Calcolo parallelo e sistemi multicore

Con l'avvento delle piattaforme multicore il calcolo parallelo non è più solo prerogativa dei supercomputer ma rappresenta ormai una realtà alla portata di tutti. L'articolo esamina i principali elementi alla base di questo paradigma e permette di comprendere le relative problematiche le cui soluzioni sono alla base di una diffusione massiva di questo nuovo approccio di computing

Silvano Iacobucci

L'industria del computing è caratterizzata dalla ricerca di prestazioni sempre crescenti ed efficienti, dalle applicazioni di fascia alta nei settori delle reti, telecomunicazioni, avionica, ai sistemi embedded a basso consumo in calcolatori desktop, portatili e videogame. Il percorso di sviluppo porta chiaramente verso i sistemi multicore, dove i processori a due, quattro, otto "core" rappresentano solo l'inizio di una prossima espansione a un numero sempre crescente di nuclei di calcolo. Questa espansione però crea una sfida, non solo nell'industria dei semiconduttori, ma anche nello sviluppo di applicazioni che possano essere eseguite con calcoli paralleli, raramente semplice e qualche volta perfino impossibile.

Da circa 40 anni è ormai ben nota la modalità per scrivere il software per un calcolo seriale, ovvero per poter essere eseguito su un singolo calcolatore con una singola Cpu, spezzando un problema in una serie discreta di istruzioni eseguite in successione ed eseguendo una sola istruzione per volta, come stabilito dal modello di Von Neumann.

Il calcolo parallelo, nella sua forma più semplice, rappresenta invece l'impiego simultaneo di risorse di calcolo multiple per risolvere un problema di elaborazione, in modo da poter essere eseguito su più Cpu, spezzando un problema in parti discrete elaborabili contemporaneamente, dove ciascuna viene ulteriormente suddivisa in una serie di istruzioni che possono essere eseguite in modo seriale su differenti Cpu (Fig. 1).

Le risorse di elaborazione possono includere un singolo computer con processori multipli, un numero arbitrario di computer connessi via rete o una combinazione di entrambi gli approcci.

Per poter essere eseguito in modo parallelo, il problema di calcolo deve avere le seguenti caratteristiche: essere scomponibile in parti risolvibili simultaneamente, eseguire istruzioni di programma multiple in qualsiasi momento ed essere risolto in un tempo inferiore, con risorse di calcolo multiple rispetto a una singola risorsa di calcolo. Il calcolo parallelo è una evoluzione del calcolo seriale che tenta di emulare ciò che spesso avviene nel mondo naturale: molteplici eventi complessi e intercorrelati che avvengono nello stesso momento e talvolta in sequenza.

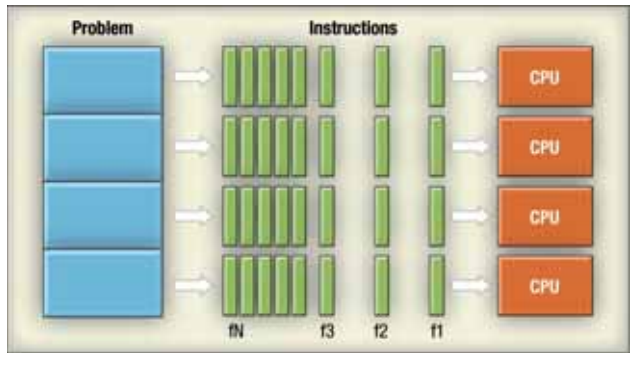

Fig. 1 – Il calcolo parallelo

Il calcolo parallelo è sempre stato considerato l'estremo apice o il futuro del computing e fino a pochi anni fa è stato motivato da simulazioni numeriche di sistemi e situazioni complesse riguardanti svariati settori: previsioni del tempo e del clima, reazioni chimiche e nucleari, genoma umano, attività sismica e geologica, comportamento di dispositivi meccanici (dalle protesi alle navette spaziali), circuiti elettronici, processi di fabbricazione. Oggi tuttavia anche applicazioni più commerciali stanno richiedendo con forza crescente lo sviluppo di computer sempre più veloci, per supportare l'elaborazione di grosse quantità di dati in modalità sofisticate, quali: data mining e database paralleli, esplorazioni petrolifere, motori di ricerca web e servizi di business in rete, diagnosi mediche assistite da calcolatore, gestione di aziende multinazionali, grafica avanzata e realtà virtuale (soprattutto nell'industria dei videogiochi), tecnologie multimediali e video in rete, ambienti di lavoro collaborativi. Non ultimo,

Fig. 2 – Sistemi Sisd

il calcolo parallelo rappresenta un tentativo di massimizzare quella risorsa infinita ma al contempo sempre più preziosa e scarsa che è il tempo. Per questo i sistemi di calcolo parallelo si stanno spostando dal mondo dei costosissimi supercomputer, riservati a pochi eletti, a soluzioni più economiche e "popolari" basate su processori multipli o computer interconnessi che possano superare i vincoli del calcolo seriale e i limiti delle Cpu singole. La realizzazione di processori singoli sempre più veloci e capaci ormai sembra aver raggiunto limiti fisici e pratici difficilmente superabili: una frequenza di clock limitata dalla velocità di trasmissione del rame (9 cm/ns) e dalla stessa velocità della luce (30 cm/ns); una miniaturizzazione delle tecnologie di costruzione dei processori ormai spinta all'estremo; fattori economici (usare molti processori o computer in parallelo è più economico che realizzare, a parità di prestazioni, un singolo processore velocissimo); limitazioni in termini di dissipazione termica che, crescendo con la frequenza di clock e con il quadrato della tensione di alimentazione, richiede sistemi di raffreddamento sempre più complessi, diminuisce l'affidabilità e riduce la longevità del dispositivo.

Architetture di calcolo parallelo

L'esecuzione parallela dei calcoli non è qualcosa di nuovo, ma l'implementazione di un sistema che possa lavorare in parallelo in un ambiente di calcolo in cui l'ordine debba essere mantenuto a tutti i costi pone alcuni problemi, derivanti dal fatto che la soluzione non può essere realizzata solo in ambito hardware, di compilatore o di sistema operativo, anche perché in queste aree la parallelizzazione è già stata implementata da tempo. Il parallelismo può essere pensato a vari livelli: bit, istruzione, task e dato.

Il parallelismo a livello di bit estende l'architettura hardware per operare simultaneamente su oggetti dati più grandi; per esempio, se un core 8-bit deve elaborare un oggetto a 16 bit richiede due istruzioni. Se i dati sono a 16-bit, l'istruzione si riduce a una sola. Per questo si è passati da core di 4 bit a core a 64 bit. Il parallelismo a livello di istruzione è una tecnica che identifica le istruzioni che non dipendono l'una dall'altra, perché ad esempio lavorano su dati differenti e possono essere eseguite in contemporanea. Questa tecnica, solitamente implementata tramite compilatore, non sempre è di semplice attuazione e la sua possibilità di realizzazione dipende dal tipo di applicazione (ad esempio, è applicabile nel digital signal processing, ma non in altri contesti).

Il parallelismo a livello di task prevede la distribuzione di differenti applicazioni, processi o thread a diverse unità di elaborazione. Questo può essere implementato manualmente o tramite

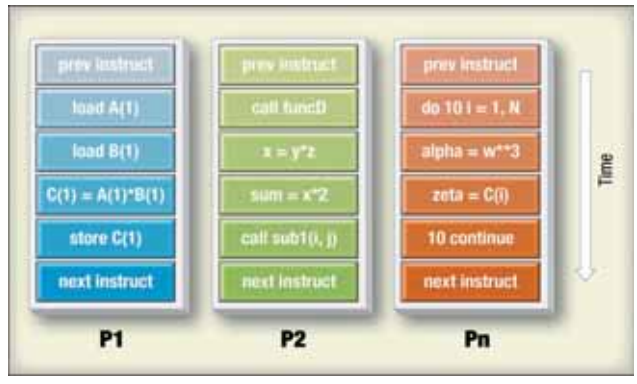

Fig. 3 – Sistemi Mimd

sistema operativo; tuttavia la difficoltà non risiede tanto nella modalità di distribuzione dei thread, ma nel capire come dividere l'applicazione in thread multipli. Il parallelismo a livello di dato è collegato con una classificazione del calcolo parallelo tramite la tassonomia di Flynn, che raggruppa in una semplice tabella le architetture di calcolo multiprocessore in funzione delle dimensioni istruzioni/dati e degli stati singolo/multiplo (Tab. 1). I sistemi Sisd (Single instruction single data) sono in pratica i calcolatori seriali, e comprendono la maggior parte dei pc e workstation a Cpu singola (Fig. 2); all'estremo opposto si trova l'approccio di tipo Mimd (Multiple instruction multiple data), dove ogni processore può lavorare in parallelo su flussi dati differenti, con esecuzione sincrona o asincrona, deterministica o non-deterministica. Questo è l'approccio usato dalla maggior parte degli attuali sistemi di parallel computing, supercomputer, "grid-computer" (calcolatori connessi a matrice) e calcolatori multiprocessore simmetrici, tra cui alcuni tipi di pc (Fig. 3). L'approccio Simd (Single instruction multiple data) prevede l'elaborazione di dati in modo simultaneo da parte di unità di calcolo multiple e viene usato in sistemi embedded commerciali, come ad esempio quelli di Freescale. La casistica Misd è stata utilizzata per lo più in passato, in poche applicazioni di supercomputing estremamente specializzate o addirittura solo in applicazioni sperimentali. Il parallelismo a livello di dato, molto difficile da applicare in alcuni casi (es. algoritmi crittografici 3DES/AES), non può essere automatizzato via hardware o tramite compilatore perchè risulta estremamente complicato assemblare un algoritmo affidabile e robusto, e perché le parti di software parallelizzabili devono essere scelte manualmente.

Nella valutazione di algoritmi per il calcolo parallelo è importante tenere conto di due leggi matematiche: la legge di Amdahl e

Tabella 1 - La tassonomia di Flynn

SISD	SIMD
Single Instruction, Single Data	Single Instruction, Multiple Data
MISD	MIMD
Multiple Instruction, Single Data	Multiple Instruction, Multiple Data

la legge di Gustafson (Fig. 4). La legge di Amdahl parte dal presupposto che in ogni applicazione ci sia una parte non parallelizzabile, che sostanzialmente limita in modo logaritmico l'aumento di velocità di calcolo ottenibile, rendendo in alcuni casi il vantaggio trascurabile pur al crescere del numero di core utilizzati:

$$Speedup = 1/(S+(1-S)/N)$$

dove S è la porzione di algoritmo in percentuale che esegue codice serializzato, e N è il numero di core usati.
Questa legge tuttavia parte dall'assunto che il problema sia fisso, ossia non possa cambiare nel tempo la percentuale di applicazione parallelizzabile rispetto a quella seriale; tuttavia in alcune applicazioni come il network routing, la porzione relativa di applicazione seriale decresce nel tempo e aumenta quella parallelizzabile (il numero di pacchetti è ignoto, ma aggiungendo core si incrementa la possibilità di elaborare in parallelo il traffico di rete). In questi sistemi si applica quindi la legge di Gustafson, che lega lo speedup al numero di core usati in modalità proporzionale ridefinendolo "scaled speedup":

$$Scaled\text{-}speedup = N + (1-N) \times S$$

dove S è la porzione di algoritmo in percentuale che esegue codice serializzato, e N è il numero di core usati.
Al crescere del numero di core utilizzati crescono le prestazioni di calcolo.

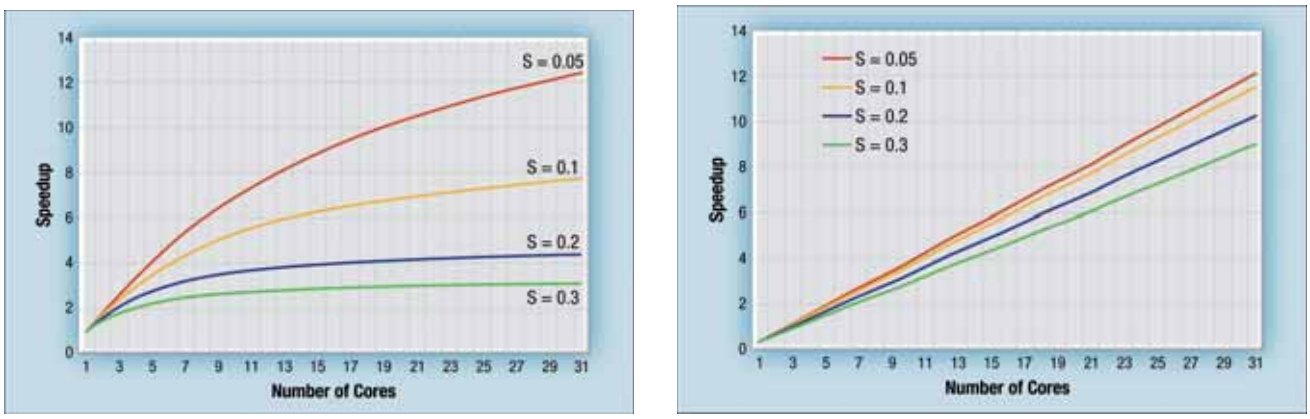

Fig. 4 – Legge di Amdahl (A) e legge di Gustafson (B). Fonte: Freescale

Architetture di memoria

Oltre alle architetture di calcolo in generale, è bene distinguere anche attraverso vari tipi di architetture di memoria utilizzabili per il calcolo parallelo, che possono essere a memoria condivisa, a memoria distribuita o ad approccio ibrido distribuito-condiviso. L'architettura a memoria condivisa, la prima a essere introdotta, prevede che diverse Cpu operino in parallelo ma utilizzando lo stesso spazio di memoria in modo condiviso (Fig. 5A). A sua volta, questa architettura può essere caratterizzata da un accesso uniforme o non uniforme alla memoria da parte delle varie Cpu (Uma, Uniform memory access oppure Numa, Non-uniform memory access). I vantaggi di questa architettura consistono in una relativa semplicità di programmazione nei confronti dell'area di memoria e in una condivisione dati tra i task veloce e uniforme, grazie alla prossimità della memoria con le Cpu. Gli svantaggi, per contro, contemplano una mancanza di scalabilità tra memoria e Cpu (l'aumento di Cpu aumenta geometricamente il traffico sui percorsi tra memoria condivisa e Cpu), una grande responsabilità del programmatore nel realizzare meccanismi di sincronizzazione per garantire accessi corretti alla memoria, e costi elevati al crescere del numero di processori.
L'architettura a memoria distribuita (Fig. 5B) permette a ogni processore di avere una memoria dedicata operando con maggiore autonomia, dove gli indirizzi di memoria di una Cpu non sono legati a un'altra Cpu, e ogni blocco Cpu-memoria può colloquiare con gli altri attraverso un bus o un segmento di rete Ethernet. L'architettura è vantaggiosa per la scalabilità (aumentando i processori, la memoria cresce proporzionalmente), per gli accessi rapidi di ogni Cpu verso la propria memoria senza interferenze o overhead necessari a mantenere coerenza di cache, e per costi ragionevoli perché è possibile usare processori e reti disponibili commercialmente, assemblandoli opportunamente. Gli svantaggi sono legati a una consistente responsabilità del programmatore verso la sincronizzazione e il trasferimento dati tra le Cpu, alla difficoltà di mappare strutture dati esistenti, basate normalmente su organizzazioni a memoria unica, e ai tempi di accesso tipici dell'approccio Numa.
Il terzo tipo di architettura deriva da un approccio ibrido tra i due precedenti (Fig. 5C) ed è quello maggiormente utilizzato attualmente nei computer più grandi e veloci; presenta vantaggi e svantaggi comuni alle due soluzioni di base.

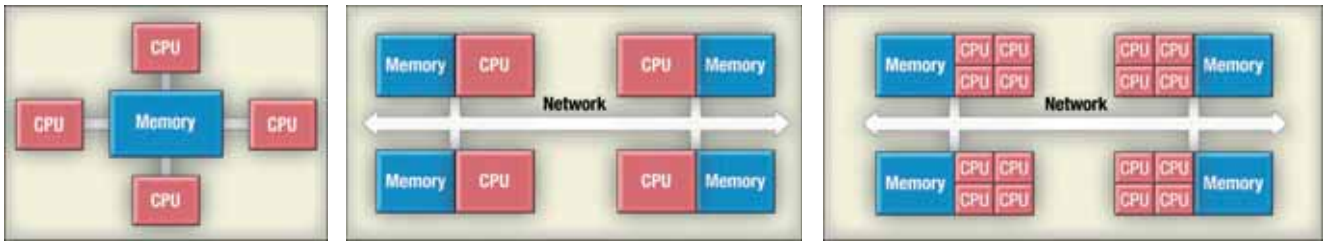

Fig. 5 – L'architettura a memoria condivisa (A), a memoria distribuita (B) e approccio ibrido (C)

Modelli di programmazione parallela

I modelli di programmazione parallela non sempre sono esattamente legati a una delle architetture di memoria suddette, in quanto sono astrazioni indipendenti dalla modalità di realizzazione. I modelli di programmazione parallela più diffusi sono svariati: a memoria condivisa, a thread, message parsing, data parallel e ibrido. Il primo, che non obbligatoriamente deve essere mappato su una architettura a memoria condivisa, prevede che lo spazio degli indirizzi di memoria sia accessibile da tutti e venga gestito con meccanismi di sincronizzazione di tipo lock o semafori. Il modello di programmazione a thread contempla un processo di elaborazione contenente diverse parti in esecuzione simultanea, e tipicamente viene usato con architetture e sistemi operativi a memoria condivisa (es. Posix e OpenMP). Il message parsing prevede che ogni gruppo di task abbia una sua memoria dedicata e comunichi con gli altri tramite scambio di messaggi, come nello standard de-facto industriale Mpi (message parsing interface). Il modello a dati paralleli è usato nel calcolo scientifico nei casi in cui tutti i processi operano ciascuno su un sottoinsieme di dati di ingresso per fornire un unico output; tipicamente impiega il linguaggio Fortran 77 con estensioni aggiuntive del '90 e '95.

La progettazione e lo sviluppo di programmi paralleli è stato sempre un processo molto manuale, dove il programmatore è responsabile di identificare e implementare il parallelismo. Molto spesso questa attività è onerosa, complessa, soggetta a errori ed è un procedimento iterativo. Esistono comunque vari tool che permettono di assistere il programmatore a convertire programmi seriali in paralleli, tipicamente dei compilatori o preprocessori di tipo parallelo, completamente automatici o gestiti dallo sviluppatore. In caso di vincoli di tempo o economici, la soluzione automatica è preferibile, anche se può presentare alcuni inconvenienti o svantaggi: risultati errati, prestazioni degradate, poca flessibilità rispetto alla parallelizzazione manuale, limitato a sottoinsiemi di codice, disponibili principalmente solo per linguaggio Fortran.

È essenziale comunque una attività di analisi preliminare del programmatore finalizzata a stabilire se un problema è effettivamente scomponibile in una soluzione parallelizzabile. Ad esempio, il calcolo della serie di Fibonacci non è intrinsecamente parallelizzabile, perché un dato dipende dal calcolo di precedenti iterazioni, mentre altri problemi quali la modellazione di ecosistemi o l'elaborazione digitale di segnali possono essere parallelizzate con grandi vantaggi.

Nella programmazione parallela occorre tenere conto anche di molti altri fattori: partizionamento dei dati e decomposizione funzionale del problema in parti discrete parallelizzabili, comunicazione fra i task, sincronizzazione, dipendenze fra i dati, load balancing, granularità di parallelismo (rapporto tra periodi di calcolo e periodi di comunicazione), gestione dell'I/O, complessità e costi del parallelismo (soprattutto in termini di tempi di progettazione, codifica, debug, tuning e manutenzione), portabilità (sicuramente migliorata dall'introduzione degli standard Mpi, thread Posix, Hpf, open, ma non sempre garantita), richiesta di risorse (la diminuzione dei tempi di elaborazione deve essere compensata da un aumento dei processori), scalabilità, necessità di analisi prestazionali e tuning.

Un percorso difficile, ma in evoluzione

Proprio per colmare il profondo divario tecnologico tra architetture multicore e i software destinati a gestirle, poco più di un anno fa Microsoft e Intel hanno stanziato un budget di venti milioni di dollari destinato a velocizzare la ricerca sul "parallel computing", in collaborazione con il mondo accademico, e a produrre nel giro di qualche anno risultati tangibili anche nelle applicazioni embedded. L'obiettivo è semplificare la programmazione parallela e migliorare le tecnologie di controllo e bilanciamento dei processori, che nel frattempo continuano ad aumentare il numero dei loro core, rendendo disponibili tool, firmware e linguaggi adeguati (sono in fase di sviluppo istruzioni aggiuntive per i linguaggi C/C++, oggi usati quasi ovunque nei sistemi embedded, e forse verrà creato un nuovo linguaggio di programmazione).

Per aiutare sviluppatori e system designer a velocizzare il time-to-market dei sistemi, i principali produttori stanno proponendo vari strumenti proprietari a supporto dei propri processori multicore, ed è in corso anche uno sforzo di standardizzazione tramite un consorzio di aziende facenti capo alla Multicore Association.

Tra gli strumenti proprietari si può annoverare la piattaforma della società RapidMind, recentemente acquisita da Intel (che aveva già nei mesi precedenti acquisito Wind River). Il tool RapidMind consente lo sviluppo di codice C++ su processori multipli e si basa più sul concetto di parallelismo del dato che

non di parallelismo di task. Un vantaggio del parallelismo di dato consiste nel poter scrivere programmi indipendenti dal numero di core usati dal target hardware. È importante sottolineare che l'Api RapidMind lavora con compilatori C++ esistenti ed è sufficiente includerla nel link come libreria; in questo modo è possibile utilizzarla con sistemi di sviluppo e Ide già esistenti. Il porting di una applicazione su processori multicore è un processo incrementale: per prima cosa si identificano tramite tool di profilazione le sezioni di applicazione che consumano molto tempo di esecuzione, poi si convertono i tipi array e numerici negli equivalenti RapidMind e si utilizza l'Api per "catturare" il calcolo. Infine, la piattaforma RapidMind mappa il calcolo sul target hardware basato su core multipli, scaricando il programma principale dell'overhead di elaborazione. Questo processo può essere ripetuto tante volte quante è necessario.
Partendo dal concetto di Simmetric Multiprocessing (SMP), un sistema operativo in un dispositivo multicore con architettura a memoria condivisa è capace di allocare risorse di elaborazione e

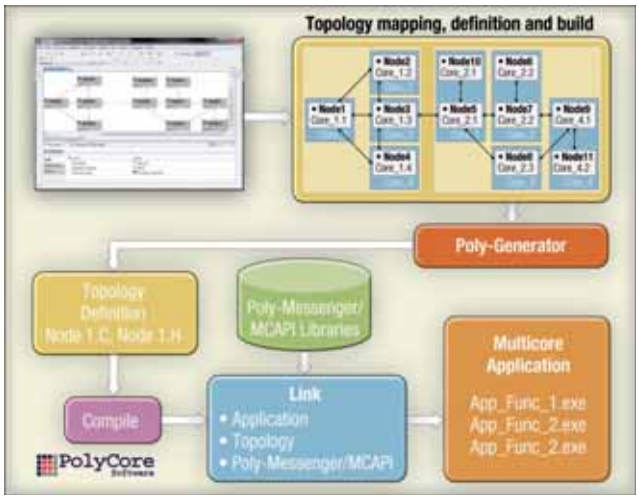

Fig. 6 – La soluzione PolyCore Poly-Messenger/Mcapi

di bilanciare i carichi dei task o dei thread attraverso i vari core, garantendo uno sviluppo software più semplice. Tra i sistemi operativi usati da Freescale si possono annoverare SMP Linux e l'Rtos multicore OSE di Enea. Quest'ultimo presenta diversi vantaggi, superando i problemi tipici di un approccio SMP e rendendo la scalabilità lineare come in un approccio distribuito tipico dell'Asimmetric Multiprocessing (AMP o ASMP). Nell'estate 2009 Freescale ha annunciato VortiQa, un set di moduli applicativi per i propri processori multicore embedded capaci di ridurre gli sforzi di sviluppo di codice parallelo. Il software supporta una varietà di modelli di elaborazione simmetrica ed asimmetrica usati nei chip Freescale QorIQ e PowerQuicc, e può fornire agli Oem fino all'80% del software applicativo necessario per alcuni progetti (codice virtual private networking, firewall stateful packet inspection per router/switch, filtri virus e spam per reti business e traduttori di indirizzi di rete per gateway, e prossimamente moduli per storage e sistemi di accesso e applicazioni per il mercato multimedia). Il software supporta Linux, OSE di Enea e VxWorks di Wind River. Due aree in forte sviluppo che vengono in aiuto ai progettisti di sistemi basati su programmazione parallela sono quelle della virtualizzazione e della simulazione, che consentono di reintrodurre controllo e determinismo nel debugging del software e nel processo di analisi, anche per i processori multicore. Le piattaforme virtuali e i simulatori (come ad esempio Labview di National instruments) offrono inoltre la possibilità di sviluppare software prima che il dispositivo hardware sia effettivamente disponibile, e costituiscono strumenti importanti nella verifica preliminare delle prestazioni e nell'analisi del progetto hardware. Il consorzio Multicore Association opera invece dal punto di vista della standardizzazione, indispensabile anche in questo settore. Circa un anno fa ha esordito con il rilascio della Mcapi (Multicore communications application programming interface), una Api in grado di gestire gli elementi base delle comunicazioni intercore on-chip richiesti dai sistemi embedded supportando fino a centinaia di core. Questa Api aiuta gli sviluppatori di software multicore a costruire applicazioni che girano su un'ampia varietà di architetture multicore, minimizzando il tempo necessario per le attività di porting e tuning delle applicazioni su differenti implementazioni. Con questo standard PolyCore Software ha realizzato un framework utilizzabile per il computing distribuito, sia a livello di core multipli su un chip sia a livello di chip multipli su una scheda (Fig. 6). Attualmente il consorzio sta lavorando su diverse direttrici: una guida di programmazione software multicore per l'industria, denominata Multicore Programming Practice (MPP), che possa migliorare la consistenza e la comprensibilità degli elementi tipici della programmazione multicore; la gestione delle risorse a livello applicativo richieste da multicore embedded SMP e AMP (Mrapi, Multicore resource management api); la virtualizzazione in ambito multicore (multicore virtualization), che migliori la portabilità e indirizzi problematiche a livello di gestione risorse, load balancing, debugging.

readerservice.it

Enea	www.enea.com
Freescale	n.16
Intel	n.17
Microsoft	n.18
Multicore Association	www.multicore-association.org
National instruments	n.19
PolyCore Software	www.polycoresoftware.com
Wind River	n.20