Programmazione parallela: evoluzione e nuove sfide

Marco Danelutto

Sommario

Le recenti evoluzioni dei componenti di calcolo hanno messo a disposizione processori sempre più veloci e potenti che pongono nuove sfide dal punto di vista degli strumenti di programmazione. I classici strumenti utilizzati fino ad ora, infatti, richiedono un notevole sforzo di programmazione quando si vogliano sfruttare appieno tutte le caratteristiche dei nuovi componenti. In questo lavoro discutiamo alcuni recenti sviluppi nei modelli di programmazione parallela strutturata che permettono di sviluppare rapidamente applicazioni molto efficienti in grado di utilizzare tutte le nuove caratteristiche disponibili.

Abstract

Recent advances in digital components provided faster and more powerful computing devices that open new challenges to the programming tools. Classical programming tools require quite a programming effort in all those cases where the advanced features of these new components must be exploited. In this work, we discuss some recent advances in the field of the structured parallel programming models that support the rapid development of efficient parallel applications suitable to exploit all the new features available.

Keywords: Parallel programming, structured parallel programming models, algorithmic skeletons, design patterns, multi-core, GPU, FPGA.

1. Introduzione

1.1 Dal sequenziale al parallelo

Fino ai primi anni 2000 l'evoluzione dell'architettura dei componenti di calcolo ha messo a disposizione processori sempre più potenti in grado di eseguire sempre più velocemente gli stessi programmi utilizzati sui processori delle generazioni precedenti. Questo succedeva perché il *modello architetturale* del processore, ovvero l'insieme di caratteristiche che ne determinano le funzionalità, non cambiava mai. Si trattava di un modello nel quale un insieme di componenti hardware era in grado di eseguire istruzioni di un unico programma alla volta. Questo modello architetturale era caratterizzato dall'esistenza di unico dispositivo in grado di eseguire in continuazione il cosiddetto ciclo *fetch-decode-execute*, ovvero di prelievo dalla memoria dell'istruzione all'indirizzo indicato da un registro detto program counter (fetch), di decodifica dell'istruzione per capirne il tipo (decode) e infine di esecuzione dell'istruzione stessa, con il conseguente aggiornamento del program counter (execute).

Questo semplice modello architetturale è evoluto nel tempo introducendo forme di parallelismo (esecuzione contemporanea e concorrente di diversi passi di calcolo) nell'esecuzione delle singole istruzioni che compongono un programma:

- si è fatto in modo che l'esecuzione degli stadi del ciclo fetch-decode-execute fossero eseguite da unità indipendenti, in modo da sovrapporre l'esecuzione di un'istruzione con la decodifica dell'istruzione successiva e con il prelievo dalla memoria dell'istruzione ancora successiva, secondo un modello simile a quello della catena di montaggio, detto *pipeline*. Successivamente si sono suddivisi ulteriormente tutti gli stadi del pipeline fino ad arrivare a pipeline di esecuzione con decine di stadi, tutti in grado di lavorare contemporaneamente su fasi diverse di diverse istruzioni dello stesso programma;
- si sono replicate le risorse interne del processore in modo da poter eseguire più istruzioni dello stesso programma contemporaneamente (modello di architettura *superscalare*);
- si sono introdotte migliorie nel sottosistema di memoria (programma e dati) che hanno portato ad una riduzione notevole del tempo necessario per l'accesso alla singola istruzione o ai dati in memoria (introduzione di uno o più livelli di cache).

Sebbene vi siano limitazioni dovute alle dipendenze fra istruzioni diverse dello stesso programma che limitano l'efficacia complessiva sia del modello pipeline che del modello superscalare, in linea di principio l'adozione di un modello pipeline a k stadi porta ad uno *speedup* (incremento della velocità di esecuzione) pari a k, così come l'adozione di un modello superscalare a k vie (capace cioè di eseguire k istruzioni alla volta) porta ugualmente ad uno *speedup* pari a k. Le due tecniche possono essere combinate fra loro.

Le migliorie nel modello architetturale di cui stiamo parlando sono state rese possibili dalla sempre maggiore densità di componenti elettronici sul silicio. I

componenti di calcolo, infatti, sono realizzati "stampando" con veri e propri processi litografici le porte logiche con cui sono realizzati i componenti direttamente sulla superficie di un cristallo di silicio. La legge di Moore, formulata negli anni '60 da Gordon Moore, co-fondatore di Intel, e rispettata fino ai giorni nostri, stabilisce che il numero di componenti (porte logiche) su una certa superficie del wafer di silicio raddoppi ogni due anni, per via dei miglioramenti delle tecnologie di produzione [1,2]. Questo ha permesso di realizzare sempre nuovi modelli architetturali che occupano sempre maggiori aree sul chip, come per esempio i modelli pipeline/superscalare appena menzionati. Ma i miglioramenti nella tecnologia di produzione dei chip hanno portato anche ad una maggior miniaturizzazione dei componenti e dunque alla possibilità di lavorare a frequenze di clock sempre maggiori. Dal momento che il clock determina in modo direttamente proporzionale la velocità finale di esecuzione delle istruzioni, questo ha portato a dispositivi che non solo erano in grado di eseguire più velocemente istruzioni per via dei miglioramenti del modello architetturale, ma le eseguivano sempre più in fretta a parità di modello per via delle frequenze dei clock sempre più alte.

Questo trend ha subito un brusco rallentamento nei primi anni 2000 per due ragioni fondamentali:

- non si è stati più in grado di trovare miglioramenti del modello architetturale del processore che permettessero incrementi sostanziali nelle prestazioni dell'esecuzione delle istruzioni provenienti da un unico programma, o perlomeno non si è stati in grado di trovare miglioramenti architetturali che portassero ad un incremento delle prestazioni proporzionale all'area di silicio occupata per introdurre tali miglioramenti. In altre parole, l'introduzione di nuove caratteristiche che potevano portare ad un incremento delle prestazioni di un X% a fronte di un corrispondente incremento dell'area di silicio necessaria del 10X% [3];
- l'utilizzo di frequenze di clock sempre più alte per migliorare le prestazioni delle CPU (Central Processing Unit) ha introdotto notevoli problemi di consumo e conseguentemente di dissipazione termica. I nuovi chip con clock che viaggiavano ormai speditamente verso i 5-6Ghz richiedevano tecnologie di raffreddamento complicate e costose, sia in termini di costruzione che di esercizio (ulteriori Watt richiesti non per far funzionare il processore ma per dissipare il calore prodotto).

Dunque a cavallo fra il secolo scorso e quello attuale i produttori si sono trovati di fronte a una situazione che vedeva da una parte la possibilità di realizzare componenti di calcolo sempre più complessi su un singolo chip e dall'altre parte la difficoltà oggettiva a realizzare nuovi componenti in grado di sfruttare queste potenzialità nella realizzazione di processori in grado di lavorare sempre più velocemente su singolo programma alla volta.

Il riconoscimento di questa situazione ha portato ad un cambio radicale di strategia:

non si è più cercato di sfruttare le maggiori capacità relative alla realizzazione di componenti hardware per realizzare processori in grado di eseguire più velocemente le istruzioni di un singolo programma, ma si sono sfruttate queste capacità per realizzare sullo stesso chip più core cioè più componenti che, operando in modo combinato ed accedendo ad un sottosistema di memoria comune, sono ognuno in grado di eseguire istruzioni di un programma diverso.

Questa svolta è radicale non perché cambia il tipo di componente, ma perché si passa da una situazione in cui un programmatore può concepire il suo codice come una serie di attività (istruzioni) da eseguire sequenzialmente, una dopo l'altra ad una situazione in cui il programmatore *deve* concepire il proprio programma come *un insieme di attività* in grado di essere eseguite in modo autonomo da core diversi, *coordinate esplicitamente* per far sì che la loro esecuzione porti al calcolo del risultato finale richiesto dall'applicazione.

All'inizio dell'era multi-core questo cambiamento non fu percepito in modo chiaro: i core disponibili, e dunque le attività necessarie per utilizzarli al meglio, furono dell'ordine delle unità e fu relativamente facile utilizzarli per eseguire in parallelo codice del sistema operativo e codice della/delle applicazioni utente. In questo caso, non vi era necessità di modificare né il codice del sistema operativo né quello delle applicazioni stesse. Subito dopo però, gli avanzamenti nella tecnologia di produzione, che continua a progredire secondo la legge di Moore "modificata" (si raddoppia il numero di core sullo stesso chip circa ogni anno), hanno permesso di realizzare decine e poi centinaia di core sullo stesso chip. Con simili numeri la necessità di programmi *paralleli*, ovvero costruiti in modo da implementare un certo numero di attività parallele da eseguire sui vari core a disposizione, è diventata assoluta ed urgente.

Ma vi è un ulteriore fattore che ha reso e rende tale necessità urgente. Per riuscire a realizzare più core sullo stesso chip, i produttori hanno cominciato ad eliminare dai singoli core tutte quelle caratteristiche che introducevano miglioramenti minimali nei tempi di esecuzione a fronte di un'occupazione significativa in termini di area sul silicio. Dunque prendendo un programma che raggiunge una certa velocità su un processore non multi-core ed eseguendolo su un singolo core dei nuovi processori multi-core si otterrebbero prestazioni (leggermente) inferiori. Questo interrompe di fatto il trend, in essere fino ai primi anni del secolo, che vedeva la possibilità di far girare il proprio codice sempre più velocemente semplicemente comprando l'ultimo modello di processore.

Sebbene questo tipo di effetto si possa verificare semplicemente fra generazioni diverse della stessa serie di processori (per esempio fra processori multi-core Intel della serie Nehalem e della successiva serie Sandy Bridge), l'effetto è particolarmente marcato nei cosiddetti *general purpose many core*. Questi dispositivi sono chip che raggruppano da decine fino a centinaia di core, di capacità più limitate rispetto ai core delle CPU tradizionali, interconnessi con reti

ad anello o a mesh in grado di eseguire ciascuno un flusso di controllo programmato con strumenti di programmazione tradizionali. Un esempio di questa classe di dispositivi è l'Intel® XEON PHI, dotato di 60 core, ciascuno 4-way hyperthreading (cioè in grado di supportare l'esecuzione contemporanea di quattro thread distinti) e dotato di unità vettoriale da 512 bit, interconnessi secondo una topologia ad anello e dotati di cache di secondo livello condivisa e coerente [4]. I singoli core del PHI sono in grado di eseguire codice x86, ma, sia per via della semplificazione del progetto architetturale del core legata alla razionalizzazione dello spazio sul chip, sia per via della frequenza di clock piuttosto bassa (intorno al GHz), l'esecuzione del codice sul singolo core risulta sensibilmente più lenta dell'esecuzione dello stesso codice su un core di una CPU tradizionale della stessa generazione. Questo non è altro che un ulteriore fattore che spinge nella direzione della ricerca di mezzi di programmazione parallela sempre più efficienti nell'utilizzo concorrente di tutte le risorse a disposizione sul dispositivo.

1.2 Dall'omogeneo all'eterogeneo

Il passaggio dal supporto all'esecuzione di un singolo flusso di controllo a quello di flussi di controllo multipli non è l'unico cambiamento significativo dettato dagli avanzamenti della tecnologia di produzione dei componenti. Vi sono almeno altri due cambiamenti che hanno avuto un grosso impatto sullo scenario attuale:

- la disponibilità di GPU (Graphics Processing Units), ovvero di acceleratori in grado di eseguire molto velocemente tutte le operazioni tipiche della grafica computazionale. Le GPU utilizzano un gran numero di componenti (migliaia su una singola GPU) a loro volta capaci di eseguire ciascuno semplici operazioni sul singolo pixel di un'immagine [5]. Vista la particolare efficacia delle GPU nel calcolo di quelle computazioni che richiedono di eseguire una singola funzione su tutti gli elementi di una matrice o di un vettore, le GPU si sono presto evolute in GP-GPU (General Purpose GPU) in grado di accelerare notevolmente l'esecuzione di calcoli *data parallel*, cioè calcoli in cui il risultato della computazione si ottiene combinando i risultati da sotto computazioni effettuate su partizioni del dato in input;
- la disponibilità di componenti di tipo completamente diverso, detti FPGA (Field Programmable Gate Array [6]) nei quali viene messa a disposizione una matrice di *celle* ognuna della quali può implementare o una semplice rete combinatoria (di pochi ingressi da 1bit, normalmente qualche unità e che produce un'uscita di un singolo bit) o un piccolo registro capace di memorizzare un singolo bit. La funzionalità delle singole celle, così come i collegamenti fra le diverse celle nella matrice, possono essere completamente riconfigurati "scrivendo" valori opportuni nel controller della FPGA. Oltre a contenere decine o centinaia di migliaia di celle, le FPGA più moderne contengono anche un certo numero di piccoli blocchi di memoria e di DSP (Digital Signal Processors, dispositivi in grado di compiere semplici operazioni aritmetiche in virgola mobile) sparsi fra le celle e configurabili per interagire con le celle stesse. I blocchi di memoria possono essere utilizzati per mantenere risultati intermedi delle

computazioni senza doverli necessariamente trasferire su moduli di memoria esterni. I DSP permettono di svolgere velocemente operazioni in virgola mobile senza dover impegnare grosse quantità di celle riconfigurabili. Utilizzando FPGA si possono "programmare" componenti hardware che, pur a velocità nettamente inferiori a quelle che si possono ottenere con un processo di produzione di chip classico, calcolano molto velocemente funzioni particolari. Per esempio, le FPGA vengono utilizzate per l'accelerazione dell'esecuzione di algoritmi crittografici particolarmente pesanti se eseguiti sulla CPU. Il vantaggio dell'uso di questi dispositivi è che se in momenti diversi dobbiamo eseguire calcoli diversi, l'FPGA può essere riprogrammata *on-the-fly* in tempi che sono dell'ordine dei micro o millisecondi.

Entrambi questi tipi di dispositivi sono stati inizialmente messi a disposizione come componenti che si affacciano sul bus del processore, e dunque interagiscono con la CPU con i meccanismi tipici delle unità di ingresso/uscita. La loro programmazione richiede ancora una volta l'orchestrazione esplicita di un insieme di flussi di controllo diversi come nel caso dei multi-core. Ma mentre i multi-core mettono a disposizione del programmatore un certo numero di risorse di calcolo completamente omogenee fra di loro e quindi in grado di eseguire tutte lo stesso insieme di compiti appartenenti ad un certo programma, FPGA e GP-GPU mettono a disposizione del programmatore modalità molto efficienti per eseguire tipi ben particolari di computazioni. In aggiunta a questo, mentre per programmare i core di un'architettura multi-core si possono utilizzare linguaggi e strumenti di programmazione classici, per la programmazione di GP-GPU ed FPGA è necessario adottare linguaggi e strumenti di programmazione *ad hoc*, spesso di non facile utilizzo per i programmatori che si sono formati utilizzando ambienti di programmazione classici.

Inoltre, la tendenza più recente è quella di integrare direttamente GP-GPU e FPGA sullo stesso chip del processore multi-core mettendo a disposizione i cosiddetti *system on chip* (SOC). Le GP-GPU sono già parte integrante dei processori dei maggiori produttori mondiali, sia di quelli dedicati al mercato consumer che di quelli dedicato al mercato high end e HPC (High Performance Computing). Diversi produttori stanno già proponendo modelli con FPGA integrate sul processore, da utilizzare per l'accelerazione di quei compiti per i quali non esiste un'implementazione efficiente o abbastanza veloce su CPU o GP-GPU. Un esempio di SOC che integra multi-core e FPGA è il sistema ZYNQ della Xilinx®, che integra sullo stesso chip due core ARM®, una FPGA che può contenere da 28K a 444K celle riconfigurabili e tutta la logica di interconnessioni e di supporto necessaria.

2. Lo scenario software

Le CPU multi-core rendono dunque lo scenario che si presenta al programmatore di applicazioni uno scenario decisamente parallelo. Applicazioni che sfruttino efficientemente un'architettura multi core devono necessariamente mettere a disposizione più attività parallele (flussi di controllo) per permettere di mantenere occupati i diversi core disponibili. Le GP-GPU rendono questo

scenario ulteriormente parallelo, visto il numero di core presenti sulle GP-GPU, ma anziché richiedere generiche attività parallele, richiedono attività che calcolano, in parallelo, lo stesso tipo di funzione su tutti i componenti di un certo dato. GP-GPU e FPGA richiedendo entrambe tecniche piuttosto particolari di programmazione il che rende lo scenario decisamente eterogeneo. Ma quali sono gli strumenti a disposizione del programmatore per realizzare applicazioni in grado di sfruttare al meglio le caratteristiche di questi nuovi dispositivi di calcolo?

Per realizzare applicazioni su queste architetture sono disponibili svariati tipi di modelli e strumenti di programmazione, ed in particolare ogni tipo di architettura ha i propri strumenti.

Su CPU multi-core gli standard di fatto sono OpenMP [7,8], per i sistemi multi core a memoria condivisa e MPI [9], per le reti/cluster di calcolatori. OpenMP è un modello di programmazione *ad ambiente globale*, ovvero nel quale si assume che tutte le attività che avvengono in parallelo nel corso dell'esecuzione di un programma abbiano accesso ad un livello di memoria condivisa. MPI invece è un modello di programmazione a *scambio di messaggi* (o ad *ambiente locale*), ovvero nel quale si assume che tutte le attività parallele all'interno del programma abbiano accesso solo ad una propria memoria locale ed eventuali dati da condividere debbano essere esplicitamente spediti alle attività che li condividono. Da un altro punto di vista, OpenMP richiede la sola introduzione nel codice sequenziale di *annotazioni* che creano e orchestrano le attività parallele. La compilazione tradizionale del codice OpenMP ignora le annotazioni ed esegue dunque il codice sequenzialmente, senza che il programmatore debba modificare alcunché nel sorgente. Per contro, MPI richiede l'introduzione di un discreta quantità di codice vero e proprio nell'applicazione, secondo un modello SPMD (Single Program Multiple Data [10]). Sostanzialmente il programmatore deve fornire esplicitamente il codice per tutte le attività che verranno eseguite in parallelo, comprensivo delle istruzioni per la comunicazione e sincronizzazione delle informazioni da condividere fra le varie attività parallele.

Su GP-GPU abbiamo OpenCL [11] e CUDA [12] (quest'ultimo disponibile solo sulle GP-GPU nVidia) anche se recentemente stanno prendendo piede anche standard leggermente di più alto livello come AAC. Entrambi i modelli espongono al programmatore un livello di astrazione molto basso. Concetti come la gestione dell'allocazione dei dati in memoria, che sono completamente nascosti al programmatore di applicazioni sequenziali da decenni, qui tornano ad essere sotto la responsabilità del programmatore delle applicazioni e determinano in modo significativo le performance che si riescono ad ottenere dalla GPU. Lo standard AAC [13] cerca di ridurre lo sforzo di programmazione richiesto, utilizzando annotazioni nello stile utilizzato da OpenMP per le CPU, ma ancora al prezzo di prestazioni sensibilmente inferiori rispetto a quelle ottenibili dall'uso diretto di OpenCL o CUDA.

Infine, sulle FPGA lo standard di fatto è rappresentato dall'utilizzo di linguaggi come VHDL [14] e Verilog [15], che sono veri e propri *linguaggi per la descrizione dell'hardware* (HDL) e che richiedono una profonda conoscenza

delle tecnologie di realizzazione delle FPGA per poter essere utilizzati efficientemente. In Verilog o VHDL si descrivono una serie di componenti hardware e collegamenti fra i componenti in termini delle funzionalità elementari messe a disposizione dalle celle delle FPGA e successivamente un compilatore provvede a trasformare queste descrizioni nei file di configurazione da caricare nelle memorie di controllo delle FPGA. Il procedimento di compilazione, fra l'altro, è estremamente complicato e richiede la soluzione di diversi problemi molto complessi e non è raro che richieda ore per il completamento. Recentemente, si stanno muovendo i primi passi verso l'adozione, anche sulle FPGA, di OpenCL (Altera, CodePlay) così come di modelli di programmazione più evoluti e ad alto livello come Chisel [16], che introduce la tecnologia Object Oriented per descrivere componenti hardware e successivamente compila il codice a oggetti in Verilog o VDHL.

Al di là degli standard *di fatto* che vengono utilizzati sui diversi tipi di componenti, vanno citati tutta una serie di ambienti di programmazione in grado di supportare lo sviluppo di applicazioni parallele, in grado di utilizzare al meglio almeno le architetture multi e many core di tipo general purpose e, in qualche caso, anche le architetture con GPU. Questi modelli di programmazione sono vari e offrono livelli di astrazione e meccanismi per le computazioni parallele diversi al programmatore di applicazioni. Per esempio:

- i modelli basati su *PGAS* (Partitioned Global Address Space [17]) espongono al programmatore uno spazio di memoria comune partizionato fra i diversi esecutori dei flussi di controllo. I linguaggi di programmazione Unified parallel C [44], Fortress [45], X10 [46] e Global Array [44] fanno tutti parte di questa categoria;
- i modelli basati su *fork/join* (operazione di creazione di un flusso di controllo indipendente e di attesa esplicita della sua terminazione) espongono al programmatore la possibilità di lanciare computazioni parallele arbitrarie ma indipendenti e di attenderne la terminazione quando i loro risultati risultino necessari per il prosieguo dell'applicazione. Cilk [18] e la sua variante Cilk++ [19] sono linguaggi che fanno parte di questa categoria, così come Erlang [20]. Il già citato OpenMP, Intel TBB [25] e la Task Parallel Library di Microsoft [26] sono ambienti di programmazione che di fatto adottano questo modello;
- i modelli basati su *mapreduce* (esecuzione di una funzione su tutti gli elementi di una grossa collezione di dati e calcolo di una seconda funzione, binaria, associativa e commutativa su tutti i risultati prodotti dalla prima funzione) danno al programmatore la possibilità di esprimere in forma molto sintetica una vasta classe di funzioni su moli di dati di grosse dimensioni, dal calcolo delle occorrenze di una stringa in un insieme di testi al ranking delle pagine utilizzato da Google nei suoi motori di ricerca. Il framework di ricerca e classificazione delle informazioni di Google [21] e il framework Hadoop [22], che ne rappresenta la versione open source, fanno parte dei questa categoria;

- i modelli basati su *BSP* (Bulk synchronous parallel) permettono ai programmatori di programmare attività parallele organizzate in *superstep*, passi in cui per un certo tempo un insieme di flussi di controllo procedono in maniera completamente indipendente alla fine dei quali si ha una fase di scambio di informazioni ottimizzata e bloccante fra tutti i flussi di controllo. A questa categoria appartengono Pregel [23] e Python BSP [24].

Ognuno di questi modelli di programmazione (e ce ne sono molti altri, in realtà, a partire da quelli object oriented fino a quelli puramente funzionali) offrono modalità diverse per programmare applicazioni parallele, ma tutti però condividono una caratteristica: mettono a disposizione del programmatore o strumenti di basso livello che il programmatore deve utilizzare per implementare le astrazioni di più alto livello di cui ha bisogno (è il caso di TBB, per esempio), o meccanismi di più alto livello ma adatti a modellare solo certe forme di calcolo parallelo (è il caso del modello *mapreduce*, per esempio) e dunque difficili da utilizzare qualora si vogliano utilizzare forme di calcolo parallelo diverse.

3. Pattern per la programmazione parallela

Fin dagli anni '90 ci si è resi conto che molte applicazioni parallele, programmate utilizzando framework di programmazione diversi, in realtà utilizzavano meccanismi di computazione parallela simili nel modo in cui le diverse attività parallele venivano create e fatte interagire fra di loro.

Ad esempio, in molti casi le computazioni parallele sono organizzate in *stadi* secondo il modello *pipeline* già menzionato a proposito dei modelli di processore nella sezione 1.1 [27]. Le applicazioni di video processing utilizzano spesso stadi in cascata per filtrare e correggere il rendering dei singoli frame di un filmato. Alcune applicazioni matematiche utilizzano lo stesso schema di calcolo con stadi che calcolano risultati parziali sempre più vicini al risultato finale dell'applicazione. In questo caso, il *pattern* di calcolo parallelo si può rappresentare come in Fig. 1 (parte sinistra) con una serie di attività concorrenti S_1, … S_m dove l'attività S_i riceve i propri dati di input dall'attività S_{i-1} e invia i propri risultati all'attività S_{i+1}. L'attività S_1 legge i dati in input dall'esterno e l'attività S_m spedisce all'esterno i propri risultati. Il pattern è utile solo nel caso in cui si debbano calcolare una serie di risultati corrispondenti a una seri di dati in input: serie di frame per il video processing, serie di dati strutturati (matrici o vettori) per le applicazioni matematiche. In questo caso, la computazione dei vari stadi S_i può avvenire in parallelo su dati relativi a ingressi diversi.

Un altro pattern tipico è quello detto *master-worker* ([28] detto anche *task farm* o semplicemente *farm*, vedi Fig. 1 (parte destra). In questo caso un'attività *master* deriva una serie di dati (detti *task*) sui quali va effettuata una certa computazione *f* o da una serie di dati in ingresso o da un unico dato complesso, e distribuisce l'esecuzione dei vari task a un insieme di *m* attività *worker* in grado di eseguire il calcolo di *f* sui task e restituire al master i risultati. Il calcolo della moltiplicazione di due matrici può essere organizzato secondo uno schema master-worker in cui il master manda ai worker coppie formate da righe della prima matrice e colonne della seconda e i worker calcolano il prodotto vettore-vettore restituendo un singolo elemento della matrice risultato al master.

In un ambito completamente diverso, un pattern master-worker può essere utilizzato per svolgere computazioni di reti neurali, con il master che assegna ai worker sotto partizioni della rete neurale per la computazione e riceve dai worker i valori calcolati da propagare verso altre partizioni o da utilizzare per iterazioni successive del calcolo della rete neurale.

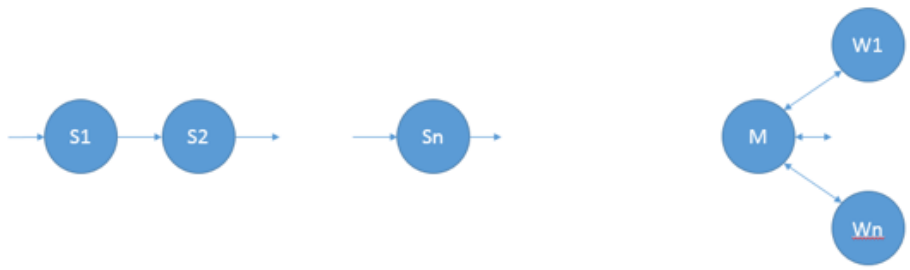

Figura 1
Pattern pipeline (a sinistra) e master/worker (a destra).
I cerchi indicano attività concorrenti e le frecce le comunicazioni fra attività.

3.1 Algorithmic skeleton

Riconosciuto questo fatto, si è cercato di implementare ambienti di programmazione che mettessero a disposizione astrazioni in grado di modellare completamente questi pattern:

- in *forma parametrica*, per poter specializzare quale computazione calcolare, per esempio specificando cosa calcolare negli stadi di un pipeline o nei worker del master-worker;
- realizzati in modo *efficiente*, sfruttando al meglio i meccanismi di supporto a basso livello;
- in modo che possano essere liberamente composti per ottenere forme di parallelismo più complesse (per esempio pipeline con stadi paralleli organizzati secondo pattern master worker)
- e che mettessero a disposizione del programmatore di applicazioni tutti i pattern più comunemente utilizzati per programmare applicazioni parallele.

Ambienti simili sono stati sviluppati fin dai primi anni ’90 e sono noti come ambienti basati su *algorithmic skeleton* dal nome dato ai costrutti che implementano i pattern paralleli da Murray Cole nella sua tesi di dottorato [29] e nel successivo “manifesto” della programmazione parallela strutturata [30].

Sviluppare applicazioni parallele quando sia disponibile uno di questi ambienti vuol dire essenzialmente seguire un procedimento in fasi (vedi Fig. 2):

- capire quali pattern si vuole/devono utilizzare per modellare il parallelismo all'interno dell'applicazione
- istanziare gli algorithmic skeleton corrispondenti con il codice necessario ad implementarne le parti sequenziali (stadi di un pipeline o worker di un farm)
- verificare i risultati dell'esecuzione dell'applicazione ed eventualmente ripetere il processo utilizzando skeleton o composizioni di skeleton diverse fino a che non si raggiungono risultati ragionevoli in termini di prestazione (performance tuning).

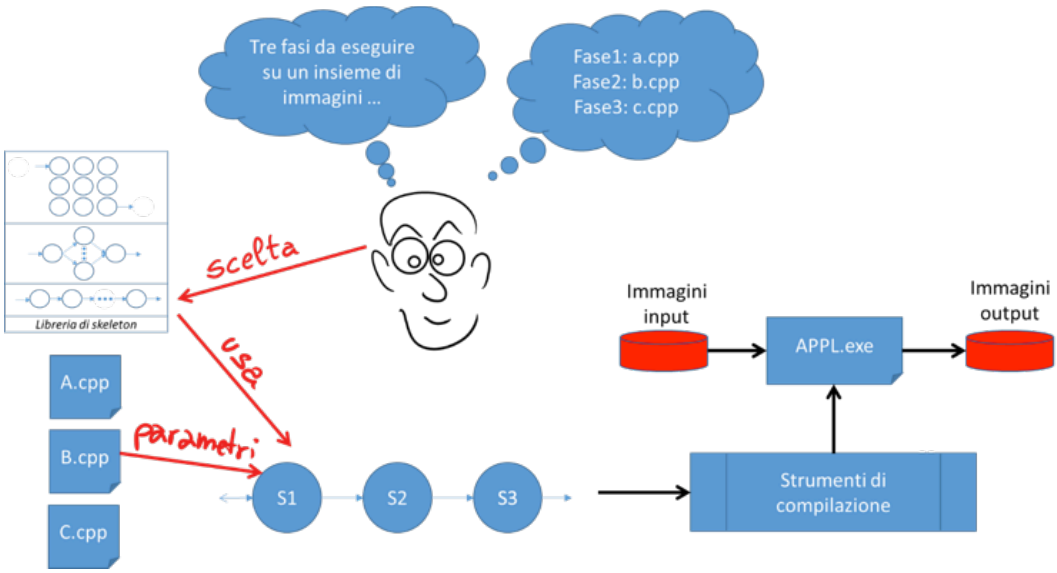

Figura 2
Realizzazione di applicazioni parallele utilizzando un framework basato su algorithmic skeleton

Questo procedimento è molto diverso da quello utilizzato quando si programmano applicazioni parallele utilizzando strumenti di programmazione tradizionali e di basso livello (thread con primitive di sincronizzazione, processi con primitive di comunicazione e sincronizzazione, etc.). In questo caso, infatti, oltre ad identificare le forme di parallelismo necessarie per implementare la propria applicazione e a scrivere il codice per la *business logic* dell'applicazione, occorre anche implementare tutto ciò che serve per mettere a disposizione queste forme di parallelismo utilizzando i meccanismi di supporto a basso livello disponibili (vedi Fig.3).

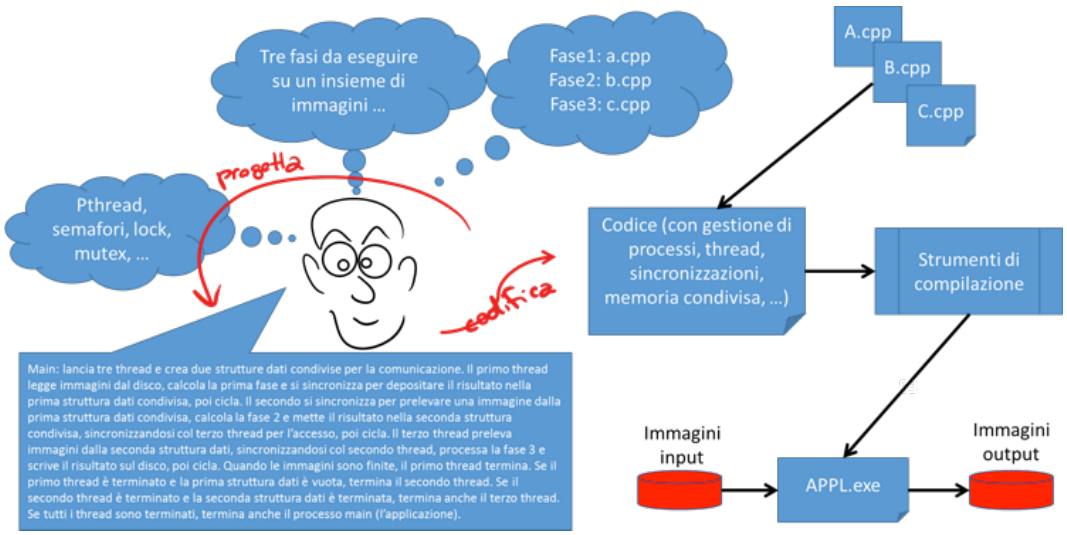

Figura 3
Realizzazione di un'applicazione parallela utilizzando framework non strutturati.

L'utilizzo di composizioni di skeleton per implementare applicazioni parallele e la conseguente conoscenza esplicita, a livello degli strumenti di compilazione e di run time, di ogni dettaglio relativo al parallelismo, ha diverse conseguenze:

- al programmatore dell'applicazione non è più richiesta alcuna conoscenza riguardo le caratteristiche dell'architettura su cui andrà ad eseguire l'applicazione né tantomeno dei meccanismi e delle tecniche necessarie per implementare efficientemente pattern di calcolo parallelo. Queste conoscenze saranno invece richieste a chi implementa gli skeleton. Si realizza dunque una completa e vantaggiosa *separazione delle responsabilità*: il programmatore "applicativo" è responsabile della correttezza del codice che calcola il risultato dell'applicazione. Il programmatore "di sistema" è responsabile della correttezza ed efficienza del codice che implementa i pattern paralleli al programmatore delle applicazioni mediante gli algorithmic skeleton. Entrambi sfruttano al meglio solo le loro specifiche capacità e conoscenze.
- per ognuno degli skeleton utilizzati dal programmatore applicativo si possono scegliere le implementazioni a disposizione per gli strumenti di supporto a basso livello e per l'architettura su cui si farà girare l'applicazione. Portare un'applicazione su un'altra architettura richiederà dunque una semplice ricompilazione del sorgente che utilizza gli skeleton.
- si possono adottare, a livello di compilatore e run time, tutta una serie di ottimizzazioni relative all'implementazione efficiente di forme di computazione parallele. Tali ottimizzazioni includono tecniche automatiche di trasformazione di composizioni di skeleton in altre composizioni funzionalmente equivalenti ma con diverse caratteristiche di performance ed efficienza nell'utilizzo delle risorse, che altrimenti sarebbero ancora a carico del programmatore applicativo.

- si possono infine utilizzare, in modo automatico, eventuali acceleratori disponibili (per esempio GPU) per realizzare in maniera più veloce o meno costosa in termini di potenza consumata tutti quegli skeleton per i quali esistono implementazioni sia per CPU che per gli acceleratori a disposizione [42, 43].

3.2 Parallel design pattern

L'idea di algorithmic skeleton ha portato allo sviluppo di un certo numero di ambienti di programmazione fra i quali ricordiamo gli ambienti FastFlow [32,33], Muesli [34], SKEPU [35], Sketo [36] e OSL [37]. Più o meno negli stessi anni però, in una comunità quasi completamente disgiunta rispetto alla comunità dell'high performance computing che ha dato vita agli algorithmic skeleton, ovvero nella comunità che si occupa di software engineering, è stato sviluppato un concetto molto simile a quello degli algorithmic skeleton: quello di *parallel design pattern*. Un *design pattern* descrive una metodologia di programmazione (per lo più object oriented) tipica ed utilizzata in molti contesti diversi, dandole un nome, discutendone le caratteristiche, gli usi tipici e le tipiche modalità di implementazione. Ad esempio, il design pattern chiamato *proxy* definisce la metodologia di programmazione in cui si usa un oggetto per filtrare le richieste dirette ad un altro oggetto. E' la metodologia utilizzata per esempio sui server web per far sì che le richieste dirette ad un server vengano invece processate da un altro server che le filtra e le inoltra ai server cui sono veramente dirette, tipicamente per implementare una qualche forma di controllo degli accessi. Un *parallel design pattern* definisce, analogamente, una metodologia di programmazione parallela, dandole un nome, discutendone le caratteristiche, etc. Tipici esempi di parallel design pattern (a diversi livelli di astrazione) sono il *dividi&conquista*, il *master/worker*, il *pipeline*.

A differenza degli algorithmic skeleton, i parallel design pattern *non mettono a disposizione del programmatore astrazioni tipiche del modello di programmazione* (chiamate di libreria, oggetti, funzioni di ordine superiore) che implementano direttamente il pattern parallelo. Piuttosto i parallel design pattern rappresentano delle vere e proprie *ricette* da utilizzare per implementare i pattern paralleli nelle applicazioni.

Ciò nonostante, i parallel design pattern hanno riscosso un successo anche maggiore di quanto non abbiano riscosso gli algorithmic skeleton. Gli esperti americani che hanno realizzato il celebre Berkeley report [38] hanno chiaramente identificato i parallel design pattern come il mezzo più adatto a formare una nuova classe di programmatori capaci di sviluppare rapidamente applicazioni parallele efficaci per le architetture parallele del nuovo millennio, auspicando adozione di approcci sintetizzabili come:

> architecting parallel software with design patterns, not just parallel programming languages.

Inoltre, alcune grandi industrie hanno abbracciato il modello dei parallel design pattern, mettendo a disposizione librerie di componenti parallele utilizzabili per implementare parallel design pattern e dunque, in ultima istanza, di realizzare rapidamente efficienti applicazioni parallele. E' il caso dei Parallel design pattern di Intel® costruiti sopra alla loro libreria TBB o della Task Parallel Library di Microsoft, che lavora su C#.

3.3 Un buon risultato è sempre frutto di un buon compromesso

Parallel design pattern e algorithmic skeleton (di cui vediamo le principali caratteristiche nel riquadro 1) hanno entrambi vantaggi e svantaggi:

- i parallel design pattern aiutano il programmatore a capire esattamente cosa succede quando si implementa una certa forma di parallelismo, ma non mettono a disposizione un modo diretto per sperimentare pattern diversi in una certa applicazione, visto che di fatto non mettono a disposizione delle vere e proprie implementazioni;
- gli algorithmic skeleton, per contro, permettono di realizzare rapidamente un'applicazione quando esistano uno o più skeleton (o composizioni di skeleton) che implementano il pattern parallelo che il programmatore vuole implementare.

E' evidente che un ambiente di programmazione che possa presentare almeno in parte i vantaggi di entrambi gli approcci senza pagarne tutti gli svantaggi costituirebbe un grosso risultato in termini di modelli di programmazione per architetture parallele. In particolare, sarebbe molto produttivo avere un modello in cui il programmatore possa ragionare in termini di parallel design patterns in fase di progetto della propria applicazione per poi utilizzare algorithmic skeleton, anziché librerie/linguaggi a più basso livello di astrazione, per la realizzazione del pattern (o della composizione di pattern) scelta.

Recentemente, sono stati finanziati diversi progetti nei quali si è cercato di adottare questo approccio, fra i quali ricordiamo alcuni recenti progetti europei *ParaPhrase* (2012-2105 [40]) e *REPARA* (2013-2016 [41]), entrambi finanziati nel programma quadro FP7, e *RePhrase* (2015-2018), finanziato nel programma H2020. Tutti e tre questi progetti adottano in qualche modo un approccio combinato *parallel design patterns + algorithmic skeletons*, con diverse sfumature e obiettivi.

ParaPhrase ha avuto come obiettivo lo sviluppo di una metodologia di programmazione e dei relativi strumenti di programmazione nella quale si utilizzano i parallel pattern per modellare l'esplicitazione del parallelismo e si utilizzano tecniche semiautomatiche per introdurre e modificare (composizioni di) skeleton in codice sequenziale C++ o Erlang al fine di parallelizzare l'applicazione o di ottimizzarne l'esecuzione parallela. REPARA si propone di mettere a disposizione strumenti automatici per introdurre parallelismo (sotto forma di istanze opportune di algorithmic skeleton) in codice C++ sequenziale arricchito con annotazioni che identificano le possibilità di trasformazione del codice secondo parallel design pattern in modo da migliorarne sia i tempi di esecuzione che i consumi energetici (l'obiettivo è raddoppiare la velocità di esecuzione dimezzando il consumo energetico). In entrambi i progetti le

architetture target prese in considerazione includono CPU multi-core con acceleratori GPU e many core. REPARA considera anche la possibilità di utilizzare schede con FPGA. Sia per la parte C++ di ParaPhrase che per tutto il progetto REPARA, l'ambiente a skeleton utilizzato come run time è FastFlow [32] (vedi riquadro 2). RePhrase, infine, intende utilizzare parallel design pattern per l'orchestrazione delle attività parallele dell'applicazione e strumenti di programmazione di tipo classico (OpenMP, MPI, TBB) o basati su algorithmic skeleton (FastFLow) per l'implementazione dei pattern individuati nell'applicazione. All'interno del progetto verranno utilizzate massicciamente tecniche tipiche del software engineering sia per l'ottimizzazione dell'utilizzo dei design pattern che della loro implementazione in termini dei componenti messi a disposizione dagli ambienti a basso livello o di tipo algorithmic skeleton.

Alcuni risultati già disponibili dal progetto *ParaPhrase* hanno dimostrato che l'uso combinato dei pattern (per ragionare ad alto livello sulle forme di parallelismo da utilizzare nelle applicazioni) e degli algorithmic skeleton (per fornire al programmatore di applicazioni strumenti adatti ad implementare i parallel design pattern individuati) permettono la realizzazione di applicazioni parallele di efficienza comparabile o leggermente superiore rispetto ad analoghe applicazioni programmate con sistemi tradizionali ma con una drastica riduzione del tempo necessario a sviluppare e mettere a punto l'applicazione (ore invece che settimane per la parallelizzazione di tipiche applicazione numeriche o di grafica computazionale) [40].

Va notato infine che esistono già diversi ambienti "di produzione" che utilizzando pesantemente la programmazione parallela strutturata, anche se in forme decisamente più limitate di quanto non venga fatto in questi progetti. Google ha fatto la sua fortuna su un framework che permette di realizzare velocemente efficaci applicazioni parallele che processano grandi quantità di dati secondo il pattern parallelo *MapReduce* (prima trasforma ogni dato in una coppia <chiave, risultato> poi "somma" tutti i risultati con la stessa chiave). Microsoft ha rilasciato la Task Parallel Library che permette di scrivere applicazioni parallele in cui il parallelismo è di fatto implementato opportune funzioni di libreria simili agli algorithmic skeletons. OpenCL, ambiente di programmazione utilizzato prevalentemente per le GPU, mette a disposizione del programmatore applicativo diversi pattern data parallel implementati come primitive del linguaggio (o algorithimc skeleton, se vogliamo).

4. Conclusioni

I recenti ed importanti miglioramenti nelle tecnologie di produzione dei componenti di calcolo hanno prodotto macchine parallele ed eterogenee potenzialmente in grado di eseguire applicazioni con prestazioni molto elevate. Sfortunatamente, la programmazione di queste nuove macchine utilizzando gli strumenti di sviluppo classici richiede uno sforzo consistente ai programmatori applicativi, che si trovano a dover scrivere e gestire sia il codice necessario a implementare la *business logic* dell'applicazione che quello necessario a implementare e gestire la struttura parallela dell'applicazione stessa.

Per ovviare a questo problema, comunità di ricercatori appartenenti ad ambienti di ricerca diversi hanno prodotto nuovi strumenti di sviluppo basati su concetti di programmazione parallela strutturata. Questi ambienti di sviluppo permettono di semplificare la vita ai programmatori applicativi ma, al contempo, supportano lo sviluppo di applicazioni molto efficienti sulle nuove architetture disponibili.

In questo lavoro abbiamo introdotto brevemente le caratteristiche principali degli ambienti di programmazione basati su *algorithmic skeleton* e su *parallel design pattern*. Questi ambienti sono stati sviluppati principalmente dalle comunità di ricercatori che si interessano di *High Performance Computing* e *Software Engineering* ed hanno, al momento, una diffusione limitata ai gruppi di ricerca di università e grossi centri di ricerca. Diversi progetti, alcuni dei quali ancora in corso di svolgimento, hanno contribuito e contribuiscono al loro sviluppo e li stanno portando a livelli adatti alla loro adozione nel mondo dell'industria software. Dal canto loro, alcuni grossi nomi dell'industria informatica mondiale (fra cui Intel e Microsoft) hanno recepito parte dei concetti relativi alla programmazione parallela strutturata e li stanno adottando in alcuni dei loro prodotti.

A breve, ci dovremmo aspettare che queste tecnologie prendano il sopravvento, come accadde negli anni '80 nel mondo della programmazione sequenziale quando i linguaggi strutturati e object oriented presero il sopravvento sui linguaggi non strutturati usati fino a quell'epoca. Questo porterà ad uno sviluppo sempre più consistente degli strumenti di supporto alla programmazione parallela (compilatori, interpreti, librerie di run time) e ad un corrispondente sempre maggior livello di astrazione messo a disposizione ai programmatori applicativi. Questo permetterà di ridurre il *time-to-market* delle nuove applicazioni e lo sviluppo di applicativi sempre più complessi con un minore sforzo, in proporzione, da parte dei programmatori.

Riquadro 1: Algorithmic skeleton e parallel design pattern

	Algorithmic skeleton	Parallel design pattern
Cosa sono	Costrutti predefiniti, parametrici, che modellano forme di parallelismo note.	Descrizioni di forme di parallelismo (nome, uso tipico, problemi di implementazione, esempi di codice)
Chi li promuove	Comunità HPC	Comunità Sw Engineering
Quando sono stati introdotti	Inizio anni '90	Primi anni 2000
Pubblicazione chiave	M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming", Journal Parallel Computing Volume 30 Issue 3, March 2004 Pages 389 - 406	T. G. Mattson, B. A. Sanders, B. L. Massingill, Patterns for parallel programming, Addison Wesley, 2013, ISBN-13: 978-0321940780
Framework che li adottano	FastFlow, Muesli, Sketo, SKEPU, OSL	INTEL TBB Building Block Design Patterns White paper
Come sono messi a disposizione	Costrutti del linguaggio sequenziale ospite (chiamate di libreria, oggetti, funzioni higher order)	Descrizione (solo testo)

Riquadro 2: FastFlow

Cos'è:	Framework di programmazione parallela C++ che mette a disposizione come oggetti primitivi astrazioni che modellano i più comuni pattern paralleli (algorithmic skeletons)
Com'è fornita:	Libreria HPP-only: header files da includere e compilare insieme al codice applicativo
Per quali architetture:	CPU multi-core e many core (Intel, AMD, ARM, Tilera), GPU (nVidia CUDA, AMD OpenCL), reti di multi/many core con GPU (Ethernet, Infiniband)
Requisiti:	Compilatore C++11, libreria pthread
Quali pattern:	stream parallel (pipeline, task farm), data parallel (parallel for, map, reduce, stencil), high level (divide&conquer, pool evolution, macro data flow)
Scenari applicativi:	parallelismo a grana fine su architetture shared memory e cache coherent, applicazioni streaming, uso congiunto di multi-core e acceleratori
Use cases portfolio:	Bowtie2 porting, two phase video image restoring (CPU+GPU), Yadt C4.5 classifier, pbzip2 porting, block-based Cholesky & LU decomposition, …
Licenza:	Open Source LGPL
Download:	svn checkout svn://svn.code.sf.net/p/mc-fastflow/code/ mc-fastflow-code
Sito web:	http://calvados.di.unipi.it/fastflow
Sample code:	Codice necessario per l'implementazione di un master/worker con NW workers:

```
using namespace ff;

//calcolo di un task
struct Worker: ff_node<float> {
    float *svc(float *task) { .... }
    ... òp
};
...
int main() {
   …
  std::vector<std::unique_ptr<ff_node>> W;
  for(size_t i=0;i<NW;++i)
    W.push_back(make_unique<Worker>());
  ff_Farm<> unFarm(W);
  if (unFarm.run_and_wait_end()<0)
     error("running unFarm");
  ...
}
```

Riquadro 3: Sigle utilizzate nel testo

SIGLA	TP
BSP	Bulk Synchronous Computing
CPU	Central Processing Unit
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
GP-GPU	General Purpose Graphics Processing Unit
HPC	High Performance Computing
MPI	Message Passing Interface
PGAS	Partitioned Global Address Space
SOC	System On Chip
SPMD	Single Program Multiple Data
TBB	Thread Building Blocks
TPL	Task Parallel Library

Bibliografia

[1] R. R. Schaller, “Moore's law: past, present, and future”, *IEEE Spectrum*, Volume 34 Issue 6, June 1997, Page 52-59, IEEE Press

[2] C. A. Mack, “Fifty Years of Moore's Law”, *Semiconductor Manufacturing, IEEE Transactions* on Vol. 24 , Issue 2, pp 202-207, 2011

[3] D. Ivosevic, N. Frid, “Performance-Occupation trade-off examination in custom processor design”, Proceedings of MIPRO 2014, pp. 1024-1029, DOI: http://dx.doi.org/10.1109/MIPRO.2014.6859719, 2014

[4] J. Jeffers, J. Reinders, Intel Xeon PHI Coprocessor High Performance Computing, Morgan Kaufmann, 2013, ISBN: 9780124104945

[5] K. Fatahalian, M. Houston, “A Closer Look at GPUs”, *Communications of the ACM*, Vol. 51 No. 10, Pages 50-57, 2008, DOI: 10.1145/1400181.1400197

[6] I. Kuon, R. Tessier, J. Rose, “FPGA architecture: survey and challenges”, *Foundations and Trends in Electronic Design Automation*, Vol 2, Issue 2, Feb 2008, pp. 135-253, Now Publishers

[7] L. Dagum, R. Menon, “OpenMP: An Industry-Standard API for Shared-Memory Programming”, *IEEE Computational Science & Engineerin*, Vol. 5 Issue 1, Jan 1998, pp 46 – 55, IEEE Computer Society Press, DOI: 10.1109/99.660313

[8] The OpenMP API specification for parallel programming, http://openmp.org/wp/, ultimo accesso febbraio 2015

[9] W, Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, 3rd edition, MIT Press, 2014.

[10] F. Darema, “The SPMD Model: Past, Present and Future”, Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer-Verlag London, 2001

[11] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, D. Schaa, *Heterogeneous Computing with OpenCL*, 2011, Morgan Kaufmann

[12] J. Nickolls, I. Buck, M. Garland, M. Skadron, “Scalable Parallel Programming with CUDA”, *Queue*, Vol. 6 Issue 2, March/April 2008, pp. 40-53, ACM, DOI: 10.1145/1365490.1365500

[13] OpenACC Directives for Accelerators, http://www.openacc-standard.org/, ultimo accesso, 2015

[14] S. Mazor, P. Langstraat, *A guide to VHDL*, Springer Verlag, 2013, ISBN: 1475721161

[15] S. Lee, Advanced Digital Logic Design Using Verilog, State Machines, and Synthesis for FPGA's, Thomson-Engineering, 2005, ISBN:0534551610

[16] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, K. Asanovic, “Chisel: constructing hardware in Scala embedded language”, DAC '12 Proceedings of the 49th Annual Design Automation Conference, pp. 1216-1225, ACM, 2012, ISBN: 978-1-4503-1199-1

[17] PGAS Partitioned Global Address Space, http://www.pgas.org/, ultimo accesso 2014

[18] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, Y. Zhou, “Cilk: an efficient multithreaded runtime system”, PPOPP '95 Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 207-216, ACM, 1995

[19] Intel, “Intel Cilk++ SDK Programmer’s Guide”, https://www.clear.rice.edu/comp422/resources/ Intel_Cilk++_Programmers_Guide.pdf, 2009

[20] F. Cesarini, S. Thompson, Erlang programming: a concurrent approach to software development, O’Reilly, 2009

[21] J. Dean, S. Ghemawat, “MapReduce: simplified data processing on large clusters”, CACM, Vol. 51, Issue 1, pp 107-113, ACM, 2008

[22] T. White, Hadoop, the definitive guide, O’Reilly, 2009

[23] G. Malewicz, M. H. Austern, A. J.C. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowsky, “Pregel: a system for large scale graph processing”, SIGMOD '10 Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pp. 135-146, ACM, 2010

[24] K Hinsen, H P Langtangen, O Skavhaug, and A Odegard, “Using BSP and Python to simplify parallel programming”, *Future Generation Computer Systems* 22(1-2), pp. 123-157, 2006

[25] Intel, “Intel Thread Building Blocks”, https://www.threadingbuildingblocks.org/, ultimo accesso, 2015

[26] Microsoft, Task parallel library (TPL), https://msdn.microsoft.com/it-it/library/dd460717%28v=vs.110%29.aspx, ultimo accesso 2015

[27] J. A. Pienaar, S. Chakradhar, A. Raghunathan, “Automatic generation of software pipelines for heterogeneous parallel systems”, Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society Press, 2012

[28] J. Berthold, M. Dieterle, R. Logen, S. Priebe, “Hierarchical Master-Worker Skeletons”, in *Practical Aspects of Declarative Languages*, LNCS, Vol. 4902, pp. 248-264, Springer Verlag, 2008

[29] Murray I. Cole. Algorithmic Skeletons: Structured Management of. Parallel Computation, MIT Press, 1989.

[30] M. Cole, “Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming”, Parallel Computing, Vol. 30 Issue 3, pp. 389 – 406, 2004, Elsevier Science Publishers

[31] K. Matsuzaki, K. Emoto, “Implementing fusion-equipped parallel skeletons by expression templates”, Proceeding IFL'09 Proceedings of the 21st international conference on Implementation and application of functional languages, pp. 72-89, Springer Verlag, 2010

[32] Fastflow home page, http://calvados.di.unipi.it/fastflow, ultimo accesso, marzo 2015

[33] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, M. Torquati, “Accelerating Code on Multi-cores with FastFlow”, Euro-Par 2011 Parallel Processing, LNCS, Vol. 6853, pp. 170-178, Springer Verlag, 2011

[34] S. Ernsting, H. Kuchen, “Algorithmic skeletons for multi-core, multi-GPU systems and clusters”, *International Journal of High Performance Computing and Networking*, vol. 7 Issue 2, April 2012, pp. 129-138, Inderscience Publishers

[35] J. Enmyren, C. W. Kessler, SkePU: a multi-backend skeleton programming library for multi-GPU systems, Proceedings of HLPP '10 Proceedings of the fourth international workshop on High-level parallel programming and applications, pp. 5-14, ACM, 2010

[36] SkeTo project, http://sketo.ipl-lab.org/, ultimo accesso 2015

[37] N. Javed, F. Loulergue, "OSL: Optimized Bulk Synchronous Parallel Skeletons on Distributed Arrays"; *Advanced Parallel Processing Technologies*, LNCS, Vol. 5737, pp. 436-451, Springer Verlag, 2009

[38] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, K. Yelick, "A View of the Parallel Computing Landscape", *Communications of the ACM*, Vol. 52 No. 10, Pages 56-67

[40] ParaPhrase home page, http://www.paraphrase-ict.eu/, ultimo accesso 2015

[41] REPARA home page, http://parrot.arcos.inf.uc3m.es/wordpress/, ultimo accesso 2015

[42] T. Serban, M. Danelutto, P. Kilpatrick, "Autonomic scheduling of tasks from data parallel patterns to CPU/GPU core mixes", High Performance Computing and Simulation (HPCS, pp. 72-79, ISBN 978-1-4799-0836-3), IEEE Press, 2013

[43] M. Goli, H. González-Vélez, "Heterogeneous Algorithmic Skeletons for Fast Flow with Seamless Coordination over Hybrid Architectures", Proceedings of PDP 2013 Euromicro International Conference on Parallel, Distributed and Network based processing, pp. 148-156, IEEE Computer Society Press, 2013

[44] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarra-Miranda. "An evaluation of global address space languages: co-array fortran and unified parallel C", In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '05)*, ACM, 2005

[45] G. L. Steele, Jr.. 2006. "Parallel programming and code selection in fortress", in Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '06), ACM, 2006

[46] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing", *SIGPLAN,* No. 40, 10 (October 2005)

Biografia

Marco Danelutto ha ottenuto il Dottorato di Ricerca in Informatica nel 1990, è attualmente professore associato presso il Dipartimento di Informatica dell'Università di Pisa ed ha recentemente ottenuto l'abilitazione a professore di prima fascia. I suoi interessi di ricerca sono concentrati su modelli e strumenti per la programmazione parallela strutturata. E' autore di oltre 140 pubblicazioni internazionali su riviste e conferenze con peer review. Attualmente è presidente del Corso di Laurea Magistrale congiunto (Università di Pisa e Scuola Superiore Sant'Anna) in Informatica e Networking.

email: marco.danelutto@unipi.it