Università del Salento

Dipartimento di Ingegneria dell'Innovazione

Corso di Laurea Magistrale in Ingegneria Informatica

DOCUMENTAZIONE DI PROGETTO

Corso di Algoritmi Paralleli

Variante dell'Algoritmo DNS con n^2r ***processori con*** $1 \leq r \leq n$

Docenti

Prof. Massimo Cafaro

Autore

Culcea Cezar Narcis

Matricola n° 20086225

ANNO ACCADEMICO 2023/2024

Indice

1 Abstract **3**

2 Fondamenti **4**

2.1 Cannon 4

2.1.1 Pseudocodice 6

2.2 DNS 7

2.2.1 Pseudocodice 8

3 DNS Variant **9**

3.1 Algoritmo 10

3.2 Pseudocodice 12

3.3 Analisi teorica delle prestazioni 13

3.4 Implementazione 14

4 Testing **19**

4.1 Utility Script 19

4.2 Simulazione 20

4.3 Controllo correttezza 20

4.4 Benchmark 21

4.5 Analisi sperimentale delle prestazioni 22

5 Conclusione **23**

Capitolo 1

Abstract

Il problema della moltiplicazione matriciale è di vitale importanza in molti ambiti scientifici. Riuscire a moltiplicare matrici di grandi dimensioni in tempi ragionevoli è una sfida quanto mai attuale.

La moltiplicazione matriciale sequenziale consta di n^2 problemi di taglia n per una complessità pari a $O(n^3)$.Tra le varie soluzioni proposte, sicuramente quelle basate su calcolo parallelo vengono in aiuto. Tra queste il **DNS** che riesce ad eseguire la moltiplicazione in $O(log(n))$, a patto di riuscire ad avere un numero di processori enorme (n^3). Numerose versioni alternative del DNS sono state proposte con lo scopo di abbassare i requisiti sul numero di processori, senza compromettere eccessivamente la complessità. Questo progetto tratta una versione del DNS che utilizza n^2r con $1 \leq r \leq n$ processori.

Capitolo 2

Fondamenti

Ci poniamo di fronte al problema di eseguire una moltiplicazione tra due matrici $A, B \in R^{n \times n}$ per ottenere una matrice $C \in R^{n \times n}$ risultante. In particolare, la matrice C è ottenuta risolvendo n^2 sottoproblemi, uno per ogni elemento. Sia $c_{i,j}$ l'elemento della matrice risultante collocato nella riga i e nella colonna j, abbiamo che i singoli elementi si calcolano con la seguente operazione.

$$c_{i,j} = \sum_{l=0}^{n-1} a_{i,l} b_{l,j}$$

Al fine di comprendere al meglio la variante che verrà proposta nei capitoli successivi, è bene analizzare i due algoritmi su cui si basa. Entrambi partono da un layer di n^2 processori posizionati su una griglia indicizzata su $0 \leq i < n$ e $0 \leq j < n$, in modo che ogni processore sia in possesso degli elementi $a_{i,j}$ e $b_{i,j}$ delle rispettive matrici A, B.

2.1 Cannon

L'algoritmo di Cannon generalizzato [1] , consente di moltiplicare due matrici di taglia n^2 su un numero di processori pari a m^2, con $n \bmod m = 0$. In questa versione, ogni processore contiene una sottomatrice di taglia $(n/m)^2$.

La versione che verrà utilizzata nell'algoritmo finale fa considera $m = n$, quindi ogni processore memorizza un solo valore di A e di B. Di conseguenza eventuali calcoli saranno moltiplicazioni scalari e non matriciali.

Il fulcro dell'algoritmo è lo shift delle righe di A e delle colonne di B. Attraverso questa procedura, ogni singolo processore, vede tutti gli n valori delle matrici di input necessari a calcolare l'elemento c finale.

Prima di iniziare il calcolo, le righe 1-6 si occupano di effettuare l'allineamento degli elementi:

- nella matrice A, la riga $i - esima$ viene shiftata di i posizioni a sinistra
- nella matrice B, la colonna $j - esima$ viene shiftata di j posizioni in alto

Una volta effettuato l'allineamento, ogni processore è in possesso della prima coppia di elementi $a_{i,j}$ e $b_{i,j}$ da moltiplicare. Dopo il primo calcolo, in un ciclo di n iterazioni, ogni processore invia il suo elemento a al processore a sinistra e lo riceve da destra. Analogamente farà con b, inviato al processore in alto e ricevuto dal processore in basso. Successivamente moltiplicherà nuovamente i due elementi e li sommerà al $c_{i,j}$. Dopo $n-1$ iterazioni, ogni processore avrà il valore c finale.

Comunicazioni

Consideriamo t_s tempo per iniziare una comunicazione e t_w tempo per inviare una *word*. Dato che questa versione invia singoli elementi e l'allineamento è fatto in maniera concorrente da n^2 processori, il tempo speso per effettuare le due send è pari a $2(t_s + t_w)$. Per quanto riguarda la fase dello *shift multiply*, ogni processore esegue in maniera concorrente il proprio ciclo, che ha una complessità pari a $(n-1) * 2(t_s + t_w)$. Sommando tutto e portando in notazione *Big-O*, otteniamo un costo pari a $O(n)$.

2.1.1 Pseudocodice

Algorithm 1 Cannon matrix multiply con n^2 processori

```
1: procedure cannonMultiply(A,B,C):
2: //Allineamento iniziale
3: for i = 0 to n − 1 do
4:     A(i,j) ← A(i, (i+j) mod n) //left circular shift
5:     B(i,j) ← B((i+j) mod n, j) //up circular shift
6: end for
7: C(i,j) ← A(i,j)*B(i,j)
8: //shift multiply
9: for l = 1 to n − 1 do
10:    A(i,j) ← A(i, (i+1) mod n)
11:    B(i,j) ← B((i+1) mod n, j)
12:    C(i,j) ← C(i,j) + A(i,j)*B(i,j)
13: end for
14: end procedure cannonMultiply
```

2.2 DNS

L'algoritmo **DNS** [1] per la moltiplicazione di matrici quadrate sfutta una decomposizione 3D. Sappiamo che moltiplicare due matrici quadrate di taglia n^2, comporta eseguire n^3 operazioni. L'idea è di assegnare ogni operazione ad un processore.

L'algoritmo inizia distribuendo le matrici A e B nel layer a livello 0, formato da n^2 processori. Consideriamo ora il generico processore del layer 0 come $P(i, j, 0)$ con $0 \leq i, j < n$. Al termine, varrà la seguente relazione

$$A(i, j, 0) = a_{i,j}$$

$$B(i, j, 0) = b_{i,j}$$

Dopo questo passaggio, vi è la distribuzione delle $j - esime$ colonne di A nel layer j e la distribuzione delle $i - esime$ righe di B nel layer i. Al termine varrà la seguente relazione

$$A(i, j, j) = A(i, j, 0) = a_{i,j}$$

$$B(i, j, i) = B(i, j, 0) = b_{i,j}$$

Infine, l'ultimo passaggio è la distribuzione degli elementi a lungo le righe del proprio layer e degli elementi b lungo le colonne del proprio layer. Al termine varrà la relazione

$$A(i, l, j) = a_{i,j} \text{ con } 0 \leq l < n$$

$$B(l, j, i) = b_{i,j} \text{ con } 0 \leq l < n$$

Dopo aver distribuito i valori, ogni processore può procedere con la moltiplicazione del proprio a e b. Infine, avremo che $c_{i,j} = \sum_{k=0}^{n-1} C(i, j, k)$

Comunicazioni

Calcoliamo il costo delle comunicazioni:

- La prima fase dell'algoritmo invia i valori presenti nei processori al layer 0, negli altri n layer. Questa operazione si può effettuare con una *Broadcast* lungo la dimensione Z. Dato che i processori inviano sia il valore $a_{i,j}$ che $b_{i,j}$, il costo delle comunicazioni di questa parte sarà $2log(n)$

- La seconda parte replica i valori di a e di b rispettivamente lungo le righe e lungo le colonne, di conseguenza anche questa parte avrà un costo pari a $2log(n)$, dovuto alle due *Broadcast*

- Infine per ottenere il risultato è necessario fare una *Reduce* lungo la dimensione Z, con un costo ancora una volta pari a $log(n)$

Il costo totale risulta quindi $5log(n) = O(log(n))$.

2.2.1 Pseudocodice

Algorithm 2 DNS matrix multiply con n^3 processori

1: **procedure** dns(A,B,C):
2: //Consideriamo A(i,j,0)=$a_{i,j}$ e B(i,j,0)=$b_{i,j}$
3: **for** k=0 **to** n-1 **do**
4: A(i,k,k) ← A(i,k,0) //Inviamo le colonne k di A al livello k
5: B(k,j,k) ← B(k,j,0) //Inviamo le righe k di B al livello k
6: **end for**
7: **for** l=0 **to** n-1 **do**
8: A(i,l,j) ← A(i,j,j) //Replichiamo A lungo le righe i
9: B(l,j,i) ← B(i,j,i) //Replichiamo B lungo le colonne j
10: **end for**
11: C(i,j,k) = A(i,j,k)*B(i,j,k) //Singola operazione
12: **for** k=1 **to** n-1 **do**
13: C(i,j,0) = C(i,j,0) + C(i,j,k) //Lungo l'altezza Z sono presenti tutte le n operazioni che portano ad un $c_{i,j}$
14: **end for**
15: **end procedure** dns

Capitolo 3

DNS Variant

L'algoritmo in questione [1] nasce come una variante del DNS con lo scopo di moltiplicare due matrici quadrate A e B di taglia $n \times n$. L'algoritmo richiede un numero di processori pari a $n^2 r$ con $1 \leq r \leq n$. Inoltre è richiesto che r sia un divisore di n.

L'idea è ancora una volta quella di distribuire le matrici A e B sugli n^2 processori $P(i, j, 0)$ con $0 \leq i, j < n$. A differenza del DNS il numero di layer ora è r, non più n.

Il passo successivo è suddividere le matrici n^2 in sottomatrici di taglia $(n/r)^2$. Consideriamo ora le matrici A e B come matrici di taglia r^2 formate da sottomatrici di dimensione $(n/r)^2$. Questa visione alternativa dei processori, trasforma la struttura fisica di $n^2 r$ processori in una logica di r^3 iperprocessori.

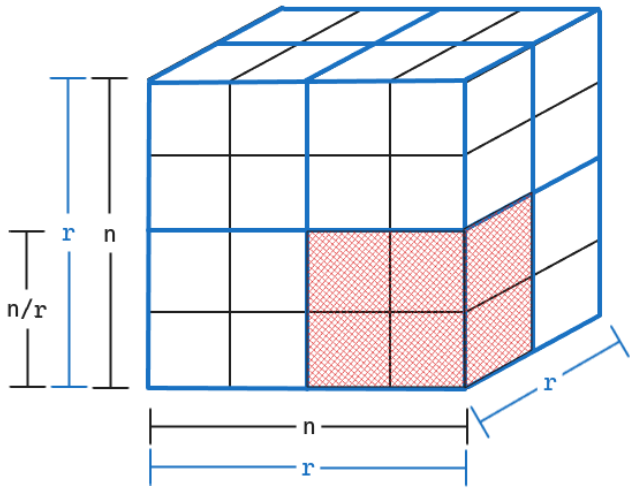

Figura 3.1: Una rappresentazione dell'ipercubo ($n = 4, r = 2$): in **nero** la griglia di processi fisici, in **blu** la griglia di iperprocessori logici, in **rosso** evidenziato un superprocessore.

3.1 Algoritmo

L'algoritmo inizia distribuendo le matrici A e B sugli n^2 processori al layer 0 in modo che ogni processore in posizione $(i, j, 0)$ abbia gli elementi $a_{i,j}$ e $b_{i,j}$. Dopo questa fase, avviene una distribuzione dei dati negli r layer, con un ragionamento analogo a quello del **DNS**, inviando però le righe e le colonne della matrice di iperprocessori r^2.

In particolare, considerando la matrice degli iperprocessori $\alpha(s, t, 0)$ e $\beta(s, t, 0)$ con $0 \leq s, t < r$:

- inviamo la colonna $t - esima$ della matrice α al layer t
- inviamo la riga $s - esima$ della matrice β al layer s

Dopo questo passaggio, ci occupiamo di replicare iperprocessori:

- replichiamo l'iperprocessore $\alpha(s, k, k)$ sulla propria riga $\alpha(s, l, k)$ con $0 \leq l < r$
- replichiamo l'iperprocessore $\beta(k, t, k)$ sulla propria colonna $\beta(l, t, k)$ con $0 \leq l < r$

Di fatto, questo procedimento è identico alla distribuzione dei dati nel DNS, solo che replichiamo negli r layer, sottomatrici di taglia $(n/r)^2$ invece di singoli elementi. In particolare se $r = n$, le sottomatrici che replichiamo hanno taglia $(n/n)^2 = 1^2$, quindi sono singoli elementi e si ritorna nello stesso caso del **DNS**.

L'algoritmo prosegue moltiplicando le sottomatrici in possesso degli iperprocessori e calcolando $\chi(s, t, k)$, la sottomatrice di C in possesso dell'iperprocessore (s, t, k).

Un iperprocessore di fatto è un'insieme di processori, nello specifico $(n/r)^2$. In particolare abbiamo il seguente mapping di coordinate (riga, colonna, layer):

- i $\rightarrow \lfloor i/r \rfloor$
- j $\rightarrow \lfloor j/r \rfloor$
- k $\rightarrow$ k

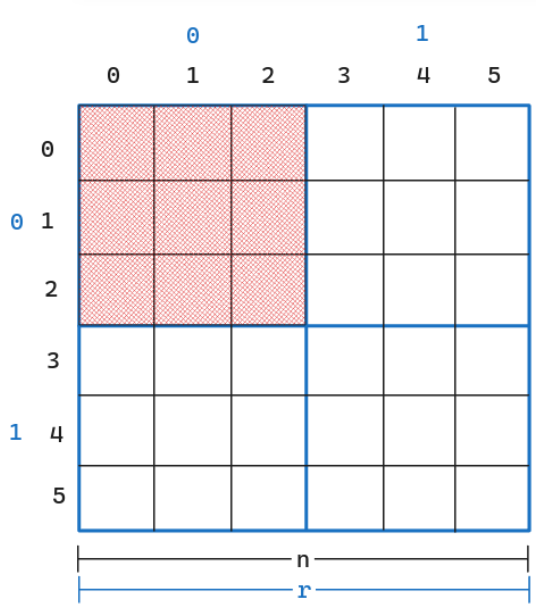

Figura 3.2: Mapping delle coordinate processore-iperprocessore ($n = 6, r = 2$)

La moltiplicazione che avviene all'interno del iperprocessore viene eseguita tramite l'algoritmo di Cannon, in particolare due matrici taglia $(n/r)^2$ vengono moltiplicate da una matrice di taglia $(n/r)^2$ di processori.

Infine i processori del layer 0 si occupano di raccogliere i valori dagli altri layer, implementando di fatto la seguente relazione

$$\chi(s,t,0) = \sum_{k=0}^{r-1} \chi(s,t,k) \rightarrow C(i,j,0) = \sum_{k=0}^{r-1} C(i,j,k)$$

.

Al termine, gli iperprocessori al layer 0 avranno le sottomatrici χ finali e di conseguenza i processori avranno gli elementi di C.

3.2 Pseudocodice

Algorithm 3 DNS Variant con n^2r processori

1: **procedure** dnsVariant(A,B,C):
2: //Utiliziamo la notazione α, β, χ per indicare le sottomatrici (iperprocessori) come descritto sopra
3: **for** $k = 0$ **to** $r - 1$ **do**
4: α(s, k, k) $\leftarrow$ α(s, k, 0)//Copiamo i valori degli iperproc. della colonna k sul layer k
5: β(k, t, k) $\leftarrow$ β(k, t, 0) //Copiamo i valori degli iperproc. della riga k sul layer k
6: **end for**
7: **for** $l = 0$ **to** $r - 1$ **do**
8: α(s, k, k) $\leftarrow$ α(s, l, k) //Replichiamo gli iperproc. sulla propria riga s
9: β(k, t, k) $\leftarrow$ β(l, t, k) //Replichiamo gli iperproc. sulla propria colonna t
10: **end for**
11: //Allineamento iniziale da fare in parallelo all'interno di ogni sottomatrice, i,j,k sono da considerarsi relativi alle sottomatrici $(n/r)^2$ e non alla matrici A, B complete
12: **for** $i = 0$ **to** $n/r - 1$ **do**
13: A(i, j, k) $\leftarrow$ A(i, (i+j) mod (n/r), k) //left circular shift
14: B(i, j, k) $\leftarrow$ B((i+j) mod (n/r), j, k) //up circular shift
15: **end for**
16: C(i, j, k) $\leftarrow$ A(i, j, k)*B(i, j, k)
17: //shift multiply
18: **for** $l = 1$ **to** $n/r - 1$ **do**
19: A(i, j, k) $\leftarrow$ A(i, (i+1) mod (n/r), k)
20: B(i, j, k) $\leftarrow$ B((i+1) mod (n/r), j, k)
21: C(i, j, k) $\leftarrow$ C(i, j, k) + A(i, j, k)*B(i, j, k)
22: **end for**
23: //Reduce finale da fare in parallelo
24: **for** $k = 1$ **to** $r - 1$ **do**
25: C(i, j, 0) = C(i, j, 0) + C(i, j, k) //Lungo l'altezza sono presenti tutte le n operazioni che portano ad un $c_{i,j}$
26: **end for**
27: **end procedure** dnsVariant

3.3 Analisi teorica delle prestazioni

Iniziamo con l'analisi delle comunicazioni [2], considerando ancora una volta t_s tempo per iniziare la comunicazione e t_w tempo necessario per inviare una *word*. Le righe 1-10 corrispondono all'algoritmo DNS e contengono in totale 4 *Broadcast*, due relative alle sottomatrici α e due alle sottomatrici β. Essendo esse inviate logicamente agli iperprocessori, il tempo totale di comunicazione è $4(t_s + t_w)log(r)$

Le righe 11-22 corrispondono all'algoritmo Cannon eseguito su sottomatrici di taglia $(n/r)^2$. La sua complessità risulta quindi $2(t_s + t_w)\frac{n}{r}$.

Infine le righe 23-26 corrispondono ad una *Reduce* lungo gli r layer, portando ad una complessità pari a $(t_s + t_w)log(r)$.

La comunicazione totale risulta essere:

$$\text{COMM} = (t_s + t_w)(5log(r) + 2\frac{n}{r}) = O(log(r) + \frac{n}{r})$$

Su p processori, il calcolo di ogni processore per moltiplicare le matrici quadrate $n \times n$ sarà:

$$\text{CALC} = \frac{n^3}{p}$$

Il T_p tempo parallelo, considerando p processori, è il seguente:

$$T_p = \text{CALC} + \text{COMM} = \frac{n^3}{p} + (t_s + t_w)(5log(\frac{p}{n^2}) + 2\frac{n^3}{p}) = O(\frac{n^3}{p} + log(\frac{p}{n^2}))$$

Calcoliamo ora l'overhead:

$$T_o = pT_p - T_1 = p * (\frac{n^3}{p} + (t_s + t_w)(5log(\frac{p}{n^2}) + 2\frac{n^3}{p})) - n^3 = (t_s + t_w)(5p * log(\frac{p}{n^2}) + 2n^3)$$

L'efficienza risulta quindi:

$$E = \frac{1}{1 + \frac{T_o}{T_1}} = \frac{1}{1 + \frac{(t_s+t_w)(5p*log(\frac{p}{n^2})+2n^3)}{n^3}} = \frac{1}{1 + (t_s + t_w)(\frac{5p*log(\frac{p}{n^2})}{n^3} + 2)}$$

Per matrici di dimensioni tendenti ad infinito, notiamo che l'efficienza ha un upper-bound pari al seguente risultato:

$$\lim_{n\to\infty} E = \lim_{n\to\infty} \frac{1}{1 + (t_s + t_w)(\frac{5p*log(\frac{p}{n^2})}{n^3} + 2)} = \frac{1}{1 + 2(t_s + t_w)}$$

L'isoefficienza reale infine risulta la seguente:

$$T_1 > CT_o \Rightarrow n^3 > C(t_s + t_w)(5p * log(\frac{p}{n^2}) + 2n^3) \Rightarrow n^3 > Cp * log(\frac{p}{n^2})$$

3.4 Implementazione

Il codice fa uso delle funzioni della libreria custom `inOutUtils.h` per generare le matrici, leggerle, scriverle e stamparle sul terminale. La libreria utilizza un seed di generazione statico per garantire la riproducibilità.

Il file `dnsVariant.c` implementa l'algoritmo esposto nel capitolo precedente e deve essere eseguito con $p = n^2r$ processori. Esso prende come parametro a linea di comando la problem size n.

Prima di partire con l'esecuzione esegue un controllo di divisibilità tra n ed r, in modo che le sottomatrici siano della stessa taglia.

Attraverso la funzione `void createCommunicators(struct Communicators* comms, int* dims, int* periods, int* cartRank, int* coords, int discriminanteColore);` vengono creati tutti i comunicatori necessari all'intera esecuzione. La funzione li crea a partire da un comunicatore cartesiano di taglia $n * n * r$. Solitamente i processi sono indicizzati nella griglia secondo le seguenti coordinate *(profondità, riga, colonna)*. Il codice effettua un mapping sulle macro X,Y,Z, trattando le coordinate come se fossero cartesiane:

- X -> corrisponde all'indice della colonna
- Y -> corrisponde all'indice della riga
- Z -> corrisponde al livello di processori

Il processo di *rank 0* si occupa allora di leggere le matrici di input dai file (*matrixA.bin* e *matrixB.bin*) ed effettuare uno scatter a tutti i processi del layer 0, in modo che ognuno abbia l'elemento della matrice A e B corrispondente alle proprie coordinate.

Seguendo lo pseudocodice, considerando la matrice degli iperprocessori r^2, ogni colonna di iperprocessori di indice x dovrebbe inviare i valori di A alla colonna di indice x e layer x. In maniera speculare ogni riga di indice y dovrebbe inviare i valori di A alla riga di indice y e layer y.

Per velocizzare la coordinazione tra i processi, questo passo viene sostituito con una *Broadcast* da parte dei processi del layer 0, lungo la dimensione Z.

```
/****************************** BCAST A Columns ***********************************/
MPI_Bcast(&localA, 1, MPI_INT, 0, comms->commZsingleDim);
/****************************** BCAST B Rows ***********************************/
MPI_Bcast(&localB, 1, MPI_INT, 0, comms->commZsingleDim);
```

Al termine di questo passaggio alternativo, il risultato è analogo negli iperprocessori di nostro interesse.

Successivamente, avviene la distribuzione delle sottomatrici contenute in un iperprocessore, lungo la propria riga per i valori di A e lungo la propria colonna per i valori di B.

Per ricopiare le sottomatrici sulle rispettive righe o colonne, facciamo uso dei comunicatori per righe e per colonne. In particolare notiamo che, all'interno del comunicatore per riga o per colonna, i processori con che hanno lo stesso valore $rowIndex\%(n/r)$, dovranno ricevere lo stesso valore. In particolare questo può essere usato come colore per creare un sottocomunicatore della riga, contenente r processi, in modo da distribuire i valori in una chiamata *Broadcast*.

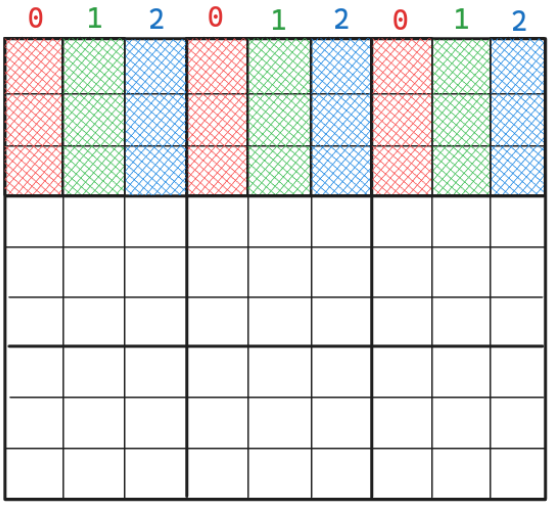

Figura 3.3: Caso d'uso con $n = 9$ ed $m = 3$. Sono evidenziate le sottomatrici ed i processori che hanno lo stesso colore e che dovranno contenere gli stessi valori. Il discorso si estende facilmente alle colonne.

La broadcast dovrà comunque essere fatta dai processi che appartengono alla sottomatrice che ha indice pari al layer in cui si trova.

```
/****************************** BCAST A values over their rows in each layer, if layer
MPI_Bcast(&localA, 1, MPI_INT, cartCoords[Z], comms->commSubMatrixX);
/****************************** BCAST B  values over their cols in each layer, if laye
MPI_Bcast(&localB, 1, MPI_INT, cartCoords[Z], comms->commSubMatrixY);
```

Arrivati a questo punto, termina la distribuzione dei valori ed inizia la sezione relativa al calcolo vero e proprio. Il ragionamento si sposta all'interno di ogni iperprocessore che

esegue l'algoritmo di Cannon. Il fulcro di questa sezione è la funzione che identifica i processori adiacenti, all'interno della sottomatrice con wrap around, partendo dal rank interno al layer in cui si trovano.

La funzione che implementa tale logica è la seguente.

```
void findAdjacentCells(int index, int n, int m, int distX, int distY,
                                                    AdjacentCells *adj){
    int sm_size = n / m;  // submatrix size
    int x = index % n;    // coord x cell
    int y = index / n;    // coord y cell

    // Get offset due to submatrix
    int sub_x = x / sm_size * sm_size;
    int sub_y = y / sm_size * sm_size;

    // Get adjacent cells coordinate in submatrix with wrap around
    int up_x = x;
    int up_y = (y - distY + sm_size) % sm_size + sub_y;
    adj->up = up_y * n + up_x;

    int down_x = x;
    int down_y = (y + distY) % sm_size + sub_y;
    adj->down = down_y * n + down_x;

    int left_x = (x - distX + sm_size) % sm_size + sub_x;
    int left_y = y;
    adj->left = left_y * n + left_x;

    int right_x = (x + distX) % sm_size + sub_x;
    int right_y = y;
    adj->right = right_y * n + right_x;
}
```

La funzione identifica le coordinate del processo relative al rank layer e successivamente le coordinate degli iperprocessori della matrice logica r^2 tramite le seguenti relazioni e

sfruttando l'approssimazione della divisione tra interi del linguaggio C:

- $rank \rightarrow (x = rank\%n, y = rank/n)$
- $(x, y) \rightarrow (sub_x = x/sm_{size} * sm_{size}, sub_y = y/sm_{size} * sm_{size})$

Le coordinate degli iperprocessori servono ad identificare l'offset relativo al posizionamento. Per esempio una matrice di processori 6^2 con $r = 2$ forma una griglia $2 * 2$ di iperprocessori, formati da sottomatrici di $3 * 3$ processori.

La prima riga di processori avrà come rank layer i numeri da 0 a 5. I rank interni alle sottomatrici saranno invece $0, 1, 2 - 0, 1, 2$. Utilizzando l'offset degli iperprocessori, riusciamo ad associare correttamente il rank interno alla sottomatrice al rank originario del layer.

Quindi una volta trovati i processi adiacenti nei rank delle sottomatrici, risulta molto semplice riottenere i rank layer originali. Questo consente di effettuare *send* e *recv* in maniera agevole per effettuare l'alignment iniziale e lo *shift-multiply* successivo.

```
/****************************** COMPUTATION ***********************************/
AdjacentCells adj;
int planeRank;
MPI_Comm_rank(comms->commXYplanes, &planeRank);

//Initial alignment
int rigaSubMatrix = cartCoords[Y] % (n/m);
int colonnaSubMatrix = cartCoords[X] % (n/m);
findAdjacentCells(planeRank, n, m, rigaSubMatrix, colonnaSubMatrix, &adj);
MPI_Sendrecv_replace(&localA, 1, MPI_INT, adj.left, 0, adj.right, 0,
    comms->commXYplanes, MPI_STATUS_IGNORE);
MPI_Sendrecv_replace(&localB, 1, MPI_INT, adj.up, 0, adj.down, 0,
    comms->commXYplanes, MPI_STATUS_IGNORE);

//Compute
for(int i=0; i<n/m; i++){
    localC += localA * localB;
    findAdjacentCells(planeRank, n, m, 1, 1, &adj);
    MPI_Sendrecv_replace(&localA, 1, MPI_INT, adj.right, 0, adj.left, 0,
```

```
        comms->commXYplanes, MPI_STATUS_IGNORE);
    MPI_Sendrecv_replace(&localB, 1, MPI_INT, adj.down, 0, adj.up, 0,
        comms->commXYplanes, MPI_STATUS_IGNORE);
}
```

Infine l'algoritmo termina effettuando una reduce lungo l'asse Z, che contiene tutti i valori di C locali, la cui somma fornisce l'elemento finale C che nella matrice risultato ha coordinate pari a quelle del processore che lo contiene.

```
/****************************** REDUCE C LOCALE ***********************************/
int finalC;
MPI_Reduce(&localC, &finalC, 1, MPI_INT, MPI_SUM, 0, comms->commZsingleDim);
```

Il codice prosegue eseguendo una *Gather* della matrice risultato nel processo di rank 0, che ne effettuerà successivamente la scrittura nel file *matrixC_ dnsVariant.bin*. Il processo si occupa inoltre di stampare sul terminale rispettivamente il tempo necessario per leggere le matrici e distribuirle con la scatter ed il tempo per eseguire la parte descritta dallo pseudocodice.

Capitolo 4

Testing

Nonostante l'algoritmo richieda meno processori del *DNS*, il numero richiesto cresce comunque molto velocemente con la problem size. Per evitare problemi è consigliato incrementare il limite di *open files* attraverso il comando `ulimit -n 1000000`.

La simulazione è eseguita su un sistema con le seguenti specifiche:

- OS -> Ubuntu 22.04.04 LTS
- OPENMPI -> versione 4.1.2
- hardware -> AMD Ryzen 9 7900X 12-Core

4.1 Utility Script

E' stato ideato uno shell script per facilitare il building del progetto, le misurazioni sui tempi e la correttezza.

In particolare sono presenti le seguenti modalità di utilizzo:

- `./script.sh -b` -> build del progetto
- `./script.sh -d` -> rimuove i file di input generati e le compilazioni
- `./script.sh -n 6 -m` -> avvia le misurazioni delle prestazioni su due matrici di taglia $6 * 6$ generate randomicamente
- `./script.sh -n 6 -c -p 36` -> esegue l'algoritmo sulle matrici $6*6$, che si suppone siano già presenti con il nome *matrixA.bin* e *matrixB.bin*; calcola la matrice risultante mediante l'algoritmo sequenziale; usando una funzione di hash stabilisce se i due risultati sono uguali.

4.2 Simulazione

Per usare l'algoritmo è necessario effettuare il build. Può essere fatto sia direttamente dal Makefile che tramite lo script con l'opzione -b.

```
gsafractal@gsafractal:~/Scrivania/dnsVariant$ ./script.sh -b
rm -f printMatrix generateMatrix seqMatrixMultiply dns dnsVariant
cc printMatrix.c inOutUtils.c -o printMatrix
cc generateMatrix.c inOutUtils.c -o generateMatrix
cc seqMatrixMultiply.c inOutUtils.c -o seqMatrixMultiply
mpicc dns.c inOutUtils.c -o dns
mpicc dnsVariant.c inOutUtils.c -o dnsVariant _
```

Figura 4.1: Build del progetto

Una volta compilati e necessario creare l'input. Se non si hanno delle matrici si può utilizzare l'eseguibile `./generateMatrix [SIZE]` che si occupa di generare due matrici quadrate della taglia specificata. L'algoritmo *dnsVariant* cerca nella propria directory due matrici chiamate *matrixA.bin* e *matrixB.bin* per effettuare i conti.

Al fine avviare l'esecuzione è consigliato il seguente comando:

```
mpirun -oversubscribe -n [SIZE*SIZE*r] ./dnsVariant [SIZE]
```

Al termine dell'esecuzione, il risultato si troverà nel file *matrixC_ dnsVariant.bin* e per poterlo visualizzare si può usare l'eseguibile *printMatrix* con il seguente comando:

```
./printMatrix [SIZE] [FILENAME]
```

```
gsafractal@gsafractal:~/Scrivania/dnsVariant$ ./generateMatrix 8
Matrices generated and saved in matrixA.bin and matrixB.bin
gsafractal@gsafractal:~/Scrivania/dnsVariant$ mpirun --oversubscribe -n 128 ./dnsVariant 8
  0.027338        0.000843
gsafractal@gsafractal:~/Scrivania/dnsVariant$ ./printMatrix 8 matrixC_dnsVariant.bin
Filename matrixC_dnsVariant.bin - Size 8*8:
Matrix n: 8*8

        232     224     163     202     245     126     180     199
        241     197     216     170     176     150     127     131
        247     243     199     231     223     138     152     176
        148     110     106     164     142     40      145     97
        212     172     151     167     169     108     78      164
        249     196     191     184     209     138     165     146
        320     317     214     267     314     196     181     261
        263     246     216     244     234     148     206     174
```

Figura 4.2: Esecuzione dell'algoritmo

4.3 Controllo correttezza

Supponendo le matrici di input siano già presenti, possiamo usare lo script per effettuare il controllo. Esso effettuerà il calcolo utilizzando prima l'algoritmo sequenziale salvando

il risultato in *matrixC_sequential.bin* e successivamente l'algoritmo parallelo salvando il risultato in *matrixC_dnsVariant.bin*. Infine utilizzando hashing md5 confronterà i file.

```
gsafractal@gsafractal:~/Scrivania/dnsVariant$ ./script.sh -n 8 -c -p 128
------------------------------------------------------------------------------------------------------
Correctness check (Matrix size: 8 * 8)
        Sequential computation ... done
        Parallel computation ... done
Matrices C are equal
Results saved in matrixC_sequential.bin and matrixC_dnsVariant.bin
gsafractal@gsafractal:~/Scrivania/dnsVariant$
```

Figura 4.3: Controllo di correttezza per un caso d'uso

4.4 Benchmark

Il benchmark è effettuato utilizzando lo script con opzione -m. Lo script contiene una variabile denominata *MAX_RUNS* che indica il numero di esecuzioni da effettuare per ottenere il tempo medio.

Inoltre per ogni problem size, l'algoritmo verrà lanciato testando con tutti i numeri di processori ammissibili tra n^2 e n^3. Viene effettuata anche una run dell'algoritmo sequenziale per confrontarne i tempi.

```
gsafractal@gsafractal:~/Scrivania/dnsVariant$ ./script.sh -n 6 -m
Matrix SIZE: 6*6
Matrices generated and saved in matrixA.bin and matrixB.bin
------------------------------------------------------------------------------------------------------
Sequential Algorithm Measurements (Avg. on 50 runs)
        Input Time:   0.000025  -  Calc Time:   0.000001
------------------------------------------------------------------------------------------------------
Parallel Algorithm Measurements (Avg. on 50 runs)
        Procs number: 36 = 6*6*1        --> Input Time:   0.003876  -  Calc Time:   0.000203
        Procs number: 72 = 6*6*2        --> Input Time:   0.009100  -  Calc Time:   0.000432
        Procs number: 108 = 6*6*3       --> Input Time:   0.014807  -  Calc Time:   0.000464
        Procs number: 216 = 6*6*6       --> Input Time:   0.020397  -  Calc Time:   0.000755
```

Figura 4.4: Misurazione tempi

Questi vengono salvati in un file csv all'interno della directory *measurements*.

	A	B	C
1	Measurements matrix SIZE 6 * 6		
2	Proc	Input Time	Calc Time
3	1	0.000025	0.000000
4	36	0.003876	0.000203
5	72	0.009100	0.000432
6	108	0.014807	0.000464
7	216	0.020397	0.000755

Figura 4.5: Esempio di un file CSV creato

4.5 Analisi sperimentale delle prestazioni

Dato l'elevato numero di processori richiesto, le esecuzioni sono tutte effettuate tramite *oversubscribing*, di conseguenza i tempi misurati sono molto più elevati di quelli che si otterrebbero su una macchina con il giusto numero di processori.

Tabella 4.1: Tabella delle prestazioni per la moltiplicazione di due matrici taglia 10^2. $T_1 = 3 * 10^{-6}$

N. Procs	T_p	Ideal T_p	Ψ	Efficienza	Karp-Flatt
100	8.37E-04	3.00E-08	3.58E-03	3.58E-05	281.81
200	1.06E-03	1.50E-08	2.84E-03	1.42E-05	353.43
500	2.82E-03	6.00E-09	1.06E-03	2.13E-06	940.88
1000	2.08E-02	3.00E-09	1.44E-04	1.44E-07	6,939.94

La tabella è formata dalle seguenti colonne:

- T_p: tempo misurato sperimentalmente
- Ideal T_p: tempo parallelo ottenuto supponendo overhead zero $T_p = T_1/p$
- Ψ: Speedup calcolato come $\Psi = T_1/T_p = pT_p/(T_o + T_1)$
- Efficienza: Calcolata come $Eff = \Psi/p$
- metrica di Karp-Flatt: calcolata come $(1/\Psi - 1/p)/(1 - 1/p)$

I valori ottenuti sperimentalmente non rispecchiano l'analisi teorica effettuata in precedenza. In particolare notiamo subito una grande differenza tra i tempi attesi e quelli reali.

Uno speedup ideale su p processori dovrebbe essere p. Ciò intuitivamente indica quanti processori stanno svolgendo lavoro utile. Nel nostro caso i valori sono ben sotto l'1. Effettuando un'analisi teorica dello Speedup, si nota la presenza dell'overhead al denominatore. Quindi molto probabilmente è l'eccessivo oversubscribe a minare le nostre prestazioni.

Conseguentemente, anche l'efficienza, direttamente legata allo speedup ne risente. Infine un'ulteriore conferma arriva dalla metrica di Karp-Flatt.

Capitolo 5

Conclusione

L'algoritmo implementato, anche se non confermato sperimentalmente a causa dei limitati mezzi, si dimostra teoricamente una buona variante dell'algoritmo DNS, permettendo di ridurre notevolmente il numero di processori richiesti, in cambio di un aumento di complessità.

Questa variante permette di effettuare il calcolo avendo delle comunicazioni pari a $O(log(r) + \frac{n}{r})$ invece del classico $O(log(n))$ del DNS.

Al variare di r si hanno i due casi limite:

- $r = 1$ -> $O(log(1) + n) = O(n)$ (Cannon con n^2 procs)
- $r = n$ -> $O(log(n) + 1) = O(log(n)$ (DNS)

Nonostante l'algoritmo si presti a risolvere il problema di moltiplicazione matriciale con un requisito sui processori più accessibile rispetto al DNS, esso resta comunque molto elevato, impedendo di fatto la moltiplicazione di matrici di dimensione estremamente grande.

Probabilmente, dal punto di vista pratico, una versione del DNS con r^3 processori con $1 \leq r \leq n$ risulterebbe molto più efficace.

Bibliografia

[1] David Nassimi Eliezer Dekel e Sartaj Sahni. «Parallel Matrix and Graph Algorithms». In: *Society for Industrial and Applied Mathematics* (1981).

[2] Anshul Gupta e Vipin Kumar. «Scalability of Parallel Algorithms for Matrix Multiplication». In: *Department of Computer Science, University of Minnesota, Minneapolis* (1994).