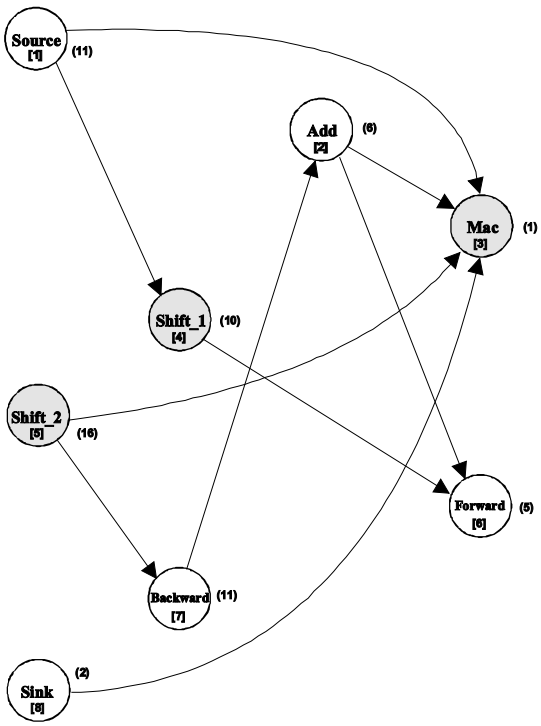
Eugenio Rossini



Schedulazione Concorrente Per Algoritmi Di Digital Signal Processing

a mia moglie,
che ha atteso pazientemente
durante i miei lunghi periodi di studio

INDICE

1. INTRODUZIONE

1.1 Oggetto della tesi

Questa tesi si propone di fornire un metodo che permetta lo sviluppo di uno strumento software in grado di risolvere il problema della ripartizione e della schedulazione temporale delle funzioni o procedure, presenti in un programma sequenziale, su piu' processori al fine di ottenerne una implementazione parallela. Il problema verra' affrontato per una classe ristretta di algoritmi che si identifica essenzialmente in quella per l'elaborazione numerica dei segnali. Una caratteristica che verra' attribuita a questa classe e' che i suoi algoritmi dovranno essere eseguiti in tempo reale intendendo con cio' quanto segue: se definiamo con il termine segnale numerico una sequenza infinita di dati x(n) di un certo tipo (ad esempio interi con prefissato numero di bits) caratterizzato da una frequenza f_x che rappresenta l'inverso del periodo di tempo tra due dati succesivi, elaborare un segnale numerico in tempo reale significa determinare una sequenza infinita di dati y(m)=g{x(n)} (g{} indica la dipendenza eventuale di y(m) da tutti gli x(n): n=$-\infty..\infty$), caratterizzata a sua volta da una frequenza f_y con la capacita' di computazione sufficiente per seguire la velocita' con cui i dati x(n) si presentano all'ingresso.
Tradizionalmente il problema dell'elaborazione in tempo reale di segnali numerici e' sempre stato risolto progettando un hardware ad hoc per ogni diversa applicazione con costi e tempi di realizzazione elevati e flessibilita' pressoche' nulla. Attualmente sono due le vie che si seguono per trattare questo tipo di problematiche, ognuna con le sue peculiarita':

- **A1:** progettazione di un hardware dedicato ma che sia single-chip, ovvero contenuto in un solo circuito integrato;

- **A2:** progettazione di un software capace di essere eseguito su sistemi multi-processore;

Entrambi gli approcci sono basati sulla stesura di un programma, descritto in due linguaggi di programmazione differenti, compilati in modo da realizzare rispettivamente un circuito logico su un supporto semiconduttore o una sequenza di istruzioni assembler su un microprocessore.
Questa tesi si propone dunque di affrontare il problema dell'elaborazione in tempo reale di un segnale numerico seguendo l'approccio A2 con l'obiettivo di permettere la descrizione dell'intero software di elaborazione tramite un programma sequenziale tradizionale per poi automatizzare la sua suddivisione sui processori del sistema.
L'impostazione del lavoro rispettera' la seguente organizzazione:

- Nel CAP.2 verra' definita rigorosamente sia la classe di algoritmi che si intende prendere in considerazione sia il sistema multiprocessore su cui si intende ripartire e schedulare un algoritmo di tale classe. Verra' inoltre introdotta la schematizzazione del problema sottoforma di grafo e del sistema sottoforma di automa a stati finiti.
- Nel CAP.3 si analizzera' il problema fondamentale del raggiungimento di uno stato di blocco della computazione che sorge in modo naturale nelle strutture concorrenti. In

particolare si dimostrera' che, sotto opportune ipotesi, si puo' evitare che la computazione subisca un arresto indesiderato.

- Nel CAP.4 verra' affrontato il problema di ottimizzazione delle risorse connesso alla ripartizione delle funzioni di un programma nativo sequenziale su piu' processori. Si otterra' anche un metodo per la schedulazione temporale dei sottoprogrammi dell'algoritmo dato sul sistema multiprocessore.
- Nel CAP.5 infine si descrivera' un procedimento automatico di trasformazione di un programma sequenziale in uno concorrente.

Verranno usati frequentemente i termini "processore" e "task". Con il termine "processore" si intendera' informalmente una qualsiasi macchina sequenziale di tipo Von Neumann. Con il termine "task" verra' a volte sottointesa informalmente una qualsiasi funzione o procedura nel significato che questi termini hanno nei comuni linguaggi di programmazione. Verra' specificato quando invece tale termine assume un diverso significato ovvero il significato che ha nei linguaggi di programmazione concorrente.

1.2 Modelli di calcolo parallelo

I modelli usati per l'analisi di algoritmi sequenziali sono essenzialmente due: gli automi a stati finiti (FSA) e le macchine di Turing (TM). La differenza fondamentale tra questi due modelli sta nel fatto che una macchina di Turing puo' usare durante la computazione una infinita' numerabile di locazioni di memoria (allocate su un nastro ad accesso sequenziale) mentre un automa a stati finiti puo' far uso solo di una memoria finita. Questa differenza comporta che ogni algoritmo emulato con un automa a stati finiti puo' essere anche eseguito su una macchina di Turing mentre non e' vero in generale il viceversa. Dunque $FSA \subset TM$.
I modelli di calcolo parallelo si sono sviluppati in due direzioni principali: le reti di automi a stati finiti (AN) e le reti neurali discrete (DNN) [1]. Entrambi i tipi di reti sono costituite da una interconnessione casuale di macchine sequenziali: gli automi a stati finiti per le AN, i neuroni per le DNN. Un neurone e' un semplice elemento di calcolo che in genere effettua la computazione $u=\sigma(\Sigma_i w_i x_i)$ dove gli x_i sono gli ingressi al neurone a cui sono associati dei pesi w_i, σ e' una funzione a soglia ed u l'uscita. Il fatto che le reti neurali siano di tipo discreto significa che i valori delle quantita' w_i e x_i sono rappresentabili con un numero finito di cifre. Poiche' un neurone puo' essere ritenuto una particolare macchina a stati, si puo' affermare $DNN \subset AN$. In entrambi i casi le reti possono essere costituite da una quantita' finita o infinitamente numerabile di elementi.
Negli ultimi anni ha suscitato molto interesse una sottoclasse di AN, gli automi cellulari (CA). La caratteristica fondamentale degli automi cellulari e' quella di possedere una interconnessione molto regolare ed elementi di calcolo molto semplici. Ciononostante, quando costituiti da una infinita' numerabile di elementi, essi possono esibire comportamenti molto complessi [2]. Ovviamente si ha $CA \subset AN$.
Il comportamento dei modelli di calcolo parallelo varia radicalmente in funzione della finitezza o meno del numero di elementi che compongono la rete. Se la rete e' infatti costituita da un numero finito di elementi, il sistema, dopo un transitorio iniziale, evolvera' sempre verso un attrattore costituito da un ciclo con un numero finito di stati. Se la rete ha invece una infinita' numerabile di elementi, la sua evoluzione puo' essere molto piu' complessa ed in certi casi puo' anche dare origine a fenomeni caotici.
Nel presente lavoro sara' necessario usare un modello di calcolo parallelo per descrivere l'evoluzione di un sistema multiprocessore. Il modello si conformera' in modo naturale come una AN con un numero finito di elementi. Un sistema di questo tipo puo' sempre essere considerato come un automa a stati finiti il cui stato sia un vettore degli stati dei singoli automi. Il modello permettera' di esaminare le caratteristiche del sistema quali lunghezze dei transitori e dei periodi dell'attrattore, gli stati di blocco della computazione, ecc.

1.3 Linguaggi per la programmazione avanzata

La progettazione di sistemi di elaborazione in tempo reale di segnali numerici, seguendo i due approcci A1 e A2, e' avvenuta fino a tempi recenti essenzialmente tramite i seguenti strumenti software:

- A1: CAD/CAE (Computer Aided Design/Computer Aided Engineering): strumenti che permettono il disegno di un circuito logico, la sua compilazione in una lista di interconnessione e la sua simulazione per la verifica del funzionamento;
- A2: Assemblatori / Compilatore-C: strumenti che permettono la stesura di un programma sequenziale, la sua compilazione in codice macchina e la sua simulazione per la verifica;

Tali metodi sono tuttora usati e validi pur presentando alcune limitazioni. Per quanto riguarda i sistemi CAD/CAE la limitazione maggiore e' dovuta alla loro inadeguatezza a trattare progetti di dimensione elevata, si pensi che e' possibile attualmente raggiungere complessita' di circuiti integrati del taglio di 1Mgates (si parla di questo campo facendo uso della sigla VLSI (Very Large Scale of Integration)). Per cio' che riguarda invece gli assemblatori/compilatori-C essi permettono la stesura di programmi sequenziali senza fornire alcun metodo per la verifica del funzionamento di piu' moduli intercomunicanti eseguiti su piu' processori. Queste limitazioni hanno evidenziato la necessita' di strumenti software piu' efficaci la cui creazione e' dovuta per entrambi gli approcci A1 e A2 al Dipartimento della Difesa degli Stati Uniti (DOD) che ha definito due linguaggi di programmazione di cui il primo inteso a sostituire il tradizionale metodo di "disegno dello schema elettrico" per la progettazione di circuiti e l'altro in grado di supportare la programmazione concorrente. I due linguaggi in questione sono:

VHDL (VHSIC(Very High Speed Integrated Circuit) Hardware Description Language):
Il linguaggio VHDL permette di descrivere l'algoritmo da implementare suddividendolo in entita' concorrenti che comunicano attraverso un tipo speciale di variabili (signal). Ogni entita' descrive una funzione allo stesso modo in cui verrebbe descritta una funzione in un linguaggio tradizionale, la differenza sostanziale sta nel fatto che mentre un programma tradizionale dovra' essere eseguito facendo uso di una risorsa ben precisa (ad esempio la CPU di computer) un programma VHDL creera' una struttura hardware per ogni operazione specificata (ad esempio se la funzione conterra' due somme verranno allocati due addizionatori costituiti da due serie di full-adders ognuna di dimensioni pari al numero di bits in cui si rappresenta il tipo di variabile usata come operando). Il passaggio dal sorgente VHDL al "compilato" avviene tramite tools, cosidetti di sintesi, che risolvono, in modo piu' o meno efficiente, il problema di minimizzazione delle "gates" logiche allocate e del tempo di risposta del circuito che realizzano.

ADA (Dal nome di Ada Byron)[1] :
Il linguaggio ADA incorpora le caratteristiche dei linguaggi Pascal-like recependo allo stesso tempo molti degli sviluppi relativi ai concetti della programmazione maturati negli anni 70 come:

- la modularita' logica: dati, tipi e sottoprogrammi possono essere organizzati in package;
- la modularita' fisica: un programma puo' essere suddiviso in moduli da compilare separatamente;
- la programmazione concorrente: una unita' di programma puo' contenere piu' "task" che evolvono in parallelo con possibilita' di sincronizzazione e comunicazione;
- il trattamento delle eccezioni;

Il linguaggio ADA e' quindi adatto sia per la programmazione convenzionale che per applicazioni dedicate ed in tempo reale.
Nella stesura della tesi, anche per descrivere gli algoritmi, verra' usato un linguaggio pseudo-ADA. Le caratteristiche sintattico-semantiche del linguaggio verranno richiamate man mano che ne verra' fatto uso.

[1] La contessa Augusta Ada Byron fu assistente del matematico Charles Babbage, ed e' passata alla storia come primo programmatore di calcolatori del mondo avendo scritto un programma per calcolare i numeri di Bernoulli su una macchina calcolatrice inventata da Babbage.

2. LA CLASSE DI ALGORITMI DSP

2.1 Introduzione

Scopo di questo capitolo e' definire rigorosamente una classe di algoritmi sequenziali e la macchina costituita da piu' processori sulla quale si intende eseguirli in modo concorrente. Per far cio' si introdurra' un grafo associato all'algoritmo ed un automa a stati finiti che descriva l'evoluzione della macchina parallela. In questo modo sara' possibile trattare l'algoritmo facendo uso della teoria dei grafi ed esaminare il comportamento del sistema studiando le proprieta' dell'automa associato.

2.2 Algoritmi DSP e macchina parallela APM

Definizione.2.1 (Classe di algoritmi DSP). Diciamo che un algoritmo e' di classe DSP, o che e' un DSPA, se esso ha una struttura del tipo riportato in figura 2-1 dove:

- "while true loop" sta ad indicare che l'elaborazione avviene senza mai arrestarsi; da notare che la presenza di questo ciclo implica la possibilita' di assegnare valori a variabili che verranno usate nella iterazione successiva;
- "source(x)" indica una procedura predefinita in grado di leggere da una sorgente infinita una prefissata quantita' di dati che tramite il vettore x sara' resa disponibile come ingresso alle procedure o funzioni successivamente chiamate;
- "block" indica la possibilita' di effettuare chiamate a funzioni o procedure innestandole fino ad un livello finito qualsiasi di cicli "for" purche', come dichiarato, il numero di cicli richiesti da ogni "for" sia costante o al piu' dipendente dall'indice di un ciclo di livello gerarchico superiore;
- "sink(y)" indica una procedura predefinita in grado di scrivere in un pozzo infinito i risultati dell'elaborazione organizzati come un vettore y di prefissata lunghezza;

.

Si e' cercato con la precedente definizione di individuare la piu' generica classe di algoritmi il cui programma principale (main) contenga solo istanze completamente decidibili a tempo di compilazione, lasciando le istanze decidibili a tempo di esecuzione (run-time) confinate nei sottoprogrammi. Questa ipotesi e' praticamente sempre vera quanto si tratta di descrivere algoritmi per l'elaborazione numerica dei segnali che sono in genere costituiti da un insieme di algoritmi di calcolo numerico interconnessi variamente in modo da formare cascate o cicli. Le istanze decidibili a tempo di esecuzione, tipicamente costrutti "if-then-else", sono rare negli algoritmi per l'elaborazione numerica dei segnali. Quando esse siano presenti si supporra' che non comportino comunque grandi varianze dei tempi di esecuzione delle funzioni che le contengono cosicche' si possa assumere come tempo di esecuzione della funzione il suo tempo medio.
Introduciamo ora il modello di macchina parallela sulla quale si vuole implementare un DSPA. Questo modello ha un forte legame con i moderni sistemi di calcolo multiprocessore nei quali esiste una

connessione bidirezionale tra due qualsiasi processori del sistema. Tutti i processori del sistema elaborano in modo asincrono e la sincronizzazione tra i tasks avviene esclusivamente in base alla disponibilita' dei dati. Inoltre si ipotizza che ogni connessione bidirezionale esistente tra due qualsiasi processori sia rappresentabile tramite un insieme finito di code bidirezionali di dimensione finita nelle quali ogni locazione sia anch'essa di dimensione finita prefissata ma in generale non unitaria, permettendo cosi' il transito di variabili strutturate tipo array. Il concetto di coda viene in questo modo ad assumere un significato astratto distinto da quello di connessione tra processori. In particolare si potra' parlare di una connessione "fisica" e di una connessione "logica" rispettivamente per intendere la connessione tra processori e la connessione tra funzioni e procedure di un DSPA.

```
procedure DSP_Algorithm is

        type vector is array(natural range <>) of integer;
        subtype source_vector is vector(0..constant);
        x: source_vector;
        subtype sink_vector is vector(0..constant);
        y: sink_vector;

        "constants and variables definitions"

begin

        "variables initializations"

        while true loop

                source(x);

                -- block 1
                for k1 in 0..constant loop
                        for k2 in 0.."constant or index" loop
                                ......
                                ......
                                        for km in 0.. "constant or index" loop
                                                "functions or procedures calls list"
                                        end loop;
                                ......
                                ......
                                "functions or procedures calls list"
                        end loop;
                        "functions or procedures calls list"
                end loop;
                "functions or procedures calls list"

                -- block 2
                .......
                -- block n

                sink(y);

        end loop;

end DSP_Algorithm;
```

Figura 2-1 - DSPA

La figura 2-2 illustra la j-esima connessione logica unidirezionale supposta costituita da n code. In ascisce e' riportata la dimensione delle code; le dimensioni k_{ij}, dei vettori elementi della i-esima coda della connessione, ed il numero delle code presenti sono invece riportati in ordinate.

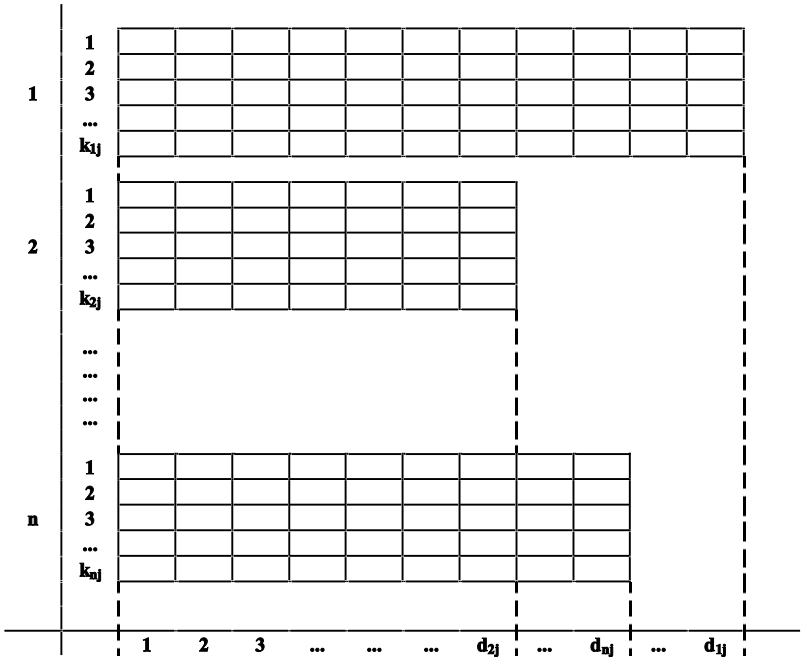


Figura 2-2. Connessione Logica Unidirezionale

Definizione.2.2 (Macchina Parallela Asincrona). Chiamiamo Macchina Parallela Asincrona, o brevemente APM, un insieme finito di macchine sequenziali di tipo Von Neumann interconnesse a due a due tramite un insieme finito di code bidirezionali di dimensione finita.

Se schematizziamo con una una freccia bidirezionale la connessione fisica tra due processori, un esempio di APM a 5 processori e' il seguente:

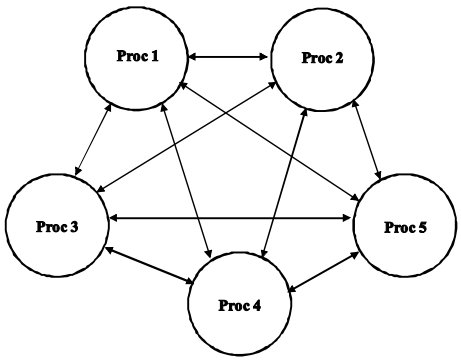


Figura 2-3. Interconnessione Fisica tra Processori

Occorre a questo punto formalizzare un metodo per mappare un DSPA su una APM. Un modo naturale e' assegnare una o piu' chiamate a funzioni o procedure presenti nel "while true loop" a ciascun processore ed ogni variabile scambiata tra due di tali chiamate ad una coda. In particolare cio' si presuppone che avvenga anche tra due funzioni o procedure assegnate allo stesso processore. La coda sara' in quest'ultimo caso di natura esclusivamente logica e non usera' nessuna connessione fisica.
Si supporra' inoltre che il tipo di variabili scambiate tra funzioni e/o procedure sia in generale un vettore di interi. Questa ipotesi non e' restrittiva per un DSPA ne' pregiudica in generale il transito di variabili strutturate di tipo diverso non appena esse siano opportunamente codificate.
Da notare che con il precedente metodo ad un dato processore viene assegnata una chiamata a procedura o funzione e non la sua dichiarazione. Cio' comporta che la dichiarazione di una stessa procedura o funzione possa essere assegnata piu' volte a processori diversi causando un aumento della memoria totale da destinare all'allocazione del codice (algoritmo compilato). Sorge cosi' un primo problema di ottimizzazione relativo alla ripartizione dei tasks di un DSPA su piu' processori:

Definizione.2.3 (I problema di ripartizione) Chiamiamo I problema di ripartizione, il problema di assegnare i tasks di un DSPA a piu' processori in modo da minimizzare lo spazio di memoria complessivamente occupato dalla dichiarazione (compilata) dei tasks, ovvero in modo da minimizzare le duplicazioni di dichiarazioni.

Il secondo problema di ottimizzazione e' di natura temporale e riguarda il tempo di esecuzione concorrente del DSPA. Esso puo' essere definito come segue:

Definizione.2.4 (II problema di ripartizione) Chiamiamo II problema di ripartizione, il problema di assegnare i tasks di un DSPA a piu' processori in modo da minimizzare il tempo di esecuzione concorrente di un "while true loop" su una APM.

L'assegnazione di funzioni e procedure ai processori della APM deve dunque avvenire in modo da soddisfare i precedenti due requisiti di ottimizzazione. I due problemi I e II dovranno essere quindi considerati contemporaneamente e risolti in modo da minimizzare entrambe le grandezze spaziali e temporali introdotte.
Piu' in generale, si potrebbe considerare una funzione obiettivo f che tenga conto, oltre che della grandezze tempo T e spazio S, anche del numero di processori N_{PROC} e del numero di interconnessioni logiche I_L e fisiche I_P, modulando l'ottimizzazione attraverso dei coefficienti peso λ_k che esaltino o attenuino i singoli aspetti del problema:

$$f := \lambda_1 T + \lambda_2 S + \lambda_3 I_L + \lambda_4 I_P + \lambda_5 N_{PROC}$$

In realta' poiche' nelle applicazioni industriali le schede multiprocessore standard hanno un numero prefissato di processori e di connessioni fisiche nel corso di questa tesi si assumera' $\lambda_4=\lambda_5=0$. Verra' inoltre anche posto $\lambda_3=0$ tralasciando l'ottimizzazione rispetto alla quantita' di dati scambiati tra processori in quanto i processori di ultima generazione sono sempre dotati di uno o piu' dispositivi DMA[2] che si occupano del trasferimento dati senza costi di elaborazione aggiuntivi per il processore stesso.

2.3 Grafo e automa associati ad un DSPA

Siamo ora in grado di dare le seguenti definizioni di grafo e automa associati ad un DSPA.

Definizione.2.5 (Grafo associato ad un DSPA) Se $\{f_i\}$ e' l'insieme delle chiamate a funzione o procedura di un DSPA e $\{var_{ij}(k)\ k>0\}$ l'insieme delle variabili scambiate tra le coppie di chiamate a funzione o

[2] DMA sta per "Direct Memory Access" ed e' un meccanismo per evitare che un trasferimento dati passi per la ALU (Arithmetic Logic Unit) della CPU (Central Processing Unit) usufruendo del tempo macchina del processore.

procedura (f_i,f_j), chiamiamo grafo associato al DSPA il grafo diretto debolmente[3] connesso G=(V,E) i cui nodi ed archi siano rispettivamente:

V={x_i: per ogni i esista una f_i};
E={a_{ij}: per ogni i,j esistano (f_i,f_j) e k>0 tali che var_{ij}(k) sia dichiarata "out" dalla f_i ed "in" dalla f_j}[4]

Nel caso in cui una variabile strutturata tipo array che sia di "out" per una f_i e di "in" per piu' f_k, k=1..r, su elementi diversi dell'array, si intende che vengano generati r archi dal nodo x_i ai nodi x_k, k=1..r. Analogamente per una variabile scalare che sia di "out" per una f_i e di "inp" per piu' f_k. La differenza nei due casi sta nel fatto che mentre nel primo gli r archi corrispondono ad r code contenenti elementi distinti dell'array, nel secondo le r code conterranno lo stesso elemento scalare. Infine se piu' variabili verranno scambiate tra due funzioni nella medesima direzione si intendera' che ad esse venga associato un unico arco. Da notare anche che per come e' definito un DSPA il grafo associato avra' un solo nodo "source" ed un solo nodo "sink".

Esempio. Per illustrare come opera la precedente definizione, consideriamo il seguente "while true loop":

```
procedure esempio is

        type vector is array(natural range <>) of integer;
        n: constant natural:=4;
        subtype source_vector is vector(1..n);
        x,a,t: source_vector;
        u,b: integer;

begin

        u:=0;
        while true loop
                source(x);
                for k in 1..n loop
                        f(k,x(k),t(k), a(k));
                end loop;
                b:=g(a,x(1),x(2),u);
                u:=h(b,x(3));
                sink(b);
        end loop;

end esempio;
```

Nel quale le funzioni e procedure presenti fanno riferimento alle seguenti dichiarazioni:

[3] Da qui in avanti con il termine "connesso" si intendera' "debolmente connesso".
[4] Il linguaggio ADA permette di dichiarare se una variabile e' di "in" o di "out" per una data procedura o funzione.

```
procedure f(x: in natural; y: in integer; w: in out integer; z out integer);

function g(x: in source_vector; y,w,z: in integer) return integer;

function h(x,y: in integer) return integer;

procedure source(x: out source vector);

procedure sink(x: in integer);
```

In accordo alla precedente definizione l'associazione tra nodi del grafo e chiamate a funzioni o procedure e' la seguente:

chiamata a funzione o procedura	**nodo del grafo**
source()	f1
f()	f2, f3, f4, f5
g()	f6
h()	f7
sink()	f8

mentre l'associazione tra variabili ed archi e' illustrata dalla seguente tabella:

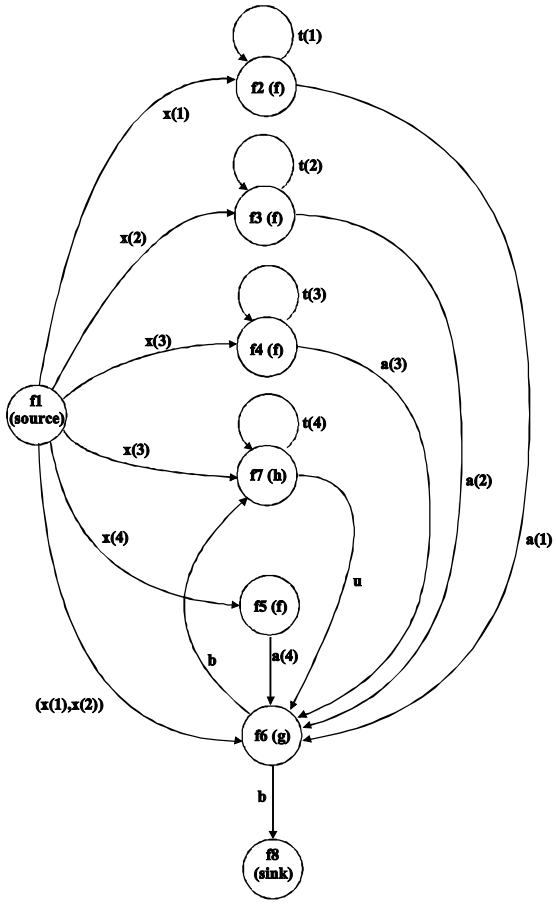
variabile		**archi del grafo associato**
x (vettore di 4 elementi)	x(1)	a_{12}=(f1,f2)
	x(2)	a_{13}=(f1,f3)
	x(3)	a_{14}=(f1,f4), a_{17}=(f1,f7)
	x(4)	a_{15}=(f1,f5)
	(x(1),x(2))	a_{16}=(f1,f6)
t (vettore di 4 elementi)	t(1)	a_{22}=(f2,f2)
	t(2)	a_{33}=(f3,f3)
	t(3)	a_{44}=(f4,f4)
	t(4)	a_{55}=(f5,f5)
a (vettore di 4 elementi)	a(1)	a_{26}=(f2,f6)
	a(2)	a_{36}=(f3,f6)
	a(3)	a_{46}=(f4,f6)
	a(4)	a_{56}=(f5,f6)
u (integer)	u	a_{76}=(f7,f6)
b (integer)	b	a_{67}=(f6,f7), a_{68}=(f6,f8)

Da notare che:

- Il vettore di due elementi (x(1),x(2)) da' luogo ad un unico arco perche' entrambi x(1) ed x(2) sono scambiati tra le stesse due funzioni f1 ed f6, e nella medesima direzione. La corrispondente coda avra' allora larghezza 2.
- Le due code individuate dagli archi (f1,f4) ed (f2,f7), in uscita dal nodo f1 (source), conterranno lo stesso dato pur essendo distinte a causa del differente nodo destinazione. Analogamente per (f6,f7) e (f6,f8).

- L'indice k non costituisce una variabile e quindi ad essa non verra' associato nessun arco del grafo e nessuna coda del sistema. L'uso di un indice come argomento di una funzione o procedura non provoca quindi violazioni dello schema generale di un DSPA ed e' utilizzabile per parametrizzare una famiglia di funzioni.

Il grafo associato al DSPA descritto dal precedente "while true loop" e' allora quello riportato nella seguente figura (si nota anche la presenza di alcuni autocicli il cui significato in termini di computazione dell'algoritmo verra' discusso successivamente):



In figura 2-4 e' riportata una possibile interconnessione logica tra processori che evidenzia il grafo associato al precedente DSPA. Le code contenenti le variabili x(3) e x(4) useranno la connessione fisica esistente tra i processori N.1 e N.2. Le code contenenti le variabili a(1), a(2) e (x(1),x(2)) useranno la connessione fisica tra i processori N.2 e N.3 ed infine le code contenenti le variabili a(3), a(4), u e b (quest'ultima tra le funzioni f6 ed f7) useranno la connessione fisica tra i processori N.1 e N.3. Tutte le altre variabili, ovvero x(1), x(2) e b (quest'ultima tra le funzioni f6 ed f8) useranno solo connessioni logiche all'interno di un medesimo processore.

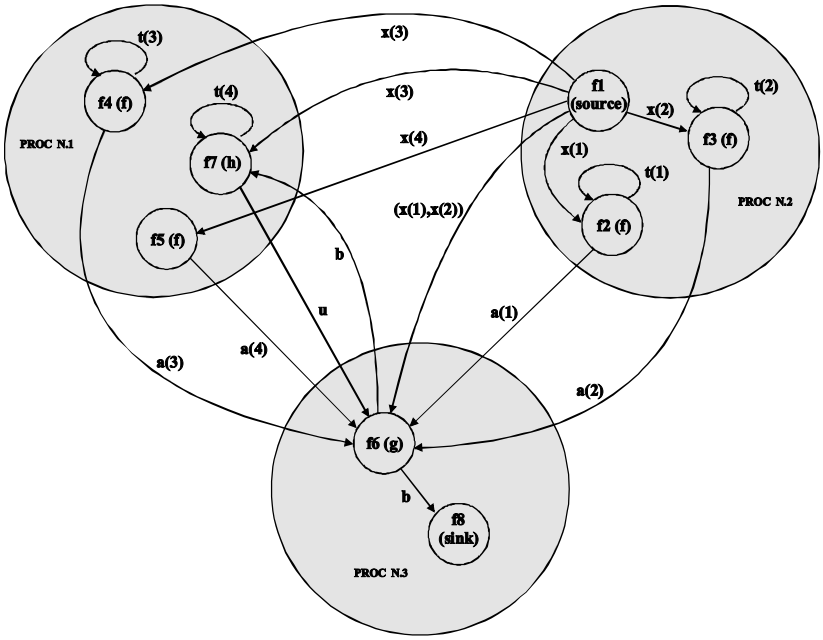


Figura 2-4 - Interconnessione Logica tra Processori

Definizione.2.6 (Automa associato ad un DSPA) Se N e' l'insieme degli interi non negativi, chiamiamo automa associato ad un DSPA una quaterna A=(G,Q,F,**q**) dove:

1. G e' il grafo associato al DSPA;
2. Q e' un insieme di |E| code di dimensione Q^k, k=1..|E|; ognuna di tali code corrisponde ad un arco di G e verra' detta coda di uscita o di ingresso ad un certo nodo rispettivamente se il corrispondente arco e' uscente o entrante nel nodo;
3. $\mathbf{q}(n)=(q^1(n), q^2(n), \ldots q^{|E|}(n))$ e' un vettore tale che, $q^k(n)\in N$, $0\le q^k(n)\le Q^k$, k=1..|E| e rappresenta lo stato dell'automa ad un certo istante (discreto) n, ovvero la quantita' di dati presenti nelle code all'istante n;
4. F e' la legge di evoluzione che permette il passaggio da uno stato al successivo; essa risulta quindi essere una applicazione $F:N^{|E|}\rightarrow N^{|E|}$ tale che $\mathbf{q}(n)=F(\mathbf{q}(n-1))$;

L'automa e' a stati finiti in quanto il numero complessivo di stati possibili e' ovviamente dato da $\Pi_{i=1..|E|}(Q^i+1)$.

La precisazione della legge F richiede alcune ulteriori considerazioni.

Definizione.2.7 (Eseguibilita' di un nodo) Un nodo x_k, ovvero la procedura o funzione ad esso associata, si dice eseguibile se la sua esecuzione, che causa la diminuzione di una unita' della quantita' di dati presenti in tutte le sue code di ingresso e l'aumento di una unita' della quantita' di dati presenti in tutte le sue code di uscita, non causi violazioni della:

$$0\le q^k(n)\le Q^k, \qquad k=1..|E|, \quad n\ge 0$$

Definizione.2.8 (Legge di evoluzione) La legge di evoluzione F e' tale che permetta l'esecuzione di un nodo sse tale nodo e' eseguibile. Se R e' l'insieme dei numeri razionali, detti:

- $\mathbf{v}(n)=(v^1(n), v^2(n), \ldots v^{|V|}(n))$ il vettore booleano tale che $v^k(n)=1$ sse x_k e' eseguibile all'istante n, 0 altrimenti;

- A_G la matrice |E|x|V| di incidenza del grafo G:

  $(A_G)_{ij}$ = 1 se l'arco i e' uscente dal nodo j;
  $(A_G)_{ij}$ = -1 se l'arco i e' entrante nel nodo j;
  $(A_G)_{ij}$ = 0 altrimenti;

- $\sigma: R \rightarrow \{0,1\}$ l'applicazione tale che $\sigma(m)=1$ sse $m>0$, 0 altrimenti;

e tenuto conto che l'aggiornamento delle code si puo' ottenere dal prodotto $A_G\mathbf{v}$, la F puo' essere formalizzata come segue:

$$F(\mathbf{q}(n)) = A_G\, \mathbf{v}(n) + \mathbf{q}(n)$$

$$v^k(n) = \sigma\left(\prod_{i=1}^{|E|}\left(\frac{1+(A_G)_{ik}}{2} Q^i - (A_G)_{ik}\, q^i(n)\right)\right) \qquad k=1..|V|$$

infatti la precedente fornisce:

$v^k(n)$=1 sse tutte le code di ingresso al nodo x_k sono non vuote e tutte le sue code di uscita sono non piene;

in quanto:

se	la coda corrispondente all' i-esimo arco e'	il corrispondente fattore della produttoria si riduce a:
$(A_G)_{ik}=1$	uscente dal nodo k-esimo	Q^i-q^i
$(A_G)_{ik}=-1$	entrante nel nodo k-esimo	q^i
$(A_G)_{ik}=0$	non interessa il nodo k-esimo	$Q^i/2$

Osservazione.2.9 (Auto-Cicli) Una matrice di incidenza non e' in grado di rappresentare la presenza di archi che collegano un nodo con se stesso e di conseguenza la presenza di code tra una medesima chiamata a procedura o funzione. D'altra parte per l'evoluzione dell'automa *A* la presenza o meno di auto-cicli non e' significativa in quanto le code ad essi associate non evolvono. Infatti il numero di dati in esse contenuti resta costante ed identico al valore iniziale in quanto l'esecuzione di una procedura o funzione contemporaneamente ne diminuisce ed aumenta il contenuto di una unita'. E' importante notare che se nel grafo e' presente un auto-ciclo la chiamata a procedura o funzione corrispondente al nodo che subisce l'auto-ciclo non sara' mai eseguita se inizialmente nella coda non fosse presente almeno un dato. Cio' introduce un problema generale di blocco della computazione che verra' trattato nel prossimo capitolo.

3. AUTOMA ASSOCIATO AD UN DSPA

3.1 Introduzione

Si ricaveranno in questo capitolo gli stati di blocco della computazione per l'automa associato ad un certo DSPA e si dimostrera' che essi sono raggiungibili solo da stati iniziali che abbiano tutti i valori delle code appartenenti a cicli coincidenti con quelli dello stato di blocco stesso. Si dimostrera' cioe' che tali stati di blocco sono in un certo senso dei punti isolati. In particolare si vedra' che se l'automa ha uno stato iniziale $\mathbf{q}(0)=0$ solo i cicli equiverso (cicli di un grafo diretto percorribili completamente seguendo il verso degli archi) possono causare un blocco della computazione. Si otterra' allora che inizializzando una coda per ciclo, ad esempio con un solo dato, in modo che cio' non causi una configurazione di blocco per eventuali altri cicli presenti, ed in accordo con la inizializzazione effettuata nel programma sequenziale originale, la computazione non si arresti.

3.2 Punti fissi e stati di blocco della computazione

Il primo problema da affrontare riguarda la possibilita' che l'automa associato ad un DSPA raggiunga uno stato che comporti il blocco della computazione. Per definire rigorosamente lo stato di blocco della computazione, premettiamo la seguente proposizione.

Proposizione.3.1 (Periodicita', Punti Fissi) Un automa a stati finiti evolve sempre, a partire da uno stato iniziale, in modo da cadere in un ciclo di periodo $p>0$ detto punto fisso di periodo p:

$$\exists\, m>0,\ p>0 \text{ tali che } \forall n>m:\ \mathbf{q}(n+p)=\mathbf{q}(n)$$

Dimostrazione.

- Dalla $\mathbf{q}(m+1)=F(\mathbf{q}(m))$ si ha anche $\mathbf{q}(m+2)=F(\mathbf{q}(m+1))=F^{(2)}(\mathbf{q}(m))$ e quindi in generale:

  $$\mathbf{q}(m+n)=F^{(n)}(\mathbf{q}(m)) \qquad \text{per ogni } n>0$$

  dalla quale si nota come ogni nuovo stato raggiunto dall'automa dopo l'istante m sia completamente determinato dallo stato $\mathbf{q}(m)$ raggiunto dall'automa all'istante m, oltreche' dalla legge di evoluzione F.
- D'altra parte il numero dei possibili stati diversi raggiungibili dall'automa dall'istante m compreso in poi e' pari a $M\leq\Pi_{i=1..|E|}\ (Q^i+1)$ dopodiche' necessariamente si dovra' avere $\mathbf{q}(m+M)=\mathbf{q}(m)$ e allora, per quanto detto al precedente punto, l'automa ripercorrera' gli stessi stati percorsi la prima volta a partire da $\mathbf{q}(m)$ cadendo cosi' in un ciclo di periodo M.

Osservazione.3.2 Un punto fisso **q** di periodo p comporta anche la periodicita' di **v**, con periodo $p^* \leq p$, in quanto la legge di evoluzione impone

$$v^k(n) = \sigma\left(\prod_{i=1}^{|E|}\left(\frac{1+(A_G)_{ik}}{2} Q^i - (A_G)_{ik}\, q^i(n)\right)\right) \qquad k=1..|V|$$

e quindi a valori identici di **q** corrispondono valori identici di **v** mentre non e' escluso che si possano avere valori identici di **v** per due diversi valori di **q**.

Definizione.3.3 (Blocco della computazione) Si dice che l'automa *A* associato ad un DSPA ha raggiunto un punto di blocco della computazione se si verifica uno dei seguenti due casi:

1. $\exists\, m>0$ tale che $\forall n>m$: $\mathbf{q}(n)=cost$, $\mathbf{v}(n)=\mathbf{0}$;
2. $\exists\, m>0, p>0$ tali che $\forall n>m, \forall c>0$: $\mathbf{q}(n+p)=\mathbf{q}(n)$, $\sum_{k=0}^{p-1} \mathbf{v}(n+k) \neq \mathbf{c}$

dove **c** indica un vettore colonna |V|-dimensionale con tutti gli elementi dello stesso valore c.

Intuitivamente i due casi precedenti corrispondono rispettivamente a:

1. da un certo istante in poi nessuno dei nodi dell'automa e' eseguibile; si e' raggiunto cioe' un punto fisso di periodo 1 per il quale **v=0**;
2. da un certo istante in poi vengono eseguiti ciclicamente alcuni nodi ma all'interno del periodo di esecuzione non tutti i nodi vengono eseguiti lo stesso numero di volte; si e' raggiunto cioe' un punto fisso di periodo p>1 che non e' utile per una corretta computazione parallela in quanto il "while true loop" presente nella definizione di un DSPA impone che tutte le chiamate a procedura o funzione, presenti al suo interno, siano eseguite lo stesso numero di volte;

Determiniamo ora le condizioni sotto le quali si ottengono dei punti fissi per l'automa *A*. Premettiamo la seguente proposizione.

Proposizione.3.4 (Rango di una matrice di incidenza) Un grafo diretto G=(V,E) e' (debolmente) connesso sse la sua matrice di incidenza A_G ha rango |V|-1.

Dimostrazione. Questa e' una proprieta' algebrica classica di teoria dei grafi. Per una dimostrazione si puo' vedere [3].

Proposizione.3.5 (Punti fissi dell'Automa) Se $\mathbf{q}(n),...,\mathbf{q}(n+p-1)$ e' un punto fisso di periodo p per l'automa *A* allora esiste un intero c tale che, detto **c** un vettore colonna |V|-dimensionale con tutti gli elementi di valore c, si abbia:

$$\sum_{k=0}^{p-1} \mathbf{v}(n+k) = \mathbf{c}$$

Dimostrazione. Se $\mathbf{q}(n),...,\mathbf{q}(n+p-1)$ e' un punto fisso di periodo p per *A*, si ha $\mathbf{q}(n+p)=\mathbf{q}(n)$ per cui:

$$\begin{aligned}
\mathbf{q}(n+1) &= \mathbf{q}(n) &&+ A_G\, \mathbf{v}(n)\\
\mathbf{q}(n+2) &= \mathbf{q}(n+1) &&+ A_G\, \mathbf{v}(n+1)\\
\mathbf{q}(n+3) &= \mathbf{q}(n+2) &&+ A_G\, \mathbf{v}(n+2)\\
\ldots &\quad \ldots &&\ldots\\
\mathbf{q}(n+p-1) &= \mathbf{q}(n+p-2) &&+ A_G\, \mathbf{v}(n+p-2)\\
\mathbf{q}(n) &= \mathbf{q}(n+p-1) &&+ A_G\, \mathbf{v}(n+p-1)
\end{aligned}$$

quindi sommando:

$$\mathbf{0} = A_G \sum_{k=0}^{p-1} \mathbf{v}(n+k) := A_G\, \mathbf{w}(n)$$

ma, per la precedente proposizione, A_G ha rango $|V|-1$ dunque tutte le autosoluzioni del precedente sistema omogeneo nelle incognite $\mathbf{w}(n)$ sono date, per un arbitrario vettore costante $\mathbf{c}$, da:

$$\mathbf{w}(n) = \mathbf{c}$$

in quanto il vettore $\mathbf{1}$ e' soluzione del sistema ed il rango e' pari al numero delle incognite meno uno.

Corollario.3.6 Un punto di blocco della computazione puo' avere solo periodo 1, si riduce cioe' ad uno stato dell'automa *A*.

Dimostrazione. Conseguenza diretta della definizione di punto di blocco della computazione e della precedente proposizione.

Proposizione.3.7 (Punti di blocco della computazione) Se G=(V,E) e' connesso:

1. Se $|E|=|V|-1$ l'automa *A* non ha punti (stati) di blocco della computazione.
2. Se $|E|\geq|V|$ tutti i punti (stati) di blocco della computazione sono dati da vettori $\mathbf{q}_B$ ad $|E|$ elementi di cui $|E|-|V|$ con valori arbitrari, mentre gli altri $|V|$ sono dati da:

$$(3\text{-}1)\quad \left((q_B)^{i_1} = \frac{Q^{i_1}}{2}(1+(A_G)_{i_1\,1}) \;\ldots\; (q_B)^{i_{|V|}} = \frac{Q^{i_{|V|}}}{2}(1+(A_G)_{i_{|V|}\,|V|}) \right)_{\substack{(i_1\, i_2 \ldots i_{|V|}) \in \mathrm{Disp}(|E|,|V|) \\ (A_G)_{i_k\,k} \neq 0 \quad k=1..|V|}}$$

dove $\mathrm{Disp}(|E|,|V|)$ e' l'insieme delle disposizioni degli indici $\{1,2, \ldots , |E|\}$ in classe $|V|$ mentre la $(A_G)_{i_k\,k} \neq 0$, $k=1..|V|$ sta ad indicare che da tutte le $|V|$-ple ottenute al variare degli indici $(i_1\, i_2 \ldots i_{|V|}) \in \mathrm{Disp}(|E|,|V|)$ vanno eliminate quelle per le quali succede che qualche elemento di A_G e' nullo. La presenza di valori arbitrari fa' si che la precedente individui insiemi di vettori, ma ad essi si fara' comunque riferimento come ad "un" punto di blocco.
Cio' equivale a dire che esiste una corrispondenza biunivoca tra i punti di blocco della computazione dell'automa *A* e le matrici $|V|x|V|$ estratte dalla matrice di incidenza A_G del relativo grafo le cui righe sono individuate dalle disposizioni di $|E|$ indici in classe $|V|$ e la cui diagonale principale sia ad elementi $\lambda_k = (A_G)_{i_k\,k}$ $k=1..|V|$ tutti non nulli.
Inoltre la presenza di un "1" sulla diagonale principale in corrispondenza al k-esimo arco del grafo comporta la presenza di Q^k sul k-esimo elemento della soluzione del sistema. La presenza di un "-1" comporta invece la presenza di uno 0.

Dimostrazione. I punti di blocco della computazione sono le soluzioni del sistema di equazioni non lineari:

$$\prod_{i=1}^{|E|} \left(\frac{1+(A_G)_{ik}}{2} Q^i - (A_G)_{ik}\, q^i(n) \right) = 0 \qquad k=1..|V|$$

in esso figurano esattamente due fattori per ciascun arco (coda): q^i e (Q^i-q^i) che si ottengono rispettivamente per $(A_G)_{ik}=-1$ e per $(A_G)_{ik}=1$ (se $(A_G)_{ik}=0$ il fattore si riduce ad una costante diversa da zero e quindi e' ininfluente). Notiamo anche che ogni fattore compare una sola volta nel sistema perche' in una matrice di incidenza non si verifica mai, quando $(A_G)_{ij}=\pm 1$:

$$(A_G)_{ik}=(A_G)_{ih} \qquad h \neq k$$

mentre su una stessa equazione la diversita' dei fattori e' garantita dalla presenza dei q^i.

1. Se $|E|=|V|-1$ il sistema ha $|V|$ equazioni e $|V|-1$ incognite. In totale abbiamo dunque esattamente $2(|V|-1)$ fattori distinti di cui $|V|-1$ del tipo q^i e $|V|-1$ del tipo (Q^i-q^i). Poiche' allora l'azzeramento delle $|V|$ equazioni richiede l'annullarsi di almeno $|V|$ fattori diversi (cioe' almeno uno per ogni equazione) se per assurdo il sistema avesse soluzioni, per qualche j dovrebbe essere contemporaneamente $q^j=0$ e $(Q^j-q^j)=0$ che e' un assurdo.

2. Se $|E|\geq|V|$ per soddisfare la k-esima equazione e' sufficiente che si annulli uno dei suoi fattori, ad esempio l'i_k-esimo ottenendo:

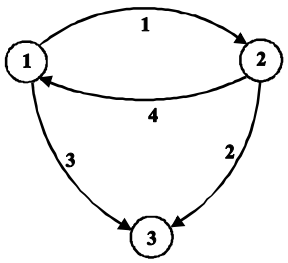
$$q^{i_k}(n)=\frac{Q^{i_k}}{2\,(A_G)_{i_k k}}(1+(A_G)_{i_k k})=\frac{Q^{i_k}}{2}(1+(A_G)_{i_k k}) \qquad k=1..|V|$$

   avendo tenuto conto che $(A_G)_{ik}=\pm 1$.
   Poiche' il sistema e' costituito da $|V|$ equazioni in $|E|\geq|V|$ incognite, $|E|-|V|$ elementi potranno assumere valori arbitrari. Inoltre il modo piu' generale in cui i precedenti fattori possono essere scelti descrive il numero di disposizioni senza ripetizioni di $|E|$ oggetti in classe $|V|$ non potendo mai verificarsi una ripetizione di scelta dello stesso fattore su equazioni diverse perche' cio' comporterebbe come gia detto $(A_G)_{ik}=(A_G)_{ih}$, $h \neq k$ cosa impossibile in una matrice di incidenza quando $(A_G)_{ij}=\pm 1$.

Corollario.3.8 L'automa *A* ha punti di blocco sse il grafo non diretto G', ottenuto ignorando i versi degli archi di G, ha almeno un ciclo.

Dimostrazione. Conseguenza diretta del fatto che un grafo non diretto e' aciclico sse [4] $|E|\leq|V|-1$ e della precedente proposizione.

Esempio. Consideriamo il seguente grafo:



a cui corrisponde la matrice di incidenza:

$$A=\begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{pmatrix}$$

Gli stati di blocco, per la precedente proposizione, sono dati da:

$$\left((q_B)^{i_1} = \frac{Q^{i_1}}{2}(1+A_{i_1\,1}),\quad (q_B)^{i_2} = \frac{Q^{i_2}}{2}(1+A_{i_2\,2}),\quad (q_B)^{i_3} = \frac{Q^{i_3}}{2}(1+A_{i_3\,3}) \right)_{\substack{(i_1\ i_2\ i_3)\in \mathrm{Disp}(4,3) \\ A_{i_k\,k}\neq 0 \quad k=1..3}}$$

Le disposizioni di 4 oggetti in classe 3 sono 24. Tra esse quelle che danno origine a matrici quadrate con diagonale principale ad elementi tutti non nulli (la disposizione contiene gli indici delle righe di A da considerare) sono quelle contrassegnate con ♦ nella seguente tabella:

	♦				♦			♦			♦			♦			♦					♦	♦	
i_1	1	1	2	2	3	3	1	1	2	2	4	4	1	1	3	3	4	4	2	2	3	3	4	4
i_2	2	3	1	3	1	2	2	4	1	4	1	2	3	4	1	4	1	3	3	4	2	4	2	3
i_3	3	2	3	1	2	1	4	2	4	1	2	1	4	3	4	1	3	1	4	3	4	2	3	2

Si hanno dunque 8 possibili stati di blocco della computazione. In corrispondenza di queste 8 disposizioni le diagonali principali delle suddette matrici valgono:

disposizione			**diagonale**		
1	2	3	1	1	-1
3	1	2	1	-1	-1
1	4	2	1	1	-1
4	1	2	-1	-1	-1
1	4	3	1	1	-1
4	1	3	-1	-1	-1
3	4	2	1	1	-1
4	2	3	-1	1	-1

ed allora usando la precedente formula si ottengono esplicitamente gli 8 stati di blocco:

1	$(q_B)^1 = Q^1$	$(q_B)^2 = Q^2$	$(q_B)^3 = 0$	$(q_B)^4 =$ arbitraria
2	$(q_B)^3 = Q^3$	$(q_B)^1 = 0$	$(q_B)^2 = 0$	$(q_B)^4 =$ arbitraria
3	$(q_B)^1 = Q^1$	$(q_B)^4 = Q^4$	$(q_B)^2 = 0$	$(q_B)^3 =$ arbitraria
4	$(q_B)^4 = 0$	$(q_B)^1 = 0$	$(q_B)^2 = 0$	$(q_B)^3 =$ arbitraria
5	$(q_B)^1 = Q^1$	$(q_B)^4 = Q^4$	$(q_B)^3 = 0$	$(q_B)^2 =$ arbitraria
6	$(q_B)^4 = 0$	$(q_B)^1 = 0$	$(q_B)^3 = 0$	$(q_B)^2 =$ arbitraria
7	$(q_B)^3 = Q^3$	$(q_B)^4 = Q^4$	$(q_B)^2 = 0$	$(q_B)^1 =$ arbitraria
8	$(q_B)^4 = 0$	$(q_B)^2 = Q^2$	$(q_B)^3 = 0$	$(q_B)^1 =$ arbitraria

3.3 Raggiungibilita' degli stati di blocco della computazione

Proposizione.3.9 (Proprieta' dei cicli) Se H=(V,E) e' un grafo diretto (in generale non connesso) tale che |E|=|V|, rappresentato da una matrice di incidenza A_H con diagonale principale ad elementi μ_k k=1..|V|, tutti non nulli[5], per un suo qualsiasi ciclo, descritto dalle righe k_1, k_2, ... k_n, e per ogni vettore |V|-dimensionale **w** vale la:

$$\sum_{i=1}^{n} \mu_{k_i} \sum_{j=1}^{|V|} (A_H)_{k_i j} w^j = 0$$

Dimostrazione. Si ha:

$$\sum_{i=1}^{n} \mu_{k_i} \sum_{j=1}^{|V|} (A_H)_{k_i j} w^j = \sum_{j=1}^{|V|} w^j \sum_{i=1}^{n} (A_H)_{k_i k_i} (A_H)_{k_i j} = \sum_{j=1}^{|V|} w^j 0 = 0$$

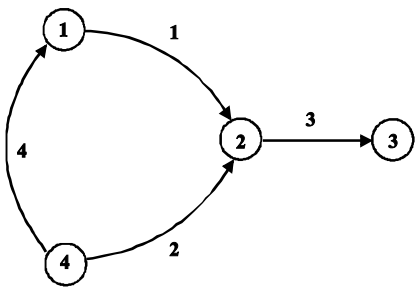
infatti, posto:

$$S_j = \sum_{i=1}^{n} (A_H)_{k_i k_i} (A_H)_{k_i j}$$

S_j si ottiene moltiplicando la diagonale principale di A_H per ogni sua colonna limitatamente agli elementi corrispondenti agli archi del ciclo (righe di A_H). S_j e' sempre nulla in quanto:

- Le colonne di A_H corrispondenti a nodi del ciclo, e limitatamente alle righe che descrivono archi del ciclo, hanno due e due soli elementi non nulli per cui S_j sara' data dalla differenza di due prodotti di cui uno di un elemento per se stesso (elemento intersezione della diagonale principale con la colonna in esame) che vale sempre 1, e l'altro di un elemento della diagonale principale per un elemento della stessa riga ma colonna diversa e dunque di segno opposto (descrive un arco) che vale sempre -1.
- Le colonne corrispondenti a nodi non coinvolti nel ciclo, ma limitatamente alle righe che descrivono archi del ciclo, hanno tutti gli elementi nulli e quindi anche la loro somma sara' nulla.

Esempio. Consideriamo il ciclo 1,2,4 del seguente grafo:



[5] L'ipotesi di diagonale ad elementi tutti non nulli richiede che tutte le righe che descrivono gli archi del ciclo abbiano uno dei loro elementi non nulli sulla diagonale.

la cui matrice di incidenza, organizzata in modo che abbia la diagonale principale ad elementi tutti non nulli, e':

$$A_H = \begin{pmatrix} 1 & -1 & 0 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 1 & -1 & 0 \\ -1 & 0 & 0 & 1 \end{pmatrix}$$

Gli archi del ciclo sono individuati dalle righe 1,2 e 4 di A_H e per esse si ha:

i	k_i	$(A_H)_{k_i k_i}$	$(A_H)_{k_i j}$ j=1..4	$(A_H)_{k_i k_i} (A_H)_{k_i j}$ j=1..4
1	1	1	(1,-1, 0, 0)	(1,-1, 0, 0)
2	2	-1	(0,-1, 0, 1)	(0, 1, 0,-1)
3	4	1	(-1, 0, 0, 1)	(-1, 0, 0, 1)
				Σ_i=(0, 0, 0, 0)

In accordo alla precedente proposizione. Se la matrice di incidenza non fosse a diagonale con elementi tutti non nulli, le conclusioni della proposizione non sarrebbero invece piu' valide come si vede subito scambiando le ultime due righe di A_H:

$$B_H = \begin{pmatrix} 1 & -1 & 0 & 0 \\ 0 & -1 & 0 & 1 \\ -1 & 0 & 0 & 1 \\ 0 & 1 & -1 & 0 \end{pmatrix}$$

Gli archi del ciclo sono ora individuati dalle righe 1,2 e 3 di B_H e per esse si ha:

i	k_i	$(B_H)_{k_i k_i}$	$(B_H)_{k_i j}$ j=1..4	$(B_H)_{k_i k_i} (B_H)_{k_i j}$ j=1..4
1	1	1	(1,-1, 0, 0)	(1,-1, 0, 0)
2	2	-1	(0,-1, 0, 1)	(0, 1, 0,-1)
3	3	0	(-1, 0, 0, 1)	(0, 0, 0, 0)
				Σ_i=(1,-1, 0,-1)

Proposizione.3.10 (Raggiungibilita') Se G=(V,E) e' il grafo associato ad un DSPA, *A* il corrispondente automa e G' il grafo non diretto ottenuto da G ignorando il verso degli archi, nessuno degli stati di blocco della computazione e' raggiungibile da uno stato iniziale che differisca da un punto di blocco per una componente corrispondente alla coda di un ciclo di G'.

Dimostrazione. Se lo stato di blocco $\mathbf{q_B}$ fosse raggiungibile all'istante m da uno stato iniziale $\mathbf{q_0}$ si avrebbe:

$$\sum_{j=1}^{|V|} (A_G)_{ij}\, w^j(m) + (q_0)^i = (q_B)^i \qquad i=1..|E|$$

cioe', usando l'espressione esplicita di $\mathbf{q_B}$ (che ha |E|-|V| elementi arbitrari i quali non impongono condizioni):

$$\sum_{j=1}^{|V|} (A_G)_{i_h j}\, w^j(m) + (q_0)^{i_h} = (1+\lambda_{i_h})\frac{Q^{i_h}}{2} \qquad h=1..|V|$$

Lo stato di blocco $\mathbf{q_B}$ e' in corrispondenza biunivoca con una matrice di incidenza quadrata di ordine |V|, a diagonale principale con elementi tutti non nulli, che descrive un grafo, in generale non connesso, con |V|

nodi e |V| archi il quale, come noto, contiene almeno un ciclo. Preso uno qualsiasi di questi cicli, che supponiamo individuato dagli archi descritti dalle righe $k_1, k_2, \ldots k_n$ tali che $\{k_1, k_2, \ldots k_n\} \subseteq \{i_1, i_2, \ldots i_{|V|}\}$, moltiplicando la precedente per gli elementi della diagonale principale relativi al ciclo e sommando sul ciclo si ha:

$$\sum_{i=1}^{n} \lambda_{k_i} \sum_{j=1}^{|V|} (A_G)_{k_i j}\, w^j(m) + \sum_{i=1}^{n} \lambda_{k_i} (q_0)^{k_i} = \sum_{i=1}^{n} \lambda_{k_i} (1+\lambda_{k_i}) \frac{Q^{k_i}}{2}$$

per la precedente proposizione il primo termine e' nullo, inoltre $\lambda_k(\lambda_k+1)= \lambda_k+1$ essendo $\lambda_k=\pm 1$, dunque:

$$\sum_{i=1}^{n} \left(\lambda_{k_i} (q_0)^{k_i} - (\lambda_{k_i}+1) \frac{Q^{k_i}}{2} \right) = 0 \quad \text{ma gli addendi valgono:} \begin{cases} (q_0)^{k_i} - Q^{k_i} \le 0 & \text{se } \lambda_{k_i} = 1 \\ -(q_0)^{k_i} \le 0 & \text{se } \lambda_{k_i} = -1 \end{cases}$$

per cui la sommatoria si annulla sse tali addendi sono tutti nulli, ovvero sse:

$$(q_0)^{k_i} = \left(1 + \frac{1}{\lambda_{k_i}}\right) \frac{Q^{k_i}}{2} = (1+\lambda_{k_i}) \frac{Q^{k_i}}{2} = (q_B)^{k_i} \qquad i=1..n$$

cioe' la tesi.

Osservazione.3.11 La precedente proposizione nulla afferma riguardo agli elementi di un punto di blocco che non corrispondano ad archi di un ciclo. Cio' comporta che $\mathbf{q}_0$ possa contenere alcune componenti identiche a quelle di un punto di blocco $\mathbf{q}_B$ purche' esse non corrispondano ad archi di un ciclo.

3.4 Inizializzazione dell'automa

Il precedente paragrafo fornisce uno dei due requisiti da soddisfare per una corretta inizializzazione delle code ovvero della scelta dello stato $\mathbf{q}_0$ dell'automa *A*:

R1. Il numero di dati inizialmente presenti in almeno una coda per ciclo di G', ottenuto da G ignorando il verso degli archi, deve essere diverso dal numero di dati che conterrebbe la coda in corrispondenza ad uno stato di blocco della computazione.

Il secondo requisito e' dovuto alla conformita' con le inizializzazioni effettuate nel DSPA:

R2. Tutte le code devono essere inizializzate in accordo alla inizializzazione delle corrispondenti variabili della formulazione sequenziale del DSPA ovvero alle inizializzazioni effettuate prima dell'esecuzione del "while true loop".

Osservazione.3.12 (Restrizioni alle inizializzazioni) Si ipotizza che le inizializzazioni delle variabili effettuate esternamente al "while true loop" del DSPA siano tutte e sole quelle necessarie e sufficienti per la sua esecuzione sequenziale corretta (nel senso che il DSPA fornisca il risultato atteso). Questa precisazione si rende necessaria per evidenziare il fatto che eventuali inizializzazioni superflue sono ininfluenti rispetto ad una corretta esecuzione del DSPA sequenziale mentre cio' non e' piu' vero nella sua versione concorrente. Infatti una inizializzazione superflua di una variabile assegna ad essa un valore che non viene mai usato all'interno del "while true loop" perche' la variabile viene riassegnata prima del suo uso, e cio' ovviamente non influenza l'esecuzione sequenziale del DSPA. La stessa inizializzazione effettuata su una coda della versione concorrente ha l'effetto di disallineare temporalmente i dati della coda in oggetto con quelli di tutte le altre comportando un risultato diverso da parte dell'algoritmo a parita' di ingresso, e cioe' una esecuzione concorrente non conforme con quella sequenziale.
Si ipotizza altresi' che le variabili interessate dall'inizializzazione siano non soggette a fenomeni di duplicazione (piu' archi del grafo associato al DSPA corrispondenti alla stessa variabile) o accorpamento

(un arco contenente anche altre variabili). Nel primo caso infatti puo' verificarsi che non tutte le code corrispondenti alla stessa variabile abbiano effettivamente necessita' di una inizializzazione, nel secondo caso invece l'inizializzazione della variabile in oggetto provocherebbe, nella versione concorrente del DSPA, delle inizializzazioni non conformi alla versione sequenziale.

Verifichiamo ora l'esistenza di un metodo di inizializzazione che soddisfi entrambi i requisiti R1 ed R2, quando siano verificate le restrizioni introdotte nella precedente osservazione.

Definizione.3.13 (Ciclo equiverso) Con il termine ciclo equiverso intendiamo un ciclo di un grafo diretto G la cui visita, in uno dei due versi possibili, classifichi solo archi di verso concorde a quello di visita.

Proposizione.3.14 (Metodo di inizializzazione delle code) Se tutte le code vengono inizializzate in accordo al requisito R2, si ottiene uno stato iniziale $\mathbf{q_0}$ dell'automa A che soddisfa anche il requisito R1 purche':

$$\begin{cases} \forall k=1..|E| \Rightarrow Q^k>1 \qquad \text{oppure} \\ \exists \text{ almeno un } k\in(1,2,...,|E|) \text{ tale che } Q^k=1 \text{ ma i cicli equiversi sono inizializzati con un solo dato} \end{cases}$$

Dimostrazione.

Cicli equiversi. Affinche' un DSPA abbia una esecuzione sequenziale corretta (nel senso che fornisca il risultato atteso), dovra' essere inizializzata, esternamente al "while true loop", almeno una variabile (arco) per ogni ciclo equiverso del grafo associato al DSPA, altrimenti il primo nodo del ciclo ad essere eseguito, nella formulazione sequenziale, usera' una variabile non inizializzata provocando un risultato errato. Inoltre non tutte le variabili di un ciclo equiverso possono essere inizializzate altrimenti il primo nodo del ciclo ad essere eseguito, nella formulazione sequenziale, riscrivera' su una variabile inizializzata che pertanto avra' subito una inizializzazione superflua. Infine una variabile non potra' mai essere inizializzata piu' di una volta perche' ancora una volta si otterrebbero delle inizializzazioni superflue.
Riportando queste considerazioni sullo stato iniziale delle code possiamo dunque affermare che, se vale R2, con le restrizioni introdotte nella precedente osservazione, allora:

$$(q_0)^{k_i} = \begin{cases} \text{al piu' 1 su almeno un arco del ciclo ma non su tutti} \\ \text{0 per tutti gli altri archi del ciclo} \end{cases}$$

Da cio' si deduce che un ciclo equiverso non puo' originare, con la sua inizializzazione, uno stato di blocco. Infatti un ciclo equiverso si immerge nella matrice di incidenza di uno stato di blocco in modo che gli elementi della diagonale principale ad esso relativi siano tutti 1 o tutti -1 dovendosi classificare in una sua visita solo archi concordi o solo archi discordi al senso di visita. Dunque:

$$(q_B)^{k_i} = (1+\lambda_{k_i})\frac{Q^{k_i}}{2} = \begin{cases} \text{0 per ogni } k_i\text{-esimo arco del ciclo se valgono i -1} \\ Q^{k_i} \text{ per ogni } k_i\text{-esimo arco del ciclo se valgono gli 1} \end{cases}$$

quindi sicuramente non tutti i $(q_0)^{k_i}$ del ciclo sono identici ai $(q_B)^{k_i}$.

Altri cicli. Le code di tutti i cicli non equiversi, al fine di evitare punti di blocco, possono essere inizializzate vuote, ovvero:

$$\forall\ k_i\text{-esimo arco del ciclo } (q_0)^{k_i} = 0$$

perche':

$$\exists\ k_j\text{-esimo arco del ciclo tale che } (q_B)^{k_j} = Q^{k_j}$$

dovendosi poter classificare, in entrambi i versi di visita del ciclo, almeno un arco concorde al verso di visita, per il quale il corrispondente elemento della diagonale principale della matrice associata allo stato di blocco vale 1. Resta dunque solo da dimostrare che l'inizializzazione effettuata sui cicli equiversi, in base al requisito R2, ed alle restrizioni della precedente osservazione, non provochi configurazioni corrispondenti a stati di blocco sugli altri cicli. Distinguiamo due casi.

1. $\forall k=1..|E| \Rightarrow Q^k>1$: allora $\forall k=1..|E| \Rightarrow (q)^k \neq Q^k$ per cui non ci sono configurazioni su cicli non equiversi corrispondenti a punti di blocco.

2. $\exists$ **almeno un** $k \in (1,2,...,|E|)$ **tale che** $Q^k=1$**, ma tutti i cicli equiversi siano inizializzati con un solo dato:** supponiamo per assurdo che le inizializzazioni delle code dei cicli equiversi causino una inizializzazione su un ciclo non equiverso corrispondente alla configurazione che esso assumerebbe in corrispondenza di un punto di blocco. Siamo allora in una situazione del tipo di quella riportata in figura 3-1 in cui si presuppone che le code contrassegnate con * siano di dimensione unitaria. Come si vede i cicli equiversi (6,1,3,7) e (2,5,4) sono inizializzati con un solo dato e cio' causa una configurazione di blocco sul ciclo non equiverso (1,2,4,3).

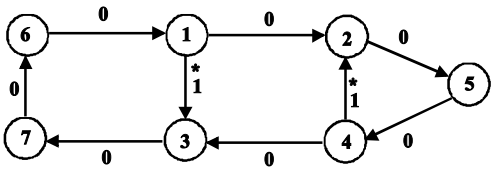


Figura 3-1 - Inizializzazione indotta di un ciclo non equiverso

Nel precedente grafo esiste pero' il ciclo equiverso (6,1,2,5,4,3,7) che non ha nessuna coda inizializzata, contro l'ipotesi che valga il requisito R2 (vedi dim. cicli equiversi). Questa situazione e' del tutto generale in quanto se l'unico arco inizializzato di piu' cicli equiversi coincide anche con alcuni archi di un ciclo non equiverso ed e' il responsabile dell'instaurarsi su quest'ultimo di una configurazione corrispondente ad un punto di blocco, allora gli archi non inizializzati dei cicli equiversi e del ciclo non equiverso in esame formano un ciclo equiverso che non ha inizializzazioni, contro il requisito R2.

Esempio. Per illustrare come agisce la precedente proposizione, consideriamo il seguente controesempio. Quando viene a mancare l'ipotesi di singola inizializzazione di un ciclo equiverso in presenza di almeno una coda di dimensione unitaria, viene a cadere la conclusione della proposizione, ovvero la validita' del requisito R1. Nella seguente figura 3-2 supponiamo che la coda contrassegnata con * sia di dimensione unitaria in accordo con l'ipotesi della proposizione. In fig.a e' riportata una inizializzazione singola del ciclo equiverso (1,3,4) che provoca una configurazione di blocco nel ciclo non equiverso (2,1,4). Cio' e' dovuto alla mancanza di inizializzazioni sul ciclo equiverso (2,1,3,4). Se inizializziamo allora quest'ultimo ciclo, non possiamo inizializzare una coda del ciclo equiverso (1,3,4) che gia' contiene una coda inizializzata (fig.b) perche' cio' sarebbe contro l'ipotesi della proposizione. Dunque siamo costretti ad inizializzare una coda del ciclo non equiverso (1,4,2) (fig.c) rimuovendo cosi' la configurazione di blocco.

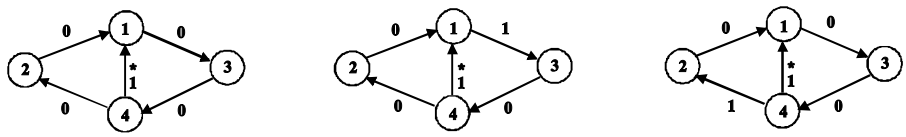


Figura 3-2 - Inizializzazione multipla su un ciclo equiverso

.

Osservazione.3.15 (Altri metodi di inizializzazione) Se si lasciasse cadere il requisito R2, o alcune delle restrizioni sulle possibili inizializzazioni delle variabili, si potrebbero ad esempio inizializzare i cicli

equiversi con piu' di un dato in una o piu' code oppure inizializzare qualche ciclo non equiverso con dati nelle code o infine, inizializzare con dati, code non appartenenti a cicli. Verificato allora che l'inizializzazione eviti punti di blocco, cosa non piu' assicurata dalla precedente proposizione, si otterrebbe una esecuzione concorrente del DSPA non conforme a quella sequenziale (nel senso che non fornirebbe piu' in generale, a parita' di ingresso, lo stesso risultato). Cio', per algoritmi di "signal processing" puo' non equivalere ad una grave degradazione delle prestazioni. Si avrebbe una alterazione della cosidetta funzione di trasferimento la quale, in particolare quando il "grain" impostato nell'algoritmo non e' troppo fine, corrisponde ad una perdita accettabile in termini di prestazioni quando cio' sia conveniente per altre ragioni. La ragione principale di cercare inizializzazioni non conformi a quelle effettuate nella descrizione sequenziale dell'algoritmo sta essenzialmente nella possibilita' di ridurre il periodo di computazione dell'automa. Sia il periodo che il transitorio dell'automa dipendono fortemente da tale inizializzazione. Si possono ricavare inizializzazioni che riducano addirittura ad 1 questo periodo. La scelta va comunque ponderata caso per caso.

Proposizione.3.16 (Conservazione dei dati in un ciclo equiverso) In un ciclo equiverso del grafo G=(V,E), avente matrice di incidenza A_G, la somma del numero di dati presenti nelle code si mantiene costante durante l'evoluzione dell'automa.

Dimostrazione. E' sempre possibile estrarre da A_G una sottomatrice quadrata |V|x|V|, con diagonale ad elementi tutti non nulli, contenente gli archi del ciclo. Se il ciclo e' equiverso inoltre su tale diagonale, limitatamente agli archi del ciclo, compariranno solo elementi μ_k di valore 1 o solo di valore -1, dovendosi classificare, durante una sua visita, solo archi concordi o solo archi discordi al senso di visita. Allora per una precedente proposizione, per ogni vettore v, vale la:

$$\sum_{i=1}^{n} \mu_{k_i} \sum_{j=1}^{|V|} (A_G)_{k_i j}\, v^j = 0 \Rightarrow$$

$$\forall n>0: \sum_{i=1}^{n} \sum_{j=1}^{|V|} (A_G)_{k_i j}\, v^j(n) = 0 \Rightarrow \sum_{i=1}^{n} (\Delta q(n))^{k_i} = 0 \Rightarrow \sum_{i=1}^{n} (q(n))^{k_i} = \sum_{i=1}^{n} (q(0))^{k_i}$$

.

Proposizione.3.17 (Lower Bound al periodo dell'automa) Se $N_1, N_2, ..., N_r$ e' una numerazione degli ordini di tutti i cicli equiversi presenti nel grafo G associato al DSPA e se il corrispondente automa e' inizializzato in accordo ai requisiti R1, R2 ed in modo che una ed una sola sua coda di ogni ciclo equiverso contenga uno ed un solo dato, per il periodo p del ciclo in cui cade l'automa, dopo la fase di transitorio, vale la:

$$p \geq \max \{N_1, N_2, ..., N_r\}$$

Dimostrazione. Per ipotesi l'inizializzazione di ogni ciclo equiverso e' tale che:

$$\sum_{i=1}^{N_k} (q(0))^{k_i} = 1 \qquad k=1..r$$

e per la precedente proposizione tale quantita' si conserva, ogni nodo del ciclo potra' essere eseguito al piu' uno alla volta perche' nel ciclo ci sono sempre N_k-1 code vuote. All'interno del periodo di ripetizione dell'automa ogni nodo del ciclo deve essere eseguito almeno una volta dunque $p \geq N_k$. Questa situazione si presenta per ciascun ciclo equiverso di G e quindi in generale sara' $p \geq \max\{N_1, N_2, ..., N_r\}$.

3.5 Entropia di un automa a stati finiti

Per avere una valutazione globale del comportamento dinamico di sistemi discreti complessi e' spesso usata la cosidetta funzione entropia. Il concetto di entropia nasce in termodinamica come misura del disordine. Nei sistemi discreti essa fornisce una misura della dimensione dell'insieme degli stati raggiungibili dopo un certo numero di iterazioni e del numero di iterazioni necessarie affinche' tale insieme resti poi costante nelle successive. E' intuitivo allora pensare che piu' grande sara' questa dimensione, piu' disordinato sara' il sistema nel senso di non poterne prevedere a priori lo stato finale con sufficiente certezza.

Definizione.3.18 (Funzione entropia) Se A e' un insieme finito di eventi la cui probabilita' di verificarsi e' $p_k \forall k \in A$, e' possibile associare ad A la seguente funzione di |A| variabili detta funzione entropia:

$$H(p_k, k \in A) = - \sum_{k \in A} p_k \log_2 p_k$$

Poiche' le quantita' p_k rappresentano delle probabilita', H sara' definita sul compatto costituito dalla parte dell'iperpiano $\Sigma_{k \in A} p_k = 1$ contenuta nell'ipercubo $0 \le p_k \le 1$. Si assume anche $0 \log_2 0 = 0$ per cui H risulta essere anche continua sul suo insieme di definizione.

Proposizione.3.19 (Limitatezza della funzione entropia) Si ha:

1. $0 \le H(p_k, k \in A) \le \log_2 |A|$;
2. H raggiunge il suo minimo H=0 negli |A| punti dati da: $p_j=1$, $p_k=0$ $\forall k \neq j$, j=1..|A| appartenenti alla frontiera del suo insieme di definizione;
3. H raggiunge il suo massimo $H=\log_2 |A|$ nell'unico punto $p_k=1/|A|$ $\forall k \in A$;

Dimostrazione. Poiche' H e' definita e continua su un compatto, ammette un massimo ed un minimo assoluti. Per il teorema dei moltiplicatori di Lagrange tali punti sono da ricercare tra i punti stazionari della:

$$H^*(p_k \mid k \in A, \lambda) = - \sum_{k \in A} p_k \log_2 p_k + \lambda \left(\sum_{k \in A} p_k - 1 \right)$$

che cadono all'interno dell'ipercubo $0 \le p_k \le 1$, oppure sulla frontiera dell'insieme di definizione. Annullando le |A|+1 derivate parziali:

$$\frac{\partial H^*}{\partial p_k} = 0, k \in A \qquad \frac{\partial H^*}{\partial \lambda} = 0$$

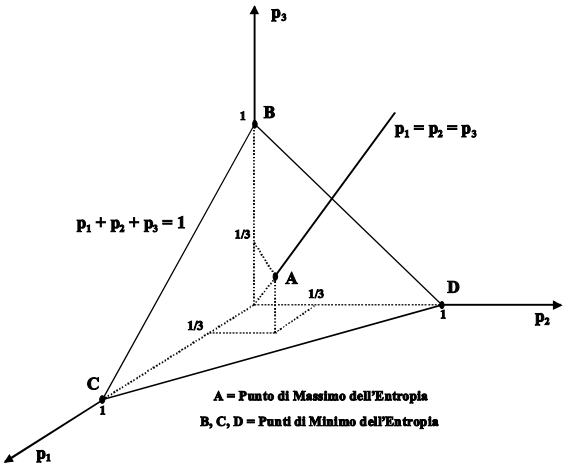
si ottiene l'unico punto ($p_k=1/|A|$, $\forall k \in A$), interno all'insieme di definizione, in corrispondenza del quale $H=\log_2 |A|$. Il fatto che si tratti di un massimo si verifica immediatamente considerando l'hessiana:

$$\frac{\partial^2 H}{\partial p_i \partial p_j} (p_k=1/|A|, \forall k \in A) = \begin{cases} 0 \text{ se } i \neq j \\ -|A| \text{ se } i=j \end{cases}$$

che risulta essere definita negativa in quanto matrice diagonale con elementi tutti negativi coincidenti con i suoi autovalori.

Per determinare il minimo osserviamo innanzitutto che dalla $0 \le p_k \le 1$ discende $-\log_2(p_k) \ge 0$ per cui e' sempre $H \ge 0$. H=0 e' effettivamente raggiunto nei punti ($p_j=1$, $p_k=0$ $\forall k \neq j$, j=1..|A|) della frontiera dell'insieme di definizione come e' immediato verificare considerando $0 \log_2 0 = 0$.

Esempio. Nel caso |A|=3 l'insieme di definizione della funzione entropia H ha la seguente rappresentazione geometrica:



La parte di piano $p_1+p_2+p_3=1$ contenuta nel cubo unitario e' il triangolo BCD ai vertici del quale si hanno i tre minimi dell'entropia. Il massimo si ha invece nel centro dell'insieme di definizione.

Osservazione.3.20 La massima entropia si ha in corrispondenza ad una equiprobabilita' degli eventi e quindi, in corrispondenza ad un elevato disordine dell'insieme nel senso di maggiore incertezza sul verificarsi o meno di un evento. Viceversa il minimo dell'entropia si ottiene in corrispondenza di una certezza del verificarsi di un evento, essendo una delle probabilita' uguale ad 1 e le altre nulle.

Proposizione.3.21 (Insieme degli stati raggiungibili da un automa a stati finiti) Se A_j, $j\geq 0$, e' l'insieme degli stati raggiungibili alla j-esima iterazione da un automa a stati finiti, il cui insieme degli stati sia A_0, si ha:

$$A_0 \supseteq A_1 \supseteq A_2 \supseteq \ldots \supseteq A_j \supseteq \ldots \supseteq A_n = A_{n+1} = \ldots \qquad n \leq |\wp(A_0)|$$

dove $\wp(A_0)$ denota l'insieme delle parti di A_0.

Dimostrazione. Per induzione:

BASE: $A_0 \supseteq A_1$ perche' A_0 contiene tutti gli stati dell'automa per ipotesi;

PASSO: Se $A_j \supseteq A_{j+1}$, ovvero se da stati dell'insieme $A_j \supseteq A_{j+1}$ si possono raggiungere solo stati di A_{j+1}, a maggior ragione cio' deve avvenire a partire da stati di A_{j+1} ovvero $A_{j+1} \supseteq A_{j+2}$.

Inoltre, poiche' il numero massimo di insiemi diversi presenti nella successione e' $|\wp(A_0)|$, da un certo $n\leq|\wp(A_0)|$ in poi essa conterra' solo elementi tutti uguali non potendo, per la monotonia, comparire dopo n, insiemi apparsi nei primi n elementi della successione.

Definizione.3.22 (Evoluzione dell'entropia di un automa a stati finiti) La raggiungibilita' di uno stato k alla j-esima iterazione puo' essere considerata un evento che si verifica con una certa probabilita' $p_k(j)$ dipendente dalla scelta dello stato iniziale dell'automa. Cio' permette di considerare la funzione entropia associata a ciascun insieme della successione introdotta nella precedente proposizione ottenendo la seguente successione:

$$H_j(p_k(j), k\in A_j) = -\sum_{k\in A_j} p_k(j)\log_2 p_k(j)$$

si tratta di una successione numerica e non di una successione di funzioni in quanto le funzioni entropia di $|A_j|$ variabili che figurano nella precedente si intendono valutate sulla distribuzione $p_k(j)$, supposta nota, delle probabilita' degli stati $k\in A_j$ al variare di j.

Proposizione.3.23 (Evoluzione dell'entropia di un automa a stati finiti) La successione numerica $H_j(p_k(j), k\in A_j)$ associata ad un automa a stati finiti e' monotona non decrescente.

Dimostrazione. Discende direttamente dalla monotonicita' della successione degli insiemi degli stati. Infatti, per la suddetta monotonicita', alla generica iterazione j si possono verificare solo due casi: $A_{j+1} = A_j$ oppure $A_{j+1} \subseteq A_j$.

CASO $A_{j+1} = A_j$ Detta F la funzione di transizione dell'automa, cio' equivale ad affermare che due diversi stati R≠S, raggiunti dall'automa alla j-esima iterazione, debbano produrre due stati distinti alla (j+1)-esima: F(R)≠F(S) (fig.a).

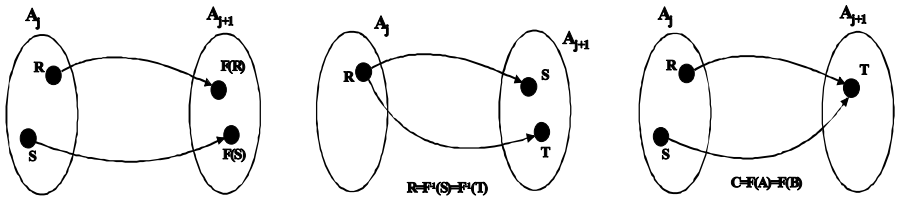
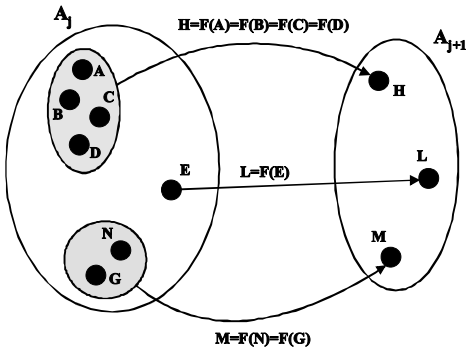


fig. a fig. b fig. c

Se per assurdo infatti due stati distinti R≠S di A_j producessero un'unico stato T di A_{j+1} (fig.c), per ottenere $A_j=A_{j+1}$ dovrebbe esistere uno stato R di A_j in grado di produrre due stati distinti R≠S di A_{j+1} (fig.b) cosa impossibile in un automa a stati finiti. Dunque necessariamente $p_{F(k)}(j+1)=p_k(j)$ $\forall k\in A_j$ e percio' $H_{j+1}=H_j$.

CASO $A_{j+1} \subseteq A_j$ In questo caso qualche sottoinsieme di stati di A_j produrra' lo stesso stato di A_{j+1}:



Consideriamo allora ad esempio il contributo $H_j(p_N(j), p_G(j))$ all'entropia H_j fornito dai due stati N e G di A_j e quello $H_{j+1}(p_M(j+1))$ fornito all'entropia H_{j+1} dallo stato M di A_{j+1}. Poiche' $p_N(j) \geq 0$ e $p_G(j)\geq 0$, vale la disuguaglianza:

$$p_N(j)\log_2 p_N(j) + p_G(j)\log_2 p_G(j) \leq (p_N(j) + p_G(j))\log_2 (p_N(j) + p_G(j))$$

che e' conseguenza immediata del fatto che:

$$\log_2 p_N(j) \leq \log_2 (p_N(j) + p_G(j)) \qquad \text{e} \qquad \log_2 p_G(j) \leq \log_2 (p_N(j) + p_G(j))$$

dunque:

$$H_j(p_N(j), p_G(j)) = - p_N(j) \log_2 p_N(j) - p_G(j) \log_2 p_G(j) \geq - (p_N(j) + p_G(j)) \log_2 (p_N(j) + p_G(j)) =$$

$$= - p_M(j+1) \log_2 p_M(j+1) = H_{j+1}(p_M(j+1))$$

Con le stesse argomentazioni si dimostra la analoga relazione per insiemi di piu' di due elementi che collassano in un'unico stato al successivo passo di iterazione. Si ha allora $H_{j+1} \leq H_j$ come si voleva.

Osservazione.3.24 Nonostante il comportamento dell'entropia al variare del tipo di automa a stati finiti sia sempre rigorosamente monotonico, e' possibile da essa estrarre alcune informazioni globali sulla lunghezza del transitorio e sul periodo del ciclo in cui inevitabilmente l'automa cade. La lunghezza massima del transitorio coincide infatti con il numero T di iterazioni necessarie alla convergenza della successione dei valori di entropia mentre la dimensione dell'insieme degli stati a cui converge la successione $A_0,A_1,A_2,...$ ci da un'indicazione sul massimo periodo del ciclo a cui l'automa converge. Se gli stati dell'insieme A a cui converge la successione $A_0,A_1,A_2,...$ fossero equiprobabili si avrebbe direttamente $|A|=2^{H(T)}$ ed il periodo del ciclo sarebbe sicuramente $<|A|$. I valori sono dei massimi per cui l'automa in genere potra' avere transitori e periodi anche molto piu' piccoli a partire addirittura da 1. E' comunque da rilevare che senza l'informazione dell'entropia nulla si potrebbe dire ne' sulla lunghezza massima del transitorio ne' su quella del periodo che risulterebbero essere limitate solo dal numero $|A_0|$ degli stati totali dell'automa.
Reti non finite di automi a stati finiti quali ad esempio automi cellulari di dimensione numerabile possono dare origine a comportamenti piu' complessi della successione dei valori di entropia. Ad esempio sistemi caotici esibiscono una evoluzione dell'entropia che comporti piccole variazioni intorno ad un valore stabilmente elevato. In altri casi la successione puo' avere grandi variazioni intorno ad un valore medio denotando un comportamento complesso del sistema.

Osservazione.3.25 E' possibile determinare numericamente l'entropia di un automa a stati finiti simulandone l'evoluzione a partire da tutti i possibili stati iniziali A_0 e contando il numero di volte $r_k(n)$ che lo stato k e' raggiunto dopo esattamente n iterazioni. Si avra' allora:

$$p_k(n) = \frac{r_k(n)}{|A_0|}$$

Tuttavia spesso $|A_0|$ e' una quantita' che cresce esponenzialmente con la dimensione del sistema. Ad esempio se il sistema e' costituito da R automi a stati finiti, ognuno con un numero di stati pari ad S, si ha $|A_0|=S^R$. Per tale motivo e' necessario determinare $p_k(n)$ in modo approssimato. Cio' puo' essere fatto generando un sottoinsieme $B_0 \subset A_0$ a distribuzione casuale uniforme di stati iniziali e ponendo:

$$p_k(n) = \frac{r_k(n)}{|B_0|}$$

3.6 Esempi

3.6.1 Filtri IIR

Un filtro IIR (Infinite Impulse Responce) di ordine N e' una trasformazione di un segnale numerico x(n) in un altro y(n) in base alla seguente formula ricorsiva:

$$y(n) = c_0\, x(n) + \sum_{i=1}^{N-1} c_1^i\, x(n-i) + \sum_{i=1}^{N-1} c_2^i\, y(n-i)$$

Il seguente programma ADA descrive un filtro IIR di ordine 5. E' riportato di seguito il solo "while true loop" mentre l'intero programma e' in appendice.

```
v1: int_vector_n:= (0,0,0,0,0);
v2: int_vector_n:= (0,0,0,0,0);

while not end_of_file(fid_inp) loop
   source(x);
   z1:=forward(v1);
   z2:=backward(v2);
   r:=add(z1,z2);
   y:=mac(c0,x,r);
   v1:=shift(v1,x);
   v2:=shift(v2,y);
   sink(y);
end loop;
```

Figura 3-3 - "while true loop" di un IIR

Le funzioni "forward" e "backward" implementano rispettivamente la sommatoria contenente il segnale numerico di ingresso x() e la sommatoria contenente il segnale di uscita retroazionato y(). La funzione "shift" effettua lo scorrimento delle sequenze x() e y() ad ogni nuovo dato in ingresso. Le funzioni "add" e "mac" sono ovvie. Al precedente programma e' associato il seguente grafo:

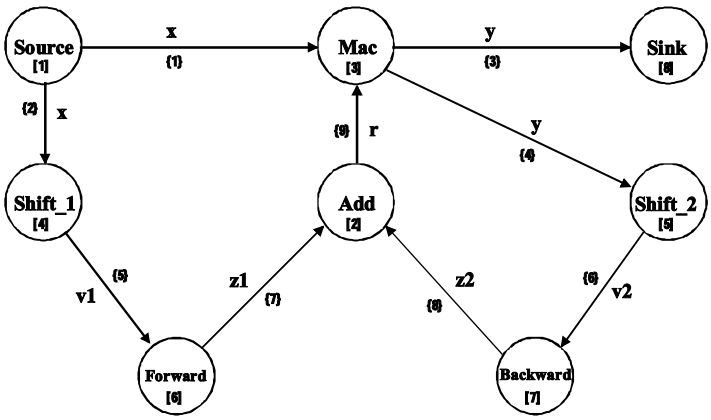


Figura 3-4 - Grafo associato ad un IIR

I nodi sono numerati con un indice tra [] mentre gli indici degli archi sono riportati tra {}. A questo grafo corrisponde la seguente matrice di incidenza:

edges	source [1]	add [2]	mac [3]	shift1 [4]	shift2 [5]	forward [6]	backward [7]	sink [8]
x {1}	1	0	-1	0	0	0	0	0
x {2}	1	0	0	-1	0	0	0	0
y {3}	0	0	1	0	0	0	0	-1
y {4}	0	0	1	0	-1	0	0	0
v1 {5}	0	0	0	1	0	-1	0	0
v2 {6}	0	0	0	0	1	0	-1	0
z1 {7}	0	-1	0	0	0	1	0	0
z2 {8}	0	-1	0	0	0	0	1	0
r {9}	0	1	-1	0	0	0	0	0

Figura 3-5 - Matrice di incidenza di un IIR

Naturalmente la descrizione non e' unica e definisce quanto accurata si vuole la granulosita' della futura implementazione concorrente: piu' le funzioni e procedure sono numerose ed elementari piu' fine sara' il "grain". Ogni funzione o procedura e' riportata tante volte quante sono le volte che e' chiamata nel programma con una numerazione crescente. Ogni arco e' etichettato con il nome della variabile di scambio tra le funzioni o procedure dei nodi che connette. Nel grafo e' possibile individuare i due cicli costituiti dagli archi {x1, r, z1, v1, x2} e {y2, v2, z2, r} dunque l'automa associato possiede stati che comportano il blocco della computazione. Tali stati non saranno comunque mai raggiungibili se l'automa parte da uno stato iniziale che differisca da uno stato di blocco per il valore di almeno una coda di un ciclo. Inizializziamo dunque le code dell'automa conformemente all'inizializzazione delle variabili effettuata nel DSPA sequenziale prima del "while true loop":

$$\mathbf{q}(0) = (0,0,0,0,1,1,0,0,0)$$

Il ciclo equiverso {y2, v2, z2, r} contiene allora una coda non vuota (quella relativa alla variabile v2) mentre l'altro ciclo non ha tutte le code inizialmente vuote, come sarebbe possibile, ma ha la coda relativa alla variabile v1 non vuota.
Le inizializzazioni delle code corrispondenti alle variabili v1 e v2 sono dovute alla presenza di auto-cicli sulle due chiamate alla funzione "shift". L'inizializzazione della variabile v2 provoca poi automaticamente anche l'inizializzazione di una coda del ciclo equiverso (sono presenti due code contenenti la stessa variabile) rimuovendo la configurazione di blocco ad esso dovuta (code tutte vuote).
Se si suppongono tutte le code della stessa lunghezza, ad esempio pari a 7, l'evoluzione dell'automa, in termini di nodi eseguiti e dati contenuti nelle code e' rappresentata dalla seguente figura 3-6:

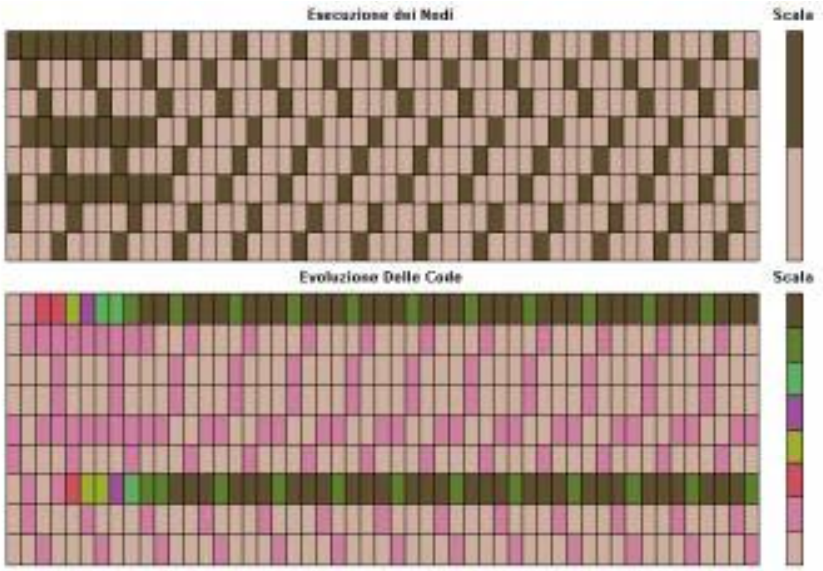


Figura 3-6 - Evoluzione dell'automa associato ad un IIR (periodo 4)

Entrambi i grafici hanno sulle ascisse gli istanti discreti mentre sulle ordinate sono presenti i nodi e le code del grafo rispettivamente, con indice crescente dall'alto verso il basso. Il colore rappresenta nei due casi rispettivamente l'esecuzione o meno del nodo del grafo e la quantita' di dati presenti nelle code.
Si nota la periodicita' raggiunta dall'automa sia per quanto riguarda l'esecuzione dei nodi sia per quanto riguarda l'evoluzione delle code. Si nota anche che durante un periodo, che risulta essere di 4 istanti discreti, vengono eseguiti tutti i nodi una sola volta ed alcuni di essi vengono eseguiti in parallelo. La quantita' di dati nelle code puo' variare da 0 a 7. La presenza di un ciclo equiverso di ordine 4 comporta, in accordo alla precedente proposizione, un periodo di esecuzione ≥ 4 quando l'inizializzazione e' tale da inserire un solo dato in una sola delle code del ciclo. Altre inizializzazioni, non conformi all'inizializzazione sequenziale del DSPA, possono avere periodo e transitorio di lunghezza diversa. Se ad esempio inizializziamo tutte le code del ciclo equiverso con un dato, otteniamo un periodo 1 per l'automa:

$$\mathbf{q}(0) = (0,0,0,1,0,1,0,1,1)$$

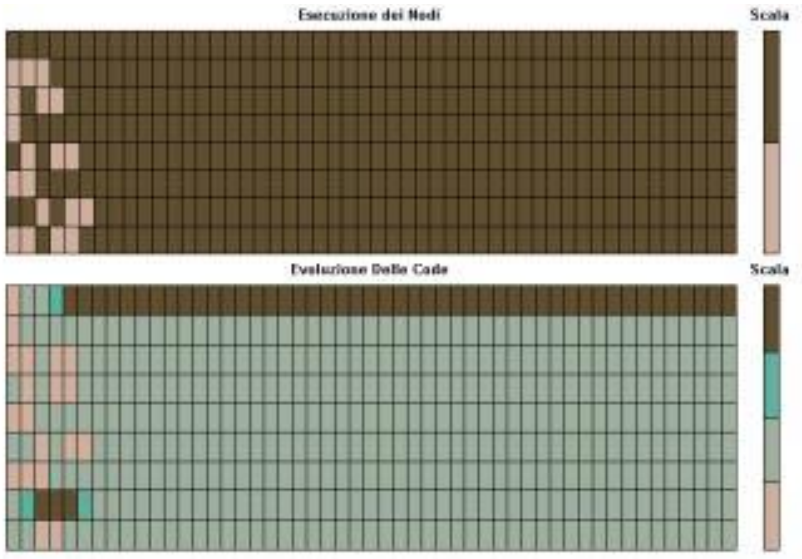


Figura 3-7 - Evoluzione dell'automa associato ad un IIR (periodo 1)

Di seguito sono riportati altri due esempi significativi di inizializzazione non conforme con il requisito R2. Se poniamo tutte le code di dimensione unitaria e:

$$\mathbf{q}(0) = (1,1,0,0,0,0,0,1,1)$$

l'automa evolve verso un periodo 5 durante il quale ogni nodo viene eseguito 2 volte:

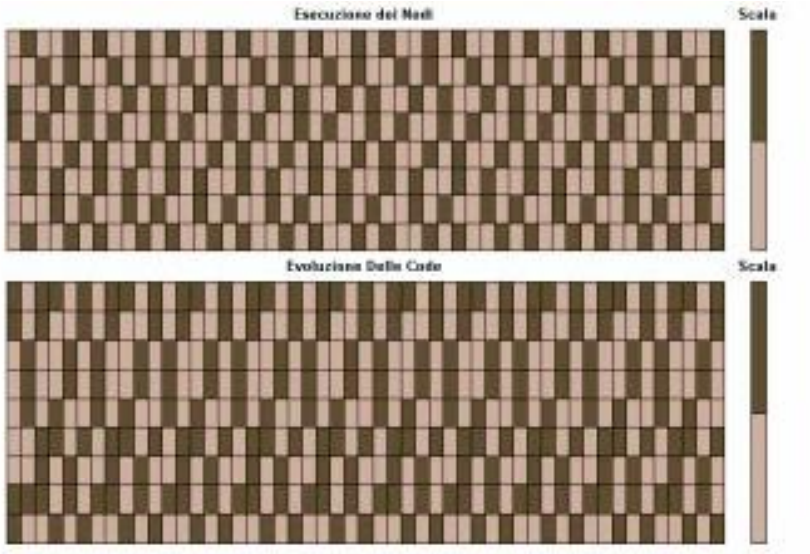


Figura 3-8 - Evoluzione dell'automa associato ad un IIR (periodo 5)

Se invece poniamo:

$$\mathbf{q}(0) = (0,0,0,0,0,0,0,0,0)$$

ripristinando la dimensione delle code a 7, l'automa evolve verso uno stato di blocco della computazione:

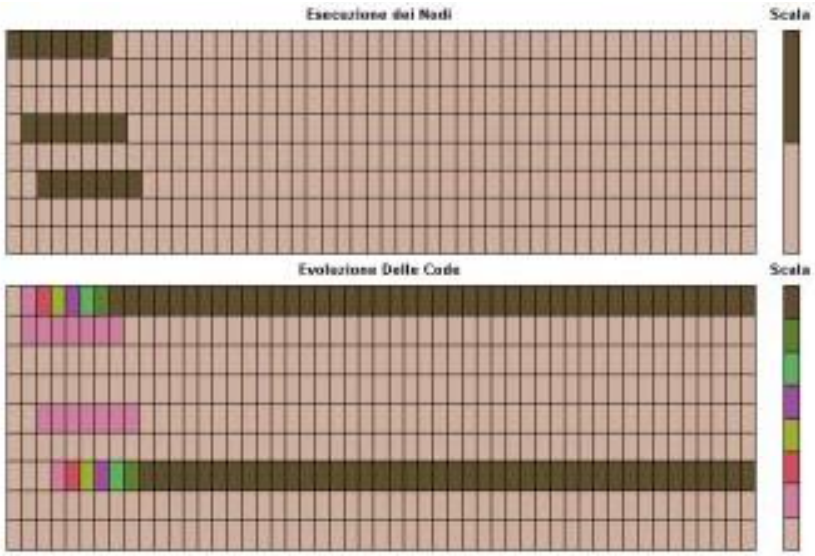
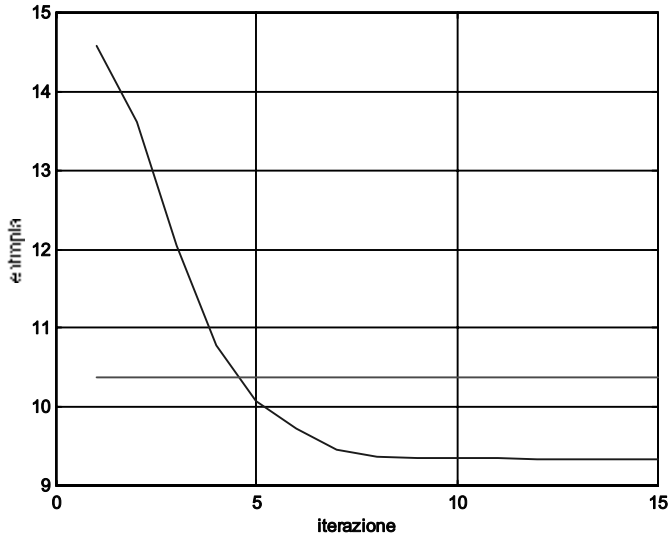


Figura 3-9 - Evoluzione dell'automa associato ad un IIR (stato di blocco)

Nel seguente grafico e' infine riportata l'evoluzione dell'entropia associata all'automa del filtro IIR in esame:



Questo grafico e' stato ottenuto facendo evolvere l'automa associato all'IIR per 15 iterazioni successive a partire da 26214 stati iniziali pari al 10% del numero totale di stati nell'ipotesi che la massima dimensione delle code sia 3. Infatti il numero totale di stati dell'automa e' dato da 4^9=262144 essendo 4 il numero di stati possibili per ciascuna coda (0,1,2,3) e 9 il numero complessivo di code (archi del grafo associato).
Come si vede la successione dei valori dell'entropia converge rapidamente (in un numero di iterazioni pari a circa il numero delle code) ad un valore compreso tra 9 e 10. Cio' ci informa che l'automa in esame ha un transitorio che si esaurisce al massimo dopo circa 9 iterazioni e dopo il quale, il numero di stati raggiungibili, nell'ipotesi che essi siano equiprobabili, e' al piu' $2^{9.4}$=675 (essendo dato come noto da $2^{H(9)}$ dove H(9) e' il valore finale dell'entropia). In realta' un conteggio diretto ha evidenziato che il numero di stati raggiungibili dopo il transitorio e' piu' elevato e precisamente corrisponde al valore dell'entropia indicato dalla linea orizzontale ovvero $2^{10.4}$= 1351. Il minore valore a cui si assesta l'entropia e' giustificato da una distribuzione delle probabilita' dei 1351 stati raggiungibili non uniforme. In altre parole l'automa e' piu' ordinato di un automa che avesse 1351 stati equiprobabili. Queste valutazioni globali ci garantiscono che il periodo massimo dell'automa sara' al piu' 1351 ma per molti stati iniziali esso sara' compreso tra 1 e 675 a causa dell'ordine evidenziato dal sistema.
In ogni caso l'entropia e' lontana dal suo minimo (che e' 0) e cio' denota la non banalita' dell'automa.

3.6.2 Filtri FIR Adattivi

Un secondo esempio, molto ricorrente nelle problematiche di elaborazione numerica dei segnali, e' costituito dai filtri FIR (Finite Inpulse Responce) adattivi. Un filtro FIR (Finite Impulse Responce) di ordine N e' una trasformazione di un segnale numerico x(n) in un altro y(n) in base alla seguente formula:

$$y(n) = c_0\, x(n) + \sum_{i=1}^{N-1} c_i\, x(n-i)$$

Un filtro FIR adattivo si differenzia da un FIR tradizionale per il fatto di usare dei coefficienti variabili, in genere stimati a partire dalla uscita stessa del filtro. Dunque:

$$y(n) = c_0(n)\,x(n) + \sum_{i=1}^{N-1} c_i(n)\,x(n-i) \qquad \text{con:} \quad c_k(n)=E(c_k(n-1),y(n-1)),\ k=0..N-1$$

dove E() e' la funzione di stima supposta del primo ordine. L'esempio seguente considera due FIR adattivi di ordine 3 in cascata:

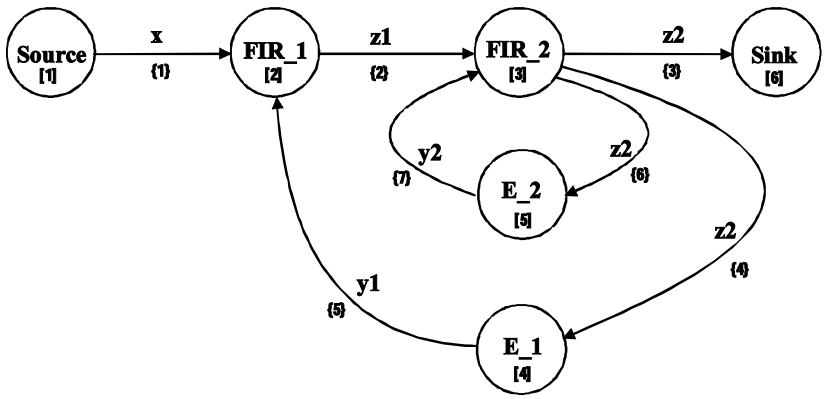


Figura 3-10 - Grafo associato a due FIR adattivi in cascata

Il "grain" e' considerevolmente superiore a quello del precedente esempio non essendo qui mai presenti operazioni elementari quali "add" e "mac". Il precedente grafo si ottiene al solito dal "while true loop" dell'implementazione sequenziale, di seguito riportato:

```
y1:=0;
y2:=0;

while not end_of_file(fid_inp) loop
   source(x);
   fir( x,y1,0,t1,z1);
   fir(z1,y2,n,t2,z2);
   y1:=coeff_estimator(z2);
   y2:=coeff_estimator(z2);
   sink(z2);
end loop;
```

Figura 3-11 - "while tue loop" di due FIR adattivi in cascata

Sono presenti degli autoloop (variabili t1 e t2) che come noto non vanno presi in considerazione ai fini del grafo da associare al DSPA ne' costituiscono interesse per quanto riguarda l'esecuzione concorrente dell'algoritmo. La funzione "fir" ha inoltre un parametro tra i suoi argomenti che, come noto, non da luogo ad archi del grafo associato. La matrice di incidenza del grafo e' la seguente:

edges	source [1]	FIR_1 [2]	FIR_2 [3]	E_1 [4]	E_2 [5]	sink [6]
x {1}	1	-1	0	0	0	0
z1 {2}	0	1	-1	0	0	0
z2 {3}	0	0	1	0	0	-1
z2 {4}	0	0	1	-1	0	0
y1 {5}	0	-1	0	1	0	0
z2 {6}	0	0	1	0	-1	0
y2 {7}	0	0	-1	0	1	0

Figura 3-12 - Matrice di incidenza di due FIR adattivi in cascata

L'inizializzazione dell'implementazione sequenziale prevede y1=0 e y2=0, ovvero:

$\mathbf{q}(0) = (0,0,0,0,1,0,1)$

ed in tal caso l'automa evolve come di seguito riportato:

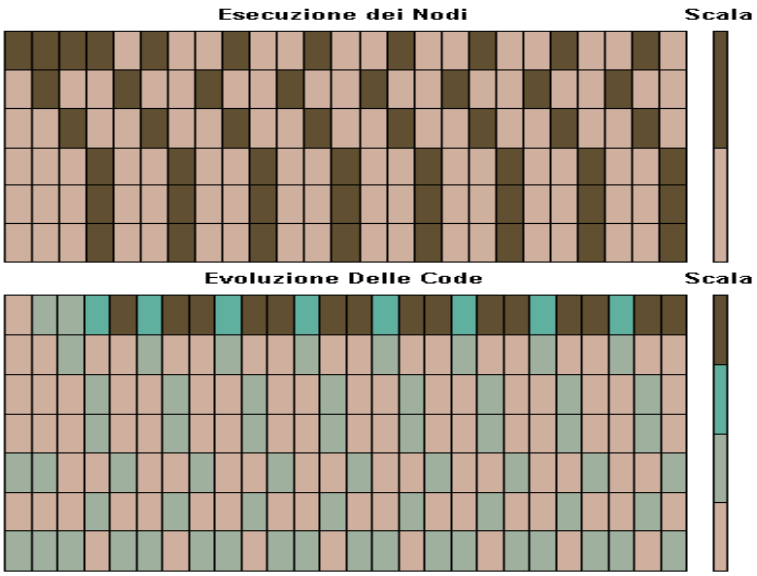


Figura 3-13 - Evoluzione dell'automa associato a due FIR adattivi in cascata (periodo 3)

L'automa raggiunge un periodo 3 in accordo con la presenza del ciclo equiverso di lunghezza 3 (archi 2,4 e 5) inizializzato con un solo elemento in una sola coda. Si nota come il ciclo equiverso di lunghezza 2 (archi 6 e 7) non provochi un aumento del periodo a 6 come potrebbe in generale verificarsi ma si rimette al passo in quanto la coda 7 mantiene il dato per due iterazioni consecutive. Per la simulazione dell'automa si e' adottata una dimensione uniforme delle code pari a 3.
Dal precedente diagramma di evoluzione si nota che i due FIRs (nodi 2 e 3) non vengono mai eseguiti contemporaneamente. Cio' e' dovuto al fatto che la coda 2 che li connette non e' inizializzata. Se una sua inizializzazione fosse accettabile, in termini di degradamento della funzione di trasferimento del sistema, ovvero fosse ammessa l'inizializzazione:

$\mathbf{q}(0) = (0,1,0,0,1,0,1)$

l'automa evolverebbe fino a raggiungere un ciclo di periodo 2:

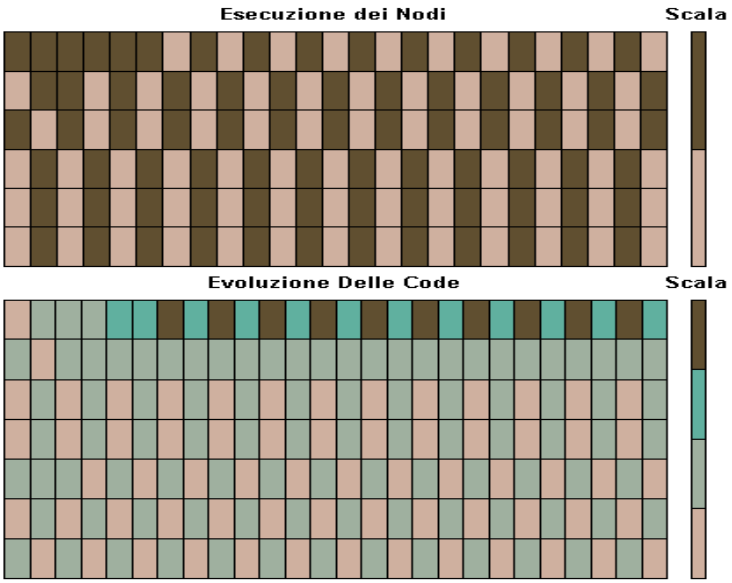
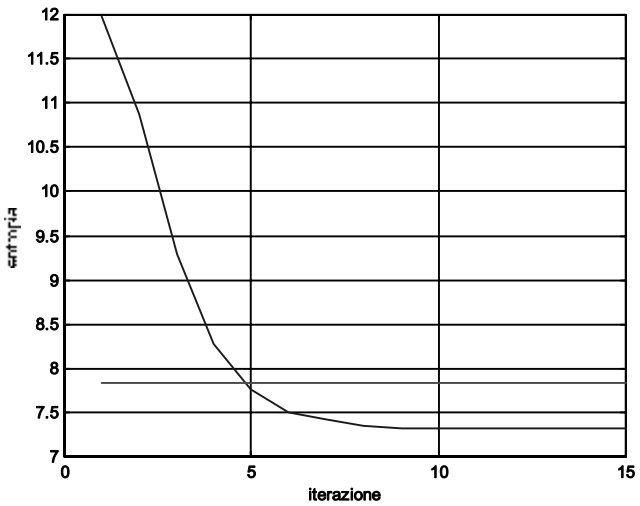


Figura 3-14 - Evoluzione dell'automa associato a due FIR adattivi in cascata (periodo 2)

Viene infine riportato di seguito il grafico di evoluzione dell'entropia associata all'automa in oggetto:



Il grafico e' stato ottenuto facendo evolvere l'automa associato ai due FIR adattivi in cascata per 15 iterazioni a partire da 4915 stati iniziali pari al 30% del numero totale di stati ipotizzando tutte le code di dimensione 3 (il numero totale di stati dell'automa e' dato da 4^7=16384 dove 4 e' il numero di stati possibili per ciascuna coda (0,1,2,3) e 7 il numero di code).
Si possono qui' ripetere le considerazioni fatte nell'esempio precedente, essendo il comportamento dell'entropia del tutto analogo. In questo caso il numero di stati raggiungibili dopo il transitorio e' al piu' $2^{7.9}$=238 (linea orizzontale). Essi non sono equiprobabili perche' l'entropia finale e' tale che se lo fossero dovrebbero essere al massimo $2^{7.3}$=157. Queste valutazioni globali ci garantiscono che il transitorio massimo dell'automa e' circa 8 iterazioni mentre il periodo massimo sara' al piu' 238 ma piu' probabilmente compreso tra 1 e 157. Anche in questo caso l'elevata entropia finale comporta una certa imprevedibilita' dello stato finale dell'automa.

4. OTTIMIZZAZIONE DELLE RISORSE

4.1 Introduzione

Il problema di ripartizione di tasks su piu' processori e la loro sequenza di esecuzione che minimizzi il tempo di esecuzione del periodo dell'automa[6], a prescindere dalla minimizzazione spaziale, e' detto problema di "schedulazione statica" (static scheduling). Il termine "statica" sta ad indicare che la ripartizione dei tasks sui processori e' effettuata una volta per tutte e non puo' essere piu' modificata durante l'esecuzione. E' noto che il problema di schedulazione statica e' in NP ed e' stato affrontato con vari algoritmi di approssimazione, tutti essenzialmente basati sul metodo denominato Hu-level [5]. Questo metodo fa uso del cosidetto grafo di precedenza (Aciclic Precedence Graph o APG) cioe' di un grafo diretto in generale non connesso che determina la dipendenza dell'esecuzione di un certo task da un determinato insieme di altri tasks (suoi nodi predecessori nell'APG). La tecnica Hu-level consiste allora nell'associare, ad ogni task T, un livello corrispondente al massimo tempo necessario al completamento di tutti i tasks dell'APG che siano successori di T lungo un percorso che connette T ad un nodo pozzo, e quindi nell'assegnare, ad un processore libero, il task disponibile (nel senso che tutti i suoi nodi predecessori siano gia' stati eseguiti su un qualche processore) di massimo livello. I casi in cui il suddetto algoritmo non raggiunge l'ottimo sono quelli in cui i livelli assegnati ai nodi siano molto omogenei o addirittura identici. In tal caso infatti la scelta del task di livello massimo corrisponde ad una scelta casuale e ne puo' risultare una ripartizione dei tasks sui processori che non minimizzi i loro tempi di attesa [6]. Nel presente capitolo non verra' usata la tecnica Hu-level sia a causa del succitato problema sia perche' non permette la minimizzazione spaziale e temporale contemporanea; verra' invece sviluppato un algoritmo basato sulla tecnica "Branch & Bound". Si usera' come funzione temporale da minimizzare quella che associa ad una data ripartizione dei tasks su piu' processori un tempo di esecuzione del periodo dell'automa ottenuto tramite una schedulazione Hu-level. Verra' infine dimostrato che il problema di ripartizione la cui funzione da ottimizzare sia la precedente sommata ad una funzione di occupazione spaziale dei task e' ancora in NP.

4.2 APG dell' automa associato ad un DSPA

Definizione. 4.1 (Matrice di esecuzione) Chiamiamo matrice di esecuzione E la matrice |V|xp ottenuta dalla simulazione dell'automa e contenente l'informazione di esecuzione di ogni nodo in ogni istante discreto del periodo p di ripetizione. Per n abbastanza grande si puo' scrivere:

$$E = (\mathbf{v}(n)^T, \mathbf{v}(n+1)^T, \mathbf{v}(n+3)^T, ..., \mathbf{v}(n+p)^T)$$

Osservazione.4.2 (Complessita' della determinazione della matrice di esecuzione E) Simulazioni fatte al calcolatore hanno mostrato lunghezze del transitorio e del periodo di ripetizione dell'automa *A*

[6] definito pero' in un contesto che non fa' uso del modello automa a stati finiti;

polinomiali in |V|. La simulazione e' stata fatta con un approccio statistico generando in modo casuale matrici di incidenza con |V| nodi e |E|=|V|+3 archi di grafi connessi con un'unico nodo sorgente ed un'unico nodo pozzo. Per ciascun valore di |V| sono state generate |V| matrici e lasciato evolvere l'automa a partire da uno stato iniziale, anch'esso generato in modo casuale, fino alla sua convergenza in un ciclo. La dimensione delle code e' stata posta uguale a 3 identica per tutte le code. I risultati ottenuti, mediati sul numero |V| di matrici generate ad ogni |V| sono riportati nel seguente grafico:

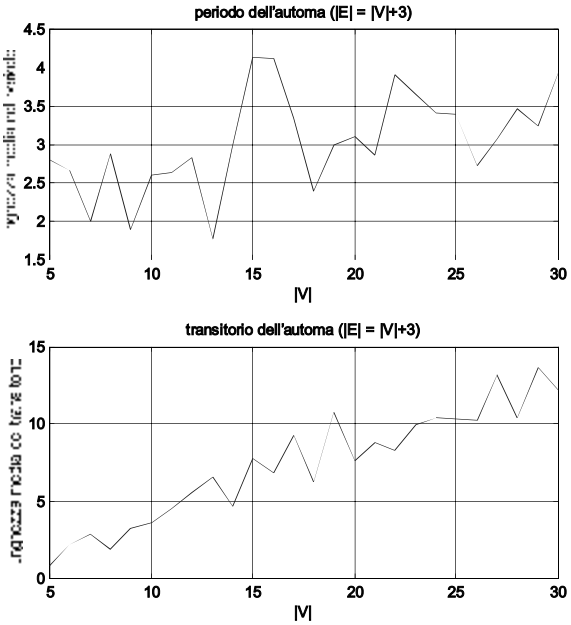


Figura 4-1 - Lunghezza di transitorio e periodo dell'automa A (|E|=|V|+3)

La stessa analisi precedente con |E|=|V|+10 e' invece riportata nella seguente figura 4-2:

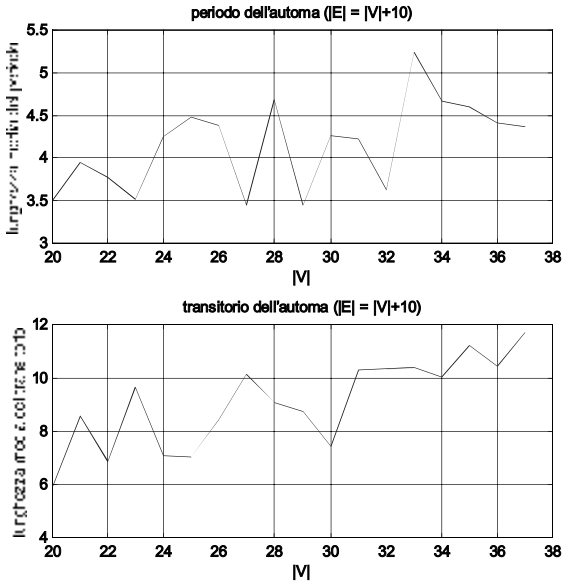


Figura 4-2 - Lunghezza di transitorio e periodo dell'automa A (|E|=|V|+10)

Infine il seguente grafico illustra l'andamento dell'entropia dello stato finale dell'automa ottenuta ancora generando |V| matrici di incidenza per ciascun |V| e facendo quindi evolvere l'automa per 15 iterazioni a partire da un numero di stati iniziali pari al 5% dei suoi stati totali. Poiche' si e' imposto anche che la massima dimensione delle code sia 1 (quindi le code possono assumere in questo caso solo i valori 0 e 1) il numero totale di stati e' dato da $2^{|E|}$ dove si e' ancora scelto |E|=|V|+3.

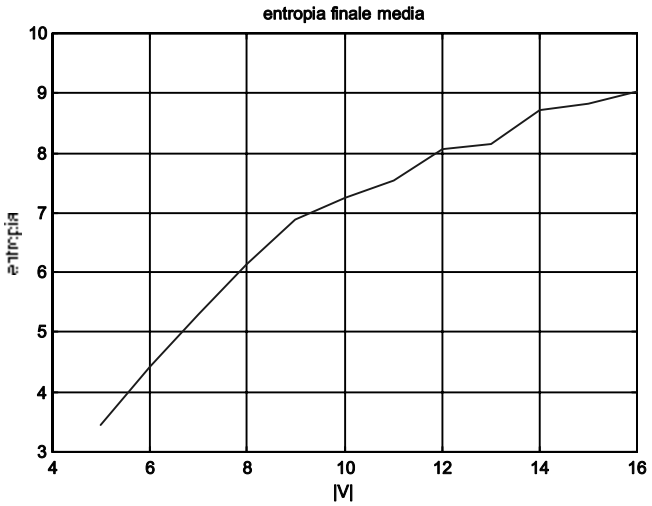


Figura 4-3 - Andamento dell'entropia all'aumentare della complessita' del grafo

Questo andamento dell'entropia conferma che la complessita' dell'automa (intesa come numero di stati finali raggiungibili e quindi come massimo periodo del ciclo finale) aumenta in modo polinomiale rispetto a |V|.

Osservazione.4.3 Se G e' al solito il grafo associato al DSPA in oggetto a cui e' anche relativo l'automa con matrice di esecuzione E, un nodo di una data colonna di E puo' essere eseguito solo dopo che lo siano stati i nodi delle colonne precedenti ad esso adiacenti nel grafo indiretto ottenuto da G ignorando il verso degli archi. Infatti l'esecuzione di un nodo per definizione dipende solo dal contenuto delle sue code di ingresso e di uscita il quale a sua volta varia in funzione dei soli nodi adiacenti al nodo in oggetto. Cio' precisa il fatto che l'esecuzione di un nodo di una data colonna di E non necessariamente deve attendere l'esecuzione di tutti i nodi delle colonne precedenti ma solo un suo sottoinsieme. In generale dunque l'appartenenza alla medesima colonna di E e' condizione sufficiente per l'esecuzione concorrente dei nodi ma non necessaria nel senso che tale esecuzione puo' essere anticipata rispetto al tempo che occorre per eseguire tutti i nodi delle colonne precedenti.

La precedente osservazione ci consiglia di introdurre un APG per tener conto delle dipendenze tra tasks.

Definizione.4.4 (APG associato all'automa) Consideriamo ciascun elemento di valore "1" della matrice di esecuzione E. A causa della presenza di periodi dell'automa che manifestano piu' di una esecuzione per ciascun nodo puo' accadere che piu' elementi di valore "1" corrispondano allo stesso nodo. Costruiamo allora il grafo $H=(V_H,E_H)$ ponendo $|V_H|=n|V|$ dove al solito G=(V,E) e' il grafo associato al DSPA in esame ed n rappresenta la molteplicita' di esecuzione di ciascun nodo all'interno della matrice di esecuzione. Si suppone che i nodi di V_H corrispondenti allo stesso nodo di V si distinguano per un indice aggiuntivo. Detta ora $col_E(k)$ la colonna della matrice di esecuzione E a cui appartiene il nodo $k \in V_H$, definiamo la matrice di adiacenza M_H di H come segue:

$$(M_H)_{ij} = 1 \quad \text{se} \sum_{k=1}^{|E|} (A_G)_{ki} (A_G)_{kj} \neq 0 \quad \text{e} \quad col_E(j)\text{-}col_E(i) \geq 1; \qquad (M_H)_{ij} = 0 \text{ altrimenti}$$

ovvero $(M_H)_{ij}=1$ sse esiste l'arco (i,j) o l'arco (j,i) nella matrice di incidenza A_G ed il nodo i sia su una colonna che precede quella di j nella matrice di esecuzione. Il costo dell'estrazione di $col_E(k)$ e' O(p) mentre quello della sommatoria e' $\Theta(|E|)$ per cui complessivamente determinare M_H da E ed A_G ha un costo $O(|V_H|^2(|E|+p))=O(n^2|V|^2(|E|+p))$ dovendo ripetere i suddetti confronti per ciascuna coppia (i,j) di nodi di H.

Conclusione. (complessita' della determinazione dell'APG dalla legge di evoluzione dell'automa) In base alle precedenti considerazioni si puo' affermare che e' possibile ottenere la matrice di adiacenza dell'APG dell'automa a stati finiti associato ad un DSPA, a partire dalla legge di evoluzione dell'automa stesso, in tempo polinomiale.

4.3 Funzioni di ottimizzazione

Definizione.4.5 (Insieme dei tasks di un processore) Indichiamo con $\theta(j)$ l'insieme dei tasks associati al processore j-esimo e con $\chi_{\theta(j)}$ la corrispondente funzione caratteristica:

$$\begin{cases} \chi_{\theta(j)}(n) = 1 & \text{se il task associato con il nodo n e' assegnato al processore j} \\ \chi_{\theta(j)}(n) = 0 & \text{altrimenti} \end{cases}$$

Indichiamo inoltre con $\varphi(j)$ l'insieme dei tasks differenti associati al processore j-esimo. Ovviamente $\varphi(j) \subseteq \theta(j)$ e $\varphi(j)$ e' completamente individuata da $\theta(j)$. Si puo' dunque anche scrivere $\varphi(j)=\Psi(\theta(j))$ per mettere in evidenza tale dipendenza.

Definizione.4.6 (Parametri caratteristici della risorsa tempo) Ad ogni nodo k=1..|V| del grafo di un DSPA associamo il parametro τ_k corrispondente al tempo di esecuzione[7] del task che il nodo rappresenta (espresso in una data unita' di misura temporale, ad esempio nsec). Il tempo di esecuzione sequenziale del "while true loop" su un unico processore sara' allora dato da:

$$T_1 = \sum_{j=1}^{|V|} \tau_j$$

Detto ora T_s il tempo usato dal processore piu' impegnato per eseguire i nodi ad esso assegnati:

$$T_s = \max_{1 \le h \le \mu} \left(\sum_{j=1}^{|V|} \chi_{\theta(h)}(j)\, \tau_j \right)$$

per il tempo T_μ di esecuzione concorrente del "while true loop" su $\mu<|V|$ processori si avra':

$$T_s \le T_\mu \le T_1$$

Il tempo T_μ viene fatto coincidere con quello ottenuto da una schedulazione basata sulla tecnica Hu-level e fornisce il tempo di esecuzione del periodo dell'automa. Il calcolo di T_μ puo' essere effettuato algoritmicamente a partire dalla famiglia di processori su cui sono ripartiti i tasks $\{\theta(h): h=1..\ \mu\}$, dai tempi di esecuzione di ciascun task $(\tau_1,\tau_2,...,\tau_{|V|})$ e dalla matrice di adiacenza M_H dell'APG H. Le funzioni usate dall'algoritmo sono le seguenti:

---

[7] In questo caso non si intende la complessita' asintotica del task ma il suo tempo effettivo di esecuzione essendo fissata la dimensione dell'input.

W:=max_path(($\tau_1,\tau_2,...,\tau_{|V|}$),$M_H$): ritorna un vettore W contenente per ciascun nodo di V_H il massimo tra tutte le possibili somme dei tempi dei nodi che occorre percorrere da esso per raggiungere un nodo pozzo del grafo H lungo un certo percorso; la funzione ha complessita' O(p|V||V_H|) come si evince dalla seguente analisi:

costo della scansione degli elementi di E per enumerare gli elementi E_{ij}=1	c_1	p|V|
costo delle scansione degli elementi di E dalla colonna j+1 alla colonna p per enumerare i potenziali successori k di i nell'APG H	c_2	(p-j-1)|V|
costo dell'esame delle colonne k ed i della matrice A_G per verificare se un potenziale successore k e' adiacente ad i (e quindi e' un successore di i)	c_3	|E|
complessita' totale	c_1 c_2 c_3	O(p^2|V|^2|E|)

Se la matrice di adiacenza M_H dell'APG H e' precalcolata, le quantita' c_2 e c_3 vengono sostituite da una semplice scansione di una riga di M_H (ricerca dei successori) ottenendo quindi una complessita' totale di O(p|V||V_H|);

z:=max_weight_node(S,W): ritorna il nodo di S a cui e' associato il massimo peso; la funzione ha un costo O(|V_H|);

r:=proc(z,{θ(h): h=1.. μ}): ritorna l'indice del processore a cui appartiene il nodo z; la funzione ha un costo O(|V|);

{P,r,T_{0min}}:=min_time_proc_0(BUSY): ritorna l'indice r del processore con il minimo carico temporale (BUSY e' un vettore con μ elementi contenente i carichi temporali correnti dei processori), T_{0min} e l'insieme dei nodi in esecuzione P che terminano la propria esecuzione in un tempo T_{0min}; la funzione ha un costo O(μ);

{P,r,T_{1min}}:=min_time_proc_1(BUSY): ritorna l'indice r del processore con il minimo carico temporale diverso da zero, T_{1min} e l'insieme dei nodi in esecuzione P che terminano la propria esecuzione in un tempo T_{1min}; la funzione ha un costo O(μ);

S:=update(S,R,Q,M_H): aggiorna l'insieme dei nodi eseguibili S in base all'insieme dei nodi R che hanno terminato la loro esecuzione, all'insieme dei nodi correntemente in esecuzione Q ed alle connessioni dell'APG H di matrice di adiacenza M_H; la funzione effettua, per ciascun nodo di V_H - R - S - Q, un test per determinare se tutti i suoi predecessori in H sono stati eseguiti (sono cioe' in R) , per far cio' usando la struttura dati M_H occorrono O(|V_H|^2) confronti come dimostra la seguente analisi:

costo della scansione dei nodi i di V_H-R-S-Q	c_1	|V_H-R-S-Q|
determinazione della colonna j di E a cui appartiene i	c_2	p
costo delle scansione degli elementi di E dalla colonna 1 alla colonna j-1 per enumerare i potenziali predecessori k di i nell'APG H	c_3	(j-1)|V|
costo dell'esame delle colonne k ed i della matrice A_G per verificare se un potenziale predecessore k e' adiacente ad i (e quindi e' un predecessore di i)	c_4	|E|
complessita' totale	c_1 (c_2 + c_3 c_4)	O(p^2|V||V_H||E|)

Se la matrice di adiacenza M_H dell'APG H e' precalcolata, le quantita' c_2, c_3 e c_4 vengono sostituite da una semplice scansione di una colonna di M_H (ricerca dei predecessori) ottenendo quindi una complessita' totale di O(p|V_H|^2);

Da notare che l'estrazione di massimi e minimi e' effettuata su strutture che cambiano contenuto ad ogni iterazione per cui non conviene effettuare un ordinamento prima della ricerca perche' cio' comporterebbe un costo complessivo O(n log n) (per l'ordinamento, mentre poi l'estrazione del massimo avrebbe costo O(1)) maggiore di O(n) per la ricerca diretta del massimo.

La presenza all'interno dei due cicli "while" della funzione update() fornisce un costo complessivo della funzione eval_time() pari a $O(|V_H|^4)=O(n^4|V|^4)$ dove n e' la molteplicita' di esecuzione di ogni nodo in E.

```
function eval_time((τ1,τ2,...,τ|V|), {θ(h): h=1.. μ},M ) return natural is

  n,x,y,z: node;
  r: natural;                          -- [illegible]
  S: set:={APG source nodes};          -- [illegible]
  R: set:={};                          -- [illegible]
  Q: set:={};                          -- [illegible]
  P: set:={};                  -- [illegible]

  -- [illegible]
  BUSY: array(1..mu) of natural:=(0,...,0);

  T[illegible],T[illegible],Tμ: natural;

  -- [illegible]
  M: array(1..|V |,1..|V |) of natural;

  W:=max_path((τ1,τ2,...,τ|V|),M );
  while ∃x∉R loop
       while ∃y∈S and BUSY(proc(y))=0 loop
           z:=max_weight_node(S,W);
           Q:= Q U {z};
           S:= S - {z};
           r:=proc(z,{θ(h): h=1.. μ});
           BUSY(r):=BUSY(r)+ τz;
           {P,r,T   }:=min_time_proc_0(BUSY);
           BUSY:=BUSY-T[illegible];        -- [illegible]
           R:= R U P;
           Q:= Q - P;
           Tμ:=Tμ+T   ;
           S:=update(S,R,Q,M );
       end loop;
       {P,r,T   }:=min_time_proc_1(BUSY);
       BUSY:=BUSY-T   ;                    -- [illegible]
       R:= R U P;
       Q:= Q - P;
       Tμ:=Tμ+T   ;
       S:=update(S,R,Q,M );
   end loop;
   return Tμ;

end eval_time_APG;
```

Figura 4-4 - Algoritmo di calcolo di T_μ

Si puo' anche dare a questo punto un indicatore del parallelismo ottenuto ripartendo i |V| tasks su μ processori. A tale proposito, detta n la molteplicita' di esecuzione di ciascun nodo di V in E definiamo il fattore:

$$\pi_\mu = \frac{n\,T_1}{T_\mu}$$

Esso vale 1 se l'introduzione di piu' processori non comporta nessuna diminuzione del tempo di esecuzione del "while true loop" mentre vale $\pi_\mu>1$ se l'uso di μ processori porta ad una esecuzione π_μ volte piu' veloce di quella sequenziale.
Ottenuto un certo valore per π_μ puo' risultare del tutto inutile un aumento ulteriore del numero di processori ed anzi tale aumento diventa una perdita netta in termini di tempo macchina superfluo messo a disposizione dell'algoritmo ma non usato. E' comunque possibile tenere sotto controllo questo fenomeno con un ulteriore indicatore definito come:

$$e_\mu = \frac{n\,T_1}{\mu\,T_\mu}$$

che rapporta il tempo nT_1 da distribuire sui μ processori con il tempo globale μT_μ messo a disposizione per la computazione. Se $e_\mu=1$ il sistema dei μ processori avra' tempi di attesa nulli perche' tutto il tempo messo a disposizione della computazione verra' effettivamente usato, se $e_\mu<1$ ci sara', durante l'esecuzione concorrente, un tempo di attesa distribuito tra i vari processori.

Definizione. 4.7 (Parametri caratteristici della risorsa spazio) Ad ogni nodo k=1..|V| del grafo di un DSPA associamo il parametro σ_k corrispondente allo spazio di memoria occupato dal codice compilato (espresso in una data unita' di misura di memoria, ad esempio byte). Lo spazio complessivo di memoria S_μ ottenuto distribuendo i |V| tasks su $\mu<|V|$ processori sara' allora dato da:

$$S_\mu = \sum_{h=1}^{\mu} \sum_{j=1}^{|V|} \chi_{\varphi(h)}(j)\,\sigma_j = \sum_{h=1}^{\mu} \sum_{j=1}^{|V|} \chi_{\psi(\theta(h))}(j)\,\sigma_j$$

nella quale la presenza di $\varphi(h)$ permette di sommare una sola volta i tasks appartenenti allo stesso processore che facciano riferimento alla stessa dichiarazione di funzione o procedura. Dalla precedente si puo' anche calcolare come caso particolare lo spazio complessivo di memoria S_1 occupato dai tasks invocati nel "while true loop" sequenziale, che per definizione usa un solo processore:

$$S_1 = \sum_{j=1}^{|V|} \chi_{\varphi(1)}(j)\,\sigma_j = \sum_{j=1}^{|V|} \chi_{\psi(\theta(1))}(j)\,\sigma_j$$

anche qui' la presenza di $\varphi(1)$ permette di sommare una sola volta i tasks che facciano riferimento alla stessa dichiarazione di funzione o procedura. Per la grandezza S_μ puo' essere dato un fattore di efficienza che fornisca un'indicazione sull'aumento di codice che si deve pagare per avere il parallelismo su μ processori:

$$\varepsilon_\mu = \frac{S_\mu}{S_1}$$

Esso vale 1 se la suddivisione delle funzioni e procedure su piu' processori non provoca aumento di memoria occupata dal codice mentre vale $\varepsilon_\mu>1$ se tale suddivisione comporta un aumento di memoria pari ad ε_μ volte quella del programma sequenziale originale.

Proposizione. 4.8 (Complessita' di $T_\mu + S_\mu$) Fissata la famiglia $F=\{\theta(h): h=1..\ \mu\}$, il calcolo della quantita' $T_\mu + S_\mu$ ha complessita' asintotica polinomiale in |V|.

Dimostrazione. Se indichiamo brevemente con *C*(P) la complessita' asintotica del generico problema P, possiamo senz'altro affermare che:

P	C(P)
Ordinamento di n oggetti (MERGE-SORT)	$C\left(\underset{1\le h\le n}{\text{sort}}\{\}\right)= O(n \log_2 n)$
Ricerca di una chiave in un insieme ordinato di n elementi (ricerca dicotomica)	$C\left(\underset{1\le h\le n}{\text{search}}\{\}\right)= O(\log_2 n)$

Calcolare a funzione $\chi_{\varphi(h)}(j)$ significa determinare se l'elemento j appartiene o no all'insieme $\varphi(h)$. Cio' puo' essere fatto ordinando i |V| oggetti di $\theta(h)$, eliminando i consecutivi identici per ottenere $\varphi(h)$, ed effettuando poi una ricerca della chiave j sull'insieme ordinato $\varphi(h)$. Dunque in base alla precedente tabella possiamo affermare che $C(\chi_{\varphi(h)}(j)) = O(|V| \log_2|V|)$. La tabella successiva descrive il costo complessivo $C(S_\mu)$.

Determinazione di $C(S_\mu)$				
calcolo	**costo**	**numero di volte**	**costo complessivo**	
$\chi_{\varphi(h)}(j)$	$O(\|V\| \log_2 \|V\|)$	$\|V\|\,\mu$	$O(\|V\|^2 \mu \log_2 \|V\|) =$	$O(\|V\|^3 \log_2 \|V\|)$
$\chi_{\varphi(h)}(j)\ \sigma_j$	$\Theta(1)$	$\|V\|\,\mu$	$\Theta(\|V\|\,\mu) =$	$O(\|V\|^2)$
$\sum_{j=1}^{\|V\|}(\dots)$	$\Theta(\|V\|-1)$	μ	$\Theta((\|V\|-1)\,\mu) =$	$O(\|V\|^2)$
$\sum_{k=1}^{\mu} (\dots)$	$\Theta(\mu-1)$	1	$\Theta(\mu-1)$	$O(\|V\|)$

dunque, sommando le quantita' in ultima colonna, si ha $C(S_\mu)=O(|V|^3\log_2|V|)$.

Per quanto riguarda $C(T_\mu)$, dalle considerazioni fatte nella precedente definizione sull'analisi dell'algoritmo di calcolo di T_μ , abbiamo $C(T_\mu)=O(n^4|V|^4)$. Dunque $C(T_\mu + S_\mu)=O(n^4|V|^4)$ che e' di tipo polinomiale.

4.4 Il problema di ottimizzazione RIP-TS

Definizione.4.9 (Problema di ripartizione) Indichiamo con RIP-P il problema di determinare una ripartizione di |V| tasks su μ processori ottimizzando una funzione calcolabile in tempo polinomiale.

Proposizione.4.10 (Complessita' di RIP-P) RIP-P $\in$ NP.

Dimostrazione. Dimostriamo per induzione sul numero μ dei processori che la dimensione dello spazio di ricerca per un prefissato numero $|V|$ di tasks e' $d(\mu,|V|)=\mu^{|V|}=2^{|V|\ \log_2(\mu)}$. Da cio', per l'ipotesi di calcolabilita' polinomiale fatta sulla funzione da ottimizzare segue la tesi essendo lo spazio di ricerca esplorabile in tempo polinomiale da una macchina di Turing non deterministica.

Base: $\mu=2$. La dimensione $d(2,|V|)$ dello spazio di ricerca coincide con il numero di modi diversi in cui e' possibile ripartire $|V|$ tasks su 2 processori. Dunque:

$$d(2,|V|)=\binom{|V|}{0}+\binom{|V|}{1}+\binom{|V|}{2}+\binom{|V|}{3}+\ldots+\binom{|V|}{|V|}=\sum_{k=0}^{|V|}\binom{|V|}{k}=2^{|V|}$$

Passo: Supponiamo per ipotesi induttiva $d(\mu-1,|V|)=(\mu-1)^{|V|}$. Allora i $|V|$ tasks possono essere assegnati al nuovo processore a gruppi di $k=1,2,\ldots$ dando luogo ancora a $2^{|V|}$ combinazioni diverse per ognuna delle quali sui primi $\mu-1$ processori si ripartiscono $|V|-k$ tasks. Quindi:

$$d(\mu,|V|)=d(\mu-1,|V|)\binom{|V|}{0}+d(\mu-1,|V|-1)\binom{|V|}{1}+d(\mu-1,|V|-2)\binom{|V|}{2}+\ldots+d(\mu-1,0)\binom{|V|}{|V|}=$$

$$=\sum_{k=0}^{|V|}d(\mu-1,|V|-k)\binom{|V|}{k}=\sum_{k=0}^{|V|}(\mu-1)^{|V|-k}\binom{|V|}{k}=\sum_{k=0}^{|V|}\binom{|V|}{k}(\mu-1)^{|V|-k}\ 1^{k}=\mu^{|V|}$$

I problemi di ottimizzazione I e II, definiti nel CAP.2, si possono riformulare allora come segue:

Definizione.4.11 (Problema di ottimizzazione delle risorse tempo e spazio) Determinare la famiglia $F=\{\theta(h): h=1..\ \mu\}$ di μ insiemi di tasks (un insieme per ciascun processore) per la quale risulti minima la quantita' (funzioni obiettivo): $T_\mu + S_\mu$.

Indichiamo brevemente questo problema di ripartizione con RIP-TS, dove TS indica che la ripartizione deve essere effettuata ottimizzando le risorse Tempo e Spazio.

Corollario.4.12 (Complessita' di RIP-TS) RIP-TS $\in$ NP.

Dimostrazione. Conseguenza diretta del fatto che RIP-P$\in$NP e che la funzione di ottimizzazione ha complessita' polinomiale.

4.5 Algoritmi per il problema di ottimizzazione RIP-TS

Poiche' RIP-TS $\in$ NP e' lecito affrontare il problema con tecniche algoritmiche che ne riducano la complessita' per ottenere una soluzione approssimata che in generale non e' la soluzione ottima ma che si discosti poco da essa. La tecnica qui' usata e' basata sul metodo "Branch & Bound" [4]. Questo approccio e' applicabile in contesti abbastanza generali ed e' possibile, variando opportunamente alcune sue caratteristiche, regolarne la complessita' e la bonta' della soluzione ottenuta. La tecnica "Branch & Bound" costruisce la soluzione un elemento alla volta, sia essa un vettore o una struttura piu' complessa S, costituita da un numero finito di elementi N. Lo schema dell'algoritmo e' costituito da due parti ben distinte attivate in successione, appunto il branch (separa) ed il bound (limita). Nella prima parte, durante la generica iterazione, l'algoritmo estrae da un insieme A, inizialmente contenente la sola soluzione vuota, una struttura S_k con k elementi e la completa usando una procedura euristica H_1 in modo da ottenere una soluzione approssimata del problema. Se la funzione obiettivo valutata su questa struttura fornisce un valore migliore di quello ottenuto fino a questo istante, la soluzione approssimata viene presa come soluzione ottima. Nella seconda parte, attivata soltanto se S_k e' incompleta, ovvero se $k<N$, si creano tutte le possibili strutture S_{k+1} e si completano usando una procedura euristica H_2 in modo da ottenere un limite

alla migliore soluzione ottenibile S_{k+1} e su di esse si valuta la funzione obiettivo. Se il valore della funzione obiettivo e' migliore di quello ottenuto finora si inserisce S_{k+1} in A per poter costruire a partire da essa una possibile soluzione tramite l'euristica H_1 durante una prossima iterazione, altrimenti si scarta, eliminando dalle successive analisi tutte le soluzioni ottenibili a partire da S_{k+1}. Le due parti dello schema "Branch e Bound" vengono eseguite in successione ripetutamente finche' A non si svuota. A questo punto la migliore soluzione ottenuta e' presa come soluzione del problema.
Le due procedure euristiche H_1 e H_2 sono quindi determinanti per ottenere una soluzione ottima con complessita' accettabile. In particolare esse dovranno essere poco costose perche' chiamate molte volte. Inoltre, affinche' il test da superare per l'inserimento di nuove strutture in A sia falso il maggior numero di volte, H_1 dovra' fornire una soluzione il piu' possibile vicina a quella ottima mentre H_2 dovra' fornire una delimitazione il piu' possibile accurata della migliore soluzione ottenibile da S_{k+1}. Ovviamente i due requisiti di basso costo di esecuzione e precisione del risultato richiesti ad H_1 e H_2 sono spesso in contrasto per cui variandone la complessita' si potranno ottenere algoritmi piu' costosi e piu' precisi (in termini di vicinanza della soluzione alla soluzione ottima) o viceversa.
Nel nostro caso la procedura H_1 usa una variante della tecnica "greedy" [2] che consiste nell'ordinare i nodi del grafo in base ai tempi di esecuzione crescenti ed assegnare tali nodi ai processori scandendo i processori secondo la legge:

$$1,2,3,\ldots,\mu,\mu,\mu\text{-}1,\mu\text{-}2,\ldots,1,1,2,3,\ldots$$

Cio' permette una piu' uniforme distribuzione della somma dei tempi di esecuzione dei nodi assegnati ai vari processori. Viene trascurata per il momento la eventuale ridondanza del codice creata da una siffatta distribuzione, lasciandone la ottimizzazione al "Branch & Bound".
La procedura H_2 e' invece fusa alla procedura di valutazione della funzione obiettivo sulla struttura che fornirebbe H_2 stessa. Essa consiste di due parti separate, una per il calcolo del tempo T_μ ed una per il calcolo dello spazio S_μ . Per il calcolo di T_μ si suppone che vengano rimossi i vincoli di interezza dei nodi, si suppone cioe' che ognuno dei nodi non ancora assegnati ai processori possa essere diviso in piu' parti da assegnare separatamente a processori diversi. In questo modo si ottiene un minimo per T_μ corrispondente alla valutazione della funzione obiettivo su una soluzione che se fosse ammissibile sarebbe sicuramente migliore di qualsiasi altra ottenibile da S_{k+1}. Per il calcolo di S_μ si suppone invece di poter distribuire sui vari processori i nodi non ancora assegnati in modo che essi non provochino ridondanze di codice con se stessi, prescindendo dai pessimi valori che una tale distribuzione potrebbe provocare per quanto riguarda i tempi di esecuzione. La quantita' $T_\mu + S_\mu$ cosi' ottenuta da H_2 sara' il valore della funzione obiettivo da usare per decidere l'eventuale inserimento di nuove strutture in A.

La struttura dati usata dall'algoritmo e' basata sul record nodo definito come segue:

```
type node is
   record
      number: natural;    -- numero di identificazione del nodo
      time: natural;      -- tempo di esecuzione del nodo
      space: natural;     -- spazio di memoria occupato dal nodo
      name: string_32;    -- nome della funzione associata al nodo
   end record;
```

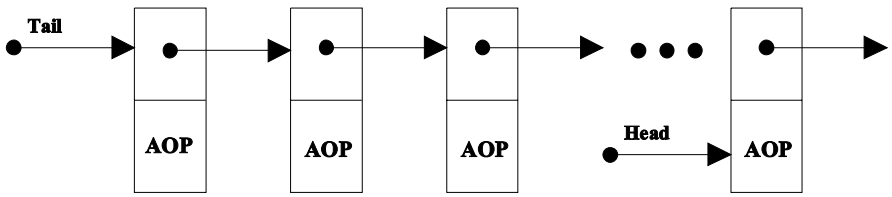
dopodiche' la struttura che implementa un insieme $\theta(h)$ e' un vettore AON (Array Of Nodes) di "record nodo", mentre la struttura che implementa la famiglia F e' un vettore AOP (Array Of Processors) di records del tipo seguente:

```
type proc is
   record
      num: natural;   -- numero di identificazione del processore
      len: natural;   -- lunghezza dell'AON associato al processore
      vec: AON;       -- AON associato al processore
   end record;
```

La famiglia F e' dunque un AOP schematizzabile come segue:

num	len	vec
1	len_1	AON_1
2	len_2	AON_2
...	...	...
μ	len_μ	AON_μ

Infine l'insieme A e' implementato tramite una coda ed e' schematizzabile come:



I parametri di ottimizzazione sono anch'essi organizzati in una struttura del tipo AON e forniscono i dati di ingresso all'algoritmo.
La struttura dell'algoritmo prevede l'uso di un certo numero di funzioni e procedure organizzate in 3 "packages" piu' un "main program". Un "package" in linguaggio ADA non e' altro che una libreria di funzioni e/o procedure che consiste di due parti, una dichiarativa ed una descrittiva (*.ads e *.adb files rispettivamente). La figura 4-5 illustra l'organizzazione di tali "packages".

Prima di commentare l'algoritmo "Branch & Bound" vero e proprio diamo una breve descrizione delle funzioni chiamate direttamente dalla funzione "branch_bound" per poterne poi fare uso nella sua descrizione. Per compattare la descrizione verranno omessi alcuni parametri delle funzioni non rilevanti ai fini della comprensione dell'algoritmo ed in generale verra' usata una notazione informale per non appesantire la descrizione. Non verra' fatta distinzione tra funzioni e procedure perche' tale distinzione e' superflua per la descrizione. Il listato ADA dell'algoritmo completo e' comunque riportato in appendice.

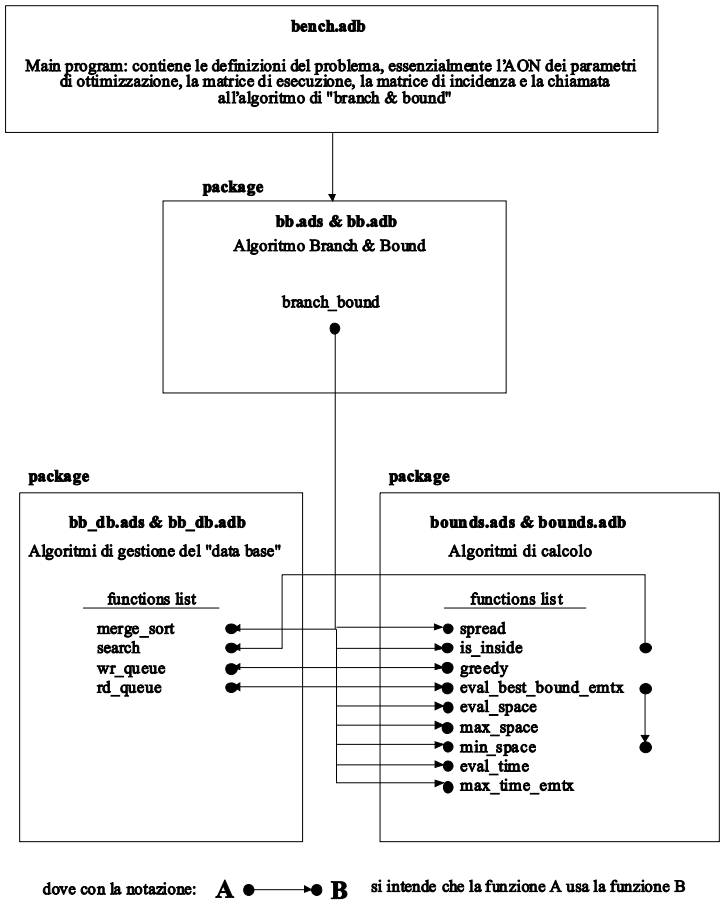


Figura 4-5 - Organizzazione dei "packages" dell'algoritmo di "Branch & Bound"

Funzione merge_sort(x:AON,"time") return AON. Ordina x, contenente i parametri di ottimizzazione forniti in ingresso all'algoritmo, secondo i tempi di esecuzione crescenti dei tasks usando un algoritmo di ordinamento "merge sort".

Funzione wr_queue(x:AOP). Scrive nella coda una struttura x di tipo AOP.

Funzione rd_queue() return x:AOP. Legge dalla coda una struttura x di tipo AOP.

Funzione eval_max(x:AON) return natural. E' in realta' la somma dei due valori che calcolano le funzioni "max_time" e "max_space". Calcola il peggiore valore della funzione da ottimizzare in base ai dati di ingresso contenuti in un AON x. Il risultato di questo calcolo e' dunque:

$$\sum_{j=1}^{|V|} \tau_j + \sum_{j=1}^{|V|} \sigma_j$$

ovvero il massimo tempo di esecuzione complessivo (coincidente con T_1) e la massima ridondanza sullo spazio di memoria occupato dal codice compilato dei tasks.

Funzione min_space(x:AON) return natural. Calcola lo spazio minimo occupato in memoria dal codice compilato delle funzioni o procedure del DSPA, cioe' lo spazio di memoria privo di ridondanze dovute alla duplicazione delle dichiarazioni di funzioni o procedure causato dal passaggio alla versione parallela

del DSPA. Il risultato di questo calcolo e' dunque S_1. Questa funzione e' usata dalla funzione eval_best_bound.

Funzione eval(x:AOP,E:matrix) return natural. E' in realta' la somma dei due valori calcolati dalle funzioni "eval_time" ed "eval_space". Fornisce il valore della funzione da ottimizzare associato ad una data struttura x di tipo AOP. Il risultato di questo calcolo e' dunque $T_\mu + S_\mu$.

Funzione greedy(x:AOP,y:AON,processor_index: natural) return AOP. Calcola una soluzione approssimata al problema usando la tecnica "greedy" rispetto al tempi di esecuzione crescenti dei tasks.

Funzione count(x:AOP) return natural. E' una sotto-funzione della funzione "branch_bound". Calcola il numero di nodi presenti nella struttura AOP.

Funzione spread(processor_index) return natural. Incrementa l'indice di scansione dei processori usando l'euristica: 1,2,3, ... ,μ, μ, μ-1, μ-2, ... , 1,1,2,3, ...

Funzione is_inside(x:AOP,n:node) return boolean. Effettua un test sulla presenza o meno del nodo n nella struttura x di tipo AOP. Ritorna un valore booleano.

Funzione eval_best_bound(x:AOP,E:matrix) return natural. Calcola il miglior valore della funzione obiettivo ottenibile da x. Cio' e' ottenuto come segue. Indichiamo con B l'insieme dei nodi assegnati dopo il k-esimo passo dell'algoritmo, con T_s(x:AOP,B:set) il tempo di esecuzione del processore piu' impegnato dell'AOP quando siano assegnati i soli nodi di $B \subset V$ e con min_space(x:AOP,B:set) la funzione min_space(x:AOP) precedentemente definita con la differenza che nell'AOP x siano presenti solo $|B|<|V|$ nodi. Detto allora TS_k il miglior valore della funzione obiettivo ottenibile completando l'AOP definiamo:

$$TS_k = T_s(x,B) + \text{min\_space}(x,V-B) + \text{bb\_complexity}\,\frac{\sum_{j=1}^{|V|} \chi_{V-B}(j)\,\tau_j}{100}$$

In essa T_s(x,B) e' il termine che tiene conto del valore assunto dalla funzione da ottimizzare sulla struttura x da completare mentre i secondi due addendi valutano rispettivamente la componente spazio e la componente tempo di tale funzione su un ipotetico completamento della struttura in modo che:

1. le funzioni e procedure ancora da assegnare non vengano duplicate (possono comunque provocare duplicazione con funzioni o procedure gia' assegnate);
2. il tempo di esecuzione complessivo delle funzioni e procedure ancora da assegnare contribuisca con una sua frazione al tempo di esecuzione del periodo dell'automa attraverso il coefficiente $0 \le$ bb_complexity ≤ 100;

Il coefficiente bb_complexity determina anche il numero di possibili soluzioni da inserire nella coda usata nel "Branch & Bound" per essere prese in considerazione nella successiva iterazione dell'algoritmo. Per tale motivo bb_complexity puo' essere usato per ottenere un buon compromesso tra tempo di esecuzione e qualita' della soluzione sempre critico in un algoritmo "Branch & Bound".
La descrizione dell'algortimo "Branch & Bound", come si nota dalla figura 4-5, e' contenuta nel package "bb.adb".

Funzione branch_bound(r:AON,E:matrix) return AOP. E' la funzione che implementa l'algoritmo "Branch & Bound" vero e proprio. Ad essa viene fornito in ingresso una struttura r di tipo AON contenente i parametri di ottimizzazione, la matrice di esecuzione E e la matrice di incidenza A_G. Usando le abbreviazioni:

1. "empty_" come prefisso ad un nome di struttura dati qualsiasi, ad indicare la struttura vuota;
2. "U" per indicare l'unione di un elemento di un insieme di tipo AON ad un insieme di tipo AOP (entrambi gli insiemi contengono elementi di tipo nodo);

Pur essendo la funzione "branch_bound" parametrica rispetto a |V| e μ, per semplicita' li supporremo prefissati. La seguente figura 4-6 schematizza tale funzione.

```
function branch_bound(r:AON,E:matrix) return AOP is
        x: AON;
        sol, approx_sol, y: AOP;
        val, z: natural;
begin
        wr_queue(empty_AOP);
        sol:=empty_AOP;
        val:=eval_max(r);
        processor_index:=1;
        x:=merge_sort(r,"time");
        while not empty_queue loop
                y:=rd_queue();
                approx_sol:=greedy(y,x,processor_index);
                z:=eval(approx_sol,E);
                if z<val then
                        sol:=approx_sol;
                        val:=z;
                end if;
                if count(y)<|V| then
                        processor_index :=spread(processor_index);
                        for h in 1..|V| loop
                                if not is_inside(y,x(h)) then
                                        z:=eval_best_bound(y U x(h),E);
                                        if z<val then
                                                wr_queue(y U x(h));
                                        end if;
                                end if;
                        end loop;
                end if;
        end loop;
end branch_bound;
```

Figura 4-6 - Algoritmo "Branch & Bound"

4.6 Esempi

4.6.1 Il problema di ottimizzazione RIP-TS per un filtro IIR

In questo paragrafo viene presentata una applicazione del precedente algoritmo al filtro IIR gia' trattato in un precedente esempio. I parametri di ottimizzazione come detto sono organizzati in una struttura di tipo AON. Con rifererimento ai campi del record "node", precedentemente definito, tali parametri sono riportati nella seguente tabella.

number	time	space	name
1	1	1	source
2	1	1	add
3	1	1	mac
4	N	3	shift
5	N	3	shift
6	N	3	forward
7	N	3	backward
8	1	1	sink

Figura 4-7 - Attributi spazio e tempo delle funzioni di un IIR

In accordo all'esempio succitato poniamo l'ordine del filtro N=5 e consideriamo la matrice di esecuzione:

$$E_{IIR} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Da essa e dalla matrice di incidenza $(A_G)_{IIR}$ si estrae il seguente APG nel quale sono riportati, per ciascun nodo, anche i tempi necessari a raggiungere un nodo pozzo lungo il peggior percorso (in termini di tempi di esecuzione dei nodi). La posizione dei nodi e' conforme a quella degli "1" nella matrice E_{IIR}.

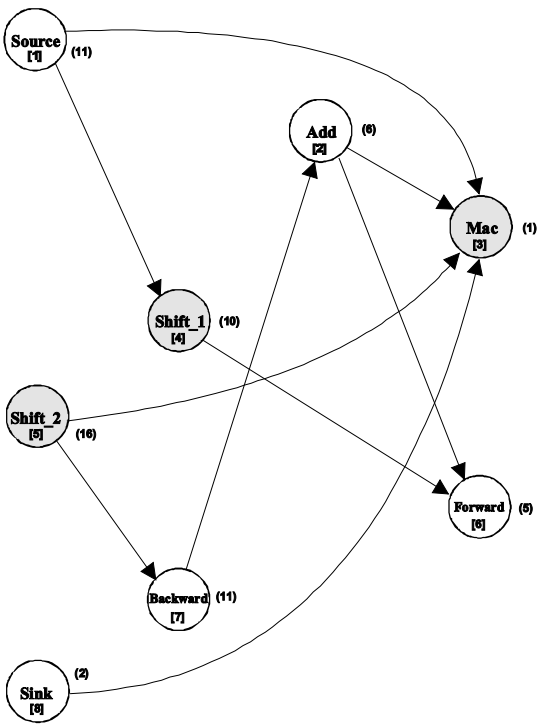


Figura 4-8 - APG dell'automa associato ad un IIR

Preso un numero di processori μ=2, l'algoritmo "Branch & Bound" fornisce la ripartizione di figura 4-9 nella quale si nota che l'unica funzione chiamata piu' di una volta ("shift") e' assegnata allo stesso processore. Non sono avvenute dunque duplicazioni di codice e quindi ε_μ=1.

num	len	vec
1	3	AON_1= Node 3 (Time 1 Space 1 Name "mac") Node 4 (Time 5 Space 5 Name "shift") Node 5 (Time 5 Space 5 Name "shift")
2	5	AON_2= Node 1 (Time 1 Space 1 Name "source") Node 6 (Time 5 Space 5 Name "forward") Node 2 (Time 1 Space 1 Name "add") Node 8 (Time 1 Space 1 Name "sink") Node 7 (Time 5 Space 5 Name "backward")

Figura 4-9 - Ripartizione dei tasks di un IIR su 2 processori

Per cio' che riguarda i parametri temporali, analizziamo la schedulazione ottenuta dall'algoritmo:

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

Figura 4-10 - Schedulazione concorrente su 2 processori di un IIR

Come si nota T_μ=16. Poiche' la molteplicita' dei nodi nella matrice di esecuzione e' in questo caso n=1 e T_1=24, si ottiene: $\pi_\mu = nT_1 / T_\mu$ =1.5. Infine $e_\mu = nT_1/\mu T_\mu$=24/(2x16)=0.75 per cui solo il 25% del tempo totale e' di attesa per i processori, mentre il 75% e' devoluto alla computazione.

4.6.2 Il problema di ottimizzazione RIP-TS per un due filtri FIR adattivi in cascata

Con riferimento al corrispondente esempio del CAP.2 consideriamo i seguenti parametri di ottimizzazione per una struttura costituita da due filtri FIR adattivi in cascata:

number	time	space	name
1	1	1	source
2	N	3	FIR
3	N	3	FIR
4	2	2	estimator
5	2	2	estimator
6	1	1	sink

Figura 4-11- Attributi spazio e tempo delle funzioni di due FIR adattivi in cascata

Qui' N indica al solito l'ordine del filtro. In accordo all'esempio succitato poniamo N=3. L'evoluzione dell'automa con l'inizializzazione q0=(0,0,0,0,1,0,1), fornisce la seguente matrice di esecuzione di periodo 3:

$$E_{ADAPT\_FIR} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

a cui corrisponde l'APG seguente che riporta anche tra () il tempo necessario ad ogni nodo per raggiungere un nodo pozzo seguendo il percorso piu' lungo in termini temporali.

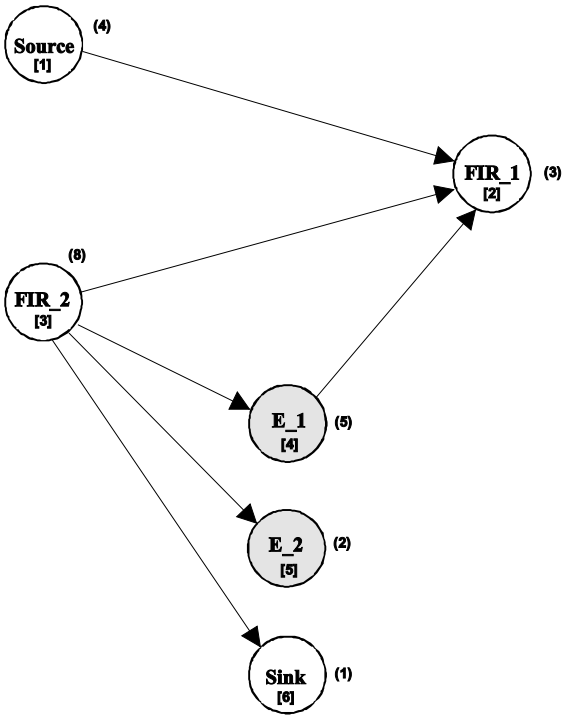


Figura 4-12 - APG dell'automa associato ad un due FIR adattivi in cascata

Eseguendo allora l'algoritmo "Branch & Bound" con un numero di processori μ=2 si ottiene:

num	len	vec
1	2	AON_1= Node 4 (Time 2 Space 2 Name "estimator") Node 5 (Time 2 Space 2 Name "estimator")
2	3	AON_2= Node 1 (Time 1 Space 1 Name "source") Node 3 (Time 3 Space 3 Name "FIR") Node 6 (Time 1 Space 1 Name "sink") Node 2 (Time 3 Space 3 Name "FIR")

Figura 4-13 - Ripartizione dei tasks di due FIR adattivi in cascata su 2 processori

Come si nota i tasks identici sono stati inseriti nel medesimo processore per cui si ha subito ε_μ=1. Inoltre l'algoritmo fornisce anche una possibile schedulazione dei task:

proc N. 1				4	4	5	5	
proc. N. 2	3	3	3	1	6	2	2	2
tempo	1	2	3	4	5	6	7	8

Figura 4-14 - Schedulazione concorrente su 2 processori di due FIR adattivi in cascata

Da essa si ricava T_μ=8. La molteplicita' dei nodi nella matrice di esecuzione e' anche in questo caso n=1 mentre T_1=12. Si ha: $\pi_\mu = nT_1 / T_\mu$ =1.5 e $e_\mu = nT_1/\mu T_\mu$=12/(2x8)=0.75 per cui solo il 25% del tempo totale e' di attesa per i processori, mentre il 75% e' devoluto alla computazione.

5. PROCESSO DI PARALLELIZZAZIONE

5.1 Introduzione

L'obiettivo di questo capitolo e' fornire un metodo automatico di trasformazione di un DSPA in una sua versione concorrente. Cio' verra' fatto sfruttando le caratteristiche del linguaggio ADA che permettono di organizzare un programma in tasks la cui esecuzione avvenga contemporaneamente. Naturalmente se il programma ADA verra' poi eseguito su una macchina singolo processore, la concorrenza sara' soltanto emulata, se si dispone invece di una macchina multi-processore, i tasks verranno effettivamente eseguiti contemporaneamente. La allocazione dei tasks sui processori e la loro schedulazione temporale e' normalmente demandata al sistema operativo presente sulla macchina; nel nostro caso invece entrambe queste azioni verranno effettuata in base ai risultati del precedente capitolo. L' unica assunzione che qui' facciamo e' quella di disporre di una macchina multi-processore del tipo APM definita nel CAP.2, basata su un microprocessore che sia dotato di un compilatore ADA. La trasformazione del DSPA dovra' quindi prevedere lo sviluppo di un sistema operativo minimo in grado di gestire lo scambio dati tra i tasks e la traduzione del "while true loop" in forma concorrente.

5.2 La programmazione concorrente in ADA

Uno dei concetti che distinguono il linguaggio ADA dai linguaggi di programmazione tradizionali e' quello di "task". Questo concetto e' analogo e affianca quello di funzione e procedura con la fondamentale differenza che i tasks sono eseguiti in modo concorrente. La sintassi con cui viene dichiarato un task T e' la seguente [7][8]:

```
task T is
        entry E(input/output variables)
end T;
task body T is
        variables declaration;
begin
        accept E(input/output variables)
                function/procedure/task calls;
        end E;
end T;
```

Si distinguono due sezioni principali, una cosidetta di specifica del task ed una detta corpo ("body") del task. Nella specifica vengono dichiarate una o piu' "entry" che individuano un sottoprogramma sequenziale contenuto nel task ed i parametri di ingresso/uscita ad esso associati. Ognuno di tali sottoprogrammi e' dichiarato nel corpo del task preceduto dalla parola chiave "accept". Mentre piu' tasks sono eseguibili in parallelo, i sottoprogrammi di un medesimo task sono eseguibili in modo esclusivo. Un task puo' trovarsi in uno dei seguenti stati:

- attivo: e' stata elaborata la parte dichiarativa del corpo fino al "begin" incluso;
- completato: e' stato elaborato l'intero corpo fino alla "end" inclusa;
- terminato: e' completato e tutti i tasks da esso chiamati (attivati) sono completati;

Tutti i tasks dichiarati staticamente (cioe' non allocati dinamicamente tramite l'istruzione "new") vengono dunque attivati, in un ordine casuale, subito dopo la messa in esecuzione del programma. In particolare quindi, se un task non possiede dichiarazioni di entry e non ha tasks figli, verra' attivato ed eseguito immediatamente raggiungendo subito la condizione di terminato. Questa situazione verra' sfruttata nella implementazione concorrente di un DSPA prevenendo la terminazione del task tramite l'inserimento di un ciclo infinito ("loop" ... "end loop") all'interno del corpo.
Quando un task esegue una "entry" ed un secondo task esegue il corrispondente "accept" si parla di "rendez-vous" tra i due tasks. E' questo il metodo messo a disposizione dal compilatore ADA per la sincronizzazione di tasks: ad ogni "entry" del task e' associata una coda in cui vengono inserite le chiamate, se il task chiamato e' occupato, quello chiamante resta in attesa, quando il task chiamato esegue l'"accept" relativo alla "entry", dalla coda viene rimossa una chiamata ed il task chiamante termina la sua attesa.

5.3 Trasformazione di un DSPA sequenziale in un DSPA concorrente

La seguente figura 5-1 illustra il procedimento di trasformazione di un DSPA sequenziale in un DSPA concorrente. In tale processo si possono distinguere le seguenti fasi:

1. progettazione del DSPA sequenziale tramite la stesura di due listati sorgenti di cui uno (DSPA While Loop) contenente il "while true loop" e l'altro (DSPA Functions Package), organizzato in un "package", contenente la descrizione delle funzioni usate nel "while true loop"; simulazione del DSPA sequenziale per verificarne la correttezza funzionale; in questa fase si useranno esclusivamente strumenti software standard quali compilatori e linkers commerciali;
2. estrazione della matrice di incidenza del grafo associato al DSPA, della matrice di esecuzione, delle informazioni di ripartizione delle funzioni sui processori del sistema e infine creazione della versione concorrente del DSPA (DSPA Task Oriented) che sostituira' il "DSPA While Loop" mentre il "DSPA Function Package" sara' ancora utilizzato intatto; in questa fase verranno utilizzati esclusivamente strumenti software proprietari sviluppati appositamente per la trasformazione del DSPA sequenziale in una sua versione concorrente;
3. linking del "DSPA Task Oriented" con l' "Operating System Package" per ottenere l'eseguibilita' del "DSPA Task Oriented" sulla APM; simulazione del DSPA concorrente per la verifica della funzionalita'; in questa fase si utilizzera' l' "Operating System Package" di sviluppo proprietario che assolve essenzialmente i compiti di gestione delle code di scambio dati tra le varie funzioni e procedure del DSPA;

Gli strumenti software che devono essere sviluppati appositamente per realizzare il precedente flusso sono dunque i seguenti:

- un PARSER che a partire dai listati sorgenti di un DSPA sequenziale sia in grado di estrarre la matrice di incidenza del grafo e lo stato iniziale dell'automa ad esso associati;

- un SIMULATORE dell'automa associato per determinare la matrice di esecuzione necessaria alla ottimizzazione delle risorse e lo stato finale del transitorio dell'automa che come si vedra' successivamente sara' necessario per implementare la schedulazione prevista dall'algoritmo di ottimizzazione;
- un OTTIMIZZATORE che a partire dai parametri tempo di esecuzione e spazio di memoria occupato dalle funzioni e procedure del DSPA sia in grado di determinare una loro ripartizione ottima sui processori messi a disposizione dal sistema;
- un TRASLATORE sequenziale/concorrente che usando le informazioni di interconnessione tra le funzioni e procedure del DSPA messe a disposizione dalla matrice di incidenza del grafo associato e le informazioni di ripartizione delle stesse funzioni e procedure sui processori del sistema, fornisca un listato sorgente che usi strutture concorrenti del linguaggio ("tasks") e che sostituisca il listato sorgente "DSPA While Loop" nella versione concorrente del DSPA;
- un SISTEMA OPERATIVO minimo che si occupi della gestione dello scambio dati tra le procedure e funzioni del DSPA che avviene tramite delle code;

Ad essi e' utile aggiungere alcuni "profiler" che effettuano una analisi dei risultati ottenuti in vari punti del flusso di progetto. I seguenti tre analizzatori sono di una certa utilita':

- S-PROFILER (Sequential Profiler Package): E' una libreria scritta in linguaggio ADA, da collegare con il DSPA sequenziale tramite un linker. Le funzioni della libreria possono essere usate all'interno del "while true loop" per estrarre dalla esecuzione sequenziale del DSPA i parametri temporali delle sue funzioni e procedure. I parametri spaziali, essendo essi di natura statica, possono essere invece calcolati senza eseguire il DSPA, effettuando una analisi della loro allocazione in memoria o, quando possibile, direttamente dai files di report del compilatore ADA;
- C-PROFILER (Concurrent Profiler Package): E' una libreria di funzioni in linguaggio ADA, da collegare con il DSPA concorrente tramite un linker. Le funzioni della libreria possono essere usate per estrarre dalla esecuzione concorrente del DSPA sia i tempi di esecuzione delle varie funzioni e procedure, sia i tempi occupati dal processore per il trasferimento dati. Questi ultimi sono indicativi di un corretto uso dei controllori DMA e possono essere utilizzati per analizzare la bonta' della ripartizione e degli stessi strumenti di sviluppo;
- F-PROFILER (Feedback Profiler): E' un analizzatore dei risultati dell'ottimizzazione in grado di estrarre i nodi dominanti in termini di tempo di esecuzione e/o dimensioni di memoria programma ai fini di fornire indicazioni al progettista per eventuali correzioni da effettuare sul DSPA sequenziale;

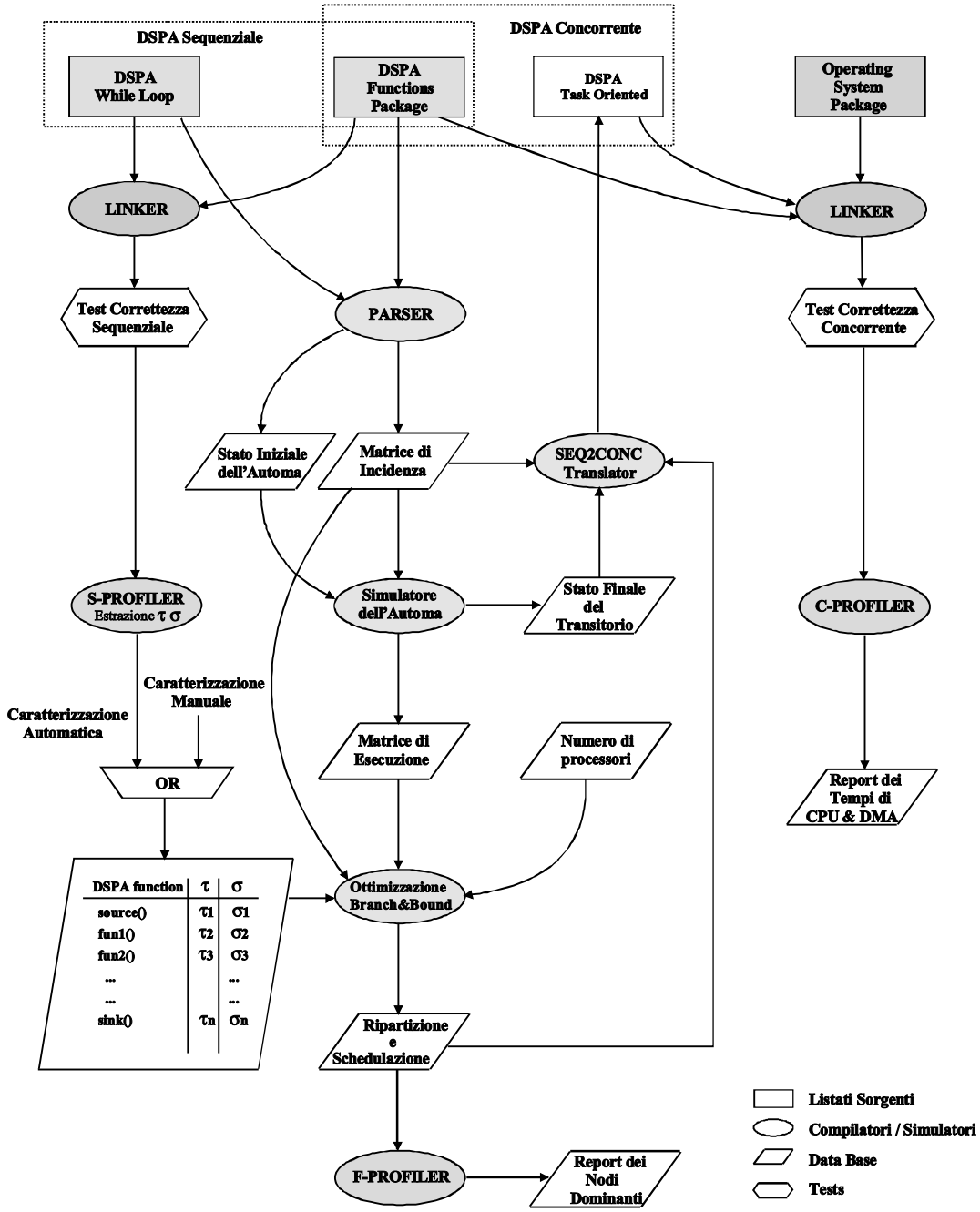


Figura 5-1 - Processo di trasformazione di un DSPA sequenziale in un DSPA concorrente

5.4 Il sistema operativo

Il sistema operativo minimo ("Operating System Package"), necessario all'implementazione concorrente del DSPA su una APM, sara' di natura distribuita, cioe' una sua copia sara' presente su ogni processore del sistema. Esso dovra' provvedere essenzialmente alla gestione delle code di scambio dati tra i "tasks" contenenti le funzioni e procedure del DSPA. Ogni coda, in accordo alla definizione di connessione logica unidirezionale data nel CAP.2, avra' come elementi degli array di interi, di dimensione "wide" e sara'

caratterizzata dai due puntatori "tail" o ultimo dato inserito e "head" o primo dato disponibile per l'estrazione oltreche' dai valori "full" di dimensione massima della coda e "level" di quantita' di dati attualmente presenti nella coda. Lo stato di una coda e' definito dunque dal record seguente [8]:

```
type queue_status is
   record
      tail: qea_ptr;     -- puntatore all'ultimo dato inserito
      head: qea_ptr;     -- puntatore al primo dato da estrarre
      level: natural;    -- numero di dati presenti nella coda
      wide: natural;     -- dimensione di un elemento della coda
      full: natural;     -- dimensione della coda
   end record;
```

Lo stato del sistema di code completo e' allora definito tramite un vettore AQS (Array of Queues Status) contenente |E| elementi del tipo "records queue_status". La gestione del sistema di code e' costituito da una struttura concorrente che fa uso del costrutto ADA "task". Tale struttura concorrente si rende necessaria per evitare eventuali conflitti dovuti all'inserimento ed alla estrazione contemporanea di dati da una coda, evento sempre possibile in un sistema asincrono. Poiche' le operazioni di inserimento ed estrazione di una data coda sono assegnate ad un medesimo "task", solo una alla volta delle due potra' essere eseguita, evitando cosi' la contemporaneita' delle azioni. Rigorosamente in pseudo-ADA cio' si ottiene definendo una struttura di "tasks" il cui tipo base e':

```
task type queue_task is
   entry tx(wr_qea_ptr,AQS,edge,node);       -- inserimento
   entry rx(rd_qea_ptr,AQS,edge,node);       -- estrazione
end queue_task;
```

la gestione dell'intero sistema di code e' allora costituita da una vettore AQT (Array of Queues Task) contenente |E| elementi del tipo "queue_task" ognuno dei quali contiene le due distine funzioni tx() ed rx() che effettuano le operazioni di inserimento ed estrazione rispettivamente. Da notare che questo meccanismo permette la allocazione di un numero di coppie di funzioni distinte tx() ed rx() pari al numero effettivo delle code presenti.
Per quanto riguarda le funzioni messe a disposizione dal sistema operativo, esse sono le seguenti:

Funzione rece_ready(x: node, qs: AQS, IM: matrix) return boolean. Effettua un test su tutte le code in ingresso al nodo x, ricavandole dalla matrice di incidenza IM, per verificare se esse siano o meno tutte non vuote. I dati da testare sono contenuti in qs.

Funzione send_ready(x: node, qs: AQS, IM: matrix) return boolean. Effettua un test su tutte le code in uscita al nodo x, ricavandole dalla matrice di incidenza IM, per verificare se esse siano o meno tutte non piene. I dati da testare sono contenuti in qs.

[8] Chiamiamo "qd" il tipo array di interi che costituisce un elemento della coda, usando l'abbreviazione di "queue data". Indichiamo invece con "qea" l'elemento della coda comprensivo del puntatore "qea_ptr" all'elemento successivo usando l'abbreviazione di "queue element array".

Procedura rece_queue(z: qd, x: node, y: edge, qs: AQS, q: AQT). Effettua l'estrazione del dato z dalla coda y in ingresso al nodo x usando la funzione rx() di q. Il puntatore al dato da estrarre e' contenuto in qs.

Procedura send_queue(z: qd, x: node, y: edge, qs: AQS, q: AQT). Effettua l'inserimento del dato z nella coda y in uscita dal nodo x usando la funzione tx() di q. Il puntatore all'elemento della coda in cui inserire il nuovo dato e' contenuto in qs.

Procedura send_broadcast(z: qd, x: node, IM: matrix, qs: AQS, q: AQT). Effettua l'inserimento del dato z su tutte le coda in uscita dal nodo x, ottenute dalla matrice di incidenza IM, usando la funzione tx() di q. Il puntatore all'elemento della coda in cui inserire il nuovo dato e' contenuto in qs.
Anche in questo caso le precedenti funzioni sono organizzate in una struttura piu' complessa il cui elemento base e':

```
type queue_man is
   record
      rece_ready_check: ready_ptr;     -- puntatore a rece_ready
      rece_make: rece_queue_ptr;       -- puntatore a rece_queue
      send_ready_check: ready_ptr;     -- puntatore a send_ready
      send_make: send_queue_ptr;       -- puntatore a send_queue
      send_broadcast: send_queue_ptr_broadcast;
   end record;
```

cosicche' la gestione dell'intero sistema di code sia possibile attraverso il vettore AQM (Array of Queues Managers) contenente |V| elementi del tipo "record queue_man". Cio' permette di assegnare uno dei precedenti records a ciascun nodo del sistema ed avere quindi effettivamente |V| copie di ciascuna delle precedenti funzioni e procedure.
L'uso di questo apparato di strutture dati e funzioni che costituisce un sistema operativo minimo e' devoluto esclusivamente al "DSPA Task Oriented" il quale potra' effettuare le chiamate a funzioni e procedure dichiarate all'interno del "DSPA Functions Package" per quanto riguarda la computazione specifica dell'algoritmo mentre dovra' ricorrere alle funzioni e procedure dell' "Operating System Package" per quanto riguarda la comunicazione tra i nodi.

5.5 Implementazione concorrente di un DSPA

Per descrivere la forma concorrente di un DSPA, premettiamo due possibili criteri di implementazione della schedulazione dei tasks su una APM.

Definizione.4.13 (Auto-Schedulazione) Se i nodi del grafo associato ad un certo DSPA vengono ripartiti tra i vari processori disponibili in base al un certo criterio di ottimizzazione, una semplice tecnica di schedulazione consiste nell'eseguire ogni funzione o procedura sse tutte le sue code in ingresso sono non vuote e tutte le sue code in uscita sono non piene come descritto nella seguente figura 5-2.

```
-- processore j-esimo

for h:=1.."numero di nodi contenuti nel processore j-esimo" loop
        execute:=1;
        for k:=1.."numero delle code di ingresso al nodo h-esimo" loop
                if "k-esima coda di ingresso del nodo h-esimo"="vuota" then
                        execute:=0;
                end if;
        end loop;
        for k:=1.."numero delle code di uscita al nodo h-esimo" loop
                if "k-esima coda di uscita del nodo h-esimo"="piena" then
                        execute:=0;
                end if;
        end loop;
        if execute=1 then
                "esegui la funzione o procedura associata al nodo h-esimo"
        end if;
end loop;
```

Figura 5-2 - Auto Schedulazione

Usando il precedente metodo di auto-schedulazione, ogni funzione e procedura di un dato processore non rispetta una sequenza prefissata di esecuzione ma viene attivata esclusivamente in base alla presenza o meno di dati nelle code incidenti al nodo che la rappresenta. Ne deriva una sorta di schedulazione dinamica che ha il pregio della semplicita' ma il difetto di non essere vincolata alla sequenza di esecuzione determinata dall'algoritmo di ottimizzazione. Il metodo puo' essere pero' modificato per eliminare questo comportamento indesiderato del sistema. Supposte infatti note una ripartizione ed una schedulazione ottime delle funzioni e procedure sulla APM, e' sufficiente introdurre all'interno di ogni processore un "token-ring". Un token-ring e' un meccanismo attraverso il quale ogni funzione stabilisce, con la sua esecuzione, quale sara' la successiva funzione da eseguire, non appena naturalmente cio' sia possibile in base alla presenza o meno di dati nelle sue code incidenti. La prefissata sequenza ottima sara' dunque facilmente implementabile, imponendo questa rigidita' alle sequenze di esecuzione delle funzioni e procedure di ogni processore della APM. Resta un'ultimo problema da superare legato al transitorio dell'automa. La schedulazione ottima non puo' essere infatti usata durante il transitorio perche' nodi e code evolvono in modo aperiodico. La soluzione sta nell'inizializzare le code della APM conformemente allo stato che esse raggiungono alla fine del transitorio, da questo istante discreto in poi infatti il comportamento dell'automa sara' esclusivamente periodico e una schedulazione prefissata sara' senz'altro possibile. I dati da inserire nelle code potranno essere precalcolati in base ad una simulazione dell'implementazione concorrente oppure semplicemente posti uguali a zero se cio' non corrisponde ad una grave degradazione delle prestazioni del sistema.

Definizione.4.14 (Schedulazione Token-Ring) Se si inizializzano le code della APM in modo che contengano un numero di dati conforme allo stato raggiunto alla fine del transitorio dall'automa associato al DSPA, e' possibile imporre un meccanismo di schedulazione completamente statico che fa' uso di una tecnica token-ring come descritto dalla seguente figura 5-3.

```
-- processore j-esimo

for h:=1.."numero di nodi contenuti nel processore j-esimo" loop
        if token(h)=true then
                execute:=1;
                for k:=1.."numero delle code di ingresso al nodo h-esimo" loop
                        if "k-esima coda di ingresso del nodo h-esimo"="vuota" then
                                execute:=0;
                        end if;
                end loop;
                for k:=1.."numero delle code di uscita al nodo h-esimo" loop
                        if "k-esima coda di uscita del nodo h-esimo"="piena" then
                                execute:=0;
                        end if;
                end loop;
                if execute=1 then
                        "esegui la funzione o procedura associata al nodo h-esimo"
                        token(h):=false;
                        token(next(h)):=true;
                end if;
        end if;
end loop;
```

Figura 5-3 - Schedulazione Token-Ring

Siamo ora in grado di descrivere come deve avvenire la trasformazione del DSPA sequenziale nella corrispondente versione concorrente. Supponiamo inizialmente che sia disponibile un processore per ciascuna chiamata a procedura o funzione presente nel "while true loop" del DSPA originario. In tal caso ognuna di tali chiamate dovra' essere contenuta in un "task". Se manteniamo la notazione del precedente paragrafo per i tipi di dati e le funzioni del sistema operativo, una funzione F(a,b) con due variabili in ingresso dovra' essere contenuta in un "task" del tipo seguente:

```
task T is
end T;
task body T is
   a,b,c: qd;
begin
   loop
      if not reset then
         if qman(x).send_ready_check(x,qs,IM) and
            qman(x).rece_ready_check(x,qs,IM) then

            qman(x).rece_make(a,x,y,qs,q);
            qman(x).rece_make(b,x,y,qs,q);
            c:=F(a,b);
            qman(x).send_make(c,x,IM,qs,q);

         end if;
      end if;
   end loop;
end T;
```

si nota che il "task" non contiene dichiarazioni di "entry" e dunque verra' attivato immediatamente dopo la messa in esecuzione del programma ma evitera' la terminazione prematura a causa della presenza del "loop" infinito presente nel corpo. La variabile "reset" sara' usata per tenere in stato di attesa tutti i "tasks" presenti finche' non siano state completate le operazioni di inizializzazione. Quando il "reset" abilita l'esecuzione delle istruzioni del corpo il "task" effettua continuamente un test sulle proprie code in ingresso ed in uscita ed eventualmente esegue la funzione ad esso associata (dichiarata nel "DSPA Functions Package") usando i dati in ingresso e spedendo il risultato su tutte le sue code di uscita.
Quando invece non siano disponibiliti tanti processori quante sono le chiamate a funzione o procedura, un singolo "task" dovra' identificare un processore del sistema e quindi racchiudere tutte le chiamate a funzione o procedura assegnate a tale processore. L'organizzazione e' riportata nella di seguito:

```
task P is
   end P;
   task body P is
      x,y,z: qd;
   begin
      loop
         if not reset then

            if qman(x1).send_ready_check(x1,qs,IM) and
               qman(x1).rece_ready_check(x1,qs,IM) then

               qman(x1).rece_make(a,x1,y1,qs,q);
               qman(x1).rece_make(b,x1,y1,qs,q);
               z:=F1(a,b);
               qman(x1).send_make(z,x1,IM,qs,q);

            end if;

            if qman(x2).send_ready_check(x2,qs,IM) and
               qman(x2).rece_ready_check(x2,qs,IM) then

               qman(x2).rece_make(a,x2,y2,qs,q);
               z:=F2(a);
               qman(x2).send_make(z,x2,IM,qs,q);

            end if;

            if qman(x3).send_ready_check(x3,qs,IM) and
               qman(x3).rece_ready_check(x3,qs,IM) then

               qman(x3).rece_make(a,x3,y3,qs,q);
               qman(x3).rece_make(b,x3,y3,qs,q);
               z:=F3(a,b);
               qman(x3).send_make(z,x3,IM,qs,q);

            end if;

         end if;
      end loop;
   end P;
```

E' illustrato il caso in cui il processore in esame effettui la computazione delle tre funzioni F1, F2, F3 ricavando al solito i dati di ingresso dalle code dei singoli nodi e spedendo i risultati in quelle di uscita.

5.6 Esempi

5.6.1 Esecuzione concorrente di un filtro IIR

Consideriamo ancora l'esempio di un filtro IIR e verifichiamo come viene eseguito in modo concorrente dopo la sua trasformazione ottenuta in accordo al procedimento illustrato nei precedenti paragrafi. L'inizializzazione scelta per algoritmo concorrente e' quella effettuata nella corrispondente implementazione sequenziale:

$$\mathbf{q}(0) = (0,0,0,0,1,1,0,0,0)$$

L'esecuzione concorrente del filtro puo' essere analizzata ricorrendo allo stesso diagramma usato per visualizzare l'automa associato all'algoritmo: nodi in ordinate, tempo discreto in ascisse e colore per distinguere se un nodo e' eseguito o meno. Verranno utilizzati differenti colori per evidenziare nodi che, in base all'algoritmo di ottimizzazione, siano assegnati a processori diversi;

La ripartizione su 2 processori fornisce allora il seguente diagramma di esecuzione[9] :

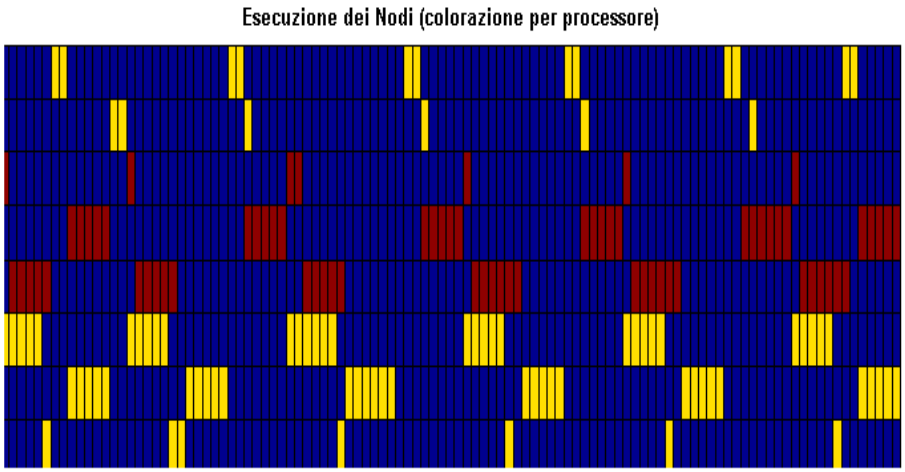


Figura 5-4 - Evoluzione dei nodi di un IIR eseguito in modalita' concorrente su 2 processori

E' evidente la non contemporaneita' dell'esecuzione di nodi appartenenti allo stesso processore (stesso colore). Se analizziamo ora i parametri temporali possiamo fare un confronto con quelli stimati dall'algoritmo di ottimizzazione. Tali parametri sono estratti da una esecuzione reale del programma concorrente per cui si possono discostare leggermente dai parametri ipotizzati, incluso il tempo di esecuzione sequenziale T_1. Prendendo dunque in considerazione una delle matrici di esecuzione del precedente diagramma, otteniamo la seguente schedulazione:

[9] In entrambi gli esempi, questo ed il seguente, i tempi di esecuzione dei singoli nodi sono stati emulati usando l'istruzione "delay" nativa del linguaggio ADA.

proc N. 1			4	4	4	4	4	3	3	5	5	5	5	5					
proc. N.2	1	1	2					6	6	6	6	6	6	8	7	7	7	7	7
tempo	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Figura 5-5 - Schedulazione ottenuta dalla simulazione concorrente di un IIR

Si nota innanzitutto che la schedulazione delle esecuzioni non e' identica a quella determinata dall'algoritmo di ottimizzazione, cosa del tutto normale in quanto l'esecuzione dei nodi su un dato processore non ha una sequenziazione prefissata ma e' asincrona per ipotesi. Cio' nonostante la casualita' di queste esecuzioni comporta mediamente una efficienza e_μ molto vicina a quella prevista.
La simulazione effettuata su T_{sim}=441 istanti discreti ha evidenziato un tempo destinato alla elaborazione pari a T_{exe}=649 istanti discreti complessivi per i due processori. Il parametro e_μ medio puo' allora essere stimato attraverso la:

$$e_\mu = T_{exe} / \mu T_{sim} = 73\ \%$$

Molto vicino al 75% previsto dall'algoritmo di ottimizzazione. Dal precedente diagramma di schedulazione si nota anche che il tempo di esecuzione del periodo e' salito a T_μ=19 anziche' T_μ=16 come previsto dall'algoritmo di ottimizzazione. Cio' e' dovuto essenzialmente alla approssimazione per difetto del tempo di esecuzione reale dei vari nodi che non tiene conto dell'ambiente di simulazione (sistema operativo dell'ambiente in background). Come si vede infatti il nodo 1, che ha tempo di esecuzione 1, impiega invece un tempo 2; analogamente il nodo 6 impiega un tempo 6 anziche' 5. Il parametro piu' attendibile per la stima della bonta' di ripartizione resta dunque il precedente e_μ .

5.6.2 Esecuzione concorrente di due FIR adattivi in cascata

Consideriamo ora l'esempio costituito da due FIR adattivi in cascata e verifichiamo la loro esecuzione concorrente su 2 processori in accordo alla ripartizione ottenuta dall'algoritmo di ottimizzazione. Anche in questo caso si suppone che lo stato iniziale coincida con quello imposto nella versione sequenziale:

$\mathbf{q}(0)=(0,0,0,0,1,0,1)$

Usando il metodo di colorazione descritto nel precedente esempio otteniamo:

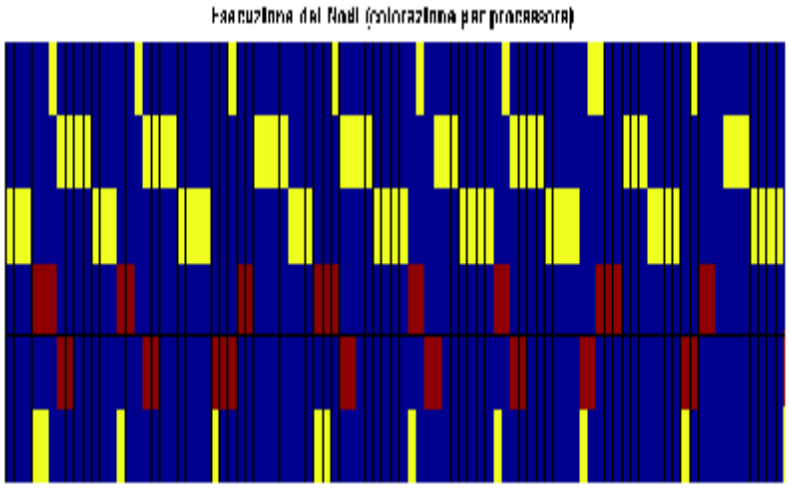


Figura 5-6 - Evoluzione dei nodi di due FIR adattivi in cascata eseguito in modalita' concorrente su 2 processori (tecnica di auto-schedulazione)

Anche in questo caso e' possibile confrontare l'esecuzione reale del DSPA concorrente con quella prevista dall'algoritmo di ottimizzazione in termini di parametri temporali. Consideriamo le seguenti due matrici di esecuzione esibite dal precedente diagramma.

proc N. 1					4	4	5	5		
proc. N.2	3	3	3	6	6	1	2	2	2	2
tempo	1	2	3	4	5	6	7	8	9	10

proc N. 1							5	5	5	4	4		
proc. N.2	3	3	3	3	6		1			2	2	2	2
tempo	1	2	3	4	5	6	7	8	9	10	11	12	13

Figura 5-7 - Schedulazioni ottenute dalla simulazione concorrente di due FIR adattivi in cascata

Si nota che l'aumento del tempo di esecuzione nella seconda schedulazione e' dovuto all'inversione della esecuzione dei nodi 4 e 5. Comunque, analogamente al precedente esempio, possiamo stimare l'efficienza raggiunta da questa implementazione concorrente misuando le grandezze di simulazione T_{sim}=259 e T_{exe}=344. Si ha[10]:

$$e_{\mu} = T_{exe} / \mu T_{sim} = 66\ \%$$

che denota una certa differenza rispetto al 75% previsto dall'algoritmo di ottimizzazione. Se si effettua pero' una schedulazione statica usando la tecnica Token-Ring, la situazione migliora notevolmente. Infatti, posto:

$\mathbf{q}(0)=(2,1,1,1,0,1,0)$

cioe' inizializzate le code allo stato finale del transitorio dell'automa associato al DSPA in esame, si ottiene il seguente diagramma di esecuzione:

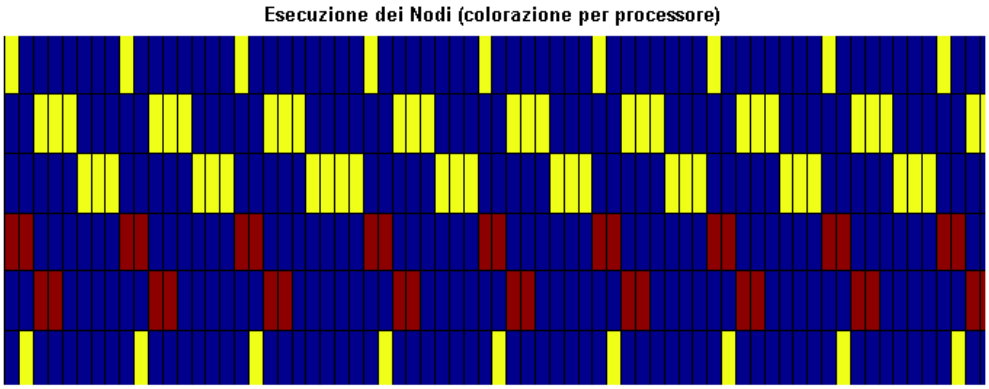


Figura 5-8 - Evoluzione dei nodi di due FIR adattivi in cascata eseguito in modalita' concorrente su 2 processori (tecnica di schedulazione Token-Ring)

[10] Anche in questo caso si possono fare considerazioni analoghe a quelle del precedente esempio riguardo l'approssimazione dei tempi di esecuzione. Il tempo T_{μ}=10 ottenuto nella prima schedulazione (che e' simile a quella individuata dall'algoritmo) e' piu' grande del tempo T_{μ}=8 previsto in quanto i nodi 6 e 2, appartenenti allo stesso processore, esibiscono un tempo di esecuzione di due istanti discreti anziche' 1.

Dal quale, estraendo una delle possibili matrici di esecuzione esibite da questa simulazione concorrente si ricava la schedulazione di seguito riportata:

proc N. 1				4	4	5	5	
proc. N. 2	3	3	3	1	6	2	2	2
tempo	1	2	3	4	5	6	7	8

che coincide esattamente con quella prevista dall'algoritmo di ottimizzazione. L'efficienza media calcolata con un post-processing dei risultati della simulazione concorrente fornisce: T_{sim}=263, T_{exe}=391 per cui:

$$e_{\mu} = T_{exe} / \mu T_{sim} = 74.3\ \%$$

Questo numero differisce dal 75% che fornisce l'ottimizzazione per i suddetti fenomeni di non uniformita' della durata dei tasks. Il meccanismo token-ring dunque permette l'implementazione di una schedulazione statica coincidente con quella ottima.

6. SVILUPPI FUTURI

Il lavoro svolto in questa tesi fornisce un metodo per la trasformazione di un algoritmo sequenziale in uno concorrente, sotto certe ipotesi sulla classe degli algoritmi trattabili. Oltre alla dimostrazione di correttezza del metodo ed alla sua applicazione a due esempi significativi, nella tesi sono stati sviluppati anche alcuni nuclei software fondamentali per risolvere il problema di ottimizzazione delle risorse e per la traslazione dell'algoritmo sequenziale su un sistema multiprocessore. Il flusso riportato in figura 5-1 illustra i passi necessari per effettuare tale trasformazione. La naturale prosecuzione del presente lavoro e' costituita dalla realizzazione di uno "shell" che inglobi i moduli software sviluppati, ne sviluppi altri quali il PARSER e il SEQ2CONC translator e che sia dotato di una interfaccia utente adeguata.
Inoltre dovra' essere esteso il nucleo di sistema operativo sviluppato per metterlo in grado di gestire le interconnessioni fisiche tra i processori. Questo aspetto dipende molto dal tipo di processore e dal protocollo hardware di comunicazione da essi messo a disposizione. Lo sviluppo di questa parte di sistema operativo si colloca dunque in modo naturale come lo sviluppo di un "package" dedicato al tipo di processore (Phisical Layer Package) ed interfacciato con quello esistente che invece prescinde da esso (Operating System Package).
La seguente figura 6-1 illustra la collocazione dei vari "tools" ancora da sviluppare. I sorgenti dei DSPA sequenziali si ipotizzano scritti in linguaggio ADA. Per ottenere il codice assembler verra' usato un linker commerciale. Il linker lavorera' su moduli sequenziali estratti dalla versione concorrente del DSPA nella quale un task e' identificato con un processore.
Ad essi si deve aggiungere lo sviluppo dei tre "profilers" introdotti nel precedente capitolo.
Un ulteriore sviluppo del lavoro svolto consiste infine nella generalizzazione del sistema proposto ai casi in cui i processori disponibili non siano omogenei, ovvero abbiano insiemi diversi di istruzioni. Cio' e' di particolare interesse quando si disponga di sistemi misti processore/FPGA[11] dove alcune funzioni sono basate su un hardware standard (quello del processore) ed altre su un hardware progettato di volta in volta (quello contenuto nelle FGPAs). L'hardware delle FPGA mette a disposizione, in un certo senso, un insieme di istruzioni personalizzabile per la realizzazione di funzioni che avrebbero un costo eccessivo se implementate tramite un programma assembler.

[11] FPGA sta per Field Programmable Gate Array ed indica dispositivi elettronici programmabili per realizzare un circuito anziche' un programma assembler come avviene per i microprocessori.

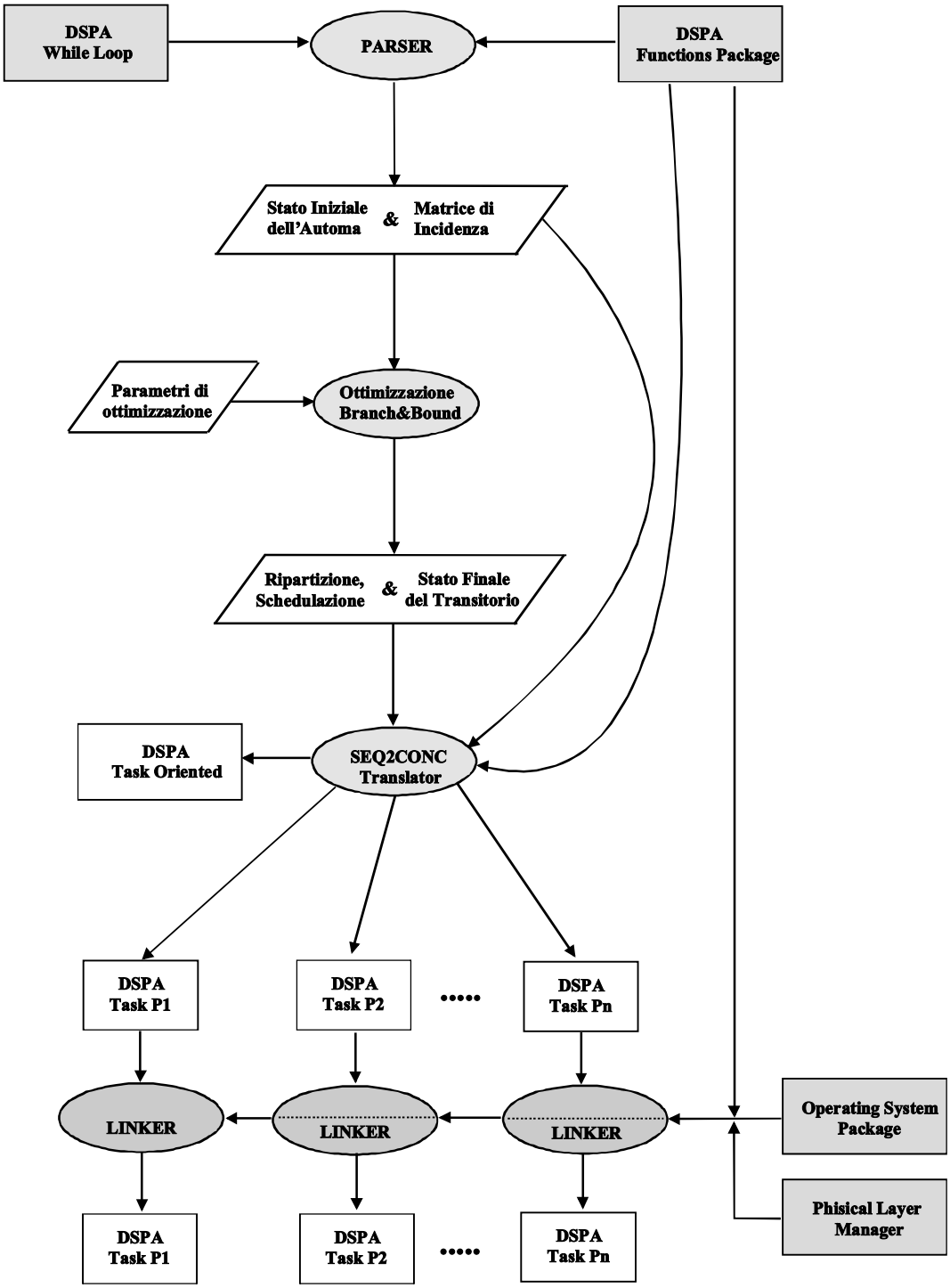


Figura 6-1 - Il sistema di sviluppo

7. BIBLIOGRAFIA

[M1] Introduction To Algorithms - Cormen / Leiserson / Rivest - McGraw-Hill;
[M2] Informatica Teorica - Ghezzi / Mandrioli - Citta' Studi Edizioni Torino;
[M3] Teoria Dei Sistemi - Rinaldi - Edizioni CLUP;
[M4] Algorithmic Graph Theory - J.A.McHugh - Prentice Hall;
[M5] Multi Microprocessor Systems - Y.Paker - Academic Press Inc (London);
[M6] Modern Signal Processing - T.Kailath - Proceedings Of The Arab School On Science And Technology;
[M7] Digital Signal Processing in VLSI - R.J.Higgins - Prentice Hall;
[M8] ADSP-2106X SHARC User's Manual - Analog Devices;

[A1] Performance Evaluation Of A Dataflow Architecture - D.Ghosal / L.N.Bhuyan - IEEE Transactions On Computer Vol.39 No.5 May 1990;
[A2] Speedup Versus Efficiency In Parallel Systems - D.L.Eager / J.Zahorjan / E.D.Lazowska - IEEE Transactions On Computer Vol.38 No.3 March 1989;
[A3] Static Scheduling Of Synchronous Data Flow Programs For Digital Signal Processing - E.A.Lee / D.G.Messerschmitt - IEEE Transactions On Computer Vol.36 No.1 January 1987;
[A4] Classifying Cellular Automata Automatically - A.Wuensche - Santa Fe Institute - 1998;
[A5] Universality And Complexity In Cellular Automata - S.Wolfram - 1984;
[A6] Cellular Automata Dynamics - R.Espericueta - Math Department Bakersfield College;
[A7] An Entropy Primer - Chris Hillman - Washington University;

---

[1] Models of Massive Parallelism - M. Garzon - Springer;
[2] Discrete Dynamical Networks And Their Attractor Basins - A.Wuensche - Santa Fe Institute - 1998;
[3] Matematica Discreta - Munarini / Zagalia Salvi - Citta' Studi Edizioni Torino;
[4] Teoria e progetto di algoritmi fondamentali - Ausiello / Marchetti / Spaccamela / Protasi - Franco Angeli Editore;
[5] Parallel sequencing and assembly line problem - T.C.Hu - Operational Research, vol. 9 pp. 841-848, 1961;
[6] Exploiting inter task dependencies for dynamic load balancing - W.Becker / G.Waldmann - IEEE proceedings of third international symposium on high performance distributed computing - 1994;
[7] ADA, Un linguaggio per la programmazione avanzata - Frosini / Lazzerini: - Addison Wesley;
[8] ADA, Una panoramica - DePaoli / Ghezzi / Mandrioli - Franco Angeli Editore;