Panoramica sulle tecnologie e sugli strumenti per la programmazione parallela (II parte)

Gianpaolo Bottoni, Maurizio Cremonesi
CILEA, Segrate

Abstract
La varietà delle piattaforme di calcolo adatte al calcolo parallelo impone lo sviluppo di metodologie di programmazione che permettano di realizzare programmi efficienti su più piattaforme mantenendo alta la leggibilità del codice e la portabilità su calcolatori di architettura diversa. Spesso l'uso di paradigmi di parallelizzazione di basso livello si rivela altrettanto efficace di altri metodi basati su paradigmi più astratti.

La prima parte dell'articolo si è conclusa con una nota di ottimismo perché non appare più così remota la possibilità di poter disporre di un sistema di direttive di parallelizzazione del tutto portabile sulle più diverse piattaforme di calcolo intensivo. Si può ben sperare che entro un tempo relativamente breve lo standard OpenMP affiancherà, o forse sostituirà, le direttive di parallelizzazione proprie delle varie piattaforme.

Le direttive di parallelizzazione hanno lo scopo di guidare con precisione e consapevolezza uno strumento automatico, il compilatore, che già di per sé è in grado di analizzare il programma ed individuare le parti di codice in cui la memoria su cui si esegue il lavoro può essere suddivisa in parti contigue ma sufficientemente indipendenti da non provocare problemi di incoerenza e generazione di risultati errati.

Questa modalità di parallelizzazione è praticamente realizzabile solo in ambienti di calcolo multiprocessore a memoria condivisa. Nella generalità degli algoritmi l'assegnazione efficiente e coerente della memoria che competerebbe ai singoli processori è una operazione troppo complessa per poter essere realizzata automaticamente. Anche fatta manualmente richiede qualche informazione in più di quella esprimibile con la semplice sintassi propria delle direttive. Il paradigma che permette di implementare questo tipo di parallelizzazione, portabile negli ambienti di calcolo parallelo sia a memoria condivisa che a memoria distribuita è l'High Performance Fortran ed ha anch'esso buone possibilità di divenire uno standard, proprio per le sue caratteristiche di portabilità e soprattutto per il fatto di essere più semplice da utilizzare rispetto alle librerie di scambio di messaggi, soprattutto se si prende in considerazione la parallelizzazione di programmi già esistenti.

Analizzando un po' più a fondo come viene realizzata la parallelizzazione a memoria condivisa con direttive sulle macchine per il calcolo parallelo del CILEA, ma in generale su tutte le piattaforme di architettura simile si scopre che quando si incontra una direttiva di parallelizzazione il flusso di esecuzione del programma, fino ad allora unitario, in qualche modo si divide in più *rivoli* (in inglese *threads* ossia *filoni di lavoro* ... metafora meno poetica) tanti quanti ne sono stati fissati nel programma. La cosa importante da tenere in considerazione è che questi thread non sono processi a sé stanti, ma tutti convivono in qualche modo nell'ambiente in cui sono stati generati, in particolare condividendo ogni indirizzo di memoria del processo genitore. Il programmatore dev'essere ben avvertito di questo e deve tenerlo sempre ben presente. Spesso i problemi di conflitto di accesso alla memoria nascono quando il programmatore non ha analizzato il programma sufficientemente e non è stato attento a privatizzare tutte le variabili che non possono essere condivise.
Fortunatamente però questi thread sono sufficientemente distinti da poter essere eseguiti da

processori diversi. Quando le risorse di calcolo non sono invece adeguate alle necessità computazionali del problema, anche i processori della macchina vengono condivisi da più thread e questo interessa un po' meno della condivisione della memoria, in quanto significa un aumento del tempo di ottenimento dei risultati.

Semplificando un po', i thread sono sottoprocessi contenuti nel processo principale e con la particolarità di essere dotati di vita propria indipendente. Per le loro caratteristiche i thread sono realizzabili proficuamente in tutte le piattaforme di calcolo multiprocessore a memoria condivisa, sia con sistema operativo Windows sia Unix.

I thread allora possono diventare un modo per realizzare programmi paralleli portabili su più piattaforme di calcolo, da Windows a Unix.

Purtroppo si deve osservare che anche in questo caso i thread utilizzabili in ambiente Windows sono diversi dai thread utilizzabili in Unix ed in effetti la portabilità è affidata ad una somiglianza piuttosto superficiale tra i due mondi.

Corrispondenza tra alcune funzioni POSIX/threads e threads/NT

POSIX	NT
pthread_create	CreateThread
pthread_exit	ExitThread
pthread_join	WaitForSingleObject
pthread_cancel	TerminateThread
pthread_self	GetCurrentThreadId
pthread_continue	ResumeThread
pthread_suspend	SuspendThread
pthread_mutex_init	InitializeCriticalSection
pthread_mutex_destroy	DeleteCriticalSection
pthread_mutex_lock	EnterCriticalSection
pthread_mutex_trylock	TryEnterCriticalSection
pthread_mutex_unlock	LeaveCriticalSection

Tabella 5

Fortunatamente per quanto riguarda la realtà del CILEA un passo importante verso la portabilità lo ha compiuto la HP anni fa, quando ha abbandonato la propria libreria CPSlib di gestione dei thread in favore della ben più portabile libreria POSIX threads. Anche in questo caso le somiglianze tra i due sistemi sono solo apparenti, ma chi è interessato può trovare un aiuto alla conversione del codice nel manuale citato in bibliografia.

Comunque le similitudini tra threads Windows e POSIX threads, di cui la *Tabella 5* dà un'idea, permettono di generare interfacce abbastanza semplici se i threads vengono applicati ai programmi di calcolo tecnico-scientifico, dove la necessità è quella di riuscire a far lavorare più processori in modo costruttivo e le difficoltà si riducono a capire quali sono le variabili da duplicare (privatizzare ad ogni thread) e a trovare i punti di sincronizzazione. Le similitudini tra threads Windows e POSIX threads non reggono più se la necessità è invece quella di controllare in dettaglio il lavoro dei vari thread, in programmi che assomiglierebbero più ad un sistema operativo che ad un codice di simulazione di fenomeni fisici.

La libreria POSIX threads è scritta in C ed è fatta per essere utilizzata da programmi C. Questo fa da repellente ai numerosi programmatori di codici applicativi Fortran e non aiuta la logica di questo articolo, orientato alla parallelizzazione di programmi Fortran. D'altra parte i thread sono nati per permettere la realizzazione di programmi di gestione operativa e non codici applicativi.

Sfortunatamente non esiste una libreria di interfaccia tra la libreria POSIX threads e il Fortran, ad eccezione di un prodotto proprietario di un importante produttore di elaboratori, che però non è pubblico né disponibile per le altre piattaforme di calcolo. Gli utenti HP, così come tutti gli altri, devono quindi cavarsela da soli se proprio vogliono applicare i thread a programmi Fortran, scavalcando la più usuale interfaccia fornita dalle direttive di parallelizzazione.

Usare una libreria C da Fortran

Tutti i programmatori sanno quanto siano distanti i due linguaggi di programmazione C e Fortran. In particolare è universalmente noto che nei due linguaggi i dati sono passati alle funzioni e procedure in modo completamente diverso: per indirizzi in Fortran e (usualmente) per valore in C.
Apparentemente non sembra quindi che ci sia alcuna possibilità di comunicazione tra i due linguaggi di programmazione. Qualche anno fa

si era diffuso nel mondo della programmazione amatoriale (e non) un programmino di pubblico dominio che rendeva disponibile il linguaggio Fortran senza alcuna licenza e praticamente su tutte le piattaforme di calcolo, in quanto semplicemente convertiva sorgenti Fortran in C. La sostituzione del Fortran 77 con il Fortran 90 (e oltre) ha fatto perdere a questa utilità molta della sua importanza. Questo suona un po' come una rivincita del Fortran, che non può più essere considerato come se fosse un sottoinsieme del linguaggio C, oltre ad essere tranquillizzante per le aziende produttrici di compilatori, che non rischiano di vedersi portar via una fetta rilevante di mercato.
Tutto ciò però fa intuire che, malgrado le differenze, non dovrebbe essere impossibile far comunicare tra loro procedure Fortran e C. In ogni caso in ambiente HP-UX 11.0 c'è la possibilità di interfacciare il C da Fortran in modo semplice ed efficace e di questo si è tenuto conto per sviluppare una prima interfaccia con lo scopo di dimostrare più che altro la fattibilità di parallelizzare codici Fortran utilizzando i POSIX threads (pthreads).
In particolare nello sviluppare questa interfaccia ci si è avvalsi della direttiva HP ALIAS, che permette di assegnare un nome *in stile Fortran* ad una procedura C, descrivendo anche come devono essere passati gli argomenti.
Inoltre ci si è avvalsi dell'uso dei moduli per definire le variabili di servizio e le dichiarazioni delle funzioni dell'interfaccia.

pthF90

L'interfaccia dimostrativa che permette di usare i pthreads in Fortran è stata chiamata *pthF90*, con nessun altro significato se non il fatto di essere un nome semplice che richiama subito alla mente gli attori coinvolti.
Le funzionalità sin qui implementate comprendono l'attivazione dei thread, l'attesa della loro terminazione, la sincronizzazione con cancelli, o semafori, in gergo: mutex. Poche altre funzionalità, meno interessanti al programmatore Fortran scientifico, sono implementate.
Soprattutto per mostrare la semplicità dell'interfaccia, sperando così di non spaventare gli eventuali interessati, ma anzi convincendoli della bontà della tecnica, vengono presentate poche ma significative parti del codice realizzato.

```
#include<pthread.h>     (1)
#include<stdlib.h>

int pthF90_create( int *thf90, &        (2)
   void *(*start_routine)(void *), int *argf90)
   /* Emula il comportamento di  pthread_create  da Fortran

      Prima di generare il nuovo thread si legge la variabile
      d'ambiente CPS_STACK_SIZE, di cui si tiene conto
      richiamando la pthread_attr_setstacksize.

      Argomenti:
      integer thf90   -  numero del thread generato (output)
      subroutine start_routine  -  codice da eseguire  (input)
      integer argf90  -  argomento da passare alla routine

      Valore:
      =  0 - va tutto bene
      <> 0 - errore nella generazione del thread
   */
{
   pthread_attr_t attr = NULL;
   int rc, stackval;
   char *varval;

   pthread_attr_init(&attr);          (3)
   varval = getenv("CPS_STACK_SIZE");
   if ( varval != NULL)
   {
       stackval = atoi(varval);
       stackval = stackval * 1024;
   } else stackval = 0;
   pthread_attr_setstacksize(&attr,stackval);
   rc = pthread_create(thf90, &         (4)
              &attr,start_routine, argf90);
   return(rc);          (5)
}
```

Tabella 6

Nella *Tabella 6* è presentata la codifica della procedura C che nella libreria *pthF90* implementa l'interfaccia per la funzione `pthread_create`, che permette di attivare un nuovo thread.

Le istruzioni (1), come ben sa ogni programmatore C, sono obbligatorie per accedere alle definizioni delle funzioni e delle variabili di ogni specifica libreria necessaria al programma. In particolare si fa riferimento qui al file di dichiarazioni `pthread.h`, proprio delle funzioni pthreads.

L'istruzione (2) dichiara che la funzione `pthF90_create` ha 3 argomenti, di cui il primo e l'ultimo sono di tipo intero e vengono passati per indirizzo (in C questo usualmente significa che la funzione vi ritorna un valore). Il secondo argomento invece è il riferimento ad una procedura, che non ritorna nessun valore (in Fortran questo significa che è una SUBROUTINE e non una FUNCTION).
Gli argomenti di `pthF90_create` sono analoghi a quelli della `pthread_create` tranne che in quest'ultima si aggiunge un altro argomento che rappresenta gli attributi da "assegnare" al nuovo thread (vedi `man pthread_create`). Per non avere troppe pretese la funzione Fortran non ammette attributi, ma come unica conces-

sione viene riassegnato, con l'istruzione (3) e successive, l'attributo STACKSIZE, nel caso si trovi definita la variabile d'ambiente CPS_STACK_SIZE. Questo è coerente con l'uso che della stessa variabile si fa in ambiente HP-UX e garantisce che il thread da attivare sarà utile anche ai programmi di calcolo intensivo che necessitano di una dimensione dello stack superiore agli 80 Mbyte.

L'istruzione (4) attiva finalmente il nuovo thread, passando come codice da eseguire la procedura start_routine, che sarà implementata come una normale SUBROUTINE Fortran con un solo argomento, di tipo INTEGER(4).
L'istruzione (5) mostra che il valore di ritorno di questa interfaccia è sempre il valore della funzione pthread_create, che ritorna zero se il thread è stato attivato correttamente.

```
include "pthF90.i"      (1)

module condiviso     (2)
   implicit none
   integer :: numproc, nsomme
   real(8) :: h, pi
end module condiviso

program pi_greco
   use modpthF90     (3)
   use condiviso
   implicit none
   external :: lavoro     (4)
   real(8) :: x, a
   integer :: nn, i, np, ios
   integer,allocatable,dimension(:) :: &
               thread_id, pid    (5)

   nn = pthf90_max_threads(numproc) (6)

   allocate(thread_id(numproc),STAT=ios)
   allocate(pid(numproc),STAT=ios)

   pi = 0.0d0    (7)
   nsomme = 1000000
   h = 1.0d0 / nsomme

   nn = pthf90_mutex_init(1) (8)

   do ii = 1, numproc    (9)
      pid(ii) = ii
      nn = pthf90_create(thread_id(ii), &
                         lavoro,pid(ii))
   end do

   do ii = 1, numproc    (10)
      nn = pthf90_join(thread_id(ii))
   end do

   nn = pthf90_mutex_destroy(1) (11)
   deallocate(thread_id,pid,STAT=ios)

   print*,"pi_greco = ",pi_greco
   stop
end program pi_greco

subroutine lavoro(nn)
   use modpthF90  (12)
   use condiviso
   implicit none
   integer, intent(in) :: pid

   integer :: i, nn
   real(8) :: f, sum, x, a

   f(a) = 4.0d0 / ( 1.0d0 + a * a )

   sum = 0.0d0
   do i = pid, nsomme, numproc
      x = h * ( dble(i) - 0.5d0 )
      sum = sum + f(x)     (13)
   end do

   nn = pthf90_mutex_lock(1)
   pi = h * sum + pi     (14)
   nn = pthf90_mutex_unlock(1)

   return
end subroutine lavoro
```

Tabella 7

L'esempio presentato nella *Tabella 7* non ha altro scopo che illustrare come viene utilizzata l'interfaccia *pthF90*.
La riga (1) deve necessariamente costituire la prima riga di ogni programma che si avvale della pthF90.
Il modulo contrassegnato da (2) permette di dichiarare la memoria condivisa da tutti i thread.
Qualsiasi procedura, sia essa PROGRAM o SUBROUTINE, che richiama funzioni *pthF90* deve contenere necessariamente l'istruzione (3), ovvero utilizzare il modulo modpthF90. Questo contiene la dichiarazione delle funzioni e delle variabili necessarie all'uso della libreria *pthF90*.
Nel caso particolare il PROGRAM accede anche alla memoria condivisa, perciò si dichiara di utilizzare anche il modulo relativo.
Le variabili thread_id e pid, dichiarate con l'istruzione (5), concettualmente quantità scalari, sono esempi di grandezze da privatizzare, perciò sono dichiarate come vettori.
La funzione pthf90_max_threads dell'istruzione (6) non ha una controparte nella libreria *pthreads* ed è stata scritta al solo scopo di rendere più semplice, portabile ed esplicita la gestione della numerosità dei thread da utilizzare. Questa funzione non fa altro che ritornare il valore della variabile d'ambiente operativo MP_NUMBER_OF_THREADS.
L'istruzione (7) inizializza la memoria condivisa da tutti i thread.

Invece l'istruzione (8) inizializza la memoria necessaria per sincronizzare i thread: in questo esempio semplicissimo è sufficiente poter disporre di un unico *cancello*.
Le istruzioni del ciclo iterativo (9) attivano i thread e passano loro come compito da svolgere la funzione `LAVORO` con l'unico argomento intero `pid`. Al proposito è importante ricordare che questo argomento dev'essere rappresentato da un indirizzo di memoria specifico e distinto per ogni thread attivato. Infatti, come è illustrato dalla *Tabella 6*, l'ultimo argomento della funzione `pthF90_create` è passato per indirizzo.
Questa funzione, dopo avere generato il thread, non attende che questi abbia terminato il proprio compito, ciò che renderebbe tutto sommato inutile l'uso dei thread, invece l'esecuzione del programma prosegue, parallelamente al lavoro di ogni thread attivato.
Indispensabile quindi è prima o poi fissare un punto nel programma dopo il quale si è assolutamente sicuri che tutti i thread attivati hanno terminato il loro compito ed è quindi possibile utilizzare in tutta sicurezza i risultati calcolati.
Le istruzioni identificate da (10) realizzano questo punto di sincronizzazione, per mezzo della funzione di interfaccia `pthF90_join`.
Al di là di questo punto quindi è possibile utilizzare il risultato calcolato con tanta fatica, ma prima è opportuno liberare la memoria che non sarà più utilizzata.
L'istruzione (11) mostra la funzione utilizzabile per liberare la memoria servita per la sincronizzazione, funzione da utilizzarsi sempre in coppia, per così dire, con la (8).
Le istruzioni appena consecutive mostrano invece il rilascio della memoria relativa alle grandezze privatizzate.
La subroutine `lavoro` deve utilizzare anch'essa le funzioni della libreria pthF90, ed inoltre deve avere visibilità della memoria condivisa, perciò dichiara (12) di dover accedere ai moduli corrispondenti.
Ogni istanza attivata della subroutine `lavoro` utilizza la propria copia privata delle variabili in essa dichiarate, perciò le istruzioni (13) possono essere fatte in parallelo da ogni thread senza necessità di sincronizzazione.
Viceversa la variabile `pi` è accessibile da tutti i thread. Per poter effettuare l'aggiornamento di `pi` in tutta sicurezza (14) è indispensabile l'utilizzo del cancello, ovvero della coppia di funzioni `pthF90_mutex_lock` e `pthF90_mutex_unlock`.

L'interfaccia *pthF90* è disponibile sulle piattaforme di calcolo intensivo del CILEA nel direttorio `/mcae/pthF90`.
Prima del suo utilizzo si consiglia di leggere lo script `/mcae/pthF90/pthF90_env.csh`.

Quanto scritto fin qui esemplifica l'uso e pubblicizza la libreria *pthF90*, utile al programmatore che vuole parallelizzare il proprio programma evitando la mediazione dell'uso delle direttive, anche per aumentarne l'efficienza.
Invece non aiuta a diffondere il messaggio che è lo scopo principale di questo articolo, ovvero la necessità di avere e la possibilità di scrivere programmi paralleli il più possibile portabili da un ambiente di calcolo all'altro. Nella realtà del CILEA questo significa poter passare da Windows (NT) a Unix (HP-UX) nel modo più trasparente possibile, ovvero senza possibilmente alterare neppure una riga del nucleo contenente le istruzioni realmente significative del programma.
Tuttavia la spiegazione di come utilizzare l'interfaccia *pthF90* serve a dimostrare che per sviluppare programmi Fortran di calcolo tecnico-scientifico paralleli esiste il modo di sfruttare i thread direttamente, in modo semplice e portabile, almeno nel senso che per fare questo è sufficiente utilizzare le funzionalità più semplici e primitive della libreria *pthreads*, quelle che presumibilmente si possono trovare anche in altri ambienti, ovvero Windows NT.
A questo punto si può riprendere il discorso portabilità tra i due ambienti e rivelare, ma lo si era ormai già capito, che quanto è stato scritto nella prima parte dell'articolo riguardo alla parallelizzazione con i moduli può essere ripetuto senza distinguere se il programma dev'essere realizzato per l'un ambiente o per l'altro: sia il programma che i moduli di servizio sono (quasi) esattamente uguali, tranne il fatto che in Windows si utilizzano i thread nativi, mentre in HP-UX si può utilizzare una libreria di interfaccia che li emula.
A lato dell'interfaccia pthF90 è stata infatti realizzata un'interfaccia che emula i thread di Windows in ambiente Fortran HP-UX, con modalità di utilizzo analoghe a quelle già spiegate per la *pthF90*.
I sorgenti relativi si possono trovare all' indirizzo:
`http://www.cilea.it/servizi/b/00/002`.

```
program test
   use modpubblico
   use parallelo_p1
```

```
   implicit none
   integer, parameter::quanti_th=4
   integer :: np
   type(info_thread), &
       dimension(quanti_th) :: th
   !__inizializzazioni varie__!
   do np=1,quanti_th
      call create_thread( &
                   th(np),main_p1)

   end do
   !__calcoli sequenziali__!
stop
end program test
```

Tabella 8

In particolare nella *Tabella 8* viene presentato il main analogo a quello illustrato dalla *Tabella 4* della prima parte di questo articolo, ma scritto per l'ambiente Windows ed utilizzabile anch'esso, grazie all'interfaccia, in ambiente HP-UX.

Bibliografia

Una presentazione semplice dei POSIX threads:
`http://www.uwo.ca/its/doc/courses/notes/hpc/pthreads.html`

Illustra una modalità proprietaria di utilizzo dei thread, ma nella prima parte presenta i POSIX threads e i threads WindowsNT in modo chiaro e semplice:
`http://threads.cs.caltech.edu/threads/Presentations/Tutorials/98-11-09-SC98/Tutorial/ppframe.htm`

Pagina HP, ricca di documentazione utile al programmatore:
`http://www.devresource.hp.com/devresource/Topics/Threads/Threads.html`

Utile confronto tra diverse implementazioni, in particolare tra POSIX threads e WindowsNT threads:
`http://www.devresource.hp.com/devresource/Docs/TechPapers/pThreads.html`

Descrizione dell'architettura degli HP Server del CILEA:
"*Programmare gli HP Server del CILEA*", Bollettino del CILEA, n.71, febbraio 2000

Dettagli sulla parallelizzazione dei programmi in ambiente HP-UX e confronto tra le librerie CPSlib e POSIX threads:
"*Parallel Programming Guide for HP-UX Systems*", H.P., B6056-96006