Implementazione algoritmo Breadth First Search per processore Cell/BE

Applicazioni di Sistemi Distribuiti

Stefano Uliari

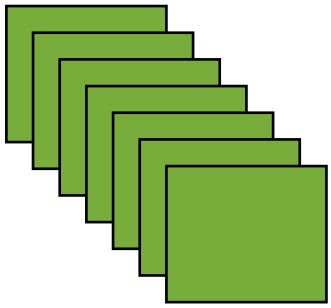
Descrizione:
L'algoritmo , dato un grafo di |V| vertici con un vertice che funge da root, ha lo scopo di visitare tutti i vertici. La descrizione matematica dell'algoritmo, con dettagli sull'implementazione per architettura Cell/BE e relativa analisi prestazionale, è nel documento "Challenges in Mapping Graph Exploration Algorithms on Advanced Multi-core Processors " (O. Villa, D.P. Scarpazza, F.Petrini, J.F. Peinador).

Obiettivo:
L'obiettivo del lavoro consiste nell'implementare, seguendo le indicazioni del documento sopraccitato, l'algoritmo per il funzionamento con processore *Cell/BE* (nel nostro caso specifico una *Playstation 3*), sfruttandone le caratteristiche di alto livello di parallelismo.

Ambiente di Sviluppo:
ho sviluppato il programma utilizzando l'*SDK* ufficiale dell' *IBM* per il *Cell/BE processor* installato su *Fedora 9* (a sua volta installato come macchina virtuale per Mac Os X). Più precisamente ho utilizzato *Eclipse* per lo sviluppo del codice, mentre per il debug *ppu-gdb* via *ssh* installato sulla *Playstation 3*.

Svolgimento:
Ho sviluppato il programma in due versioni: la prima versione,pur sfruttando il parallelismo delle 6 SPE disponibili come descritto nel documento di riferimento, fa a meno di alcune tecniche software per l'ottimizzazione delle prestazioni, quali il double buffering e le liste DMA. Nella seconda versione ho invece voluto implementare sia le lista DMA che il double buffering (quest'ultimo solamente nella prima parte dell'algoritmo). Entrambe le versioni sono state testate per verificarne la correttezza dell'output.

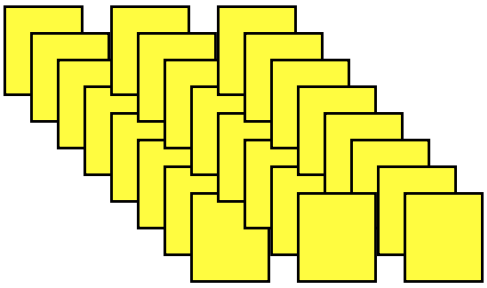
Input (PPE):



vertice G[|V|]
dove **|V|** è il numero dei vertici

Ogni elemento è un vertice ed è rappresentato dalla struct **vertice**

```
typedef struct nodo {
        unsigned int numero;
        unsigned int adj_length;
        unsigned int padding[2];
} vertice;
```



adj[|V|][|Evl_max_aligned16]
dove **|Evl_max** è l'arity massima(cioè il numero massimo di connessioni di un vertice) allineata a 4 (in modo che moltiplicata per i 4 byte di ogni int la struttura diventi allineata a 16)

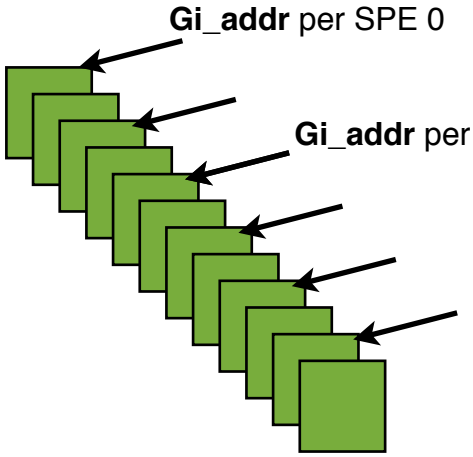
Questo è un vettore doppio, dove ogni colonna rappresenta la lista delle adiacenze del vertice considerato (ogni adiacenza è un vertice connesso al vertice considerato).

Il grafo è quindi rappresentato con una **AOS** (Array of structs)

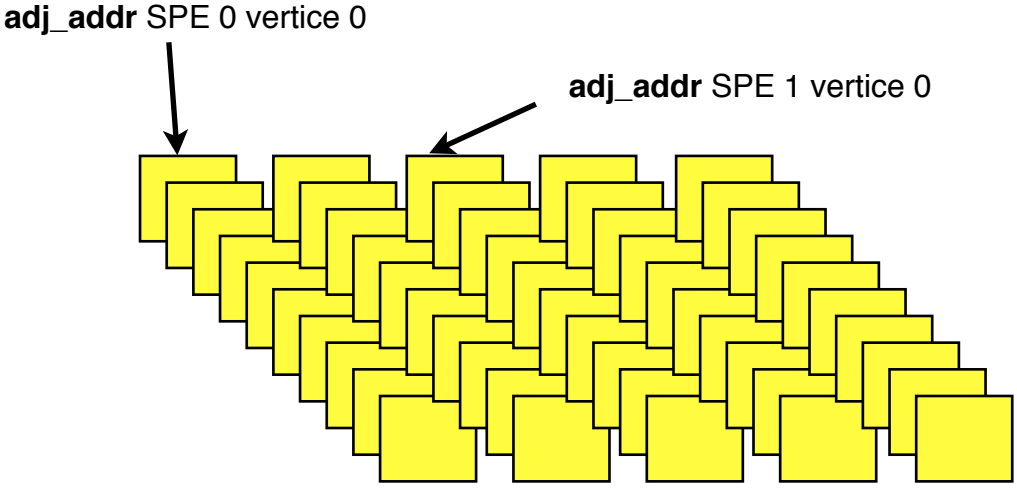
L'utente nel file common.h deve specificare il numero dei vertice, l'arity massima e quella minima, il vertice di root.
Il grafo viene generato automaticamente con connessioni random tra i vertici

Creazione delle SPE (PPE):

Vengono create 6 SPE.
Al momento della creazione a ogni SPE vengono passati gli indirizzi per caricare parti di input nel loro Local Store



Se per esempio abbiamo solo 10 vertici, a ogni SPE vengono assegnati 2 vertici. Solo la SPE proprietaria potrà esplorare (cioè vedere le adiacenze) e marcare come visitati i vertici a lei assegnati

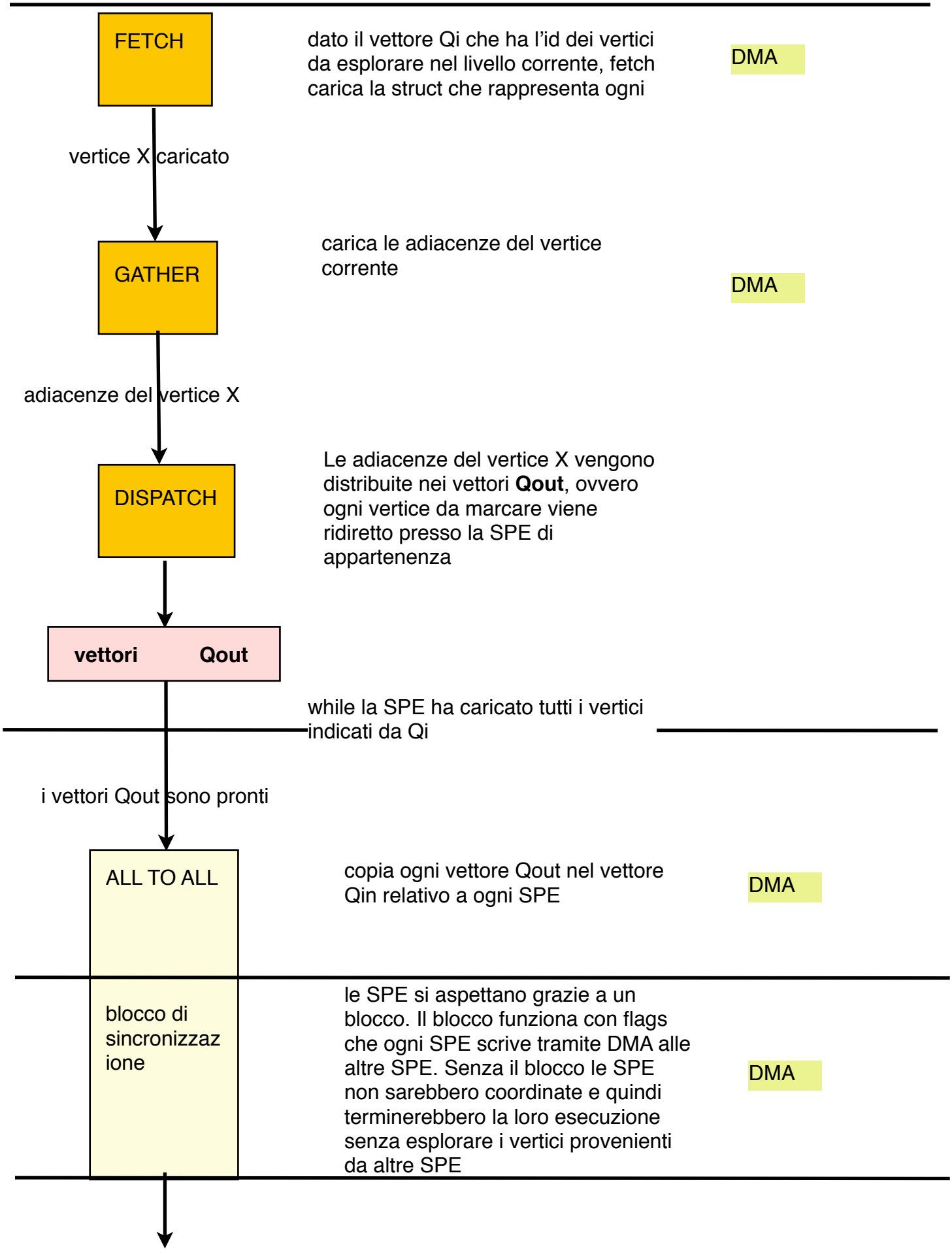


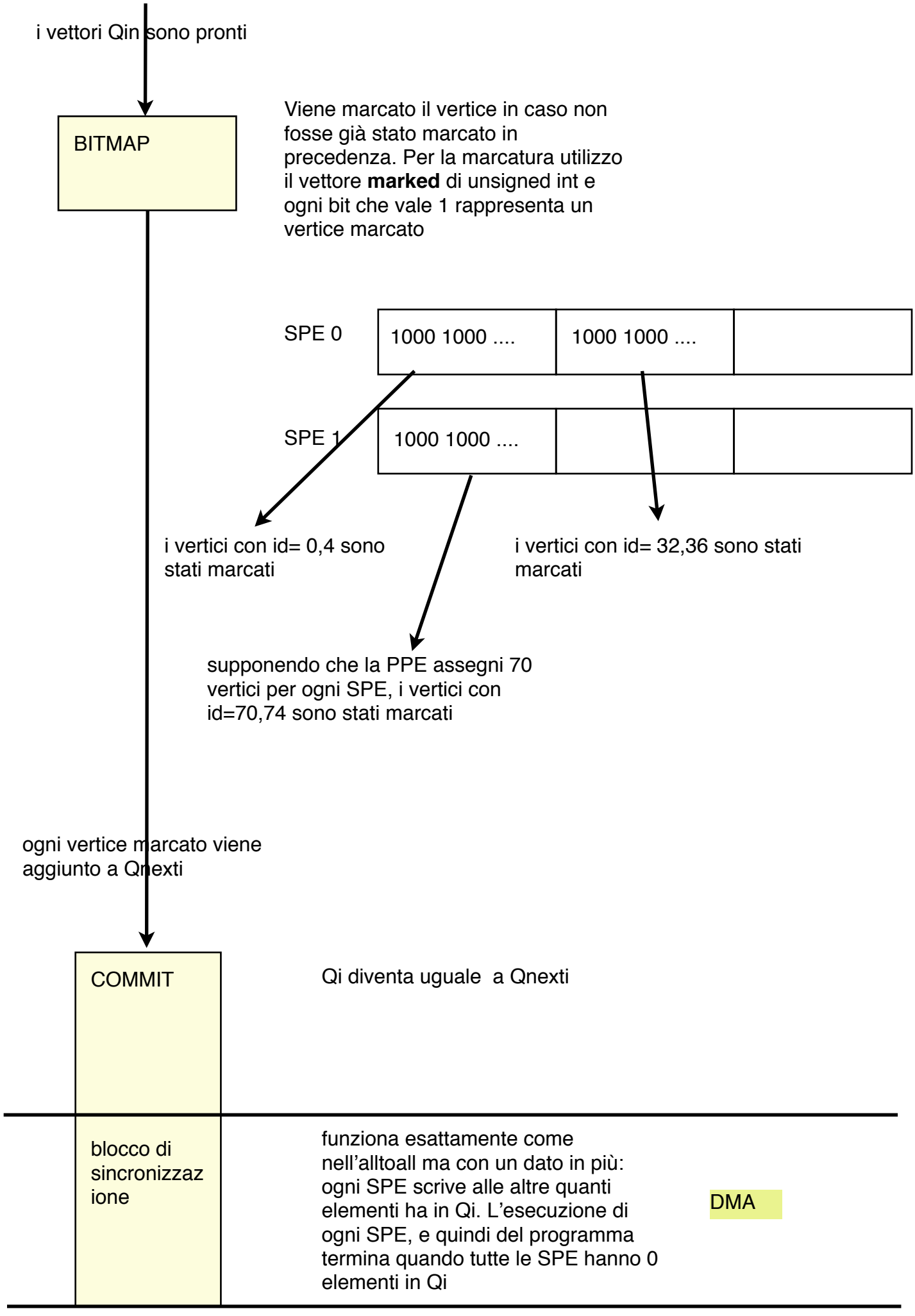

```
/*Struct con i parametri per le SPEs. Totale di 48 bytes*/
typedef struct params_t {
        unsigned int id;
        unsigned int Gi_addr;   //puntatore alla parte di grafo che deve essere analizzato dalla SPE
        unsigned int adj_addr;   //puntatore al vettore delle adiacenze
        unsigned int root;  //una sola SPE avrà il root tra i suoi vertici
        unsigned int local_stores[SPU_THREADS];  //per DMA SPE-SPE
        unsigned int padding[8 - SPU_THREADS]; //riempio la struct per arrivare a riempire 48 bytes
} params;
```

Struttura che rappresenta ogni SPE dentro la PPE.

```
typedef struct thread_spu{
      spe_context_ptr_t spe_ctx;
      pthread_t pthread;
      void *argp;   //è l'indirizzo della struct params, viene passato come argomento del main
      unsigned int entry;
} thread_spu;
```

FUNZIONAMENTO SPE ver 1




i vettori Qin sono pronti
BITMAP
Viene marcato il vertice in caso non fosse già stato marcato in precedenza. Per la marcatura utilizzo il vettore **marked** di unsigned int e ogni bit che vale 1 rappresenta un vertice marcato
SPE 0
1000 1000 ....
1000 1000 ....
SPE 1
1000 1000 ....
i vertici con id= 0,4 sono stati marcati
i vertici con id= 32,36 sono stati marcati
supponendo che la PPE assegni 70 vertici per ogni SPE, i vertici con id=70,74 sono stati marcati
ogni vertice marcato viene aggiunto a Qnexti
COMMIT
Qi diventa uguale a Qnexti
blocco di sincronizzazione
funziona esattamente come nell'alltoall ma con un dato in più: ogni SPE scrive alle altre quanti elementi ha in Qi. L'esecuzione di ogni SPE, e quindi del programma termina quando tutte le SPE hanno 0 elementi in Qi
DMA

Analisi dei limiti di questa implementazione (SPE)

Limiti della memoria:

I **256 Kb** di Local Store di ogni SPE limitano il massimo di vertici esplorabili da ogni SPE a **1010** circa , per un totale quindi di **6060** vertici.

Se vengono creati più di 1010 vertici, avviene uno *stack overflow*. Questo è visibile analizzando la memoria nel processo di debug. Per esempio, se viene aggiunto qualche "printf" nel codice, ogni SPE riesce ad analizzare al massimo circa 1000 vertici, in quanto ogni "printf" va ad occupare dello spazio nello stack.

Le strutture che occupano maggiormente il Local Store sono :

---

unsigned int Qi[IVil];
unsigned int Qnexti[IVil];

Entrambi sono vettori la cui grandezza dipende da quanti vertici vengono assegnati alla SPE.

---

unsigned int Qout[SPU_THREADS][adj_max_aligned16]
unsigned int Qin[SPU_THREADS][adj_max_aligned16]

Sono i vettori che occupano maggior spazio in quanto non solo sono doppi, ma anche ognuno di loro è di **adj_max_aligned16,** ovvero l'upper bound di quanti elementi può avere una coda di uscita o entrata. Questo numero massimo si verifica quando :
- tutti i nodi di un vertice vengono visitati allo stesso livello
- ogni nodo ha il massimo di adiacenze
- tutte le adiacenze sono nello stesso livello

E' quindi un numero dato dalla moltiplicazione dei vertici della SPE x il grado massimo dei vertici.

---

unsigned int marked[MARKED_SIZE];

La grandezza di marked dipende dal numero di vertici , ma è sicuramente inferiore a Qi e Qnext in quanto vengono occupati 4 bytes (pari a un unsigned int) ogni 32 vertici

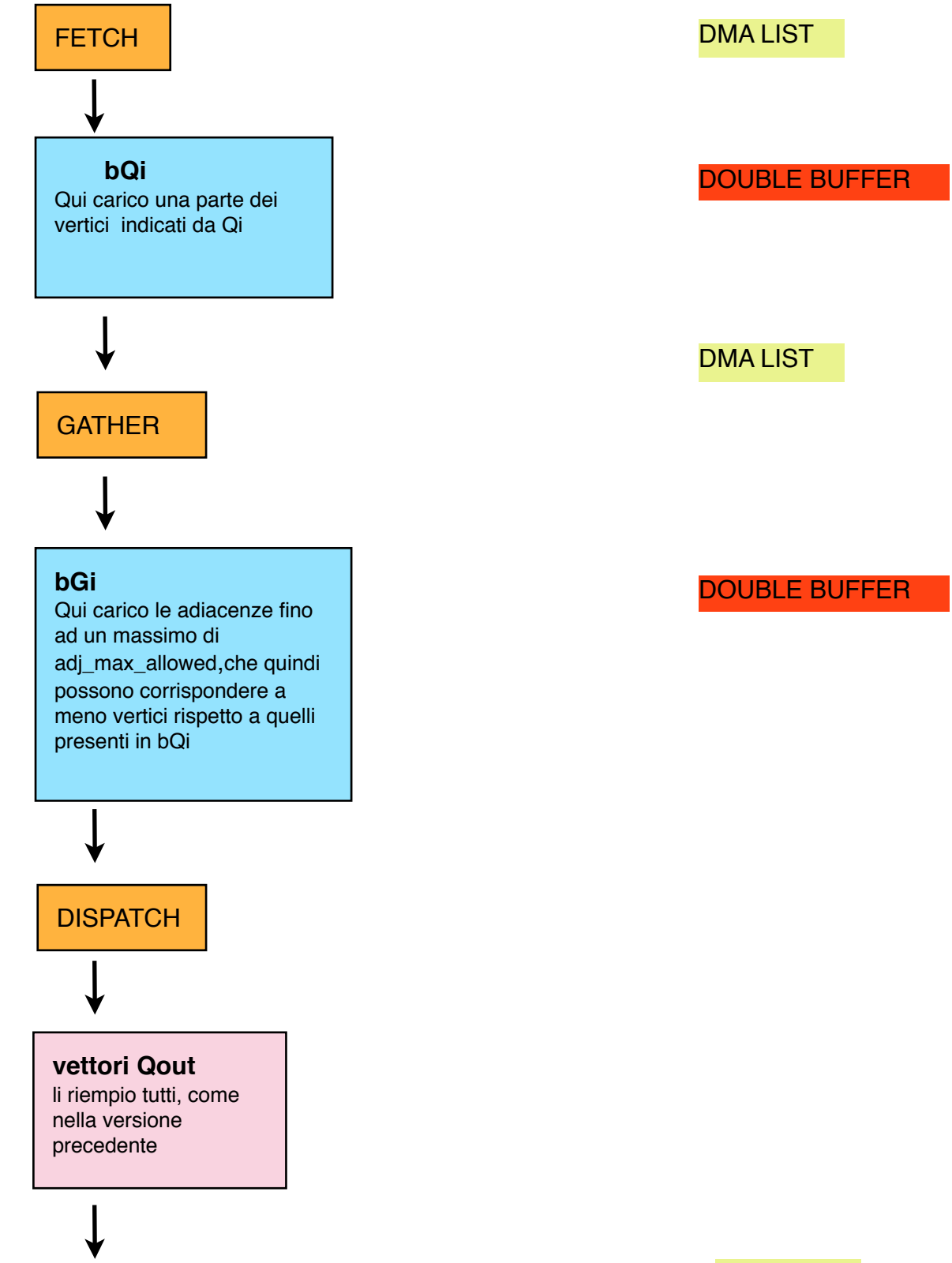
---

Limiti temporali:

Il principale ostacolo per ottenere un *throughput* migliore è la mancanza di *overlapping* tra caricamenti DMA e computazione.

Più precisamente :
- quando bisogna svolgere più operazioni DMA (perché per esempio bisogna comunicare con tutte le altre SPE) un ciclo for svolge un'operazione DMA alla volta. Sarebbe meglio andare avanti con la computazione intanto che i DMA vengono portati a termine dall'MFC.
- Le prime tre parti dell'algoritmo (che portano alle code Qout) avvengono sotto un ciclo for, fino a che tutte le adiacenze non sono state scritte nelle code. Sarebbe meglio proseguire con le ultime tre parti intanto che le prime continuano a caricare nuove adiacenze.

FUNZIONAMENTO SPE ver 1.5



ANALISI DELL'IMPLEMENTAZIONE 1.5

Così com'è questa implementazione non risolve i problemi di memoria e temporali della versione 1.0, ma sicuramente è una dimostrazione di qual è la via da percorrere per ottenere risultati sensibilmente migliori.
Infatti, dal punto di vista **temporale**, se anche la prima parte è ottimizzata grazie alla tecniche di double buffering, la seconda agisce da collo di bottiglia. Inoltre, i vantaggi sull'utilizzo della **memoria** non si vedono, perché oltre ai buffer devo utilizzare le stesse strutture della versione 1.0 per far si che la seconda parte funzioni.

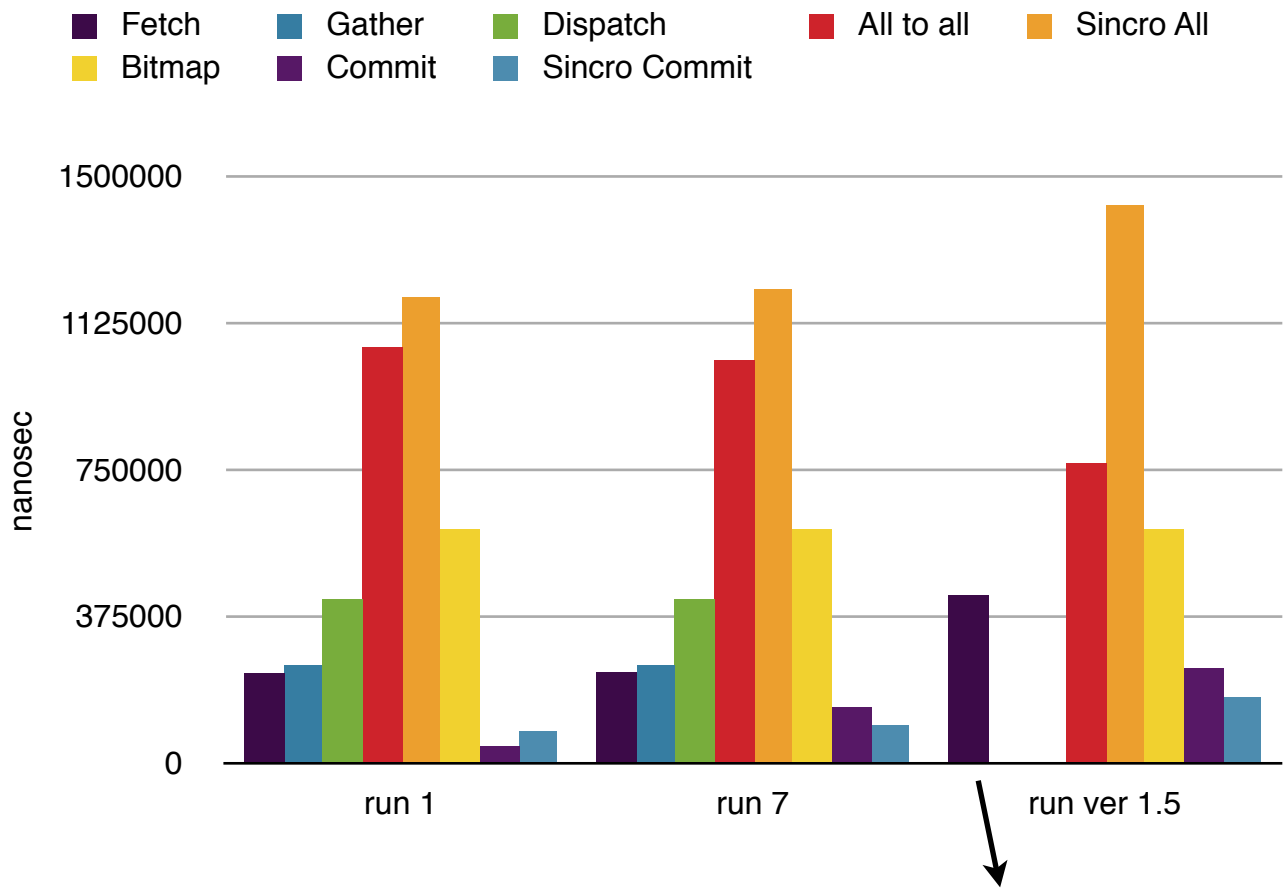
Per arrivare a una versione finale avrei dovuto implementare le tecniche di *double buffering* anche alla seconda parte dell'algoritmo. Facendo così:

- non solo avrei ottenuto un miglioramento sostanziale delle prestazioni grazie a un totale **overlapping** tra trasferimenti e computazione
- ma anche avrei potuto utilizzare delle code **Qout e Qin** molto più piccole, essendo dei buffer predefiniti e non dipendenti dal numero dei vertici assegnati alla SPE. Infine , spostando **Qi** totalmente nella memoria Ram e riducendo **Qnexti** a un buffer predefinito, lo memoria libera crescerebbe totalmente rimanendo a disposizione solo del vettore **marked** (che quindi rimarrebbe l'unico vettore con una grandezza variabile e direttamente dipendente dal numero di vertici)

PRESTAZIONI

2 run estratti da una serie di 10 run

5700 vertici (950 per SPE) con grado minimo dei vertici pari a 2 e grado massimo pari a 5. root nel vertice 0



NB: nella versione 1.5, il tempo per la parte di **fetch** comprende anche il **gather** e il **dispatch**.