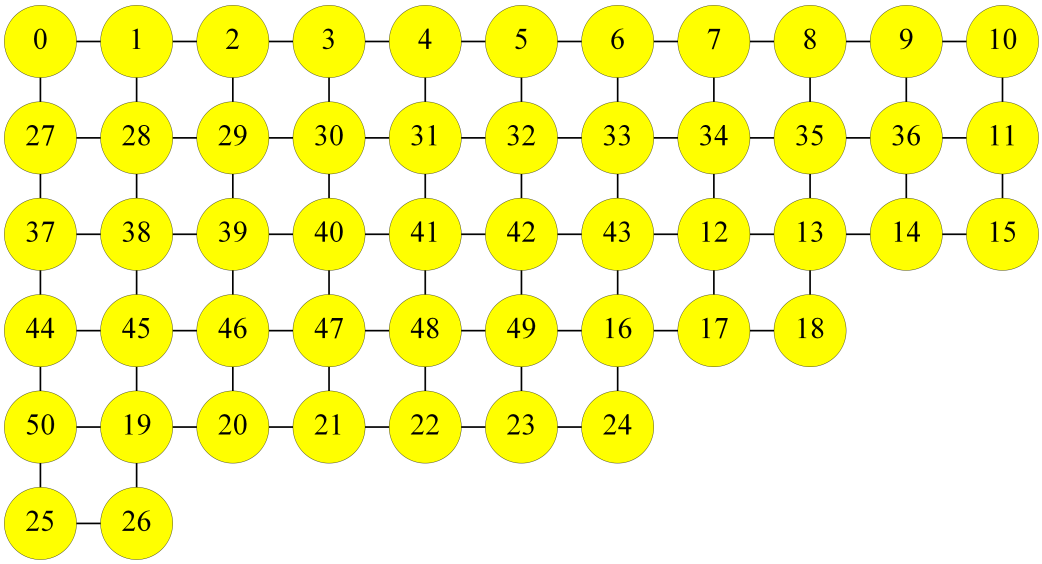


Figura 21: Grafo di un dominio reale senza multi risoluzione

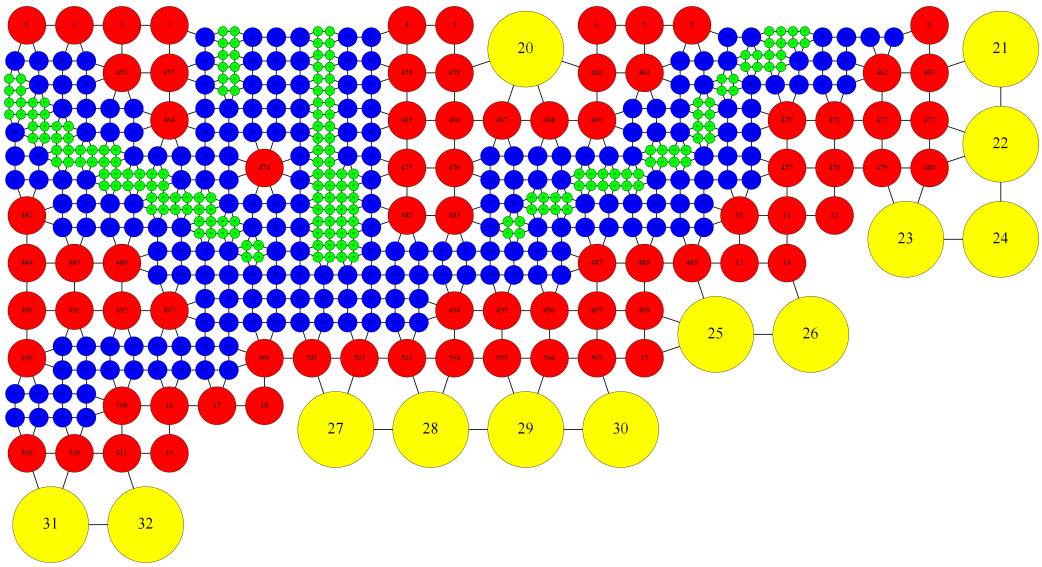
4.1.2 La rappresentazione del grafo

La rappresentazione effettiva del grafo su macchina avviene attraverso l'array *neigh* (paragrafo 1.4.1) che combina i vantaggi di una rappresentazione a lista e matrice di adiacenza (paragrafo 3.2). Questa combinazione è possibile grazie al numero piccolo (otto) di vicini di ogni blocco fissato a priori. Questa proprietà permette all'array *neigh* di poter accedere efficientemente alle informazioni dei vicini di ogni blocco mantenendo comunque un'occupazione di memoria pari a $\Theta(V + E)$.
Ad esempio dato un blocco **K** è possibile controllare le informazioni di tutti i suoi vicini scorrendo le otto celle da $neigh[\mathbf{K}.id * 8]$ a $neigh[\mathbf{K}.id * 8 + 7]$. Inoltre se si vuole controllare lo stato di un vicino in una determinata posizione (Nord, Sud, ...) basterà seguire lo schema indicato in figura 7 al capitolo 1 ed utilizzare l'indice relativo alla posizione voluta: le informazioni sul vicino

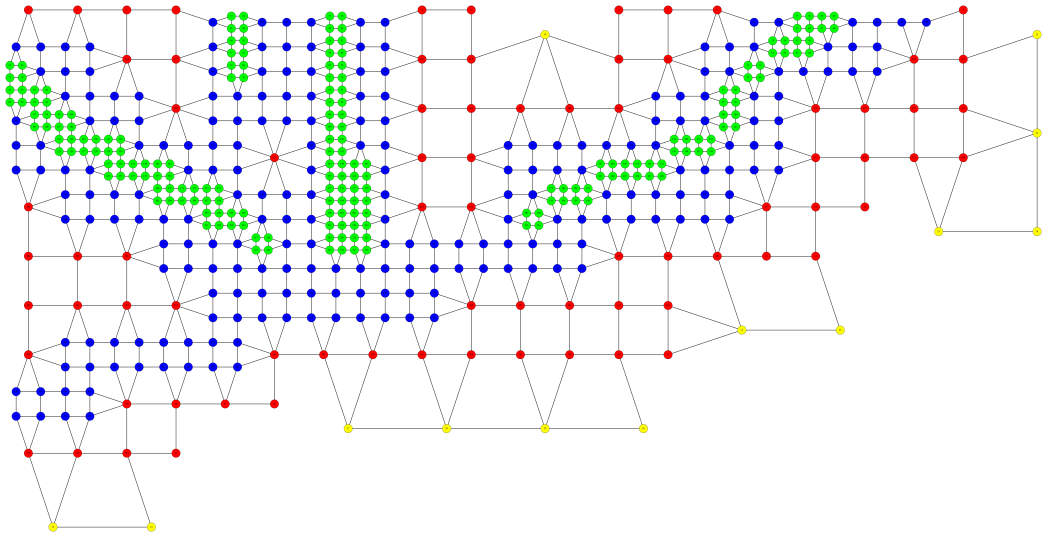
Nord di **K**, ad esempio, si possono ricavare, con un singolo accesso, nella cella $neigh[\mathbf{K}.id * 8 + 6]$, essendo il vicino Nord associato all'indice 6.

4.1.3 La multi risoluzione nel grafo

La topologia del grafo che rappresenta il dominio è profondamente influenzata dal concetto di multi risoluzione la cui presenza varia significativamente il rapporto tra la quantità di nodi presenti e l'ampiezza del terreno tra le zone di alto e basso interesse. Un'alta risoluzione crea quindi zone computazionalmente pesanti e nelle quali intercorrono grandi quantità di connessioni mentre, al contrario, zone a basse risoluzioni sono contraddistinte da un esiguo numero di blocchi e da poche comunicazioni.
Questa differenza tra le zone a diverse risoluzioni sarà il punto centrale dell'algoritmo di partizionamento. Questo infatti, cercherà di suddividere il dominio in modo che le zone ad alto costo computazionale siano distribuite equamente tra i vari sotto-domini eseguendo il *taglio* in zone con un basso numero di comunicazioni al fine di ridurre l'*overhead* dovuto allo scambio dei dati.



(a) Grafo con vertici proporzionali alla zona di terreno rappresentata



(b) Grafo con vertici uniformi

Figura 22: Grafo di un dominio reale con multi risoluzione

Il grafo d'ora in avanti sarà rappresentato con vertici rettangolari e senza rappresentare gli archi che connettono i vari nodi. Questa rappresentazione (realizzata tramite l'applicativo Graphviz [1]), mostrata in figura 23 permette una migliore visione d'insieme del dominio

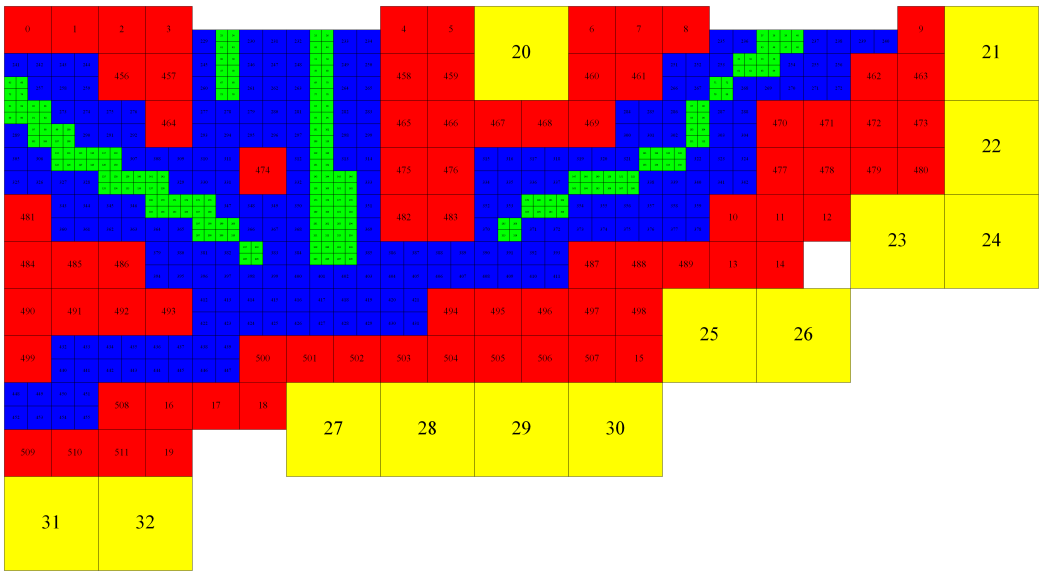


Figura 23: Modello di rappresentazione utilizzato

4.2 Le strutture dati dell'algoritmo

Lo sviluppo dell'algoritmo ha richiesto l'introduzione di alcune strutture dati che permettessero di poter elaborare efficientemente il dominio, legando ad ogni nodo del grafo (ovvero ad ogni blocco) informazioni aggiuntive.
Si è quindi sviluppato il tipo di dato *Block* (mostrato nel codice 5) che permette di salvare al suo interno, oltre alle informazioni fornite dal grafo di partenza (*id_block, id_resolution, neighbors*), altri parametri utili all'algoritmo di suddivisione.

```
struct Block
{
  int id_block;
  char resolution;
  int id_subdomain;
  int key;
  bool in_other_subdomain;
  struct neigh_t neighbors[4];
};
```

Codice 5: Struttura dati *Block*

Block cambia leggermente la modalità di memorizzazione dei vicini trasportandone tutte le informazioni nell'array *neighbors* che mantiene comunque il tipo *neigh_t* (paragrafo 1.4.1) per la memorizzazione dei vicini).

Al contrario di quello che ci si aspetterebbe la dimensione di *neighbors* è quattro e non otto. Questo è dovuto al fatto che le comunicazioni del blocco con i vicini diagonali avvengono solo in casi di adiacenza particolari e quindi non influenzano i criteri per la suddivisione. L'associazione tra indice e posizione del blocco viene comunque mantenuta introducendo un nuovo schema mostrato in figura 24.

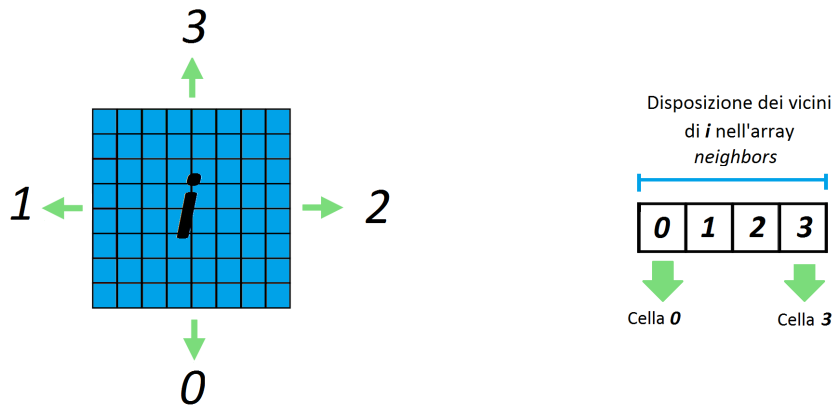


Figura 24: Il nuovo schema di associazione

Il grafo, durante il partizionamento, sarà rappresentato quindi da un array di *Block*, chiamato *adjacency_list*, che rispecchia le proprietà di *neigh*.

4.3 La suddivisione tramite multi-MST

Introdotte le strutture dati principali e spiegata la conformazione del grafo si possono passare a descrivere i primi stadi dell'algoritmo pensato per trovare una buona suddivisione.
Una buona ripartizione del dominio sui vari nodi di calcolo dovrebbe, come detto prima, soddisfare due requisiti:

- Bilanciare la quantità di informazioni, e quindi il numero di blocchi, fra tutti i sotto-domini;
- Ridurre al minimo il numero di comunicazioni tra i nodi di calcolo cercando quindi di eseguire i *tagli* al grafo in zone in cui intercorrono pochi *archi*.

Considerando la multi risoluzione e la conseguente topologia del grafo i due parametri citati precedentemente possono essere rivisti nei seguenti termini:

- Suddividere più equamente possibile le zone a risoluzione più alta;
- Eseguire i *tagli* nelle zone a risoluzione più bassa.

L'idea su cui si basa l'algoritmo di partizionamento, che cerca di rispettare questi due requisiti, si fonda sul concetto di *albero di copertura minimo* con archi equi-pesati (paragrafo 3.4).
In particolare si utilizzerà una foresta di MST che esplorano il grafo contemporaneamente.

4.3.1 Suddivisione di un dominio Ideale

Inizialmente l'idea è nata immaginando di effettuare una suddivisione su un *dominio ideale* in cui ci sono tante zone ad alta risoluzione quanti nodi di calcolo e dove queste degradano lentamente verso zone a risoluzione minima. Prendendo questo dominio è facile immaginarne una suddivisione ottimale tramite gli alberi di copertura minimi.
In particolare ogni albero di copertura minima, che rappresenta la partizione da assegnare ad un nodo di calcolo, ingloberebbe tutta una zona a risoluzione massima e i conseguenti contorni a risoluzioni intermedie, fermandosi (come mostrato nella figura 25) quando incontra le *foglie* di un altro MST (che avrà a sua volta inglobato nei suoi nodi interni le zone a risoluzione maggiore).
Data la configurazione ideale del dominio si può supporre che l'incontro fra i vari MST avvenga nelle zone di risoluzione minima individuando un'area nella quale è possibile effettuare un *taglio* che minimizza le comunicazioni.

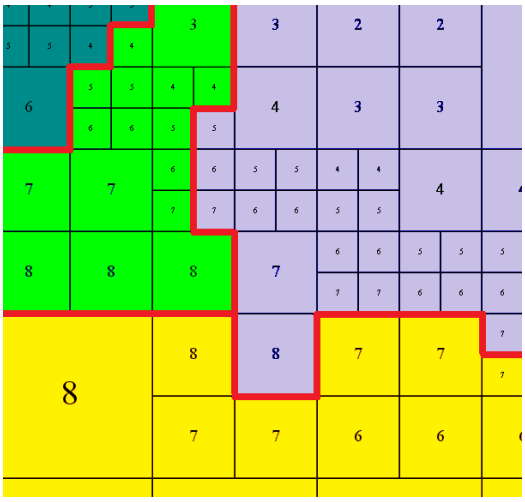


Figura 25: Le linee di arresto dei vari MST che compongono la foresta
In ogni vertice del grafo è mostrato il realtivo peso e il peso di ogni arco è considerato unitario

I problemi relativi a questo primo passo dell'algoritmo sono però evidenti: si richiede un dominio ideale e quindi non compatibile con i casi reali, gli unici che sono realmente interessanti, e inoltre si presuppone di conoscere a priori i blocchi di partenza che permettono all'algoritmo di eseguire una suddivisione ottimale.

4.3.2 Le radici Casuali

Il partizionamento tramite *multi-MST* (ovvero l'esecuzione di più MST su uno stesso dominio) rimane però un buon punto di partenza. La suddivisione ottenuta grazie a questo approccio, se scelti buoni punti di partenza, riesce a sfruttare la topologia del grafo per creare partizioni bilanciate e *tagli* con un basso numero di comunicazioni.
Partendo dalle giuste radici, i vari alberi di copertura minimi permetterebbero una divisione omogenea del numero dei blocchi.
Avendo infatti gli archi tutti lo stesso peso, la copertura di un albero si fermerà solo quando le foglie di questo incontreranno le foglie di un altro albero con un peso simile (come mostrato in figura 25), indicando che il numero di blocchi inglobati è equilibrato.
Inoltre è possibile supporre che, se le radici fossero poste nei punti centrali delle zone a densità di informazione massimale[3], questo tipo di suddivisione

[3]Con risoluzione più alta e con massima distanza dalle zone a risoluzione più bassa

permetterebbe di eseguire i *tagli* nelle zone topologicamente più leggere e quindi dove intercorrono meno comunicazioni.
Idealmente infatti posizionando le radici nei punti "più interni"(ovvero nei punti al centro delle zone più dense) del dominio, le zone sarebbero mappate sull'albero all'incirca in base alla loro concentrazione di blocchi. In particolare le zone a densità minore rappresenterebbero le foglie dell'albero e quindi le zone in cui viene eseguito il *taglio*.
Come primo stadio dell'algoritmo su casi reali è stata realizzata una suddivisione basata su multi-MST che sceglie casualmente le proprie radici e che viene eseguita ripetutamente per poter scegliere una soluzione accettabile.
Nella (figura 26) è mostrata una suddivisione con radici di partenza casuali.

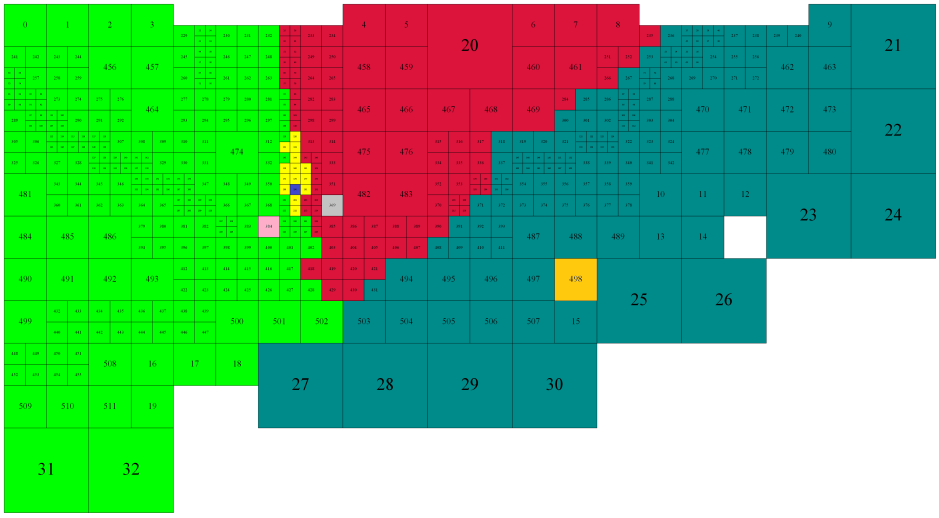


Figura 26: Ripartizione del dominio in quattro partendo da radici casuali
Le radici sono messe in evidenza con colori diversi

L'implementazione dell'algoritmo di Prim con multi-radice
Nel codice 2 si dà una descrizione in pseudo-codice della versione dell'algoritmo di Prim (algoritmo 1), utilizzato per realizzare la suddivisione, che ammette un numero di *radici*, e di conseguenza un numero di MST, parametrico.
roots è la lista delle radici e *adjacency_list* rappresenta l'array di *Block* citato precedentemente.

Algorithm 2 Prim Multi Source

```
1:  function PRIM_MULTI-MST(roots_list roots, array_Block adjacency_list)
2:      //Inizializzazione di tutti i vertici
3:      for ogni u ∈ adjacency_list do
4:          u.key ← ∞
5:          u.id_subdomains ← NIL
6:          u.in_other_subdomain ← FALSE
7:      //Inizializzazione di tutte le radici
8:      i ← 0
9:      for ogni r ∈ roots do
10:         r.key ← 0
11:         r.id_subdomains ← i
12:         i ← i + 1
13:     Q: coda di priorita ordinata in base al campo key
14:     Q ← roots
15:     count ← 0
16:     while count < adjacency_list.size do
17:         u ← EXTRACT-MIN(Q)
18:         if !(u.in_other_subdomain) then
19:             //Essendo il minimo viene aggiunto definitivamente
20:             u.in_other_subdomain ← TRUE
21:             //Si analizzano i vicini
22:             for ogni v ∈ u.neighbors do
23:                 //In questo caso specifico il peso di ogni arco vale 1
24:                 if v ∈ adjacency_list and (u.key + 1) < v.key then
25:                     v.key ← u.key + 1
26:                     v.subdomains ← u.subdomains
27:                     count ← count + 1
28:     return
```

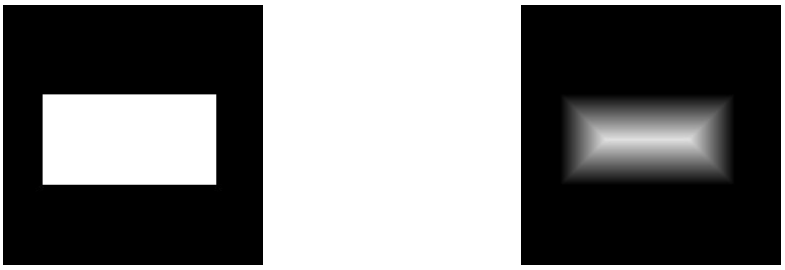
Terminata l'esecuzione, per ricostruire i vari sotto-domini, basterà analizzare i campi *id_subdomain* di tutti gli elementi (blocchi) contenuti all'interno dell'array *adjacency_list*, oppure memorizzare, per esempio in un array di liste di interi, i vari *id* relativi ai blocchi che compongono un determinato sotto-dominio ogni volta che un blocco viene inglobato definitivamente da un MST (algoritmo 2, riga 19).

4.4 La selezione delle radici

Si è implementata una suddivisione basata sulle capacità degli alberi di copertura minimi di sfruttare la conformazione topologica del grafo con multi risoluzione, che utilizza però come radici vertici casuali, richiedendo così un alto numero di tentativi per ottenere una suddivisione soddisfacente.
Per ovviare al problema quindi, si è cercato un criterio di *selezione delle radici*.
Il criterio che verrà proposto in questa tesi si baserà sul concetto di *massimo locale*.
Prima di introdurre il concetto di massimo locale però è necessario parlare di *distance transform*.

4.4.1 Distance Transform e Massimi Locali

Il concetto di *distance transform* o *distance map* è solitamente legato alla rappresentazione di immagini e consiste nel trasformare un'immagine binaria[4] in un'immagine a scala di grigi [6]. Questa trasformazione mantiene invariata la struttura dell'immagine ma "sfuma" il livello di colore partendo dai punti più interni ed arrivando, con i toni più chiari, ai bordi dell'immagine.



(a) L'immagine prima della trasformazione

(b) La stessa immagine dopo la trasformazione

Figura 27: Esempio di *distance transform* su immagine binaria

Lo stesso concetto può essere applicato al contrario, ovvero: dati i punti più esterni, si può "sfumare" verso quelli più interni cioè verso tutti quei punti con una massima distanza dai bordi rispetto ai loro vicini.
Una suddivisione di questo genere, applicata al dominio che si intende partizionare, permette di creare una distribuzione discreta e lineare di pesi.
In particolare si può utilizzare una specifica trasformazione *distance map* che, partendo dai punti centrali delle zone a risoluzione minima locale (che

[4] con solo 2 livelli di colore, tipicamente il bianco e il nero

comunicano cioè solo con zone a risoluzione maggiore), distribuisce i valori aumentandoli man mano che ci si allontana.
Dopo questa trasformazione si avrà quindi una "curva" della distribuzione delle risoluzioni ovvero, si passerà da una distribuzione di valori quantizzati[5] molto distanti, ad una distribuzione più fitta. Questa distribuzione considera le distanze dalle risoluzioni vicine in modo da poter guidare una ricerca per gradiente dei massimi locali (figura 28). I vertici del grafo che avranno associato un valore maggiore rispetto ai propri vicini, saranno identificati proprio come *massimi locali* e rappresenteranno i punti più interni delle zone che hanno localmente la massima risoluzione.

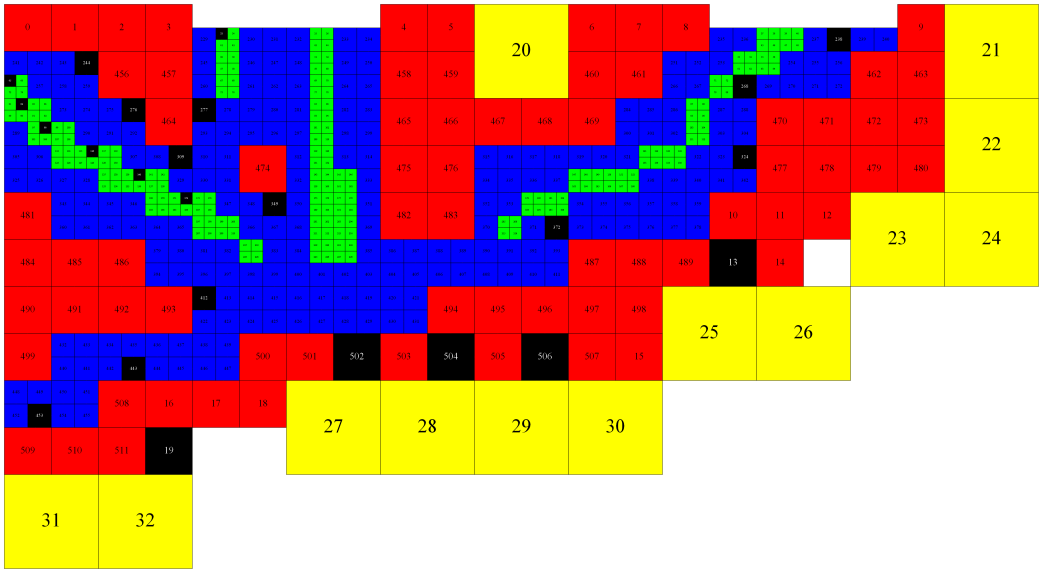


Figura 28: Grafo del dominio con i massimi locali evidenziati (in nero)

Si illustra di seguito lo pseudo codice relativo alla trasformazione utilizzata sul dominio (che sfrutta l'algoritmo 2) per l'individuazione dei massimi locali:

[5]I valori rappresentano i vari livelli di risoluzione e sono: 1, 2, 3 e 4

Algorithm 3 Map Tranform

```
1:  function MAP_TRANFORM(array_Block adjacency_list)
2:      //Inizializzazione di tutti i vertici
3:      for ogni u ∈ adjacency_list do
4:          u.key ← ∞
5:          u.in_other_subdomain ← FALSE
6:      //Si analizza il grafo un livello di risoluzione alla volta
7:      res ← min_res_value
8:      while res ⩽ max_res_value do
9:          Si definisce adjacency_list(res_value) l'insieme
10:         dei nodi a risoluzione res_value
11:         Q ← adjacency_list(res_value)
12:         is_L ← FALSE
13:         while Q ≠ 0 do
14:             if EXISTS_LOW(Q)[1] and res ≠ min_res_value then
15:                 start_node ← EXTRACT_BOUNDARY_L(Q)[2]
16:                 is_L ← TRUE
17:             else
18:                 start_node ← EXTRACT_BOUNDARY_H(Q)[3]
19:             MST_WEIGHT_DIST(start_node, is_L, adjacency_list, Q)[4]
20:     return
```

1 EXISTS_LOW (riga 10) verifica se è presente almeno un blocco che ha almeno un vicino a risoluzione minore;

2 EXTRACT_BOUNDARY_L (riga 11) estrae come blocco di partenza un blocco che comunica con vicini a risoluzione minore;

3 EXTRACT_BOUNDARY_H (riga 14) estrae un blocco che sia sul bordo del dominio o che abbia almeno un vicino a risoluzione maggiore;

4 MST_WEIGHT_DIST (riga 15) spiegato nell'algoritmo 4.

Algorithm 4 MST Weight Redistribution

```
1: function MST_WEIGHT_DIST(Block start_node, bool is_L, array_Block adjacency_list, list_same_res Q)
2:     T ← EXTRACT_COMMUNICANT(Q, res_start)[1]
3:     if is_L then
4:         list_start_node ← COMPLETE_BOUNDARY_L(start_node, T)[2]
5:         new_start ← PRIM_MULTI-MST_central(list_start_node, T)[3]
6:         T[new_start].key ← 0
7:     else
8:         list_start_node ← COMPLETE_BOUNDARY_H(start_node, T)[4]
9:         BOUNDARY_WEIGHT(list_start_node, adjacency_list)[5]
10:    PRIM_MULTI-MST_final(list_start_node, T, Q, adjacency_list)[6]
11:    return
```

1 EXTRACT_COMMUNICANT (riga 2) estrae tutti i nodi di Q che hanno comunicazione (diretta o indiretta) con *start_node* passando solo attraverso blocchi della stessa risoluzione;

2 COMPLETE_BOUNDARY_L (riga 4) estrae tutti i nodi nella zona di *start_node* (T) che hanno stessa risoluzione e che possiedono almeno un vicino a risoluzione minore;

3 Si considera che la funzione PRIM_MULTI-MST_central (riga 5) esegua l'algoritmo 2 ma ritorni il nodo con valore *key* maggiore senza modificare globalmente T;

4 COMPLETE_BOUNDARY_H (riga 8) estrae tutti i nodi nella zona di *start_node* (T) che hanno stessa risoluzione e che possiedono almeno un vicino a risoluzione maggiore;

5 BOUNDARY_WEIGHT (riga 9) ridefinisce i pesi delle radici (*list_start_node*) assegnando ad ogni radice il peso del proprio vicino (anche a risoluzione minore) "meno pesante"aumentato di 1 (tutti gli archi hanno infatti peso unitario);

6 PRIM_MULTI-MST_final (riga 10) esegue l'algoritmo 2 su T senza settare a 0 il valore *key* delle radici. Inoltre estrae da Q tutti i blocchi in T e riporta le modifiche sui valori dei blocchi in *adjacency_list*.

4.4.2 I massimi locali randomici come partenza

Grazie all'algoritmo 3 è quindi possibile determinare tutti i massimi locali del grafo; essendo questi i punti più interni del grafo, come detto nel paragrafo 4.3.2, costituiscono delle buone radici per i vari MST.
Siccome il numero di partizioni desiderate è quasi sempre di molto inferiore al numero di massimi locali trovati, nasce il problema di quali scegliere come punti di partenza.
Un primo approccio è stato quello di scegliere le radici dell'algoritmo di partizione (algoritmo 2) casualmente tra tutti i massimi locali (figura 29).
Come si nota in figura 29 però questo approccio non garantisce affatto una suddivisione ideale.

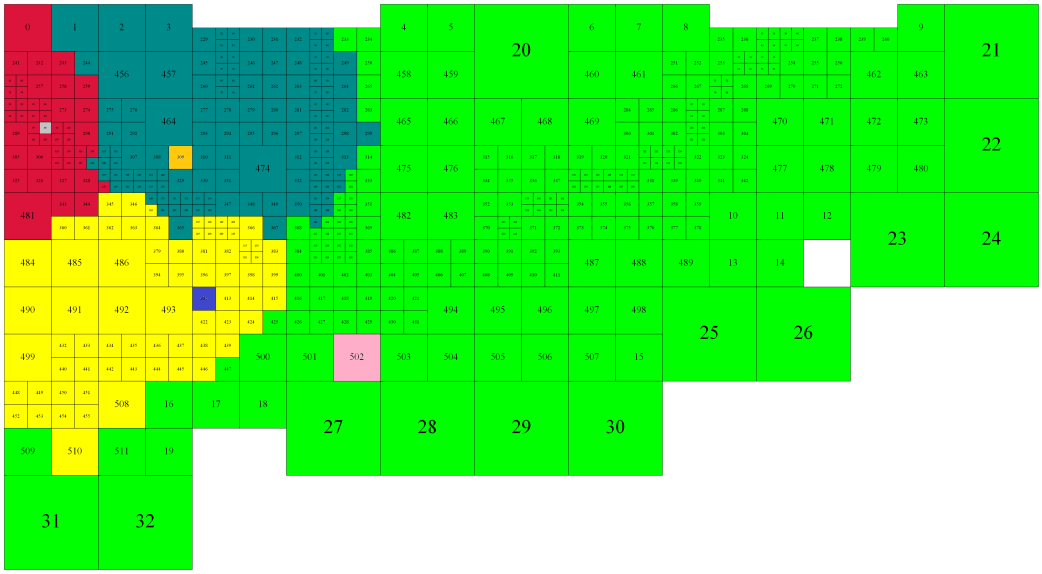


Figura 29: Ripartizione del dominio in quattro partendo da radici scelte casualmente tra i massimi locali

4.5 Multi-MST sui massimi locali

Calcolati tutti i massimi locali bisogna quindi cercare di sfruttarli per produrre una buona suddivisione.
I massimi locali sono, come definito nel paragrafo 4.4.1, i punti centrali delle zone a densità di informazione massimale (risoluzione più alta e lontana da zone a risoluzione più bassa) e quindi delle zone più dense del grafo. Vista la posizione dei massimi locali è facile immaginare che nelle loro vicinanze vi sia un'alta concentrazione di blocchi e, di conseguenza un alto numero di comunicazioni.

Se si prende, per esempio, il dominio in figura 28 nel quale sono stati calcolati i massimi locali risulta infatti evidente che le zone dove intercorrono meno comunicazioni sono quelle più distanti dai vari massimi locali.
Questo è dovuto proprio al fatto che i massimi locali sono al centro delle zone più pesanti dal grafo e quindi allontanandosi da questi si arriva su zone con una bassa densità di blocchi (zone a bassa risoluzione).

4.5.1 Zone di appartenenza e Tagli ideali

Vista questa proprietà si può quindi supporre che sia conveniente effettuare i *tagli* tra i blocchi topologicamente più distanti dai vari massimi locali.
Per individuare queste zone si è sfruttata ancora una volta l'idea di albero di copertura minimo.
In particolare si è pensato di sviluppare contemporaneamente, sullo stesso dominio, un MST per ogni massimo locale (con radice nel massimo locale stesso) utilizzano l'algoritmo 2.
Quando l'algoritmo ha terminato la sue esecuzione si potranno distinguere nel grafo le varie "*zone di appartenenza*", una per ogni massimo locale (figura 30).

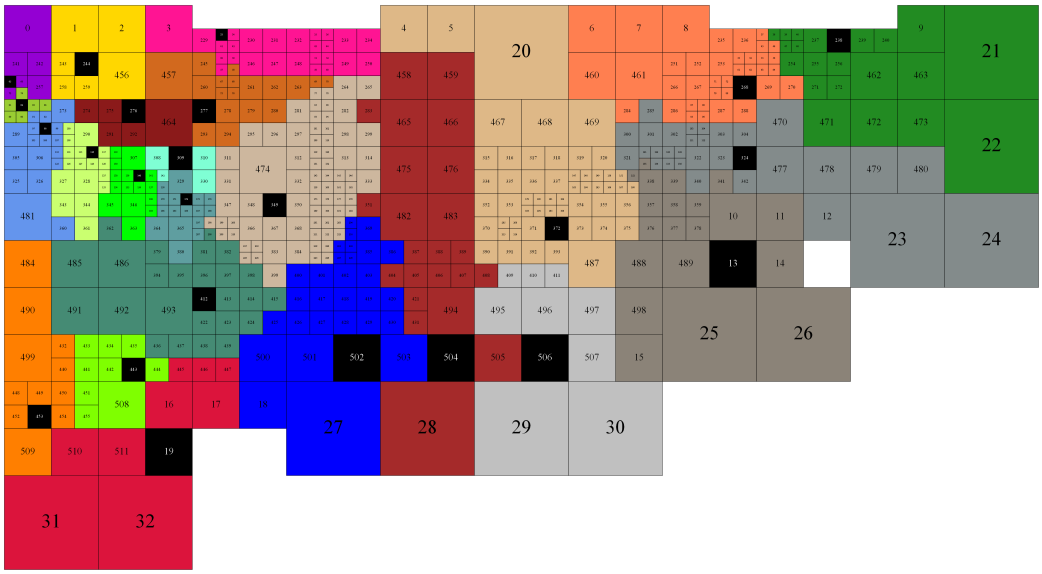


Figura 30: Le zone di appartenenza descritte dalla foresta di MST

Una volta definite le "zone di appartenenza" risulta quindi chiaro che i bordi esterni dei vari MST rappresentano le zone più distanti dai massimi locali.

In particolare, seguendo l'idea che nelle nelle zone più lontane dai massimi locali ci sia minima comunicazione, i vari *tagli* dovrebbero essere effettuati lungo *le linee di contatto* (mostrate in figura 25) tra i vari MST.
Tagliando lungo le suddette linee infatti è possibile evitare di *tagliare* dove vi è una grande densità di blocchi.
Si può dire quindi che le *linee di contatto* forniscano le zone ideali su cui effettuare i vari *tagli* necessari a partizionare il grafo.

4.5.2 Unione delle zone di appartenenza

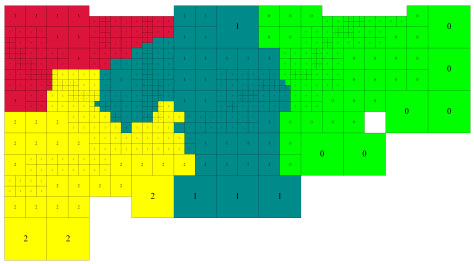
Definite le linee lungo le quali effettuare i possibili *tagli* è il momento di suddividere il grafo vero e proprio nel numero di partizioni desiderato.
Per mantenere intatte le zone di taglio sicure si è pensato di eseguire la suddivisione partendo proprio dalle *zone di appartenenza*.
Unendo tra loro le singole zone è possibile infatti creare sotto-domini maggiori che conservino le *linee di taglio* individuate precedentemente e che producano quindi un buon partizionamento del grafo.
Perciò si è usato un processo di unione iterativo che ad ogni passo seleziona i due sotto-domini migliori da unire fino a raggiunger il numero di partizioni volute.
Siccome vi sono diversi criteri di unione validi, questo algoritmo di unione è stato sviluppato in più versioni ognuna delle quali può variare sia nella scelta iniziale del sotto-dominio che si vuole unire, sia nella scelta della partizione da unire con il suddetto sotto-dominio.
In particolare si avranno i seguenti criteri:

1 Si sceglie come sotto-dominio di partenza quello con il rapporto *numero_blocchi*/*numero_comunicazioni* peggiore (cioè minore) e come partizione da unire quella che riduce maggiormente le comunicazioni del sotto-dominio di partenza (figura 31a);

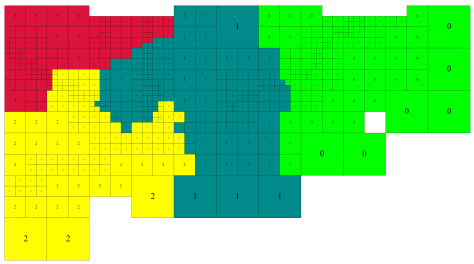
2 Si sceglie come sotto-dominio di partenza quello con il rapporto *numero_blocchi* / *numero_comunicazioni* peggiore (cioè minore) e come partizione da unire quella che, unita al sotto-dominio di partenza, produce il rapporto *numero_blocchi* / *numero_comunicazioni* migliore (figura 31b);

3 Si sceglie come sotto-dominio di partenza quello con il più alto numero di comunicazioni e come partizione da unire quella che,che riduce maggiormente le comunicazioni del sotto-dominio di partenza (figura 31c).

Tutti e tre i vari criteri sono suddivisi in altre due versioni: una che ammette la possibilità di eseguire un'unione anche tra vicini non adiacenti per permettere un bilanciamento migliore, e un'altra che non lo consente (a meno di un grande sbilanciamento nei sotto-domini finali).

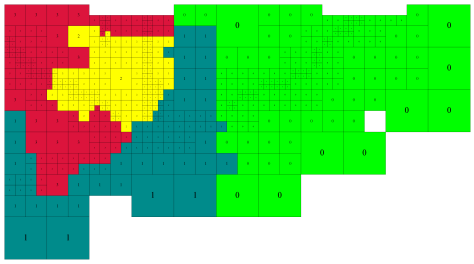


Amessi solo vicini adiacenti

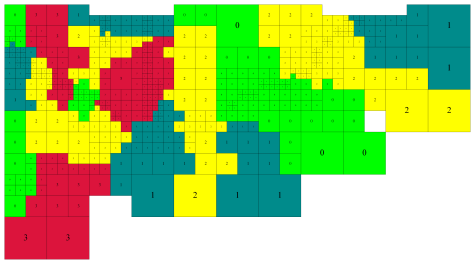


Vicini non adiacenti ammessi

(a) Ottimizzazione rapporto *numero_blocchi* / *numero_comunicazioni* e comunicazioni

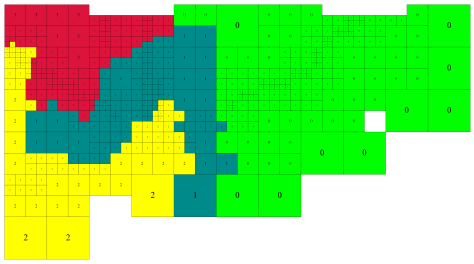


Amessi solo vicini adiacenti

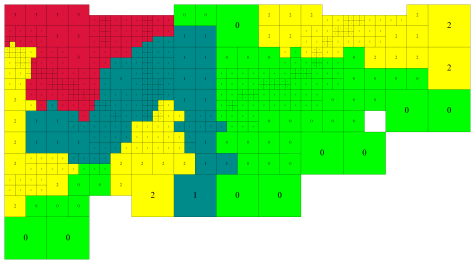


Vicini non adiacenti ammessi

(b) Ottimizzazione rapporto *numero_blocchi* / *numero_comunicazioni*



Amessi solo vicini adiacenti



Vicini non adiacenti ammessi

(c) Ottimizzazione comunicazioni

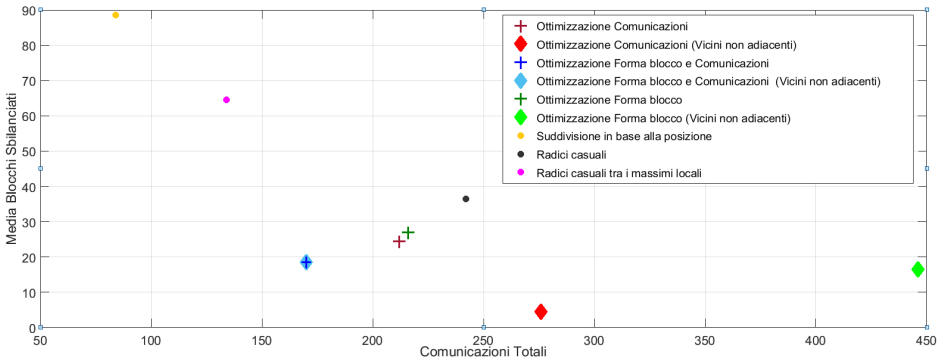
Figura 31: I vari criteri di unione a confronto

I vari criteri di unione creano diverse suddivisioni all'interno dello stesso grafo ma producono comunque delle ripartizioni accettabili, sia dal punto

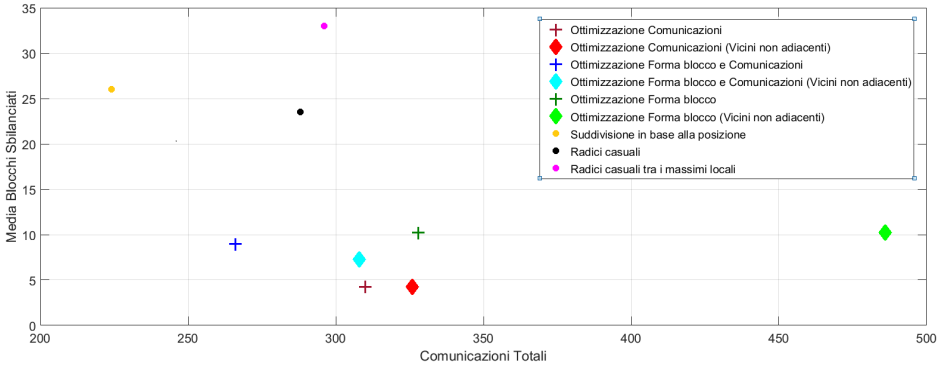
di vista del bilanciamento che dal punto di vista dell comunicazioni, senza richiedere diversi tentativi come invece accade nelle implementazioni basate su radici casuali.
Si è quindi riusciti ad ottenere una suddivisione soddisfacente del dominio, che permette di ripartire il carico adeguatamente su una struttura MPI.

4.6 I confronti

Nell'immagine 32 si mostreranno due grafici che valutano in base a due parametri (il bilanciamento e il numero delle comunicazioni totali) le varie fasi dell'algoritmo e i vari criteri di unione delle *zone di appartenenza* (paragrafo 4.5.2).
In particolare si sono messi a confronto i vari criteri di unione, tra loro e con altri algoritmi di suddivisione basati sulla posizione dei blocchi o su scelte di radici casuali.



(a) Il confronto degli algoritmi con quattro partizioni



(b) Il confronto degli algoritmi con otto partizioni

Figura 32: Le varie suddivisioni a confronto

Nei grafici in figura 32 non emerge, come ci si aspetterebbe, un miglioramento sul numero delle comunicazioni tra le suddivisioni basate sulle zone di appartenenza rispetto a quelle in cui si sono scelte radici casuali. Si nota anzi, che la suddivisione basata sull'offset ottimizza il numero di comunicazioni: questo è probabilmente dovuto alla conformazione del dominio e non è attribuibile ad una vera e proprio caratteristica della suddivisione.
Per quanto riguarda il bilanciamento invece, le suddivisioni basate sulle zone di appartenenza, ottengono dei buoni risultati rispetto alle altre. In particolare, si ottengono dei buoni risultati quando si ammettono anche unioni tra sotto-domini non adiacenti.

4.6.1 Il risultato finale

Il risultato dei confronti tra i vari algoritmi di ripartizione (figura 32), non fa emergere, come ci si aspetterebbe, un netto miglioramento sul numero di comunicazioni dovuto alla tecnica di suddivisione basata sulle *zone di appartenenza*.
Questo è dovuto al fatto che il criterio di ripartizione basato sulle *zone di appartenenza*, suddivide il dominio basando la sua esecuzione su zone che non cambiano mai la loro forma di base; infatti ogni sotto-dominio finale è semplicemente l'unione dei diversi MST iniziali.
Lo scarso miglioramento, rispetto ad algoritmi che si basano su scelte casuali, è quindi probabilmente dovuto alla struttura non ideale del *taglio*, dettata dalla rigidità dei sotto-domini di base. Si può concludere che l'algoritmo basato sulle *zone di appartenenza*, riesca ad individuare dove sia più corretto eseguire i *tagli*, ma non li esegua nel modo più adeguato.

Al contrario, l'ultima suddivisione proposta in questa tesi, garantisce sempre un buon bilanciamento dei blocchi tra le varie ripartizioni.
Questa proprietà è molto importante in quanto, anche se non se ne ha la certezza, la distribuzione non equilibrata del carico porterebbe ad un overhead molto alto. Infatti, anche se non è ancora stata sviluppata una versione che permetta di testare realmente le suddivisioni, sembra che lo squilibrio del carico sui vari nodi causi un maggiore rallentamento rispetto a quello dovuto ad un alto numero di comunicazioni.

L'algoritmo studiato, nella sua ultima versione (paragrafo 4.5.1), ha l'interessante proprietà di individuare un partizionamento con un numero di comunicazioni accettabile e di garantire un bilanciamento quasi perfetto[6].

[6]Il bilanciamento è migliore se si permette di unire tra loro sotto-domini non adiacenti

L'idea di ripartizione proposta in questa tesi potrebbe essere quindi, con qualche piccolo aggiustamento, una buona base per produrre una suddivisione ottima su tutti i vari domini che l'applicativo *sweGPU* deve elaborare, indipendentemente dalla loro confermazione specifica.
In particolare l'idea di partizionamento basata sull'esecuzione di una foresta di MST (con radici adeguate) su uno stesso dominio con multi risoluzione, sembra garantire una suddivisione migliore di quelle che verrebbero individuate con gli algoritmi di partizionamento classici[2]. In particolare l'algoritmo che ottimizza il rapporto *numero_blocchi* / *numero_comunicazioni* e le comunicazioni, e che non ammette i sotto-domini non adiacenti, è quello che produce i risultati migliori.

5 Il modello MPI

Una volta definito come partizionare il dominio (capitolo 4) è necessario predisporre l'ambiente di calcolo per eseguire l'applicativo parallelamente sui vari nodi.
Di seguito si introduce un modello MPI che assegna le diverse partizioni sui vari nodi di calcolo e che gestisce le varie comunicazioni tra i nodi stessi.

5.1 L'architettura e la distribuzione dei dati

Il tipo di architettura hardware pensata per supportare la versione parallelizzata di questo applicativo è un'architettura *MIMD* (figura 14d) a memoria distribuita.
In particolare si parla di memoria distribuita in quanto le informazioni della partizione sono note solo al nodo che la contiene e possono essere lette dagli altri solo attraverso uno scambio di messaggi.
Prima che il programma distribuito venga eseguito è quindi necessario assegnare ad ogni nodo le informazioni che gli appartengono e quelle che gli sono utili per eseguire correttamente le comunicazioni con gli altri nodi di calcolo.
Ogni nodo ha bisogno di conoscere solamente:

- Quale partizione è assegnata ad ogni nodo di calcolo;
- I dati della mappatura del terreno di ogni blocco contenuto nella propria partizione;
- La lista, identificata dall'array *neigh* (paragrafo 1.4.1), di tutti i blocchi con i relativi vicini;
- La partizione (e di conseguenza il nodo) di appartenenza di ogni blocco del dominio.

5.2 La fase di pre-computazione

Nel modello MPI avverrà quindi una fase di pre-computazione che si occuperà di distribuire sui vari nodi tutte le informazioni necessarie.
Come tipicamente avviene nelle strutture di calcolo distribuito, la fase di pre-computazione viene eseguita soltanto su un nodo, d'ora in poi indicato con il termine *root*, che si occupa di calcolare e distribuire le varie informazioni.
In particolare, durante la fase di pre-computazione per la parallelizzazione in un ambiente MPI del programma *sweGPU*, il nodo *root* sarà l'unico a conoscere le informazioni sul domino. Nello pseudo-codice 5 si descrive la fase di

pre-computazione (mostrata in figura 33a) eseguita dal nodo *root*, considerando D come l'insieme delle informazioni riguardanti il dominio e *par_id_to_node* come una funzione che associa l'id di una partizione al rispettivo nodo.

Algorithm 5 Fase di pre-computazione

```
1:  function DATA_DISTRIBUITION(program_data D, func par_id_to_node)
2:      MPI::Init()
3:      //Invio delle partizioni ai nodi relativi
4:      if rank ≡ root.rank then
5:          lists_partitions ← MAKE_PARTITION(D.domain)
6:          for ogni partition ∈ lists_partitions do
7:              node_to_send ← par_id_to_node(partition.id)
8:              if node_to_send≠root.rank then
9:                  MPI_NON_BLOCKING_SEND(partition, node_to_send)
10:                 add_to_list_block-partition(partition)
11:             else
12:                 SAVE(partition)
13:     else
14:         MPI_RECEIVE(root, partition)
15:         SAVE(partition)
16:     /*Invio in broadcast (paragrafo 2.2.4) dell'array neigh e dell'array
17:     block-partition che associa ad ogni blocco la propria partizione*/
18:     if rank ≡ root.rank then
19:         MPI_BROADCAST_SEND(D.neigh)
20:         SAVE(D.neigh)
21:         MPI_BROADCAST_SEND(block-partition)
22:         SAVE(block-partition)
23:         DELETE(D)
24:     else
25:         MPI_BROADCAST_RECEIVE(neigh)
26:         SAVE(neigh)
27:         MPI_BROADCAST_RECEIVE(block-partition)
28:         SAVE(block-partition)
29:     MPI::Finalize()
30:     return
```

5.3 Lo scambio dei Dati di esecuzione

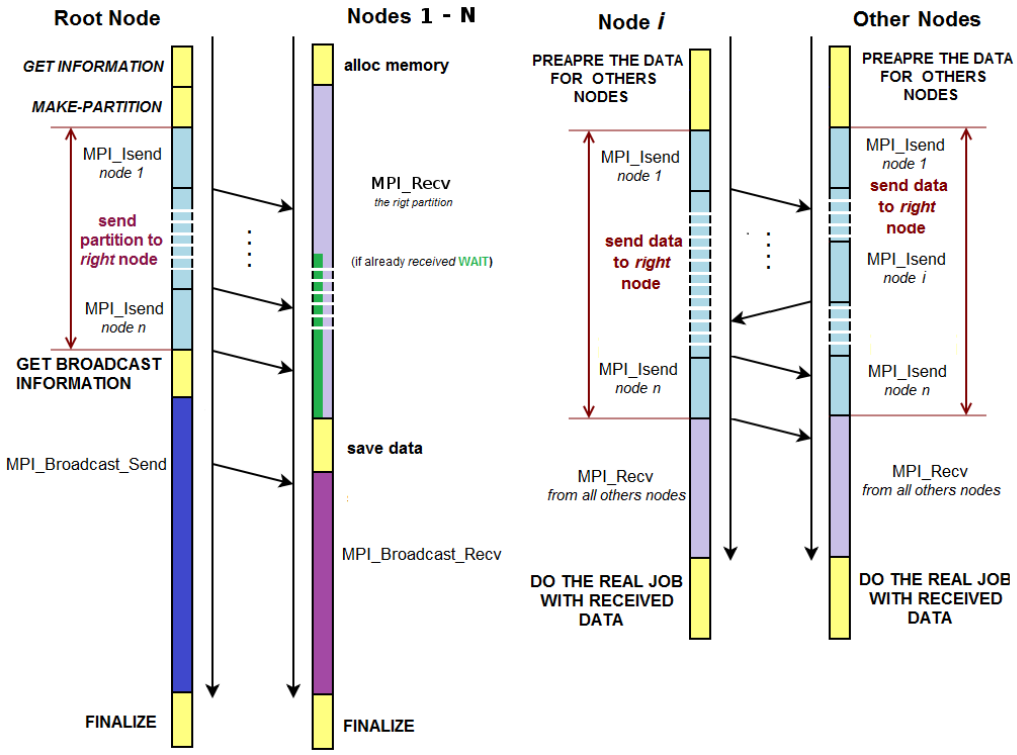
Finita la fase di pre-computazione, l'applicativo per la simulazione numerica dei flussi può iniziare ad eseguire i vari calcoli parallelamente sui vari nodi.
Come spiegato nel paragrafo 1.4, durante ogni passo di esecuzione del programma ogni cella ha bisogno delle informazioni dei propri vicini nel dominio reale. In particolare può capitare che una cella di bordo richieda informazioni che si trovano su un altro blocco (figura 6) e siccome il dominio è stato distribuito su vari nodi può capitare che il blocco in questione non si trovi sullo stesso nodo di quello che contiene la cella. Quando si verifica questa situazione il nodo deve richiedere, tramite protocolli MPI, le informazioni necessarie ai nodi che contengono i vicini.
Ad ogni passo di calcolo i nodi devono scambiare informazioni riguardanti i bordi tra blocchi adiacenti e su nodi diversi. Per sfruttare la banda di trasmissione, conviene accorpare i dati da scambiare tra coppie di nodi in un unico blocco di informazione.
In particolare si accodano tutte le informazioni dei vari bordi da trasmettere. Ciascun nodo ha informazioni sufficienti per sapere quali bordi dovranno essere spediti e ricevuti. In questo modo alla ricezione del blocco di dati, sarà possibile gestire i singoli bordi.

Nello pseudo-codice 6 viene descritto il modello MPI di comunicazione tra i nodi durante lo svolgimento del programma *sweGPU*, in cui la funzione *par_id_to_node*, associa l'id di una partizione al rispettivo nodo e il parametro *PRE-DATA* contiene le informazioni derivanti dalla pre-computazione.
Inoltre in questo modello vengono eseguite tutte le *send non bloccanti* prima delle *receive* in modo da poter inviare tutti i dati senza attendere la sincronizzazione tra i nodi.
Uno schema del modello è mostrato in figura 33b.

Algorithm 6 Modello dello scambio di informazioni

```
1: function sweGPU_MPI(func par_id_to_node, pre_data PRE-DATA)
2:     MPI::Init()
3:     for ogni passo di computazione del programma sweGPU do
4:         //Calcolo dei dati da inviare ad ogni nodo
5:         partition ← PRE-DATA.partition
6:         for ogni block ∈ partition do
7:             for ogni single_neigh ∈ PRE-DATA.neigh(block) do
8:                 if single_neigh.id_partition ≠ partition.id then
9:                     info_to_send ← PRE-DATA.block.info
10:                    node_to_send ← par_id_to_node(partition.id)
11:                    push(list_data_to_send(node_to_send), info_to_send)
12:        for ogni altro nodo node che ha almeno una comunicazione do
13:            MPI_NON_BLOCKING_SEND(list_data_to_send(node))
14:        for ogni altro nodo node che ha almeno una comunicazione do
15:            MPI_RECEIVE(node ,list_data_to_receive)
16:            SAVE(list_data_to_receive)
17:        DO_THE_REAL_JOB_ON_MY_PARTITION
18:        //Le informazioni contenute nei vari blocchi cambiano
19:    MPI::Finalize()
20:    return
```

Essendo l'*overhead* per la ricezione dei bordi (riga 15 algoritmo 6) molto penalizzante, si può pensare di ridurlo introducendo la possibilità di eseguire il calcolo durante il tempo di ricezione per i blocchi che non dipendono dai dati attesi .



(a) Schema delle comunicazioni nella fase di pre-computazione

(b) Schema delle comunicazioni del modello di scambio informazioni

Figura 33: La rappresentazione del modello MPI

6 Lavori Futuri

Variazione sui criteri di suddivisione

L'algoritmo di ripartizione studiato in questa tesi si prefigge di giungere ad una suddivisione accettabile del dominio per poterlo distribuire sui vari nodi di calcolo. In particolare non si è ricercato un algoritmo ottimale ma uno che permetta, con un overhead relativamente basso, di eseguire una suddivisione accettabile in base al bilanciamento delle partizioni e al numero di comunicazioni tra i vari nodi.
Non essendo infatti ancora implementata una versione distribuita di *sweGPU* non si conoscono i fattori penalizzanti della distribuzione, in particolare non si conosce l'overhead introdotto dalle comunicazioni e quello introdotto da un eventuale sbilanciamento.
Quando verrà implementata una versione per architetture multi-nodo si potranno valutare gli overhead dei diversi fattori e quindi ridefinire l'algoritmo in base ai vari parametri che maggiormente influenzano la suddivisione stessa. Se, per esempio, si noterà che l'overhead introdotto dalle comunicazioni è insignificante rispetto a quello introdotto da uno sbilanciamento si potrebbe pensare di forzare l'algoritmo a suddividere il numero di celle perfettamente tra i vari nodi anche a se questo dovesse introdurre un alto numero di comunicazioni.

Migliorie all'algoritmo di partizionamento

Vi sono poi alcune migliorie da apportare all'algoritmo analizzato nel capitolo 4. In particolare:

- Poiché l'ultimo stadio dell'algoritmo, per formare le varie suddivisioni, lavora sull'unione di sotto-domini più piccoli, i vari *tagli* che dividono le varie partizioni possono avere forme abbastanza irregolari. Questa irregolarità può portare ad un numero di comunicazioni leggermente maggiore di quanto ci si aspetti.
  Bisognerebbe quindi implementare una piccola procedura che al termine dell'esecuzione riassegni i blocchi che toccano il *taglio* intelligentemente.

- L'algoritmo è pensato per utilizzare un gran numero di massimi locali come base per descrivere una buona partizione. Anche se nei domini reali, vista la complessità del territorio, i massimi locali sono sempre presenti in un numero abbastanza alto, potrebbe capitare ipoteticamente che questi siano numericamente inferiori al numero di partizioni

volute. Bisognerebbe quindi predisporre un caso limite che permetta di gestire questa configurazione. In un caso del genere si potrebbe pensare, ad esempio, di suddividere il dominio in base alla posizione dei blocchi, visto che un basso numero di massimi locali comporta domini equamente distribuiti.

Le modifiche al modello MPI

Come già spiegato precedentemente la versione definitiva del programma *sweGPU* non è stata ancora del tutto sviluppata. Di conseguenza il modello *MPI* dovrà evolversi rispettando le caratteristiche dell'architettura, che si definirà nel corso dello sviluppo della versione distribuita del programma.
Oltre a questa evoluzione il modello *MPI*, dovrà prevedere la possibilità di svolgere comunicazioni considerando anche le possibili multi GPU installate su uno stesso nodo.
Bisognerà inoltre implementare un semplice algoritmo che permetta ai nodi di ricostruire l'ordine delle informazioni ricevute.
Si può pensare ad un semplice ordine nel salvataggio dei dati da inviare. Ad esempio, le informazioni dei bordi dei vari blocchi si possono memorizzare nell'array di dati da inviare, seguendo l'ordine degli identificativi dei blocchi stessi. I dati di uno stesso blocco si possono poi organizzare a seconda della posizione del bordo[7].

[7]Nord per primo, Est per secondo etc.

Conclusione

Vista la necessità di portare il progetto *sweGPU* su un'architettura distribuita, questa tesi ha studiato un primo approccio alla suddivisione del dominio e un modello base per la comunicazione dei dati tra i vari nodi.

In particolare, la suddivisione del dominio, per essere ottimale, dovrebbe cercare di ridurre le comunicazioni e bilanciare il carico tra i diversi nodi. Per ottenere questo risultato si è sviluppato un algoritmo, in varie versioni, che si basa sui concetti di *albero di copertura minimo* e di *massimo locale*.
Tutte le diverse versioni dell'algoritmo hanno garantito dei buoni risultati, sia dal punto di vista delle comunicazioni sia da quello del bilanciamento.
Inoltre, a seconda della versione, è possibile ottimizzare un parametro a dispetto di un altro. Questo permette, una volta che sarà realizzata una versione distribuita *sweGPU*, di utilizzare l'algoritmo che ottimizza il parametro che porta un overhead maggiore nel caso reale.
L'algoritmo di ripartizione studiato in questa tesi, potrebbe essere una buona base per la suddivisione del dominio e visti i risultati incoraggianti, si può supporre che la versione multi-nodo di *sweGPU* porterà dei reali miglioramenti sui tempi di esecuzione.

Si è studiato inoltre un modello per la comunicazioni MPI tra i vari nodi. Il modello si compone di una fase di pre-computazione, che consente di assegnare i dati dei vari sotto-domini tra i nodi, e una di comunicazione che permette lo scambio di informazioni essenziali tra tutti i nodi.
L'implementazione del modello è basata sulla libreria *OpenMPI* e utilizza comunicazioni sincrone, asincrone e collettive.
Il modello è stato testato con dati fittizi[8] per accertarsi dell'effettivo funzionamento.
Sono stati anche calcolati i tempi di trasmissione tra i threads di una stessa macchina ma, come prevedibile, sono risultati infinitesimali.
Bisognerà quindi aspettare un modello reale per testare l'effettivo costo delle comunicazioni.
Il modello studiato comunque, permette di ammortizzare l'eventuale overhead dello scambio di messaggi grazie all'utilizzo di comunicazioni asincrone. Infatti, se necessario, durante i tempi di attesa si potranno eseguire le operazioni che non necessitano delle informazioni derivanti da altri nodi.

[8] Che simulano l'effettivo carico di dati scambiato tra i vari nodi

Riferimenti bibliografici

[1] Graphviz: An open source graph visualization software. http://www.graphviz.org/.

[2] Metis: a set of serial programs for partitioning graphs, partitioning finite element meshes, and producing fill reducing orderings for sparse matrices. http://glaros.dtc.umn.edu/gkhome/metis/metis/overview/.

[3] Open mpi: Open source high performance computing. https://www.open-mpi.org/.

[4] Mpi: A message-passing interface standard, 2015.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill, 2009.

[6] George J. Grevera. Distance transform algorithms and their implementation and evaluation.

[7] Peter S. Pacheco. A user's guide to mpi, 1998.

[8] Alessia Ferrari Paolo Mignosa Federico Prost Alessandro Dal Palù Pier Paolo Albertoni Renato Vacondio, Francesca Aureli. Un modello 2d-swe parallelo e multi-risoluzione, 2016.

[9] Keijo Ruohonen. Graph theory, 2013.