Kamran Shah

LABVIEW E L'HYPERTHREADING

L'hyperthreading è una caratteristica avanzata delle versioni evolute del processore Intel Pentium 4 e successivi

Un computer hyperthreaded ha un solo processore, ma si comporta come se sullo stesso chip fossero incorporati più processori. Alcune delle risorse sul chip sono duplicate, come il set di registri, altre sono condivise, come le unità di esecuzione e la cache. Alcune risorse, come i buffer per le microoperazioni, sono partizionate, in modo tale che ogni processore logico ne riceva una parte.

Ottimizzare un'applicazione per sfruttare l'hyperthreading è come ottimizzare un'applicazione per un sistema multiprocessore, ma vi sono alcune differenze. Per esempio, un computer hyperthreaded condivide le unità di esecuzione mentre un computer a doppio processore contiene due set completi di unità di esecuzione. Pertanto, tutte le applicazioni che sono limitate dalle unità di esecuzione in virgola mobile hanno prestazioni migliori su un computer multiprocessore, perché non è necessario condividere le unità di esecuzione. Lo stesso principio vale per la contesa della cache. Se due thread cercano di accedere alla cache, le prestazioni sono migliori su un computer multiprocessore, dove ogni processore possiede la propria cache.

RICHIAMI GENERALI SULLE MODALITÀ DI ESECUZIONE DI LABVIEW

Il sistema di esecuzione di LabVIEW è già concepito per il multiprocessing. Nei linguaggi di programmazione testuali, per rendere multithreaded un'applicazione, è necessario creare più thread e scrivere il codice di comunicazione fra gli stessi. LabVIEW invece riconosce automaticamente le opportunità di multithreading nei VI e il sistema di esecuzione gestisce per voi la comunicazione.

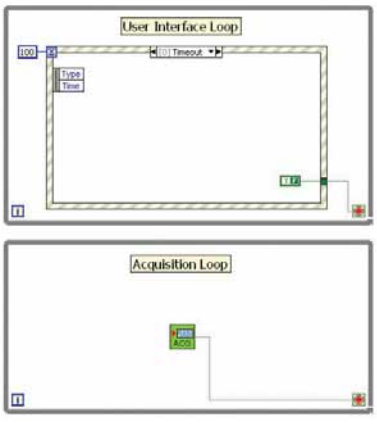

Fig. 1

L'esempio seguente sfrutta il sistema di esecuzione multithreaded di LabVIEW.

In questo VI (fig.1), LabVIEW capisce che può eseguire due loop indipendentemente e, in ambienti multiprocessore o hyperthreaded, anche simultaneamente.

ESEMPIO DI ESECUZIONE PARALLELA NELLA RICERCA DI NUMERI PRIMI

L'esempio seguente (fig. 2) calcola i numeri primi maggiori di due.

Lo schema a blocchi valuta tutti i numeri dispari fra tre e **Limit** e determina se sono primi. Il Ciclo For interno restituisce TRUE se qualche numero divide il termine con resto zero.

Il Ciclo For interno è gravoso in termini computazionali perché non include alcun I/O o funzione di attesa. L'architettura di questo VI impedisce a LabVIEW di sfruttare i benefici del funzionamento parallelo, perché esiste un

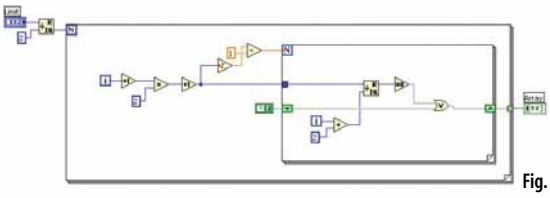

Fig. 2

ordine obbligato nell'esecuzione di ogni operazione. Tale ordine è forzato dal flusso di dati e non esistono altri possibili ordini di esecuzione, perché ogni operazione deve attendere tutti i suoi ingessi per essere eseguita.

Si possono comunque apportare delle modifiche al VI. L'esecuzione parallela richiede che nessuna iterazione del ciclo debba dipendere da altre iterazioni. Se si soddisfa questa condizione, si possono distribuire le iterazioni del ciclo fra due cicli. Tuttavia, un vincolo di LabVIEW è che nessuna iterazione di un ciclo può iniziare prima che l'iterazione precedente sia finita. Si può dividere il processo in due cicli dopo avere determinato che questo vincolo non è necessario.

Nella figura seguente (fig. 3), l'esempio di ricerca parallela dei numeri primi divide il processo in due cicli. Il ciclo superiore valuta metà dei numeri dispari e quello inferiore l'altra metà.

Su un computer multiprocessore, la versione parallela risulta più efficiente, perché LabVIEW può eseguire simultaneamente il codice dei due cicli.

Notate che i VI di questi due esempi non includono codice per la gestione esplicita dei thread. Il paradigma di programmazione a flusso di dati di LabVIEW permette al sistema di eseguire i due cicli in thread differenti. In molti linguaggi di programmazione testuali è necessario creare e gestire esplicitamente i thread.

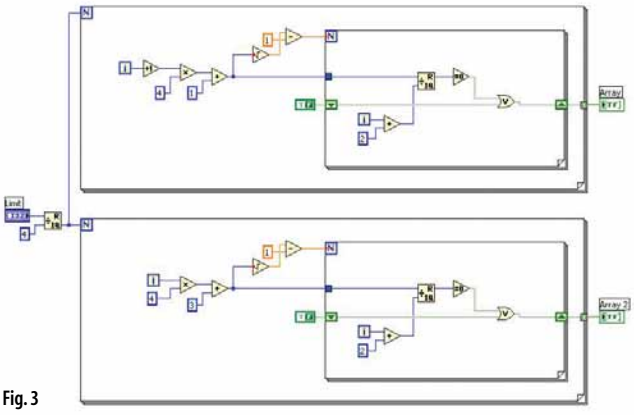

Fig. 3

IL MODELLO DI THREADING DI LABVIEW

La prima versione multithreaded di LabVIEW è stata LabVIEW 5.0. LabVIEW 5.0 separava il thread dell'interfaccia utente dai thread di esecuzione. Gli schemi a blocchi venivano eseguiti in uno o più thread e i pannelli frontali erano aggiornati in un altro thread. Il sistema operativo lavorava in multitasking preemptive, garantendo il tempo processore al thread di esecuzione, quindi al thread dell'interfaccia utente, e così via.

Prima di LabVIEW 7.0, LabVIEW allocava di default un solo thread per ogni singolo sistema di esecuzione e valore di priorità; ciò voleva dire che tutte le attività in tutti i VI appartenenti a quel sistema di esecuzione dovevano attendere insieme. LabVIEW 7.0 ha aumentato a quattro il numero di default dei thread allocati per ogni singolo sistema di esecuzione e valore di priorità; quindi, se un thread esegue un'istruzione che provoca un'attesa, l'esecuzione delle altre sezioni del diagramma a blocchi può proseguire.

Oltre al multitasking preemptive del sistema operativo, LabVIEW si serve di una forma di multitasking cooperativo. Durante la compilazione, LabVIEW analizza i VI alla ricerca dei gruppi di nodi che si possono eseguire insieme nei cosiddetti *clump*. Ogni combinazione di sistemi di esecuzione e livelli di priorità ha una struttura dati basata su una coda di run che contiene i clump eseguibili insieme. Quando il sistema operativo attiva un thread, esso recupera ed esegue un clump dalla coda di run. Quando il sistema operativo termina l'esecuzione, esso memorizza nella coda di run altri clump che soddisfano le condizioni di input, permettendo l'esecuzione del diagramma a blocchi in uno qualsiasi dei quattro thread di esecuzione disponibili.

Se lo schema a blocchi include un sufficiente grado di parallelismo, può essere eseguito simultaneamente in tutti i quattro thread.

LabVIEW non assegna permanentemente clump di codice a un particolare thread. LabVIEW può eseguire infatti un clump usando un thread differente la prossima volta che si esegue il VI.

ESEMPIO DEI NUMERI PRIMI IN C++

Il seguente esempio di codice descrive come scrivere un programma multithreaded in un linguaggio di programmazione testuale, riscrivendo in C++ l'esempio di esecuzione parallela nella ricerca di numeri primi visto in precedenza. L'esempio mostra il tipo di sforzo richiesto per scrivere il codice di gestione dei thread e illustra la programmazione speciale necessaria per proteggere i dati condivisi dai thread.

Il codice d'esempio seguente è stato scritto e testato in Microsoft Visual C++ 6.0. La versione basata su un singolo thread, che sfrutta gli stessi algoritmi del precedente esempio scritto in LabVIEW, si presenterebbe così:

```
// Versione Single-threaded.

void __cdecl CalculatePrimes(int numTerms) {
bool *resultArray = new bool[numTerms/2];
for (int i=0; i&lt;numTerms/2; ++i) {
int primeToTest = i*2+3; // Start with 3, then add 2 each iteration.
bool isPrime = true;
for (int j=2; j&lt;=sqrt(primeToTest); ++j) {
if (primeToTest % j == 0) {
isPrime = false;
break;
}
}
resultArray[i] = isPrime;
}
ReportResultsCallback(numTerms, resultArray);
delete [] resultArray;
}
```

Non esiste alcun parallelismo in questa versione a thread singolo del codice, che utilizzerebbe così il 100 per cento di un solo processore virtuale senza coinvolgere l'altro. Per muoversi nella direzione di una logica parallela di funzionamento, l'applicazione deve promuovere thread addizionali e distribuire il lavoro.

Il codice seguente è l'esempio di una versione multithreaded preliminare dello stesso esempio:

```
struct ThreadInfo {
int numTerms;
bool done;
bool *resultArray;
};
static void __cdecl CalculatePrimesThread1(void*);
static void __cdecl CalculatePrimesThread2(void*);
```

```
void __cdecl CalculatePrimesMultiThreaded(int
numTerms) {

// Inizializza l'informazione da passare ai thread.

ThreadInfo threadInfo1, threadInfo2;
threadInfo1.done = threadInfo2.done = false;
threadInfo1.numTerms = threadInfo2.numTerms =
numTerms;
threadInfo1.resultArray = threadInfo2.resultArray =
NULL;

// Inizia due thread

_beginthread(CalculatePrimesThread1,NULL,
&amp;threadInfo1);
_beginthread(CalculatePrimesThread2,NULL,
&amp;threadInfo2);

// Attende che i thread finiscano l'esecuzione.

while (!threadInfo1.done || !threadInfo2.done)
Sleep(5);

// Colleziona i risultati.

bool *resultArray = new bool[numTerms/2];
for (int i=0; i&lt;numTerms/4; ++i) {
resultArray[2*i] = threadInfo1.resultArray[i];
resultArray[2*i+1] = threadInfo2.resultArray[i];
}
ReportResultsCallback(numTerms, resultArray);
delete [] resultArray;
}
static void __cdecl CalculatePrimesThread1(void *ptr) {
ThreadInfo* tiPtr = (ThreadInfo*)ptr;
tiPtr-&gt;resultArray = new bool[tiPtr-&gt;numTerms/4];
for (int i=0; i&lt;tiPtr-&gt;numTerms/4; ++i) {
int primeToTest = (i+1)*4+1;
bool isPrime = true;
for (int j=2; j&lt;=sqrt(primeToTest); ++j) {
if (primeToTest % j == 0) {
isPrime = false;
break;
}
}
tiPtr-&gt;resultArray[i] = isPrime;
}
tiPtr-&gt;done=true;
}
static void __cdecl CalculatePrimesThread2(void *ptr) {
ThreadInfo* tiPtr = (ThreadInfo*)ptr;
tiPtr-&gt;resultArray = new bool[tiPtr-&gt;numTerms/4];
for (int i=0; i&lt;tiPtr-&gt;numTerms/4; ++i) {
int primeToTest = (i+1)*4+3;
bool isPrime = true;
for (int j=2; j&lt;=sqrt(primeToTest); ++j) {
if (primeToTest % j == 0) {
isPrime = false;
break;
}
}
tiPtr-&gt;resultArray[i] = isPrime;
}
tiPtr-&gt;done=true;
}
```

In questo esempio, la funzione CalculatePrimesMultiThreaded() crea due thread usando la funzione _beginthread(). Il primo thread richiama la funzione CalculatePrimesThread1(), che testa metà dei numeri dispari. Il secondo thread richiama la funzione CalculatePrimesThread2 e testa l'altra metà dei numeri dispari. Il thread originale, che esegue ancora la funzione CalculatePrimesMultiThreaded(), deve attendere che i due thread in esecuzione finiscano, creando la struttura dati ThreadInfo per ogni thread e passandola alla funzione _beginthread(). Quando l'esecuzione di un thread finisce, viene scritto true in ThreadInfo::done.

Il thread originale deve interrogare continuamente ThreadInfo::done finché il thread originale legge un valore true per ogni thread di calcolo. A quel punto il thread originale può accedere con sicurezza ai risultati del calcolo. Il programma colleziona quindi i valori in un singolo array, in modo che siano identici all'output della versione a singolo thread di questo esempio.

Nota: il passo precedente non è mostrato nell'esempio LabVIEW, ma farlo è banale.

Quando si scrive codice multithreading in un linguaggio di programmazione testuale come il C++, è necessario proteggere le locazioni dei dati a cui più thread possono accedere. Se non si proteggono le locazioni dei dati, le conseguenze sono imprevedibili e spesso difficili da riprodurre e localizzare. In ogni caso in cui l'assegnazione di tiPtr->done a true non sia atomica, è necessario usare un mutex per proteggere l'assegnazione e l'accesso.

È possibile migliorare ancora l'esempio precedente. In particolare, non vi è ragione per iniziare due thread addizionali e lasciarne uno inattivo. Poiché questo esempio contiene codice per un computer con due processori virtuali, è possibile iniziare un thread addizionale ed usare quello originale per eseguire metà dei calcoli. Inoltre, è possibile passare entrambi i thread attraverso la stessa funzione anziché scrivere due volte quella che, fondamentalmente, è la stessa funzione. È necessario passare un parametro addizionale alla funzione per indicare in quale thread può

essere eseguita. **Nota**: si può eseguire la stessa ottimizzazione nell'esempio scitto in LabVIEW creando un subVI rientrante che contiene il Ciclo For. Il VI chiamante dovrà avere due istanze del subVI invece di due Cicli For.

Il codice segunte rappresenta un'applicazione multithreaded più efficiente:

```
struct ThreadInfo2 {
int threadNum;
int numTerms;
bool done;
bool *resultArray;
};
static void __cdecl CalculatePrimesThread(void*);
void __cdecl CalculatePrimesMultiThreadedSingleFunc(int numTerms) {

// Inizializza le informazioni da passare ai thread.

ThreadInfo2 threadInfo1, threadInfo2;
threadInfo1.done = threadInfo2.done = false;
threadInfo1.numTerms = threadInfo2.numTerms = numTerms;
threadInfo1.resultArray = threadInfo2.resultArray = NULL;
threadInfo1.threadNum = 1;
threadInfo2.threadNum = 2;

// Inizia un thread

_beginthread(CalculatePrimesThread, NULL, &amp;threadInfo1);

// Usa il thread per l'altro ramo invece di lanciare un altro thread.

CalculatePrimesThread(&amp;threadInfo2);

// Questo thread potrebbe finire per primo. In questo caso, aspetta l'altro.

while (!threadInfo1.done)
Sleep(5);

// Colleziona i risultati.

bool *resultArray = new bool[numTerms/2];
for (int i=0; i&lt;numTerms/4; ++i) {
resultArray[2*i] = threadInfo1.resultArray[i];
resultArray[2*i+1] = threadInfo2.resultArray[i];
}
ReportResultsCallback(numTerms, resultArray);
delete [] resultArray;
}
static void __cdecl CalculatePrimesThread(void *ptr) {
ThreadInfo2* tiPtr = (ThreadInfo2*)ptr;
int offset = (tiPtr-&gt;threadNum==1) ? 1 : 3;
tiPtr-&gt;resultArray = new bool[tiPtr-&gt;numTerms/4];
for (int i=0; i&lt;tiPtr-&gt;numTerms/4; ++i) {
int primeToTest = (i+1)*4+offset;
bool isPrime = true;
for (int j=2; j&lt;=sqrt(primeToTest); ++j) {
if (primeToTest % j == 0) {
isPrime = false;
break;
}
}
tiPtr-&gt;resultArray[i] = isPrime;
}
tiPtr-&gt;done=true;
}
```

Questa versione del programma è più efficiente perché si possono attivare meno thread e aspettare meno tempo in attesa che i thread operativi completino il lavoro. Inoltre, poiché il codice esegue il calcolo all'interno di una singola funzione, c'è meno codice duplicato, cosa che rende l'applicazione più facile da mantenere. Tuttavia, una grossa parte del codice esegue la gestione dei thread, cosa che è intrinsecamente insicura e difficile da testare e debuggare.

UN ESEMPIO PIÙ COMPLESSO IN LABVIEW

L'esempio di gestione parallela nella ricerca di numeri primi è un semplice esempio che dimostra molti dei concetti coinvolti nel multiprocessing. La maggior parte delle applicazioni reali non è così semplice. Anche se fosse necessario scrivere un generatore di numeri primi, l'algoritmo utilizzato negli esempi presenti in questo articolo non rappresenterebbe un metodo efficiente.
Questo paragrafo descrive un altro algoritmo impegnativo dal punto di vista computazionale che può trarre vantaggio dal sistema di esecuzione multithreaded di LabVIEW.
Lo schema a blocchi in fig. 4 calcola pi greco con il numero di cifre decimali specificato.

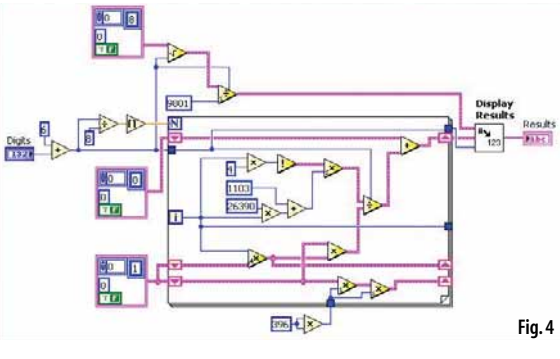

Fig. 4

I fili rosa sono cluster che servono come valori numerici a precisione arbitraria. Se desiderate calcolare pi greco fino a 1000 cifre decimali, non è più sufficiente un valore in virgola mobile, a precisione estesa. I nodi che assomigliano a funzioni di LabVIEW ma che operano sui cluster sono VI che eseguono calcoli sui numeri a precisione arbitraria.
Anche questo VI risulta impegnativo da un punto di vista computazionale. Esso calcola pi greco in base alla formula di fig. 5.

Fig. 5

$$\frac{1}{\Pi} = \frac{\sqrt{8}}{9801} \sum_{i=0}^{\infty} \frac{(4i)!\,[1103 + 26390i]}{(i!)^4\, 396^{4i}}$$

Data la complessità numerica di questa equazione, sarebbe difficile, nella maggior parte dei linguaggi di programmazione testuali, scrivere un programma che utilizzi entrambi i processori logici su un computer hyperthreaded. LabVIEW, invece, può analizzare la precedente equazione e riconoscerla come un'opportunità per il multiprocessing.
Quando LabVIEW analizza lo schema a blocchi di fig. 6, identifica il parallelismo intrinseco. Notate che non esiste alcuna dipendenza nel flusso di dati fra le parti evidenziate in rosso e quelle in verde.
A prima vista, sembra che in questo schema a blocchi sia presente un basso livello di parallelismo. Sembra infatti che solo due operazioni possano essere eseguite in parallelo con il resto del VI, e tali due operazioni sono eseguite una sola volta. In effetti, la radice quadrata richiede

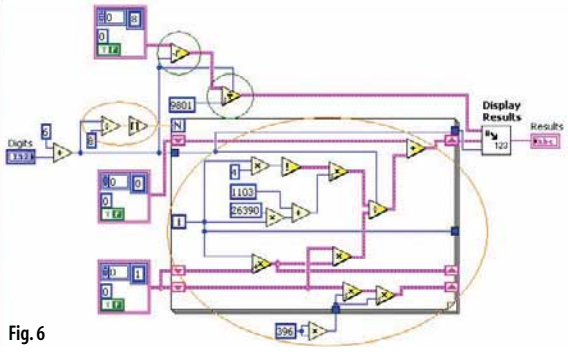

Fig. 6

tempo, e questo VI dedica circa metà del tempo di esecuzione totale ad eseguire l'operazione radice quadrata in parallelo con il Ciclo For. Se dividete il VI in due cicli non correlati, LabVIEW può riconoscere la possibilità di gestire i thread in parallelo usando entrambi i processori logici, con un elevato guadagno in termini di prestazioni. Questa soluzione funziona su un computer hyperthreaded come su un computer multiprocessore.
È difficile analizzare un'applicazione per identificare il parallelismo. Se scrivete un'applicazione in un linguaggio di programmazione testuale come il C++, dovete identificare le opportunità di esecuzione parallela e creare ed eseguire esplicitamente i thread per sfruttare le potenzialità dei computer hyperthreaded. Il vantaggio di usare LabVIEW è che in molti casi, come nel nostro esempio, LabVIEW identifica automaticamente le opportunità di esecuzione parallela e permette di utilizzare entrambi i processori logici. In certe applicazioni, è vantaggioso usare un algoritmo che crei opportunità di esecuzione parallela, in modo da poter tratte vantaggio dalle capacità di multithreading offerte da LabVIEW.

La tabella seguente riporta il tempo impiegato dal VI per calcolare pi greco con 1500 cifre decimali senza alcuna forma di ottimizzazione delle prestazioni:

Hyperthreading: 27,9 s
Senza hyperthreading: 25,3 s

Notate che il computer hyperthreaded risulta più lento. Poiché i due processori logici di un computer hyperthreaded condividono una cache, è più facile sovraccaricare le linee della cache e provocare problemi.
LabVIEW memorizza le informazioni necessarie per il debugging in un'area di memoria a cui tutti i thread di esecuzione devono accedere, sia in scrittura che in lettura. In alcune situazioni, i thread sono eseguiti simultaneamente su processori logici differenti, e un thread deve leggere le informazioni di debug mentre l'altro thread vi sta accedendo in scrittura.
Se due thread eseguono operazioni sulla stessa linea di cache in un computer hyperthreaded, la perdita di prestazioni può essere sostanziale.
Potete migliorare le prestazioni del VI disabilitando il debugging. Potete disabilitare il debugging per una parte del VI critica per le prestazioni e quindi riabilitarlo se dovete debuggare quella parte del VI.
La tabella seguente riporta il tempo di esecuzione richiesto dal VI per calcolare pi greco con 1500 cifre decimali dopo avere disabilitato il debugging per l'intero VI:

Hyperthreading: 21,6 s
Senza hyperthreading: 21,5 s

Notate che il computer hyperthreaded ha quasi le stesse prestazioni del computer senza hyperthreading. La suddivisione del carico di lavoro non sembra migliorare le prestazioni, anche se LabVIEW lavora in parallelo.

La tabella seguente riporta le prestazioni dell'esempio di esecuzione parallela nella ricerca di numeri primi in LabVIEW. Il VI ha trovato tutti i numeri primi nei primi 400.000 numeri naturali su un computer Windows XP con Pentium 4 a 3,06 GHz.

Hyperthreading senza debugging: 1,97 s
Hyperthreading con debugging: 10,47 s

La differenza con il debugging abilitato è significativa, ma tale differenza si verifica solo quando le due sezioni parallele vengono eseguite nello stesso VI. Se scriveste l'applicazione con più di un VI o se il VI fosse abbastanza complesso da gestire la maggior parte delle informazioni di debugging di sezioni distinte in diverse linee della cache, la differenza con il debugging abilitato non sarebbe così significativa.

Un'altra ottimizzazione che può migliorare le prestazioni su un computer hyperthreaded è quella di rendere rientranti alcuni subVI.Diversi thread possono chiamare simultaneamente lo stesso subVI se questo è rientrante.
È importante capire come lavora l'esecuzione rientrante nel sistema di esecuzione di LabVIEW quando si ottimizza

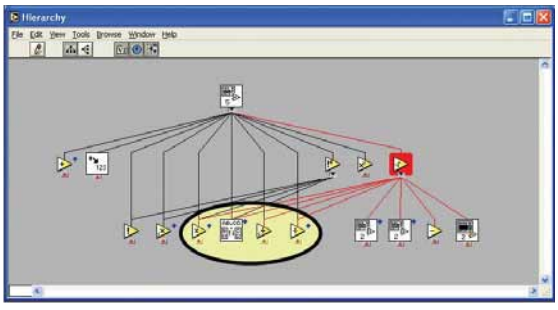

Fig. 7

un VI per un computer multiprocessore o hyperthreaded. Marcare erroneamente un VI come rientrante può provocare inutili ritardi, specialmente in un ambiente multiprocessore. Per determinare quali subVI possono essere rientranti, trovate i VI richiamati da entrambi i rami dell'esecuzione parallela. La fig. 7 mostra la gerarchia del VI: la funzione Square Root e il Ciclo For operano in parallelo.
Il VI di livello superiore richiama le funzioni Square Root, Add e Multiply e i VI Multiply Scalar e Divide Scalar in un Ciclo For. Queste sono due parti ideali da rendere rientranti, perché LabVIEW richiama queste parti da thread differenti.
La tabella seguente riporta il tempo di esecuzione richiesto dal VI per calcolare pi greco con 1500 cifre decimali dopo avere marcato come rientranti le funzioni Add e Multiply e i VI Multiply Scalar e Divide Scalar:

Hyperthreading: 20,5 s
Senza hyperthreading: 21,9 s

Marcare come rientranti i VI rende l'esecuzione dei VI leggermente più veloce sul computer hyperthreading.
Il tempo di esecuzione per il computer senza hyperthreading era leggermente più lento dopo avere reso rientranti i quattro VI, perché LabVIEW crea uno spazio dati addizionale quando richiama VI rientranti.

Potete continuare ad ottimizzare un'applicazione usando il VI Profiler e poi definire che qualche VI venga eseguito con priorità di subroutine. Potete inoltre marcare più VI come rientranti. La tabella seguente riporta il tempo di esecuzione richiesto dal VI per calcolare pi greco con 1500 cifre decimali dopo diverse azioni di ottimizzazione:

Hyperthreading: 18,0 s
Senza hyperthreading: 20,4 s

Questi risultati indicano un miglioramento di prestazioni del 10 percento sul computer hyperthreaded. Notate che non è necessario modificare il codice del VI per ottenere tale miglioramento. Dovete semplicemente disabilitare il debugging, rendere rientranti alcuni VI ed eseguire alcuni VI con priorità di subroutine. Complessivamente, queste modifiche permettono di ottenere un miglioramento di prestazioni pari circa al 35 percento su un computer hyperthreaded.

LABVIEW OTTIMIZZATO PER LAVORARE SU COMPUTER HYPERTHREADED

Già dalla versione 7.1 LabVIEW è stato ottimizzato per condividere la cache fra i due processori logici di un computer hyperthreaded. Quando è stato introdotto l'hyperthreading, è stato necessario apportare modifiche opportune alle applicazioni multithreaded, incluso LabVIEW, per sfruttare il nuovo potenziale offerto in termini di prestazioni.

La tabella seguente riporta i risultati in secondi di esecuzione dell'esempio di parallelismo sui numeri primi.
Il VI ha trovato tutti i numeri primi nei primi 400.000 numeri naturali su un computer Windows XP con Pentium 4 a 3,06 GHz.

Versione	Hyperthreading	Senza hyperthreading
LabVIEW 7.0	6,77	3,39
LabVIEW 7.1	1,91	3,40

Tutti i benchmark di prestazioni nella parte precedente sono stati testati su computer Windows, ma anche i sistemi Linux possono essere eseguiti su computer hyperthreaded. I test dimostrano che LabVIEW 7.1 supera in prestazioni le precedenti versioni di LabVIEW su sistemi Linux multiprocessore su computer hyperthreaded.

Note sull'autore
Kamran Shah, Group Manager, LabVIEW Product Marketing, National Instruments