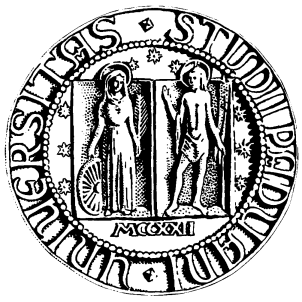
UNIVERSITÀ DEGLI STUDI DI PADOVA



FACOLTÀ DI SCIENZE STATISTICHE

CORSO DI LAUREA SPECIALISTICA IN STATISTICA E INFORMATICA

TESI DI LAUREA

DATA PARALLELISM PER PROCESSORI MULTICORE

VIRTUAL PARALLELISM SU CPU MULTICORE

Relatore : Ch.mo Prof. Mauro Migliardi

Candidato : Alberto Cavalin / 528086-STF

ANNO ACCADEMICO 2007–2008

Indice

1 Il linguaggio **3**

1.1 La macchina di riferimento 3

1.2 Le variabili 4

1.2.1 Dichiarazione 5

1.2.2 Inizializzazione e filtering 6

1.3 Stato di attività dei PE 7

1.3.1 Funzioni di gestione dello stato locale 7

1.3.2 Funzioni di gestione dello stato globale 8

1.4 Operatori ed espressioni 8

1.4.1 Operatori aritmetici 9

1.4.2 Operatori di incremento/decremento 9

1.4.3 Operatori logici 9

1.4.4 Operatori relazionali 10

1.4.5 Operatori di riduzione/espansione 10

1.5 Istruzioni di Input/Output 12

1.5.1 Input 12

1.5.2 Output 12

1.5.3 I/O sul singolo PE 12

1.6 Strutture di controllo 12

1.6.1 Struttura condizionale 12
1.6.2 Struttura di ciclo 13
1.6.3 Struttura di stato 14
1.7 Primitive di comunicazione fra i processori 14
1.7.1 La primitiva `shift` 14
1.7.2 La primitiva `broadcast` 15
1.8 Funzioni standard 15

2 L'implementazione **17**
2.1 Gestione dei thread e dei task 17
2.2 Task per operatori e gestione dello stato 19
2.3 Task per operatori di riduzione 20
2.4 Task per `shift` 20
2.5 Task per `broadcast` 24
2.6 Confronto con un mapping alternativo 27
2.7 Uso della libreria 29

A Manuale di riferimento (ver.1.2-pacifica) **31**

Bibliografia **65**

Elenco delle figure

1.1 Modello della macchina a dati paralleli. 4
1.2 Disposizione delle celle all'interno della matrice allocata. 5

2.1 Gestione di un'istruzione. 18
2.2 Mapping di default tra thread e celle della griglia. 19
2.3 Shift verso est e relative celle interessate. 20
2.4 Mapping per primitive a direzione verticale `NORTH/SOUTH`. . . . 24
2.5 Ricerca del capo del cluster a partire da una cella. 26
2.6 Mapping "direzionale" e a scacchiera. 27
2.7 Ananlisi dei tempi su matrice 255x255 con 4 Core. 28
2.8 Ananlisi dei tempi maggiorati su matrice 255x255 con 4 Core. . . 29

Introduzione

La generazione di processori Pentium IV ha segnato un cambiamento sostanziale nella legge di Moore, il numero di transistori continua ad aumentare, ma la capacità elaborativa dedicata al singolo thread di esecuzione no.

Le nuove CPU sono dotate di architettura multicore, con il numero di core sul singolo processore in rapido aumento. Questo cambiamento richiede alle applicazioni di diventare intrinsecamente parallele e di adottare una granularità di parallelismo sempre più fine.

Il paradigma data parallel si presta in modo ottimo alla generazione di programmi con un parallelismo a grana molto fine e, allo stesso tempo, tramite il virtual parallelism è possibile modulare dinamicamente tale granularità adattandola a numero e topologia dei core.

L'obiettivo di questa tesi è stato quindi l'implementazione di un linguaggio data parallel per tali architetture cioè, prendendo come riferimento il linguaggio *RPC++*, è stato sviluppato un software per l'esecuzione di programmi scritti in tale linguaggio in modalità virtual parallelism su CPU multicore.

Capitolo 1

Il linguaggio

In questo capitolo vengono illustrati il modello della macchina SIMD (Single Instruction Multiple Data) di riferimento, ed la variante del relativo linguaggio di programmazione *RPC++* (Reconfigurable Parallel C++) originariamente ideato nel 1991 da Boreanaz-Dirosa-Migliardi-Orione.

1.1 La macchina di riferimento

Il programmatore che si cimenta nella scrittura di un programma in *RPC++* deve tenere a mente che sta operando su un calcolatore di tipo SIMD, cioè una particolare macchina che in un solo passo riesce ad operare su tutta la mole di dati.

La macchina di riferimento è costituita da N processori detti PE (Processor Element), interconnessi tra loro tramite una rete a MESH bidimensionale richiusa ai bordi (figura 1.1), nella quale ogni PE può comunicare con gli adiacenti secondo le quattro direzioni cardinali (Nord, Est, Sud e Ovest).

Ogni PE è dotato di un flag indicante lo stato di attività (PESHORT/PEOPEN) il quale ne abilita od inibisce il funzionamento. Quando viene impartita una sequen-

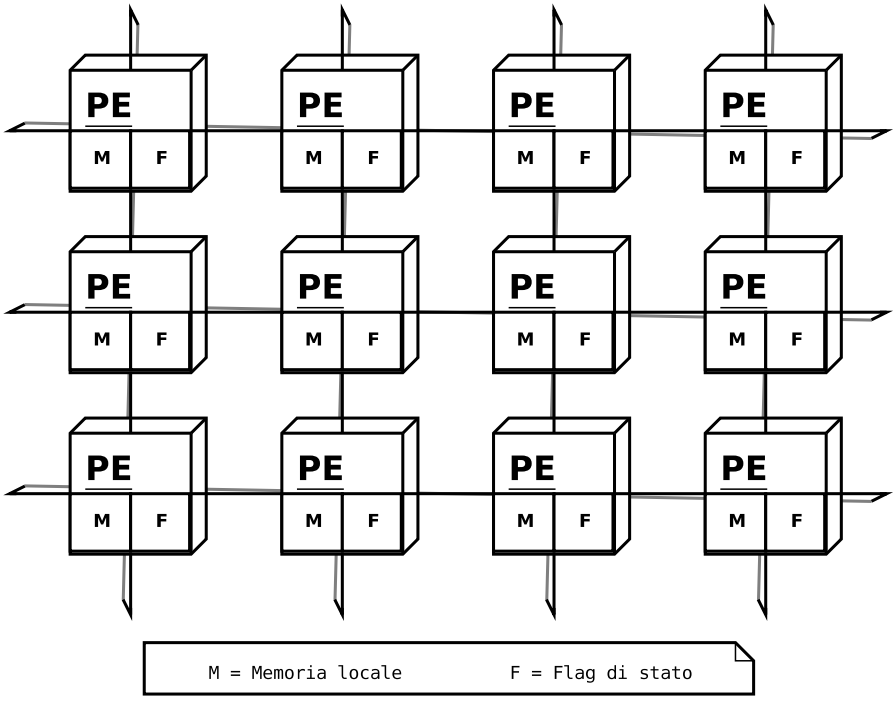


Figura 1.1: Modello della macchina a dati paralleli.

za di istruzioni, a seconda dello stato in cui esso si trova, ogni PE la applica o meno alla propria area di memoria associata. In questo modo, durante l'esecuzione del programma, si potranno avere alcuni PE utilizzati ed altri no.

1.2 Le variabili

Il linguaggio *RPC++* estende il set di variabili del C++ introducendo quelle di tipo parallelo. Quando si crea una tale variabile viene in pratica allocata una griglia virtuale di PE, in cui ogni cella contiene la sua memoria ed un flag di stato.

Le operazioni che vi si effettuano vengono eseguite in parallelo su ogni cella, o meglio, i PE virtuali vengono suddivisi in modo equo tra i core fisici disponibili, e quest'ultimi lavoreranno in parallelo gli $N/\#core$ PE assegnati.

1.2.1 Dichiarazione

Le variabili parallele sono di fatto implementate tramite una classe template, vanno quindi dichiarate attraverso i diversi costruttori a disposizione, opzionalmente specificando anche il tipo primitivo che si intende utilizzare:

```
Poly<int> A;
```

la dichiarazione soprastante alloca in memoria una matrice di interi congiuntamente ad una matrice di valori booleani utilizzando le dimensioni di default.

È altresì possibile specificare le dimensioni della griglia e/o associare un vettore od una matrice ad un singolo PE come riportato in tabella 1.1. In quest'ultimo caso la disposizione delle celle all'interno della matrice avviene come in figura 1.2.

costruttore	**spazio di allocazione**	**dimensioni matrice**
`Poly<> A;`	singolo	di default: RxC
`Poly<> A(l);`	vettore	$(R \cdot l)$xC
`Poly<> A(r,c);`	singolo	rxc
`Poly<> A(r,c,l);`	vettore	$(r \cdot l)$xc
`Poly<> A(r,c,d1,d2);`	matrice	$(r \cdot d1)$x$(c \cdot d2)$
`Poly<> B(A);`	come A	identiche a quelle di A

Tabella 1.1: Costruttori per le variabili parallele.

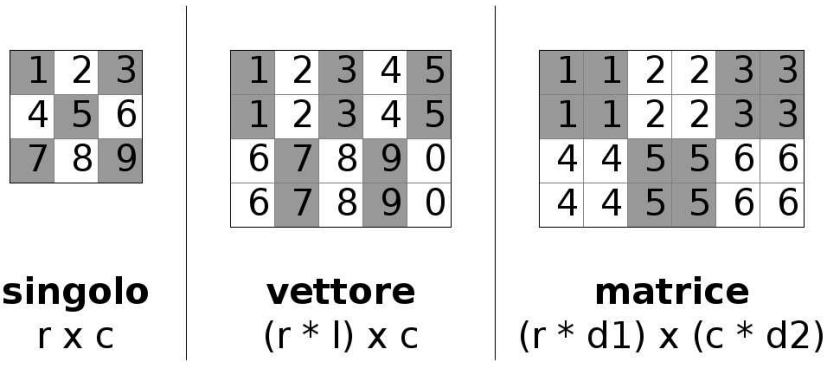


Figura 1.2: Disposizione delle celle all'interno della matrice allocata.

I nomi di varibile proibiti, oltre alle parole chiave del C++, si sono ampliati ed in particolare sono stati aggiunti i seguenti:

```
atleastone
direction
elsewhere, endwhere
polyChunk, polyState, polyTask, polyStateTask
pwhile, pwend, pglobal, pgend, pestatus
Poly, PolyLogic
threadTask
where
```

1.2.2 Inizializzazione e filtering

In *RPC++* non è possibile contemporaneamente dichiarare ed inizializzare una variabile, perciò al momento della sua dichiarazione essa contiene una matrice di valori casuali; la sua inizializzazione avviene in modo del tutto analogo alle variabili classiche, l'istruzione:

```
A=3;
```

fa in modo che tutti i valori della griglia siano inizializzati a 3.

Nel caso di assegnazione di vettori o matrici al singolo PE, è possibile inizializzare solo alcune componenti di questi, ad esempio:

```
Poly<> A(2);  // assegna un vettore di 2 elementi
A[1]=3;       // modifica le componenti di indice 1

Poly<> B(r,c,2,2);  // assegna una matrice 2x2
B[0][1]=3;    // modifica le componenti di indici (0,1)
```

L'operatore `[.]` è detto di *filtering* in quanto permette di selezionare una singola componente delle sottomatrici/sottovettori, ma il suo valore di ritorno è semplicemente una reference all'oggetto stesso. Questo significa che le istruzioni seguenti sono equivalenti a quelle precedenti:

```
B[0][1];      // filtra le componenti di indici (0,1)
B=3;          // modifica le componenti filtrate
```

i riferimenti alle componenti filtrate vengono cancellati subito dopo la prima modifica sulla variabile stessa.

1.3 Stato di attività dei PE

In fase di allocazione della variabile, oltre alla matrice di dati, viene anche allocata una matrice booleana di eguali dimensioni rappresentante lo stato di ogni PE: acceso o spento. A seconda dello stato in cui si trova, il PE esegue o meno le istruzioni che gli sono state impartite.

1.3.1 Funzioni di gestione dello stato locale

Per modificare lo stato dei singoli PE all'interno di una variabile, è possibile applicarle le seguenti funzioni:

funzione	**descrizione**
`set(VAR,r,c);`	abilita il PE di coordinate (r, c)
`unset(VAR,r,c);`	disabilita il PE di coordinate (r, c)
`setall(VAR);`	abilita tutti i PE
`unsetall(VAR);`	disabilita tutti i PE

1.3.2 Funzioni di gestione dello stato globale

Nella libreria esiste uno stack di stati globali che viene aggiornato tramite particolari direttive e/o funzioni. *Quando una variabile deve eseguire un'operazione, prima di tutto controlla se lo stack non è vuoto, e se non lo è controlla che l'elemento alla sua cima sia di dimensioni compatibili con le proprie; in tal caso prende quello come lo stato di riferimento, altrimenti considera quello locale.*

Le direttive per aggiornare lo stack (push/pop) sono le seguenti:

```
where(COND), elsewhere, endwhere
pwhile(COND), pwend
pglobal(VAR), pgend
```

mentre le funzioni per modificare lo stato in cima allo stack sono:

```
gset(r,c), gunset(r,c)
gsetall(), gunsetall()
```

e hanno una funzione analoga alle corrispettive per lo stato locale.

È da far presente che la disabilitazione di un PE tramite `unset()` o `unsetall()` prevale sull'eventuale stato globale di riferimento.

1.4 Operatori ed espressioni

Tutte le variabili parallele messe in gioco dai vari operatori devono essere compatibili tra loro, cioè di pari dimensioni. Ogni operatore applicato a variabili parallele, salvo diversa indicazione, assume lo stesso significato definito dal C++, ma va sottolineato che esso viene applicato ad ogni PE cioè ad ogni cella della matrice di dati.

1.4.1 Operatori aritmetici

Anche per le variabili parallele vi sono a disposizione i classici operatori aritmetici applicabili a qualunque tipo numerico: `+, -, *, /, %`, comprese le loro forme di assegnazione contratta `+=, -=, *=, /=, %=`.

Fissato un operando come una variabile parallela, l'altro può essere un'altra variabile parallela, una variabile *mono* (non parallela), oppure una parallela booleana (`PolyLogic`):

```
C=A+B;        B=A*3;        C=A+(A&&B);
```

il loro valore di ritorno è sempre un'altra variabile parallela.

1.4.2 Operatori di incremento/decremento

Nel caso di variabili parallele gli operatori `++` e `--` hanno una priorità più elevata rispetto a quelli aritmetici, e questo implica che le loro versioni postfisse e prefisse danno lo stesso risultato; ad esempio le due istruzioni seguenti sono da considerarsi equivalenti:

```
C=A*B--;      C=A*--B;
```

1.4.3 Operatori logici

Gli operatori logici a disposizione sono `&&, ||`, e `!`, i quali rappresentano rispettivamente le operazioni AND, OR, e NOT. I primi due sono binari mentre l'ultimo è unario, e tutti e tre restituiscono una variabile parallela booleana di tipo `PolyLogic` rappresentante il risultato.

Qualora il programmatore volesse riutilizzare più volte il risultato di uno di questi operatori basta semplicemente creare una varibile `PolyLogic` come nel seguente esempio:

```
PolyLogic RIS(A && B);      // forma abbreviata
PolyLogic RIS; RIS=A && B;  // forma estesa
```

È possibile operare congiuntamente sia su variabili parallele che mono:

```
int i=0;
...
PolyLogic RIS(A && B || i>0);
```

1.4.4 Operatori relazionali

Gli operatori logici non sono gli unici a restituire una variabile `PolyLogic`, ma anche gli operatori binari: `==, >=, <=, !=, >, <`.

Anche in questo caso, fissato un operando come una variabile parallela, il secondo può essere una variabile parallela, mono, oppure una `PolyLogic`.

1.4.5 Operatori di riduzione/espansione

Gli operatori di riduzione sono caratterizzati dal fatto che, data una variabile parallela, essi restituiscono una variabile mono. Tali operatori sono:

```
min, max   // min/max tra tutti le celle
sum, prod  // somma/prodotto di tutte le celle
```

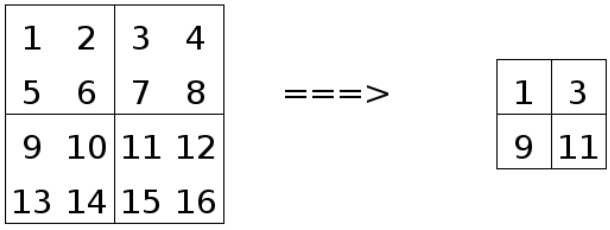
Esiste tuttavia un operatore di riduzione applicabile solo a variabili in cui ad un PE viene associata una matrice od un vettore:

```
Poly<> A(r,c,2,2),    // matrice (2x2) ad ogni PE
       B(r,c);        // var di destinazione
A[0][0];              // seleziona l'elemento (0,0)
B=~A;   /*oppure*/   B=reduce(A);
```

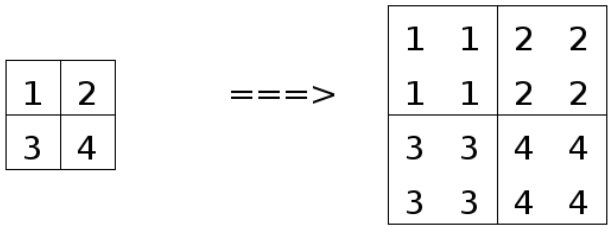
nell’esempio soprastante viene creata una variabile B la cui matrice di dati contiene gli elementi della variabile A di posizione `(0,0)` all’interno dei suoi PE. Per ogni evenienza è stato implementato anche l’operatore di espansione che effettua l’operazione inversa del precedente:

```
Poly<> A(r,c), B(r,c,2,2);
B=expand(A,2,2);           // matrice (2x2) ad ogni PE
```

Per maggiore chiarezza si vedano i semplici esempi seguenti:




```
Poly<> A(2,2,2,2)    A[0][0]; ~A;
```




```
Poly<> B(2,2)      B.expand(2,2);
```

1.5 Istruzioni di Input/Output

1.5.1 Input

È possibile impostare un'intera variabile leggendo i dati da un file TSV (valori separati da tabulatore) tramite la funzione `readFromFile(VAR,"nomefile.tsv")`.

1.5.2 Output

L'output è strettamente su video ed è formattato a colonne di larghezza fissa. Le funzioni per visualizzare i contenuti di una variabile parallela sono:

```
print();       // stampa la matrice di dati
printState();  // stampa la matrice di stato
```

1.5.3 I/O sul singolo PE

Se si desidera impostare il valore di una cella della matrice di dati, basta utilizzare l'operatore `·(r,c,value)` o equivalentemente `write(·,r,c,value)`; ad esempio l'operazione `A(3,7,1)` imposta ad 1 la cella di coordinate `(3,7)` della variabile A.

Per la lettura invece si deve utilizzare l'operatore `·(r,c)` o equivalentemente `read(·,r,c)`) il quale restituisce il valore alle coordinate `(r,c)`.

1.6 Strutture di controllo

1.6.1 Struttura condizionale

Il seguente costrutto permette di far eseguire certe istruzioni ad un sottoinsieme dei PE prima, ed altre istruzioni all'insieme complementare dopo; questo signi-

fica che entrambi i rami vengono eseguiti in successione. Per dividere i PE in due insiemi disgiunti basta semplicemente fornire una condizione (che ritorna una variabile `PolyLogic`) alla clausola `where`:

```
where(A==B && (A*2)!=C) {
   // qui i PE attivi soddisfano la condizione
} elsewhere {
   // qui i PE attivi soddisfano la condizione negata
} endwhere
```

È possibile tuttavia ottenere un costrutto identico all'`if-else` classico ricorrendo all'utilizzo della variabile globale `atleastone`; quest'ultima assume valore `true` se almeno un PE verifica la condizione, e `false` altrimenti:

```
where(A==B && (A*2)!=C) {
   if(atleastone)
      ...
} elsewhere {
   if(atleastone)
      ...
} endwhere
```

Occorre inoltre sottolineare che, nel caso di `where` annidati, è consigliato usare sempre variabili parallele delle stesse dimensioni (vedi §1.3.2).

1.6.2 Struttura di ciclo

Il seguente costrutto permette di far eseguire iterativamente delle istruzioni ad un sottoinsieme dei PE identificato da una data condizione. Finchè almeno un PE soddisfa tale condizione, sia quest'ultima che le istruzioni vengono valutate ad ogni singolo passo del ciclo.

```
int i=0;
pwhile(A==B*C && i<100) {
   ...
   i++;
} pwend
```

1.6.3 Struttura di stato

Il seguente costrutto serve unicamente a porre in cima allo stack degli stati globali lo stato locale della variabile specificata (vedi §1.3.2).

```
pglobal(VAR) {
   ...
} pgend
```

Questo torna utile quando si devono attivare/disattivare in modo identico i PE di molte variabili, basta quindi farlo per una sola variabile e poi racchiudere le successive istruzioni in esso.

1.7 Primitive di comunicazione fra i processori

Per scambiare i dati tra un PE ed i suoi adiacenti all'interno della griglia, vi sono a disposizione due primitive per le quali bisogna anche specificare una direzione; a questo scopo è stata definita una variabile ti tipo enumerativo per elencare i quattro punti cardinali: `NORTH`, `EAST`, `SOUTH`, `WEST`.

1.7.1 La primitiva `shift`

L'istruzione `shift` consente di spostare i dati da ogni PE al suo adiacente lungo una data direzione. Ad esempio:

```
Poly<> A(2,2,2,2), B(2,2,2,2);
B=shift(A, EAST);
```

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

===>

4	1	2	3
8	5	6	7
12	9	10	11
16	13	14	15

1.7.2 La primitiva `broadcast`

L'istruzione `broadcast` permette di definire dei segmenti (cluster) di comunicazione lungo una data direzione, entro i quali il contenuto del PE a capo del cluster viene trasmesso a tutti i PE seguenti, fino ad incontrare il nuovo cluster. Per definire i cluster è sufficiente indicare quali sono i PE al loro capo fornendo una variabile `PolyLogic`. Ad esempio:

```
Poly<> A(2,2,2,2), B(2,2,2,2), C(2,2,2,2);
B=broadcast(A,  EAST, A%3==0);
C=broadcast(A, SOUTH, A%3==0);
```

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

A

==>

3	3	3	3
6	6	6	6
9	9	9	12
15	15	15	15

B

,

9	6	3	12
9	6	3	12
9	6	3	12
9	6	15	12

C

dove in grigio sono evidenziate le celle per cui la condizione `A%3==0` risulta vera.

1.8 Funzioni standard

Esistono a disposizione altre funzioni che possono tornare particolarmente utili nella stesura di un programma:

– Effettua una copia 1:1 della variabile `src` nella variabile `dest`;

```
copy(dest, src);
```

– Ritorna il numero di (sotto)colonne/righe della variabile `obj`:

```
columns(obj);       rows(obj);
subcolumns(obj);    subrows(obj);
```

– Ritorna una variabile `Poly<T>` in cui ogni colonna/riga contiene i numeri da `0` a `rows(obj)/columns(obj)`:

```
col(obj);           row(obj);
```

ad esempio, considerando una `Poly<> A(4,4)`, si ottiene:

0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3

`col(A)`

0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

`row(A)`

– Imposta le dimensioni di default della la griglia di PE (influenza solo la creazione di nuove variabili):

```
set_mesh(nrows, ncols);
```

Capitolo 2

L'implementazione

In questo capitolo vengono illustrate le soluzioni adottate per i principali problemi che sono stati affrontati durante le fasi di progettazione e sviluppo.

Il linguaggio *RPC++* (Reconfigurable Parallel C++) è stato implementato a mezzo di una libreria per C++, utilizzando costrutti come *classi*, *template*, e *macro*, i quali agevolano lo sviluppo ed il debug e, tramite il paradigma di programmazione OOP, facilitano l'utilizzo della libreria all'utente finale nascondendone la complessità sottostante.

Per per la gestione del multithreading si è deciso di utilizzare la libreria *libpthread* implementante i *Pthreads* (POSIX Threads), in quanto essi sono uno standard multipiattaforma ben noto.

2.1 Gestione dei thread e dei task

Grazie all'uso dei puntatori a funzione e a quelli generici di tipo `void` è stato possibile generalizzare il concetto di thread/*esecutore* e task/*comando*; in questo modo ogni singolo thread esegue dei pacchetti di istruzioni di cui ignora sia il tipo che i parametri.

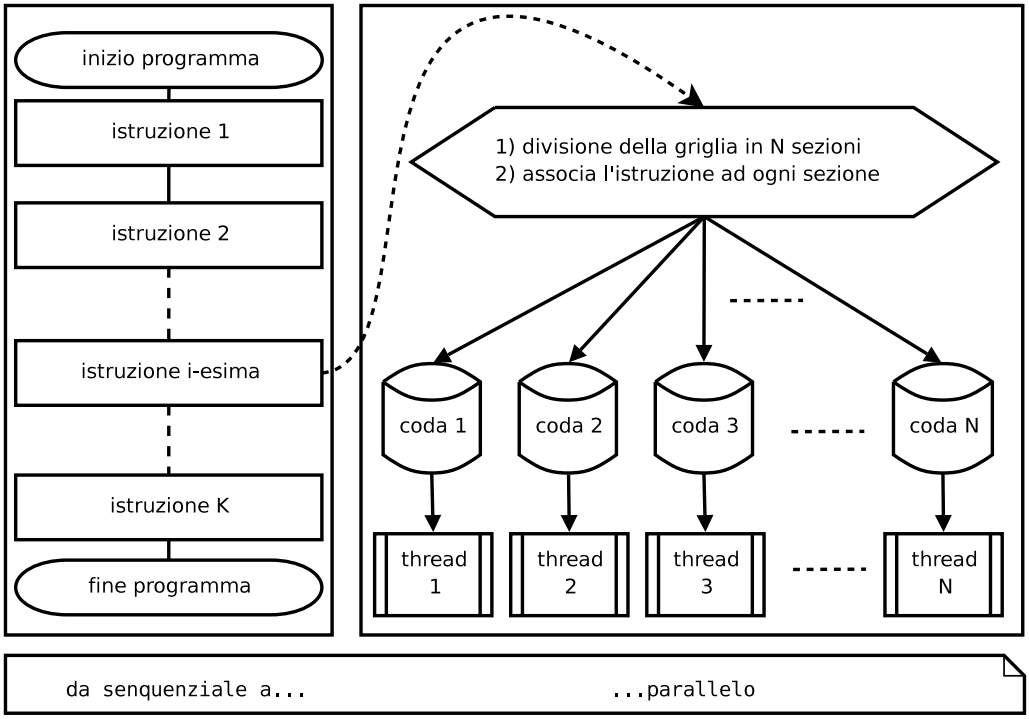


Figura 2.1: Gestione di un'istruzione.

Osservando la figura 2.1 è possibile capire più facilmente cosa avviene di ogni istruzione parallelizzabile incontrata in un programma scritto in *RPC++*:

- la griglia viene divisa equamente tra gli N Core disponibili:
  - viene calcolato il numero di celle per Core: $(r \cdot c/N)$
  - se la precedente divisione ha resto non nullo allora le rimanenti celle, a partire dal primo Core, vengono spartite una per una fino ad esauriento
- viene creato un task per ogni Core, cioè un pacchetto che contiene la funzione da eseguire assieme ai suoi parametri (compresa la sezione della griglia su cui essa andrà applicata)
- ogni task viene inserito nella coda del Core corrispondete
- ogni thread estrae un task alla volta dalla propria coda e lo esegue

Il mapping di dafault tra i thread e le celle della griglia è sequenziale a partire dalla cella `(0,0)` fino alla `(r-1,c-1)`; la matrice viene praticamente affettata orizzontalmente e le righe vengono affiancate in successione in modo da creare un lungo vettore, infine quest'ultimo viene suddiviso tra i thread. In figura 2.2 vengono presentati alcuni mapping d'esempio dove il numero all'interno delle celle rappresenta quello del Core di appartenenza:

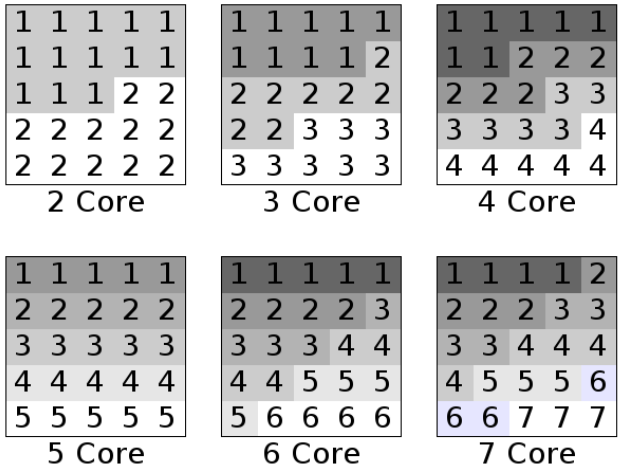


Figura 2.2: Mapping di default tra thread e celle della griglia.

2.2 Task per operatori e gestione dello stato

Gli operatori aritmetici, di incremento/decremento, logici, relazionali, e le operazioni sulle matrici di stato, non necessitano di alcuna sincronizzazione tra i thread, questo perchè l'operazione viene applicata strettamente sull'assegnata sezione della griglia; un thread può quindi eseguire tali operazioni senza preoccuparsi di quello che stanno facendo i suoi simili.

2.3 Task per operatori di riduzione

Prima di restituire il risultato di un operatore di riduzione bisogna prima collezionare i risultati di tutti i thread, è per questo necessaria una sincronizzazione per l'elaborazione del risultato finale.

Il thread padre dopo aver inserito i task nelle code, attende un segnale di fine elaborazione da parte di tutti i thread figlio; quando tutti hanno terminato il loro task allora il risultato è finalmente pronto e prelevabile dal thread padre, il quale potrà ora continuare l'esecuzione del programma.

2.4 Task per `shift`

Questa primitiva di comunicazione non lavora solo sulla propria sezione di griglia ma opera anche su una piccola parte di un'altra adiacente; se si devono infatti spostare i contenuti di alcune celle in altre, una parte di esse rimarrà nella sezione dedicata, mentre al massimo due andranno a finire in un'altra.

In figura 2.3 sono rappresentati tre esempi di `shift` verso est di una variabile 5x5; le celle in grigio chiaro indicano la sezione che si sta attualmente spostando, mentre quelle in grigio scuro indicano le celle di altre sezioni su cui bisogna scrivere.

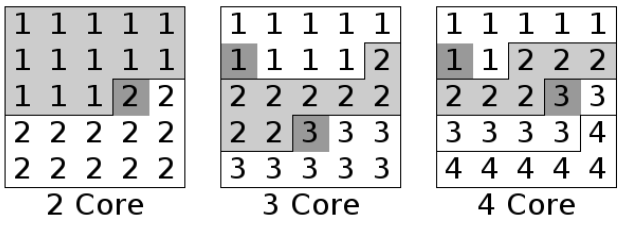


Figura 2.3: Shift verso est e relative celle interessate.

Di seguito viene riportata la pseudocodifica dell'algoritmo eseguito in parallelo da ogni thread; è da tener presente che la sincronizzazione è stata gestita tramite

mutex e *variabili condizionali*, e che l'area di memoria dalla quale vengono letti i dati è diversa da quella in cui essi vengono scritti:

```
// la risorsa mutex_block[thread_id] e' ora bloccata

// nel caso di piu' shift consecutive attendi la fine
// dell'operazione precedente
lock(mutex_nextop);
while (this_op > currentop[thread_id])
    cond_wait(cond_nextop, mutex_nextop);
unlock(mutex_nextop);

// calcola il numero di accessi a questa sezione
// da parte delle altre sezioni
used_by[thread_id]=get_num_accesses();

// rendi disponibile la sezione di questo thread
unlock(mutex_block[thread_id]);

// passa in rassegna tutte le celle di questa sezione
for (int i=celle.start; i<=celle.end; i++) {
    // controlla se bisogna scrivere in una sezione
    // diversa da quella di questo thread
    if (needs_neighbour_block(i))
        ... // *aggiorna la lista celle speciali
    else
        ... // *esegui l'operazione per la cella i-ma
}//for
```

```
// passa in rassegna tutte le celle speciali
for (cella in lista celle speciali) {
    // blocca la sezione della cella
    lock(mutex_block[block_of(cella)]);

    ... // esegui l'operazione per questa cella

    // decrementa il contatore degli usi della
    // sezione appena usata
    used_by[block_of(cella)]--;

    // risveglia un eventuale thread che attende
    // di usare anch'esso la sezione della cella
    wake(cond[block_of(cella)]);

    // rendi disponibile la sezione della cella
    unlock(mutex_block[block_of(cella)]);
}//for

// aspetta che gli altri thread abbiamo finito
// di utilizzare questa sezione prima di bloccarla
lock(mutex_block[thread_id]);
while (used_by[thread_id]>0)
    wait(cond[thread_id], mutex_block[thread_id])
unlock(mutex_block[thread_id]);
```

```
// aggiorna il contatore dell'operazione corrente
lock(mutex_nextop);
currentop_subop++;
// se questo e' l'ultimo thread ad eseguire
// questa operazione
if (currentop_subop==numero_tot_thread) {
    currentop_subop=0;
    // aggiorna il contatore di ogni thread
    unsigned char newop=currentop[i]++;
    for (i=0; i<cpus; i++)
        currentop[i]=newop;
    delete this_op;
}//if
unlock(mutex_nextop);

// risveglia tutti i thread che attendono
// nel caso di piu' shift consecutive
wake_all(cond_nextop);
```

tramite le variabili `cond_nextop` e `mutex_nextop` è stato introdotto un meccanismo di protezione dei dati in caso si presenti una serie consecutiva di `shift`, per evitare l'evento in cui l'istruzione successiva va a leggere un dato di una cella non ancora aggiornata dall'operazione precedente.

Nella pseudocodifica non è riportato ma, quando viene effettuato un cambio di direzione, il mapping tra le celle ed i thread cambia. All'inizio del programma la direzione attuale è orizzontale (`WEST/EAST`), se viene incontrata una primitiva di comunicazione che fa viaggiare i dati nel senso oppposto, cioè verticale

(NORTH/SOUTH), allora avviene una sincronizzazione tra i thread ed il mapping cambia come illustrato in figura 2.4.

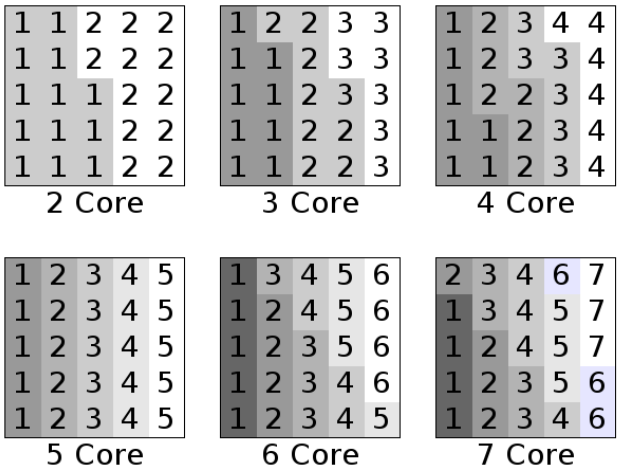


Figura 2.4: Mapping per primitive a direzione verticale NORTH/SOUTH.

2.5 Task per `broadcast`

Il funzionamento di questa primitiva di comunicazione è molto simile a quello della `shift`. Il ragionamento e le celle interessate sono gli stessi di figura 2.3 ed 2.4, ma bisogna aggiungere all'algoritmo la ricerca del capo del cluster per poterne propagare il valore nella direzione indicata.

Di seguito viene riportata la pseudocodifica dell'algoritmo eseguito in parallelo da ogni thread; sono da tener presente le stesse considerazioni fatte per la `shift`:

```
// la risorsa mutex_block[thread_id] e' ora bloccata

// nel caso di piu' broadcast consecutive attendi la
// fine dell'operazione precedente
lock(mutex_nextop);
```

```
while (this_op > currentop[thread_id])
    cond_wait(cond_nextop, mutex_nextop);
unlock(mutex_nextop);

// rendi disponibile la sezione di questo thread
unlock(mutex_block[thread_id]);

// passa in rassegna tutte le celle di questa sezione
current_row=-1;
for (int i=celle.start; i<=celle.end; i++) {
    // se avviene un cambio di riga
    if (current_row != get_row(i)) {
        current_row=get_row(i);
        // trova il valore del capo del cluster
        value=get_propagation_value(i);
    }//if
    ... // *esegui l'operazione per la cella i-ma
}//for

// aspetta che gli altri thread abbiamo finito
// di utilizzare questa sezione prima di bloccarla
lock(mutex_sharedres);
if ((params.bcast_threads--) > 0)
    wait(cond_sharedres, mutex_sharedres)
lock(mutex_block[thread_id]);
wakeall(cond_sharedres);
unlock(mutex_sharedres);
```

```
// aggiorna il contatore dell'operazione corrente
lock(mutex_nextop);
currentop_subop++;
// se questo e' l'ultimo thread ad eseguire
// questa operazione
if (currentop_subop==numero_tot_thread) {
     currentop_subop=0;
     // aggiorna il contatore di ogni thread
     unsigned char newop=currentop[i]++;
     for (i=0; i<cpus; i++)
          currentop[i]=newop;
     delete this_op;
}// if
unlock(mutex_nextop);

// risveglia tutti i thread che attendono
// nel caso di piu' shift consecutive
wake_all(cond_nextop);
```

A partire dalla cella fornita, la funzione `get_propagation_value` cerca nella direzione opposta la cella a capo del cluster, effettuando prima l'eventuale blocco della sezione su cui va a leggere.

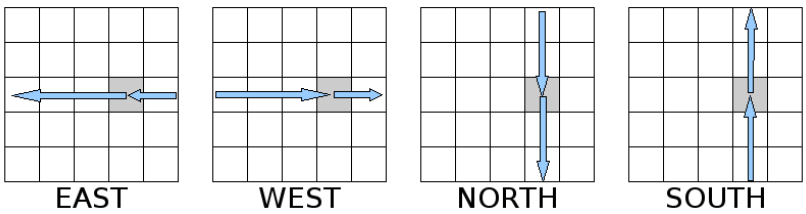


Figura 2.5: Ricerca del capo del cluster a partire da una cella.

2.6 Confronto con un mapping alternativo

Almeno in linea teorica, è interessante vedere per le primitive di comunicazione quali vantaggi e svantaggi si hanno dovendo scegliere tra il mapping qui implementato ed uno statico come quello a scacchiera (vedi figura 2.6).

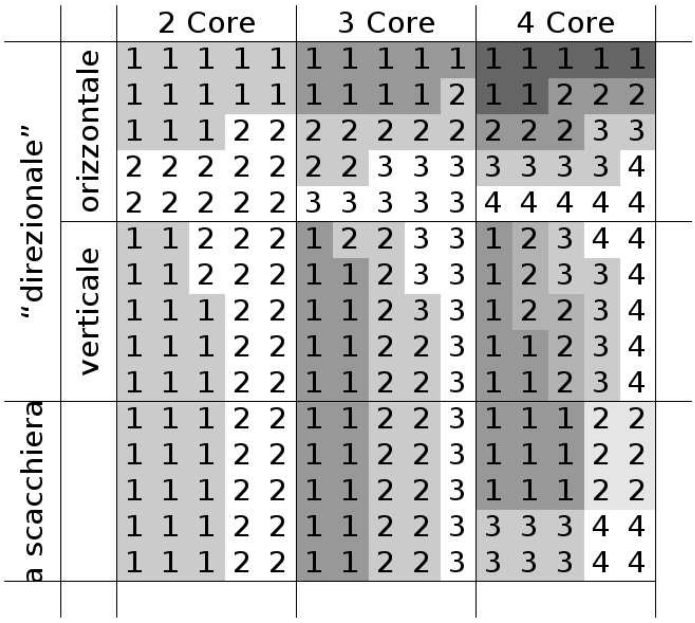


Figura 2.6: Mapping "direzionale" e a scacchiera.

Il confronto riportato in tabella 2.1 si basa sul numero di *lock* delle risorse, la quantità di celle da scrivere durante lo stesso, ed il numero di sincronizzazioni tra i thread; questi tre parametri decidono infatti i tempi d'esecuzione dei singoli task, senza contare la quantità di memoria usata, e lo stress impartito al sistema operativo per la gestione dei thread.

singolo thread	**mapping "direzionale"**	**mapping a scacchiera**
# lock	da 0 a 2	1 (oppure $\leq n$)
# celle per lock	1	$\leq n$ (oppure 1)
# sincronizzazioni	1 per cambio di direzione	0

Tabella 2.1: Confronto tra mapping diversi considerando una matrice quadrata nxn.

Di per sè la sola analisi quantitativa delle risorse utilizzate non aiuta più di tanto nel confronto tra i due mapping, infatti una visione migliore si ha nel grafico riportato in figura 2.7, nel quale viene rappresentato il tempo impiegato contro il numero consecutivo di istruzioni; le diverse versioni del tempo impiegato dal mapping direzionale, dipendono dal fatto che nella sequenza di istruzioni è contenuta una percentuale diversa di cambi di direzione.

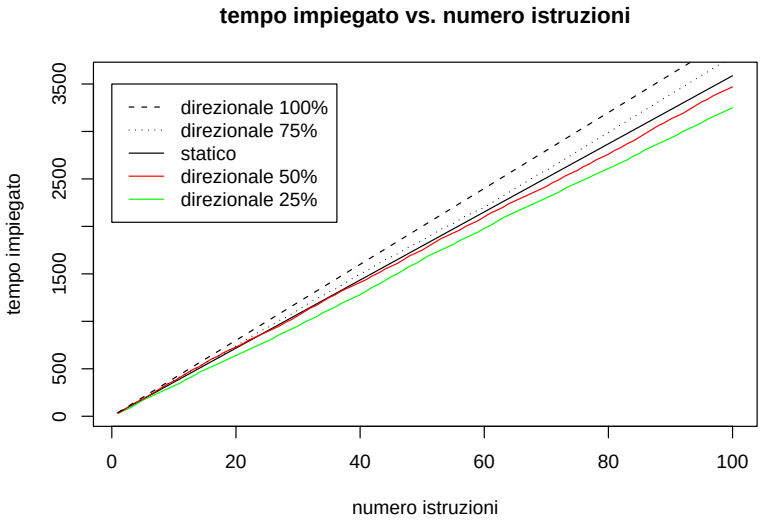


Figura 2.7: Ananlisi dei tempi su matrice 255x255 con 4 Core.

Dal grafico si deduce che in alcuni casi il mapping direzionale conviene rispetto a quello statico a scacchiera e viceversa; in particolare la massima percentuale di cambi di direzione ammessi, affinchè il mapping direzionale resti il più conveniente, è stata calcolata pari a circa il 60%.

Un'altro grafico interessante si ottiene conteggiando anche anche il tempo impiegato dal cambio di mapping quando ne avviene uno di direzione; ciò serve per quantificare il peso del tempo di trasmissione dei dati sui bus nel caso in cui si operi sulla reale macchina di riferimento.

Il numero di celle che vengono rimappate durante il cambio di direzione (quelle

che fanno quindi perdere tempo) è inferiore a circa $(n^2/\#cpu)/3$; in figura 2.8 il tempo di trasferimento è stato impostato per assurdo ad un centesimo del tempo di scrittura di un dato locale, eppure si nota lo stesso come il mapping statico sia nettamente superiore.

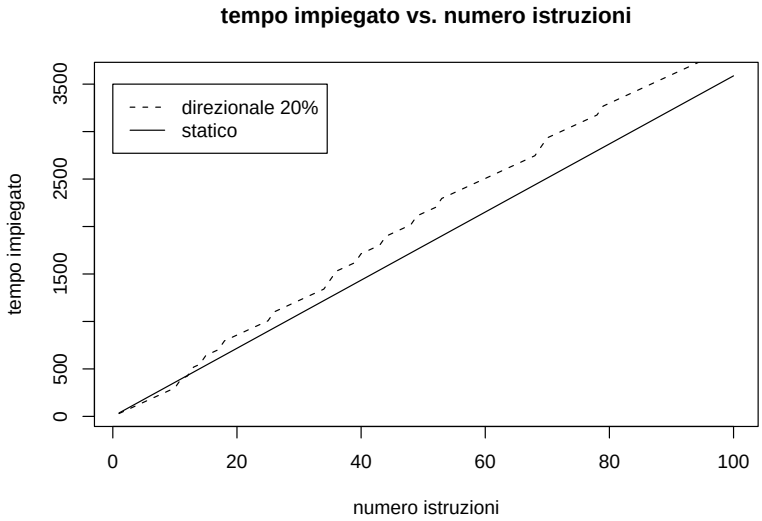


Figura 2.8: Ananlisi dei tempi maggiorati su matrice 255x255 con 4 Core.

Questi risultati indicano come non sia sensato usare il mapping direzionale sulla macchina di riferimento, in quanto la velocità di trasmissione sui bus dovrebbe essere molto più elevata rispetto a quella della scrittura su di un dato locale.

2.7 Uso della libreria

La libreria *librcpp* è stata sviluppata in C++ e testata tramite il compilatore GNU gcc. Per la sua compilazione basta eseguire il comando GNU make:

```
make CONF=Release build  # compila per la distribuzione
make CONF=Debug build    # compila per il debug
```

il quale produrrà la libreria nella cartella "./dist/CONF/target/librpcpp.ext>".

Nel codice del proprio programma bisogna includere l'header della libreria, e opzionalmente utilizzare il namespace `poly`:

```
// test.cpp
#include "rpcpp.h"
using namespace poly;

int main(int argc, char** argv) {
  Poly<int> A(2,2);
  col(A).print();  // stampa la matrice colonna

  return 0;
}//main
```

mentre per la sua compilazione basta eseguire il comando:

```
g++ -o test.exe -lpthread -lrpcpp test.cpp
```

assicurandosi di avere `librpcpp.so` o `rpcpp.dll` nel proprio path di ricerca.

Appendice A

Manuale di riferimento

(ver.1.2-pacifica)

atleastone

Tipologia: variabile globale

Prototipo:

```
bool atleastone;
```

Esempio:

```
where(COND) {
   // qui i PE attivi soddisfano la condizione
   if(atleastone) { ... }
} elsewhere {
   // qui i PE attivi soddisfano la condizione negata
   if(atleastone) { ... }
} endwhere
```

Descrizione:

Essa assume valore `true` se almeno un PE verifica la condizione, e `false` altrimenti. Se inserita in un costrutto `where` è possibile ottenere un costrutto identico all'`if-else` classico.

broadcast

Tipologia: primitiva di comunicazione

Prototipo:

```
Poly<T> broadcast(Poly<T> &obj, direction dir, PolyLogic s);
```

Esempio:

```
Poly<> A(2,2,2,2), B(2,2,2,2), C(2,2,2,2);
B=broadcast(A,  EAST, A%3==0);
C=broadcast(B, SOUTH, B%2==0);
```

Descrizione:

Permette di definire dei segmenti (cluster) di comunicazione lungo una data direzione, entro i quali il contenuto del PE a capo del cluster viene trasmesso a tutti i PE seguenti, fino ad incontrare il nuovo cluster. Per definire i cluster è sufficiente indicare quali sono i PE al loro capo fornendo il risultato di una condizione (variabile `PolyLogic`).

col

Tipologia: funzione di libreria

Prototipo:

```
Poly<T> col(Poly<T> &obj);
```

Esempio:

```
Poly<> A, C;
C=col(A);
```

Descrizione:

Data una variabile parallela A, essa ne ritorna un'altra di pari dimensioni in cui ogni colonna contiene i numeri da `0` a `rows(A)`. Questa funzione, congiuntamente all'analoga `row`, risulta una comoda scorciatoia nel caso si vogliano esprimere condizioni particolari, come ad esempio:

```
pglobal(col(A)==row(A))
  ...
pgend
```

la quale permette l'attivazione di tutti i PE sulla diagonale.

columns

Tipologia: funzione di libreria

Prototipo:

```
int columns(Poly<T> &obj);
```

Esempio:

```
Poly<> A;
int numero_colonne=columns(A);
```

Descrizione:

Ritorna il numero di colonne della variabile specificata.

copy

Tipologia: funzione di libreria

Prototipo:

```
bool copy(Poly<T> &dest, Poly<T> &src);
```

Esempio:

```
Poly<> A, B;
A=3;
unset(A,0,0);
copy(B,A);     // copia A in B
```

Descrizione:

Copia sia la matrice di dati che quella di stato dalla variabile `src` nella variabile `dest`. La funzione effettua la copia e ritorna `true` se le due variabili hanno pari dimensioni, altrimenti ritorna `false`.

DEFMESHROWS

Tipologia: costante di compilazione

Definizione:

```
#define DEFMESHROWS 256
```

Esempio:

```
Poly<> A(DEFMESHROWS,3);
```

Descrizione:

Questa costante indica il numero di righe di default della griglia di PE.

DEFMESHCOLS

Tipologia: variabile del compilatore

Definizione:

```
#define DEFMESHCOLS 256
```

Esempio:

```
Poly<> A(3,DEFMESHCOLS);
```

Descrizione:

Questa costante indica il numero di colonne di default della griglia di PE.

direction

Tipologia: variabile enumerativa globale

Prototipo:

```
enum direction { NORTH='n', EAST='e', SOUTH='s', WEST='w' };
```

Esempio:

```
A=shift(B, SOUTH);
C=broadcast(D, EAST, COND);
```

Descrizione:

Definisce le possibili direzioni per le primitive di comunicazione `shift` e `broadcast`.

expand

Tipologia: funzione di upsizing

Prototipo:

```
Poly<T> expand(Poly<T> &obj, int r, int c);
```

Esempio:

```
Poly<> A(2,2), B(2,2,3,3);
B=expand(A,3,3);
```

Descrizione:

Ritorna una variabile espansa: da una variabile in cui ad ogni PE è stato associato un singolo valore, questa funzione ritorna un'altra variabile la quale ad ogni PE associa una matrice `rxc`, replicando i singoli valori iniziali nelle nuove celle vuote.

get_cpu_count

Tipologia: funzione di libreria

Prototipo:

```
int get_cpu_count();
```

Esempio:

```
int ncpus=get_cpu_count();
```

Descrizione:

Ritorna il numero di Core presenti nella CPU della macchina corrente. Attualmente sono supportate le piattaforme Windows e Linux.

gset

Tipologia: funzione di manipolazione dello stato

Prototipo:

```
void gset(int r, int c);
```

Esempio:

```
gset(0,0);
```

Descrizione:

Attiva il PE di coordinate `(r,c)` modificando lo stato in cima allo stack degli stati globali.

gsetall

Tipologia: funzione di manipolazione dello stato

Prototipo:

```
void gsetall();
```

Esempio:

```
gsetall();
```

Descrizione:

Attiva tutti i PE modificando lo stato in cima allo stack degli stati globali.

gunset

Tipologia: funzione di manipolazione dello stato

Prototipo:

```
void gunset(int i, int j);
```

Esempio:

```
gunset(0,0);
```

Descrizione:

Disattiva il PE di coordinate `(r,c)` modificando lo stato in cima allo stack degli stati globali.

gunsetall

Tipologia: funzione di manipolazione dello stato

Prototipo:

```
void gunsetall();
```

Esempio:

```
gunsetall();
```

Descrizione:

Disattiva tutti i PE modificando lo stato in cima allo stack degli stati globali.

max

Tipologia: funzione di riduzione

Prototipo:

```
long double max(Poly<T> &obj);
```

Esempio:

```
Poly<> A;
long double m=max(A);
```

Descrizione:

Restituisce il valore massimo tra tutti i valori contenuti in tutti PE.

min

Tipologia: funzione di riduzione

Prototipo:

```
long double min(Poly<T> &obj);
```

Esempio:

```
Poly<> A;
long double m=min(A);
```

Descrizione:

Restituisce il valore minimo tra tutti i valori contenuti in tutti PE.

pglobal-pgend

Tipologia: costrutto di manipolazione dello stato

Prototipo:

```
pglobal(VAR) { ... } pgend
```

Esempio:

```
Poly<> A,B,C;
unset(A,0,0);
pglobal(A) {
  A=B+C;
  B=C*2;
  C=A-B;
} pgend
```

Descrizione:

Pone in cima allo stack degli stati globali lo stato locale della variabile VAR.

Poly<T>

Tipologia: tipo di variabile

Prototipo:

```
template <class T> class Poly;
```

Esempio:

```
Poly<> A;
Poly<int> B, *ptB=&B;
```

Descrizione:

È la classe che definisce una variabile di tipo parallelo. Se non specificato, il tipo di default è `double`.

PolyLogic

Tipologia: tipo di variabile

Prototipo:

```
PolyLogic VAR;
```

Esempio:

```
Poly<> A,B;
PolyLogic COND1, COND2(A!=0 || B==3);
COND1=(A!=0 && B==3);

where(COND1)
  ...
```

Descrizione:

È la classe che definisce una variabile di tipo parallelo per la sola gestione degli stati e le eventuali operazioni su di essi: `&&`, `||`, `==`, `!=`, `=`, `!`.

print

Tipologia: funzione di libreria

Prototipo:

```
void print(Poly<T> &obj);
```

Esempio:

```
Poly<> A;
print(A);
```

Descrizione:

Stampa su video i dati di tutti i PE, in forma matriciale a colonne di larghezza fissa.

printState

Tipologia: funzione di libreria

Prototipo:

```
void printState(Poly<T> &obj);
```

Esempio:

```
Poly<> A;
printState(A);
```

Descrizione:

Stampa su video lo stato di tutti i PE in forma matriciale a colonne di larghezza fissa.

prod

Tipologia: funzione di riduzione

Prototipo:

```
long double prod(Poly<T> &obj);
```

Esempio:

```
Poly<> A;
long double p=prod(A);
```

Descrizione:

Restituisce il prodotto tra tutti i valori contenuti in tutti PE.

pwhile-pwend

Tipologia: costrutto di iterazione

Prototipo:

```
pwhile(COND) { ... } pwend
```

Esempio:

```
int i=0;
pwhile(A==B*C && i<100) {
  ...
  i++;
} pwend
```

Descrizione:

Questo costrutto permette di far eseguire iterativamente delle istruzioni ad un sottoinsieme dei PE identificato da una data condizione. Finchè almeno un PE soddisfa tale condizione, sia quest'ultima che le istruzioni vengono valutate ad ogni singolo passo del ciclo.

Sovente risulta difficile prevedere il numero di iterazioni che verrano eseguite, quindi, se possibile, si consiglia di utilizzare una variabile contatore in modo da limitare un eventuale ciclo infinito.

`read`

Tipologia: funzione di libreria

Prototipo:

```
T read(Poly<T> &obj, int r, int c);
```

Esempio:

```
Poly<> A;
read(A,0,0);
```

Descrizione:

Restituisce il valore alle coordinate `(r,c)` contenuto nella matrice di dati della variabile `obj`.

`readFromFile`

Tipologia: funzione di libreria

Prototipo:

```
Poly<T> readFromFile(Poly<T> &obj, char *fname);
```

Esempio:

```
Poly<> A;
readFromFile(A, "file.tsv");
```

Descrizione:

Imposta tutti i valori della matrice di dati della varibile `obj` a quelli letti dal file specificato. Il file dev'essere in formato TSV (*tab separated values*), cioè deve contenere una matrice di numeri separati da tabulatore.

reduce / ~

Tipologia: funzione di downsizing

Prototipo:

```
Poly<T> reduce(Poly<T> &obj, int r, int c);
Poly<T> operator ~();
```

Esempio:

```
Poly<> A(2,2,3,3), B(2,2);
B=~(A[0][0]);
B=reduce(A,0,0);
```

Descrizione:

Resituisce una variabile la cui matrice di dati contiene gli elementi della variabile `obj` di posizione `(r,c)` all'interno dei suoi PE.

row

Tipologia: funzione di libreria

Prototipo:

```
Poly<T> row(Poly<T> &obj);
```

Esempio:

```
Poly<> A, R;
R=row(A);
```

Descrizione:

Data una variabile parallela A, essa ne ritorna un'altra di pari dimensioni in cui ogni riga contiene i numeri da `0` a `columns(A)`. Questa funzione, congiuntamente all'analoga `col`, risulta una comoda scorciatoia nel caso si vogliano esprimere condizioni particolari, come ad esempio:

```
pglobal(col(A)==row(A))
  ...
pgend
```

la quale permette l'attivazione di tutti i PE sulla diagonale.

rows

Tipologia: funzione di libreria

Prototipo:

```
int rows(Poly<T> &obj);
```

Esempio:

```
Poly<> A;
int numero_righe=rows(A);
```

Descrizione:

Ritorna il numero di righe della variabile specificata.

RPCPPLIB_*

Tipologia: costante di compilazione

Definizione:

```
#define RPCPPLIB_VER_MAJOR "1"
#define RPCPPLIB_VER_MINOR "2"
#define RPCPPLIB_CODENAME  "pacifica"
```

Esempio:

```
cout<<"Verisione liberia: "
    <<RPCPPLIB_VER_MAJOR<<"."
    <<RPCPPLIB_VER_MINOR<<"-"
    <<RPCPPLIB_CODENAME;
```

Descrizione:

Contiene le informazioni sulla versione corrente della libreria.

set

Tipologia: funzione di manipolazione dello stato

Prototipo:

```
void set(Poly<T> &obj, int r, int c);
```

Esempio:

```
Poly<> A;
set(A,0,0);
```

Descrizione:

Attiva il PE di coordinate `(r,c)` modificando lo stato locale della variabile `obj`.

setall

Tipologia: funzione di manipolazione dello stato

Prototipo:

```
void setall(Poly<T> &obj);
```

Esempio:

```
Poly<> A;
setall(A);
```

Descrizione:

Attiva tutti i PE modificando lo stato locale della variabile `obj`.

set_mesh

Tipologia: funzione di libreria

Prototipo:

```
void set_mesh(int rows, int cols);
```

Esempio:

```
set_mesh(10,3);
```

Descrizione:

Imposta le dimensioni di default della griglia di PE; influenza solo la creazione di nuove variabili.

shift

Tipologia: primitiva di comunicazione

Prototipo:

```
Poly<T> shift(Poly<T> &obj, direction dir);
```

Esempio:

```
Poly<> A,B,C;
B=shift(A, EAST);
C=shift(shift(A, SOUTH), NORTH);
```

Descrizione:

Questa funzione effettua una traslazione dei dati di una unità verso la direzione specificata. Si ricorda che la griglia è chiusa agli estremi, e che quindi i dati delle celle che sforando da un bordo vanno a finire sul bordo diametralmente opposto.

subcolumns

Tipologia: funzione di libreria

Prototipo:

```
int subcolumns(Poly<T> &obj);
```

Esempio:

```
Poly<> A(2,2,3,4);
int sc=subcolumns(A);  // --> 4
```

Descrizione:

Ritorna il numero di colonne della sottomatrice associata al singolo PE.

subrows

Tipologia: funzione di libreria

Prototipo:

```
int subrows(Poly<T> &obj);
```

Esempio:

```
Poly<> A(2,2,3,4);
int sc=subrows(A);  // --> 3
```

Descrizione:

Ritorna il numero di righe della sottomatrice associata al singolo PE.

sum

Tipologia: funzione di riduzione

Prototipo:

```
long double sum(Poly<T> &obj);
```

Esempio:

```
Poly<> A;
long double s=sum(A);
```

Descrizione:

Restituisce la somma di tutti i valori contenuti in tutti PE.

unset

Tipologia: funzione di manipolazione dello stato

Prototipo:

```
void unset(Poly<T> &obj, int r, int c);
```

Esempio:

```
Poly<> A;
unset(A,0,0);
```

Descrizione:

Disattiva il PE di coordinate `(r,c)` modificando lo stato locale della variabile `obj`.

unsetall

Tipologia: funzione di manipolazione dello stato

Prototipo:

```
void unsetall(Poly<T> &obj);
```

Esempio:

```
Poly<> A;
unsetall(A);
```

Descrizione:

Disattiva tutti i PE modificando lo stato locale della variabile `obj`.

`where-[elsewhere]-endwhere`

Tipologia: costrutto condizionale

Prototipo:

```
where(COND) { ... } elsewhere { ... } endwhere
```

Esempio:

```
where(A==B && (A*2)!=C) {
   if(atleastone)
      ...
} elsewhere {
   if(atleastone)
      ...
} endwhere
```

Descrizione:

Questo costrutto permette di far eseguire certe istruzioni ad un sottoinsieme dei PE prima, ed altre istruzioni all'insieme complementare dopo; questo significa che entrambi i rami vengono eseguiti in successione. Per dividere i PE in due insiemi disgiunti basta semplicemente fornire una condizione (che ritorna una variabile `PolyLogic`) alla clausola `where`.

L'uso della variabile globale `atleastone` permette la creazione di due blocchi di istruzioni mutuamente esclusivi, ottenendo così un costrutto identico al classico `if-else`.

write

Tipologia: funzione di libreria

Prototipo:

```
bool write(Poly<T> &obj, int r, int c, T value);
bool operator ()(int r, int c, T value);
```

Esempio:

```
Poly<> A;
write(A,0,0,1000);
A(0,0,1000);
```

Descrizione:

Imposta a `value` il valore della cella di coordinate `(r,c)` della matrice di dati nella variabile `obj`.

Bibliografia

[1] “RPC++: Manuale di riferimento”
G. Boreanaz, G. Dirosa, M. Migliardi, E. Orione – Giungno 1991

[2] “Programming with POSIX Threads”
David R. Butenhof – Addison-Wesley, Maggio 1997

[3] “Pthreads Programming”
B. Nichols, D. Buttlar, J.P. Farrell – O’Reilly, Febbraio 1998

[4] “Pthreads Programming”
B. Barney – Marzo 2008
`http://www.llnl.gov/computing/tutorials/pthreads/`

Finito di stampare il 17 settembre 2008 con LaTeX