STRUMENTI PER LA PARALLELIZZAZIONE DI APPLICAZIONI

CASO DI STUDIO: IBM PARALLEL FORTRAN

Rapporto Interno C89-54

Novembre 1989

Ranieri Baraglia
Domenico Laforenza
Pasquale Lazzareschi
Giovanni Lombardi
Raffaele Perego

STRUMENTI PER LA PARALLELIZZAZIONE DI APPLICAZIONI

CASO DI STUDIO: IBM PARALLEL FORTRAN

R. Baraglia, D. Laforenza , P. Lazzareschi,
G. Lombardi, R. Perego

Reparto Calcolo Parallelo

Consiglio Nazionale delle Ricerche
Istituto CNUCE
via S. Maria, 36
56100 Pisa

Rapporto interno C89-54

Sommario

La presente nota riporta le modalità di parallelizzazione di codici scientifici e le esperienze acquisite nella conversione di applicazioni da sequenziale a parallelo. La nota fà specifico riferimento all'ambiente parallelo shared-memory IBM 3090 multiprocessore ed allo IBM Parallel Fortran PRPQ (program number 5799-CTX).

Sono descritte le funzioni di parallelizzazione automatica fornite dal compilatore (parallelismo implicito) nonché le estensioni al linguaggio Fortran ed ai servizi di libreria per permettere la codifica di specifici costrutti paralleli (parallelismo esplicito).

Sono specificate le modalità di compilazione ed esecuzione di programmi paralleli sia utilizzando il sistema operativo MVS che il sistema CMS. Infine sono presentati alcuni criteri per la conversione di applicazioni da sequenziali a parallelo e i principali indici per la valutazione delle prestazioni dei programmi.

1. Introduzione

L'elaborazione di complesse applicazioni tecnico scientifiche richiede la disponibilità di *computer* con sempre maggiore potenza elaborativa per ridurne i tempi di esecuzione. Tale riduzione si ottiene principalmente utilizzando l'elaborazione vettoriale su architetture di tipo SIMD uniprocessore e l'elaborazione parallela su architetture di tipo MIMD.

Con elaborazione parallela si intende la modalità con cui parti indipendenti dello stesso programma (processi o *task* paralleli) sono eseguite simultaneamente su più processori di un stesso *computer*. L'utilizzo del parallelismo riduce il tempo di permanenza dell'applicazione nel computer (*elapsed time*) aumentando, generalmente, il tempo di CPU totale.

Le interazioni tra le porzioni del programma che vengono eseguite parallelamente avvengono tramite una singola memoria condivisa da tutti i processori, nel caso di architetture realizzate secondo il modello computazionale ad ambiente globale, tramite scambio di messaggi nel modello ad ambiente locale.

Il modello ad ambiente globale, al quale ci riferiamo in questa nota, prevede l'utilizzo di più processori ciascuno dei quali elabora segmenti indipendenti di codice memorizzati in un'unica memoria centrale. I processi in esecuzione comunicano accedendo a variabili in memoria centrale usate in condivisione. Appartengono a questa categoria elaboratori quali CRAY Y-MP, IBM 3090, Alliant FX, ecc.

Il modello ad ambiente locale prevede invece che ciascun processore disponga di una propria memoria privata e che i processi in esecuzione su ogni processore comunichino tra loro tramite collegamenti fisici dedicati. Appartengono a questa categoria gli elaboratori NCUBE, Intel iPSC/2, ecc.

Dal punto di vista *software*, per sfruttare il parallelismo previsto da queste architetture, sono stati sviluppati, strumenti (compilatori e precompilatori) parallelizzanti che in modo automatico (parallelismo implicito) od utilizzando primitive di parallelizzazione proprie del linguaggio di programmazione (parallelismo esplicito), permettono di sfruttare il parallelismo dell'architettura. Attualmente non esistono strumenti per la totale parallelizzazione automatica di tutte le forme di parallelismo presenti in un programma. Gli strumenti si limitano alla parallelizzazione dei DO-loop, analizzando il codice sequenziale per individuare eventuali inibitori, ed in loro assenza generano il corrispondente codice parallelo. Altre forme di parallelismo sono costituite da sottoprogrammi eseguibili contemporaneamente su differenti dati e da gruppi di istruzioni eseguibili concorrentemente con altre sequenze di istruzioni.

I linguaggi di programmazione parallela usati sugli attuali *Supercomputer* sono generalmente ottenuti estendendo le corrispondenti versioni sequenziali. Un esempio è costituito dal Parallel Fortran utilizzato sulle configurazioni multiprocessore della serie IBM 3090.

2. Parallel Fortran: Ambiente di esecuzione

La figura 1 mostra l'ambiente di esecuzione del Parallel Fortran. In fase di compilazione del programma Fortran parallelo viene prodotto, in modo implicito od esplicito, il codice per la creazione a *run time* dei *task* Fortran. Questi costituiscono delle entità parallele, a cui viene assegnato lavoro con l'esecuzione di opportune primitive di parallelizzazione. I *task* Fortran sono accodati per l'esecuzione sui processori Fortran. Quest'ultimi, creati a *run time*, sono implementati come *task* MVS oppure come CPU virtuali sotto VM ed a loro volta allocati dal sistema operativo ai processori reali.

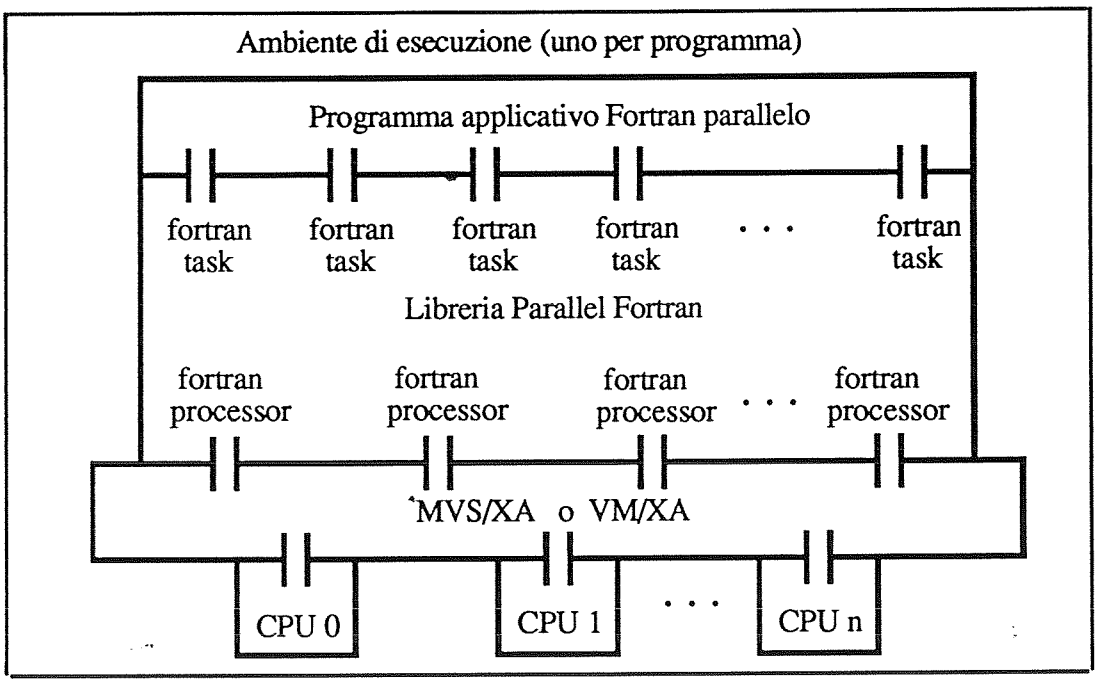


Figura 1: *Ambiente di esecuzione del Parallel Fortran*

3. Il compilatore Fortran Parallelo

Il Parallel Fortran[2] (PF) IBM è realizzato come estensione del linguaggio e dei servizi di libreria del Fortran VS ed è stato sviluppato utilizzando come base la versione 2.1 del Fortran VS. Esso costituisce attualmente un prodotto separato ma, le estensioni relative alle primitive di parallelizzazione ed ai servizi di libreria saranno integrate nel Fortran VS a partire dalla versione 2 release 5 di prossima disponibilità.

Il PF può essere utilizzato con i sistemi operativi MVS, MVS/XA e VM/XA, mantiene tutte le caratteristiche del compilatore Fortran VS versione 2.1 (vettorizzazione, ottimizzazione, ecc.) e permette due tipi di parallelismo: implicito o automatico ed esplicito. Il primo è realizzato con estensioni ai servizi del compilatore, il secondo è ottenuto con estensioni del linguaggio di programmazione. Inoltre, routine per la sincronizzazione dell'esecuzione di regioni critiche di programma e di variabili accedute contemporaneamente sono state aggiunte come servizi di libreria.

---

2 IBM Parallel Fortran PRPQ (Program Number 5799-CTX)

Il PF permette di realizzare un parallelismo a più livelli cioè, un programma può creare più *task* paralleli e questi a loro volta possono creare altri *task* ; ciascun *task* parallelo può eseguire operazioni di I/O.

3.1 Estensioni ai servizi del compilatore

Le estensioni ai servizi del compilatore realizzano la parallelizzazione automatica dei DO-loop, la figura 2 ne mostra un esempio. Questi servizi sono richiedibili con l'opzione di compilazione PARALLEL. L'opzione PARALLEL(AUTO) permette di attivare questa funzione di parallelizzazione automatica.
Un DO-loop viene parallelizzato quando:

- non esistono inibitori (il risultato dell'elaborazione parallela è uguale a quello dell'elaborazione seriale)
- la stima del tempo di esecuzione è favorevole all'elaborazione parallela.

Un DO-loop può essere contemporaneamente parallelizzato e vettorizzato. Il compilatore analizza i DO-loop e, se non esistono inibitori, sulla base di una stima dei costi di esecuzione, decide se produrre codice scalare, vettoriale e/o parallelo. Il lavoro del compilatore in questa fase è simile a quello svolto per la vettorizzazione automatica dei DO-loop. Le scelte fatte dal compilatore possono essere modificate usando opportunamente la direttiva PREFER. Quest'ultima permette inoltre di specificare il numero di iterazioni da assegnare ad ogni *task* parallelo nonché il numero di *task* da generare.

Il DO-loop:

```
        DO 10 I = 1,NMAX
          A(I) = B(I)*B(I) + C(I)*C(I)
10      CONTINUE
```

viene suddiviso come segue:

```
        DO 11 I = 1,N1
          A(I) = B(I)*B(I) + C(I)*C(I)      ==> un task parallelo
11      CONTINUE

        DO 12 I = N1+1,N2
          A(I) = B(I)*B(I) + C(I)*C(I)      ==> un task parallelo
12      CONTINUE
                                            .....
                                            .....
        DO XX I = NXX+1,NMAX
          A(I) = B(I)*B(I) + C(I)*C(I)      ==> un task parallelo
XX      CONTINUE
```

Figura 2: *Esempio di parallelizzazione automatica di un DO-loop*

I principali inibitori alla parallelizzazione automatica sono:

- salti fuori dal loop (GO TO)
- operazioni di I/O
- chiamate a routine (CALL, FUNCTION) di utente o di sistema non parallelizzate
- dipendenze tra dati
- variabili induttive

3.1.1 Dipendenze tra i dati e variabili induttive

Esiste un dipendenza sui dati quando l'ordine di esecuzione delle istruzioni o delle iterazioni di un DO-loop determina il risultato dell'elaborazione. Le dipendenze sono ovviamente accettabili all'interno di un *task* ma non sempre tra *task* paralleli. Le dipendenze e le variabili induttive non costituiscono sempre un inibitore alla parallelizzazione. Per quanto riguarda le dipendenze tra dati si distingue tra: dipendenza vera, antidipendenza e dipendenza di output.

Una dipendenza vera, figura 3, si ha quando S1 accede una locazione di memoria in scrittura e S2 accede la stessa locazione in lettura.

```
          DO 40 K = 1,N                      DO 40 K = 2,N
S1        C(K) = B(K)*A(K)          S1,S2    C(K) = C(K-1)+1
S2        D(K) = C(K)+1                  40  CONTINUE
     40   CONTINUE
               (a1)                               (a2)
```

Figura 3: *Esempio di dipendenza vera*

Nel caso (a1) la dipendenza non inibisce la parallelizzazione del loop perché ogni iterazione è indipendente dalle altre, mentre il caso (a2) non è parallelizzabile, perché ogni k-esima iterazione deve essere eseguita solo al termine dell'iterazione precedente.

Una antidipendenza, figura 4, si ha quando S1 e S2 accedono la stessa locazione di memoria, rispettivamente, in lettura ed in scrittura.

```
          DO 40 K = 1,N                      DO 40 K = 1,N-1
S1        C(K) = B(K)*A(K)          S1,2     C(K) = C(K+1)+1
S2        B(K) = A(K)+1                  40  CONTINUE
     40   CONTINUE

               (b1)                               (b2)
```

Figura 4: *Esempio di antidipendenza*

Nel caso (b1) la dipendenza non inibisce la parallelizzazione, mentre il caso (b2) non può essere parallelizzato perché la k-esima iterazione deve essere completata prima dell'iterazione successiva. Un semplice metodo per ovviare a questa semplice forma di inibizione alla parallelizzazione è quella di ricopiare il vettore C in un vettore temporaneo che nel LOOP sarà utilizzato nel lato destro dell'assegnamento.

Una dipendenza di output, figura 5, si ha quando S1 e S2 eseguono due accessi in scrittura alla stessa locazione di memoria.

```
      DO 40 K = 1,N                    DO 40 K = 1,N
S1    C(K) = A(K)+1            S1,2    T = A(K)
      . . . . . .                      A(K) = B(K)
S2    C(K) = B(K)+1                    B(K) = T
   40 CONTINUE                      40 CONTINUE
```

(c1) (c2)

Figura 5: *Esempio di dipendenza di output*

Il DO-loop (c1) può essere eseguito in parallelo, il caso (c2) può essere parallelizzato solo se la variabile T e utilizzata solo all'interno del loop come variabile temporanea e può quindi essere replicata su ogni *task* parallelo.

Una variabile induttiva, figura 6, è una variabile la cui sequenza di valori forma una progressione aritmetica. La presenza di una variabile induttiva in un DO-loop può inibire la parallelizzazione automatica.

```
      J = 1                            DO 40 I = 2,N,2
      DO 40 I = 2,N,2                  C(I-1) = C(I)*2
      C(J) = C(I) * 2            40    CONTINUE
      J = J + 2
40    CONTINUE
```

(d1) (d2)

Figura 6: *Esempio di variabile induttiva*

La parallelizzazione automatica di DO-loop contenenti variabili di questo tipo, caso (d1), può presentare delle difficoltà per il compilatore. Una ristrutturazione del tipo mostrato in (d2) rende il DO-loop sicuramente parallelizzabile in modo implicito.

3.1.2 Direttiva PREFER

La direttiva PREFER già prevista per l'elaborazione vettoriale è stata estesa all'elaborazione parallela, questa indica al compilatore:

- quale tipo di codice deve generare per il DO_loop che segue la direttiva, senza tener conto del risultato della *cost analysis*
- il numero di processori Fortran da usarsi per l'elaborazione parallela di un DO_loop, di un PARALLEL LOOP o di un PARALLEL CASES
- il numero di iterazioni che si vuole assegnare ad un *task* Fortran per eseguire un DO_loop od un PARALLEL LOOP.

{C | * } string

PREFER [SCALAR | VECTOR] [SERIAL | PARALLEL]
[PROCS(n)] [MINPROCS(l)] [MAXPROCS(m)]
[CHUNK(n)] [MINCHUNK(l)] [MAXCHUNK(m)]

abbreviazioni:

VECTOR	===>	VEC
PARALLEL	===>	PAR
MINPROCS	===>	MINPR
MAXPROCS	===>	MAXPR
MINCHUNK	===>	MINCH
MAXCHUNK	===>	MAXCH

"string" deve essere uguale a quella specificata nell'opzione di compilazione DIRECTIVE.

PREFER PARALLEL

richiede la parallelizzazione del primo DO_loop che segue la direttiva; esso viene parallelizzato solo se non presenta inibitori.

PREFER SERIAL

obbliga il compilatore a produrre codice seriale per il primo DO_loop che segue la direttiva; questa direttiva è sempre rispettata.

PREFER PROC(*n*) MINPROCS(*l*) MAXPROCS(*m*)

indica il numero o limita il numero di processori Fortran che si vogliono usare per eseguire un DO_loop, un PARALLEL CASES oppure un PARALLEL LOOP.

PROCS(*n*)

raccomanda l'uso di *n* processori Fortran.

MINPROCS(*l*)

raccomanda l'uso di almeno *l* processori Fortran.

MAXPROCS(*m*)

raccomanda l'uso di al più *m* processori Fortran.

PREFER CHUNK(*n*) MINCHUNK(*l*) MAXCHUNK(*m*)

indica il numero di iterazioni che si vuole assegnare ad un *task* Fortran per eseguire un DO_loop o PARALLEL LOOP.

CHUNK(*n*)

raccomanda l'assegnazione di *n* iterazioni.

MINCHUNK(*l*)

raccomanda l'assegnazione di almeno *l* iterazioni.

MAXCHUNK(*m*)

raccomanda l'assegnazione di al più *m* iterazioni.

3.2 Estensioni al linguaggio Fortran

Le primitive di parallelizzazione descritte nel seguito sono state implementate per permettere l'inserimento nei programmi di espliciti costrutti paralleli. Esse permettono di eseguire in parallelo: iterazioni di un loop, blocchi di istruzioni e subroutines.

Iterazioni in parallelo

PARALLEL LOOP

Blocchi di istruzioni in parallelo

PARALLEL CASES

Subroutines in parallelo

ORIGINATE
SCHEDULE
DISPATCH
WAIT
TERMINATE

3.2.1 Iterazioni in parallelo

PARALLEL LOOP permette di eseguire in parallelo sottoinsiemi di iterazioni di un loop. Il compilatore genera il corrispondente codice parallelo, ed è compito del programmatore verificare se le iterazioni sono o no indipendenti.

```
      PARALLEL LOOP lab ind=in,term[,step]
      [PRIVATE(var[,var] ....)]
      [DOFIRST [LOCK]]
      ..........
      prologo, eseguito un volta per task
      ..........
      [DOEVERY]
      ..........
      corpo del DO-loop parallelo
      ..........
      [STOP LOOP label]
      ..........
      [DOFINAL [LOCK]]
      ..........
      epilogo, eseguito un volta per task
      ..........
lab   CONTINUE
```

Le variabili ***lab***, ***ind***, ***in***, ***term*** e ***step*** seguono le regole delle corrispondenti variabili presenti in un DO-loop.

L'istruzione PRIVATE può essere usata per specificare i nomi delle variabili private di ciascun *task* parallelo. Esse possono essere variabili artmetiche o logiche il cui valore è indefinito al di fuori del loop.

Le istruzioni DOFIRST e DOFINAL sono opzionali e possono essere usate per specificare eventuali istruzioni da eseguire una sola volta come prologo ed epilogo di ciascun *task*.

Il parametro LOCK può essere usato per serializzare l'esecuzione di detti blocchi di istruzioni.

Un Parallel Loop:

- può essere eseguito da più CPU
- ciascuna CPU può eseguire un sottoinsieme di iterazioni per volta
- l'esecuzione sequenziale continua solo dopo che tutte le iterazioni sono state eseguite
- può contenere altri costrutti paralleli (parallel loop, parallel cases e DO_loop), nidificati fino ad un massimo di 25 livelli
- l'indice del loop parallelo è implicitamente definito privato
- possono essere usate funzioni intrinsiche ed operazioni di I/O
- DOFIRST e DOFINAL non possono contenere la variabile indice del parallel loop

Le figure seguenti mostrano alcuni esempi di Parallel Loop.

```
      codice seriale                codice parallelo

      SUM = 0.D0                    SUM=0.D0
      DO 10 I = 1,N                 PARALLEL LOOP 10 I=1,N
      DO 10 J = 1,N                  PRIVATE(PSUM)
      SUM = SUM + A(I,J)             DOFIRST
10    CONTINUE                        PSUM=0.D0
                                     DOEVERY
                                      DO 20 J = 1,N
                                       PSUM = PSUM + A(I,J)
                                20     CONTINUE
                                     DOFINAL LOCK
                                      SUM=SUM+PSUM
                                10  CONTINUE
```

Figura 7: *Esempio di Parallel Loop*

```
            PARALLEL LOOP 10 I=1,N
            PRIVATE(T)
            T = A(I)
            A(I) = B(I)
            B(I) = T
10          CONTINUE
```

Figura 8: *Esempio di Parallel Loop*

```
      PARALLEL  LOOP 10 I=1,N
          ..........
          ..........
        IF (A(I).EQ.V) THEN
           LI = I
           STOP  LOOP 10
        ENDIF
          ..........
          ..........
10      CONTINUE
```

Figura 9: *Esempio di uso dello statement STOP LOOP nel Parallel Loop*

Lo statement STOP LOOP interrompe l'esecuzione del Parallel Loop; al verificarsi della condizione specificata l'esecuzione di tutti i *task* paralleli coinvolti nell'elaborazione del Parallel Loop viene interrotta.

Un Parallel Loop viene eseguito in modo seriale se contiene:

- variabili di tipo carattere
- salti nel DOFIRST
- salti fuori dal DOFIRST
- salti nel DOEVERY
- salti fuori dal DOEVERY
- salti nel DOFINAL
- salti fuori dal DOFINAL
- variabili indice non INTEGER*4
- chiamate a subroutine o function di utente

3.2.2 Istruzioni in parallelo

L'istruzione PARALLEL CASES permette di eseguire sequenze di istruzioni in parallelo; identifica l'inizio di una struttura costituita da una serie di gruppi di istruzioni detti *case blocks*.

```
PARALLEL CASES
[PRIVATE(var[,var] ....)]
CASE
. . . . .
. . . . .
CASE 1
. . . . .
. . . . .
CASE 2,WAIT FOR CASE 1
. . . . .
. . . . .
CASE[m[,WAIT FOR CASE(n1[,n2]....)]]
. . . . .
. . . . .
END CASES
```

L'istruzione PRIVATE può essere usata per specificare i nomi delle variabili che devono essere private per ciascun *task*. Esse possono essere variabili aritmetiche o logiche. Viene creata una copia di dette variabili per ogni *case*, ed il loro valore è indefinito al di fuori del gruppo *case*.

Per un *Parallel Cases* valgono le seguenti considerazioni:

- può essere eseguito da più CPU
- ciascuna CPU può eseguire un case per volta
- l'esecuzione sequenziale riprende solo dopo che tutti i cases sono stati eseguiti
- può contenere altri costrutti paralleli (parallel cases, parallel loop e DO_loop)
- le variabili indice di un *loop* parallelo, contenuto in un case, sono private
- l'istruzione WAITING FOR CASE definisce un ordinamento nell'esecuzione di più *case*, all'interno dello stesso *Parallel Cases*
- un *case* può usare funzioni intrinsiche ed operazioni di I/O
- l'indipendenza tra i *case* deve essere controllata dal programmatore.

Nell'esempio di figura 10 il *Parallel Cases* è costituito da quattro blocchi di istruzioni; l'esecuzione dei *case* 3 e 4 è ritardata fino a che i *case* 1 e 2, non sono completati.

```
      PARALLEL CASES
       CASE 1
         DO 10 I = 1, N
          C(I) = SQRT(ABS(A(I)))
10       CONTINUE
       CASE 2
         DO 20 I = 1,N
          D(I) = SQRT(ABS(B(I)))
20       CONTINUE
       CASE 3, WAITING FOR CASE 1
         DO 30 I = 1,N
         E(I) = E(I) + C(I)
30       CONTINUE
        CASE 4, WAITING FOR CASE 2
          DO 40 I = 1,N
          F(I) = F(I) + D(I)
40       CONTINUE
      END CASES
```

Figura 10: *Esempio di Parallel Cases*

Un Parallel Cases viene eseguito in modo seriale se contiene:

- variabili di tipo carattere
- salti all'interno di un *case*
- salti fuori da un *case*
- variabili indice non INTEGER*4
- chiamate a *subroutine* o *function* di utente

3.2.3 Subroutine in parallelo

Per l'esecuzione di *subroutines* in parallelo è necessario prima creare, in modo esplicito, i *task* Fortran, a cui successivamente verrano assegnate le *subroutine* per l'esecuzione parallela. La gestione dei *task* Fortran viene fatta con le istruzioni seguenti.

```
ORIGINATE ANY TASK taskid
```

- crea un *task* parallelo
- ritorna al programma, nella variabile *taskid*, l'identificatore del task

```
TERMINATE TASK taskid
```

cancella un *task* e rilascia la memoria ad esso allocata

```
      INTEGER*4. TASKID(4)
      ..............
      ..............
      DO 10 I = 1,4
        ORIGINATE ANY TASK TASKID(I)
10    CONTINUE
      ..............
      ..............
      ORIGINATE ANY TASK ITASK
      ..............
      ..............
      DO 20 I = 1,4
        TERMINATE TASK TASKID(I)
20    CONTINUE
      ..............
      ..............
      TERMINATE TASK ITASK
      ..............
```

Figura 11: *Esempio di uso delle istruzioni ORIGINATE e TERMINATE*

In figura 11, l'esecuzione della prima istruzione Originate crea 4 *task* Fortran i cui identificatori sono posti nel vettore TASKID. Successivamente questi identificatori sono utilizzati in una istruzione Terminate per la cancellazione dei *task* Fortran. La seconda istruzione Originate crea un solo *task* Fortran il cui identificatore è memorizzato nella nella variabile ITASK.

Per l'assegnazione di una *subroutine* ad un *task* Fortran, per l'esecuzione parallela, si usano le istruzioni Schedule e Dispatch .

```
SCHEDULE TASK taskid,                          <== formato 1
[TAGGING(tag1[,tag2] .....),]
[SHARING(common1[,common2] ....),]
[COPYING(common1[,common2] ....),]
[COPYINGI(common1[,common2] ....),]
[COPYINGO(common1[,common2] ...),]
CALLING nomesub[arg1[,arg2] ...)]

SCHEDULE ANY TASK taskid, ....                 <== formato 2
[TAGGING(tag1[,tag2] .....),]
[SHARING(common1[,common2] ....),]
[COPYING(common1[,common2] ....),]
[COPYINGI(common1[,common2] ....),]
[COPYINGO(common1[,common2] ...),]
CALLING nomesub[arg1[,arg2] ...)]
```

```
DISPATCH TASK taskid,                          <== formato 1
[TAGGING(tag1[,tag2] .....),]
[SHARING(common1[,common2] ....),]
[COPYING(common1[,common2] ....),]
[COPYINGI(common1[,common2] ....),]
[COPYINGO(common1[,common2] ...),]
CALLING nomesub[arg1[,arg2] ...)]

DISPATCH ANY TASK taskid, .....                <== formato 2
[TAGGING(tag1[,tag2] .....),]
[SHARING(common1[,common2] ....),]
[COPYING(common1[,common2] ....),]
[COPYINGI(common1[,common2] ....),]
[COPYINGO(common1[,common2] ...),]
CALLING nomesub[arg1[,arg2] ...)]
```

L'istruzione Dispatch può assegnare del nuovo lavoro ad un *task* Fortran senza attendere il completamento del lavoro ad esso assegnato in precedenza, mentre l'istruzione Schedule può riassegnare lavoro ad un *task* Fortran, solo dopo che il lavoro ad esso assegnato è completato. Il formato 1 permette di assegnare assegnare lavoro ad uno specifico *task* Fortran il cui identificatore è contenuto nella variabile ***taskid;*** il formato 2 è usato per assegnare lavoro ad un qualsiasi *task* Fortran inattivo il cui identificatore è posto nella variabile *taskid.* Nell'esempio di figura 12 il valore

memorizzato nella variabile TASKID(I) identifica il *task* parallelo a cui può essere assegnato lavoro con l'esecuzione delle primitive SCHEDULE o DISPACTH. La *subroutine* assegnata può a sua volta creare altri *task* paralleli, quest'ultimi sono proprietà del *task* che li ha generati.

L'istruzione TERMINATE cancella il *task* il cui identificatore è memorizzato nella variabile TASKID(I) e tutti i *task* di più basso livello da esso generati.

```
      .........
      INTEGER*4  TASKID(4)
      .........
      .........
      DO 10 I = 1,4
        ORIGINATE ANY TASK TASKID(I)
10    CONTINUE
      .........
      .........
      DO 20 I = 1,4
        SCHEDULE TASK TASKID(I),CALLING SUB3
20    CONTINUE
      .........
      .........
      DO 30 I = 1,4
        TERMINATE TASK TASKID(I)
30    CONTINUE
      STOP
      END
      SUBROUTINE SUB3
      .........
      .........
      STOP
      END
```

Figura 12: *Esempio di uso della istruzionie SCHEDULE*

L'istruzione WAIT definisce un punto di sincronizzazione, in cui si aspetta per la fine di uno o più i *task* paralleli

```
WAIT FOR TASK taskid                          <== formato 1
[,TAGGING(,tag1[,tag2]  .....)]

WAIT FOR ANY TASK taskid                      <== formato 2
[,TAGGING(,tag1[,tag2]  .....)]

WAIT FOR ALL TASK                             <== formato 3
```

Il formato 1 può essere usato per aspettare la fine di uno specifico *task* il cui identificatore si trova in *taskid*. Il formato 2 può essere usato per aspettare la fine un qualsiasi *task*, l'identificatore del *task*, appena terminato, viene posto nella variabile *taskid*. Il formato 3 può essere usato per aspettare la fine di tutti i task paralleli.

In figura 13 sono mostrati alcuni esempi di utilizzo combinato delle istruzioni SCHEDULE, DISPATCH e WAIT.
La *subroutine* SUB1 viene eseguita nel *task* il cui identificatore è contenuto nella variabile TASKID(1); la WAIT successiva permette di attendere il completamento del lavoro assegnato al *task* TASKID(1) per poter assegnare ad esso la *subroutine* SUB2. "WAIT FOR ANY TASK J" sincronizza per la terminazione di un qualsiasi *task* Fortran, il cui identificatore è posto nella variabile J, in cui verrà eseguita la *subroutine* SUB4.
"WAIT FOR ALL TASKS" sincronizza per la terminazione del lavoro assegnato a tutti i *task* paralleli, e successivamente l'istruzione DISPATCH assegna nuovamente lavoro ai *task* eventualmente accodandolo se questi sono tutti attivi.

```
          .........
          INTEGER*4  TASKID(4),TASK2(8)
          .........
          .........
          DO 10 I = 1,4
            ORIGINATE ANY TASK TASKID(I)
10        CONTINUE
          .........
          SCHEDULE TASK TASKID(1),CALLING SUB1
          .........
          WAIT FOR TASK TASKID(1)
          SCHEDULE TASK TASKID(1),CALLING SUB2
          .........
          DO 20 I = 1,4
            SCHEDULE TASK TASKID(I),CALLING SUB3
20        CONTINUE
          .........
          WAIT FOR ANY TASK J
          .........
          DISPATCH TASK J,CALLING SUB4
          WAIT FOR ALL TASKS
          .........
          DO 30 I = 1,8
            DISPATCH ANY TASK TASK2(I),
            CALLING SUB5
30        CONTINUE
          WAIT FOR ALL TASKS
          .........
          DO 40 I = 1,4
            TERMINATE TASK TASKID(I)
40        CONTINUE
          STOP
          END
```

Figura 13: *Esempio di uso delle istruzione SHEDULE, DISPATCH e WAIT*

La clausola *TAGGING* permette di associare un identificatore ad un *task* parallelo.

```
SCHEDULE TASK taskid,TAGGING(tag1,..),CALLING...
   .............
   .............
WAIT FOR TASK taskid,TAGGING(,tagx,...)
```

Alla sincronizzazione per il completamento dell'esecuzione del lavoro assegnato ad un *task* è possibile memorizzare in tagx il contenuto di tag1 ed identificare cosi la terminazione di una parte del programma parallelo. Un esempio è riportato in figura 14; al completamento dell'esecuzione della subroutine SUB asegnata al *task* IT(1) può essere eseguito un certo processo se K=3 e J=2.

```
      .........
      INTEGER*4 IT(4)
      .........
      .........
      DO 10 I = 1,4
      ORIGINATE ANY TASK IT(I)
10    CONTINUE
      DO 10 K = 1,NTIMES
      DO 20 J = 1,N
      SCHEDULE TASK IT(1),TAGGING(K,J),CALLING SUB
      SCHEDULE TASK IT(2),TAGGING(K,J),CALLING SUB
      SCHEDULE TASK IT(3),TAGGING(K,J),CALLING SUB
      SCHEDULE TASK IT(4),TAGGING(K,J),CALLING SUB
      WAIT FOR TASK IT(1),TAGGING(I1,I2)
      IF(I1.EQ.3.AND.I2.EQ.2) THEN
      .........
      .........
      ENDIF
      WAIT FOR ALL TASKS
      .........
      .........
20    CONTINUE
      .........
      .........
10    CONTINUE
```

Figura 14: *Esempio di uso clausola TAGGING*

Alle subroutine parallele la cui esecuzione è attivata con le istruzioni SCHEDULE o DISPATCH possono essere passati degli argomenti nello stesso modo previsto dalla normale istruzione CALL. Detti argomenti sono condivisi tra il programma chiamante e la *subroutine* parallela.

3.2.4 Comunicazione tra *task* paralleli

Ogni *task* parallelo ha la sua memoria privata e la comunicazione tra i *task* può avvenire tramite gli argomenti specificati nella istruzione SCHEDULE o DISPATCH o *common block*. Mentre gli argomenti sono condivisi tra il *task* chiamante ed il *task* chiamato allo stesso modo della istruzione CALL i nomi dei **common block** usati per lo scambio di dati debbono essere indicati nelle istruzioni SCHEDULE e DISPATCH.
I parametri possibili sono SHARING, COPYING, COPYINGI e COPYINGO.

SHARING (nome,.....)

i *common* sono messi in comune tra tutte le subroutine in esecuzione parallela. In figura 15 il *common* C1 può essere acceduto in lettura e/o scrittura da ambedue le subroutine SUB1 e SUB2.

```
      .........
      COMMON /C1/Z(1000)
      Z(1) = 1
      Z(2) = 2
      .........
      .........
      SCHEDULE TASK T1,SHARING(C1),CALLING SUB1(X)
      SCHEDULE TASK T2,SHARING(C1),CALLING SUB2(Y)
           .........
      WAIT FOR ALL TASKS
C     Z(1)  contiene 2
C     Z(2)  contiene 3
           .........
      SUBROUTINE SUB1(X)
      COMMON /C1/Z(1000)
C     Z(1)  contiene 1 quando SUB1 è chiamata
      Z(1) = Z(1) + 1
      .........
      .........
      SUBROUTINE SUB2(X)
      COMMON /C1/Z(1000)
C     Z(2)  contiene 2 quando SUB2 è chiamata
      Z(2) = Z(2) + 1
      .........
```

Figura 15: *Esempio di uso del parametro SHARING*

COPYING (nome,.....)

i *common* del programma chiamante sono copiati nei corrispondenti *common* della *subroutine* parallela, all'esecuzione della istruzione SHEDULE o DISPATCH e sono ricopiati indietro al completamento della istruzione WAIT. La figura 16 ne mostra un esempio.

```
      .........
      COMMON /C1/Z(1000)
      Z(1) = 1
      Z(2) = 2
      .........
      .........
      SCHEDULE TASK T1,COPYING(C1),CALLING SUB1(X)
      SCHEDULE TASK T2,COPYING(C1),CALLING SUB2(Y)
           .........
      WAIT FOR ALL TASKS
C     il contenuto di Z dipende quale subroutine termina prima
C     Z(1)  contiene 1 o 2
C     Z(2)  contiene 3 o 2
           .........
      SUBROUTINE SUB1(X)
      COMMON /C1/Z(1000)
C     Z(1)  contiene 1 quando SUB1 è chiamata
      Z(1) = Z(1) + 1
      .........
      .........
      SUBROUTINE SUB2(X)
      COMMON /C1/Z(1000)
C     Z(2)  contiene 2 quando SUB2 è chiamata
      Z(2) = Z(2) + 1
      .........
```

Figura 16: *Esempio di uso del parametro COPYING*

COPYINGI (nome,.....)

i *common* del programma chiamante sono copiati nei corrispondenti *common* della *subroutine* parallela, all'inizio dell'esecuzione del *task*. La copia avviene come parte della istruzione SCHEDULE o DISPATCH. Nelle figure 17 e 18 sono mostrati due esempi di uso di questo parametro.

```
      .........
      COMMON /C1/Z(1000)
      COMMON /C2/Y(1000)
      Y(1) = 101
      Y(2) = 102
      .........
      Z(1) = 1
      Z(2) = 2
      .........
      SCHEDULE TASK T1,COPYINGI(C1),SHARING(C2),
      CALLING SUB1
      SCHEDULE TASK T2,SHARING(C1),CALLING SUB2
           .........
      WAIT FOR ALL TASKS
C     Y(1)  contiene 2
C     Z(1)  contiene 0
C     Z(2)  contiene 3
           .........
      SUBROUTINE SUB1
      COMMON /C1/Z(1000)
      COMMON /C2/Y(1000)
C     Z(1)  contiene 1
      Y(1) = 2 * Z(1)
      .........
      SUBROUTINE SUB2
      COMMON /C1/Z(1000)
C     Z(1)  contiene 1
C     Z(2)  contiene 2
      Z(1) = 0
      Z(2) = 3
      .........
```

Figura 17: *Esempio di uso del parametro COPYINGI*

```
      .........
      COMMON /C1/Z(1000)
      Z(1) = 1
      Z(2) = 2
           .........
      SCHEDULE TASK T1,COPYINGI(C1),CALLING SUB1
           .........
      WAIT FOR ALL TASKS
C     le modifiche fatte da SUB1 non sono ritornate al task radice
C     Z(1)  contiene 1
C     Z(2)  contiene 2
           .........
      SUBROUTINE SUB1
      COMMON /C1/Z(1000)
C     Z(1)  contiene 1
C     Z(2)  contiene 2
      Z(1) = 0
      Z(2) = 3
      .........
```

Figura 18: *Esempio di uso del parametro COPYINGI*

COPYINGO (nome,.....)

i *common* della *subroutine* parallela sono copiati nei corrispondenti *common* del programma chiamante, al completamento della istruzione WAIT. Le figure 19 e 20 ne mostrano due esempi.

```
      .........
      COMMON /C1/Z(1000)
      COMMON /C2/Y(1000)
      Y(1) = 101
      Y(2) = 102
      .........
      Z(1) = 1
      Z(2) = 2
      .........
      SCHEDULE TASK T1,SHARING(C1,C2),CALLING SUB1
      SCHEDULE TASK T2,COPYINGO(C1),CALLING SUB2
           .........
      WAIT FOR ALL TASKS
C     Y(1)  contiene 2
C     Z(1)  contiene 0
C     Z(2)  contiene 3
           .........
      SUBROUTINE SUB1
      COMMON /C1/Z(1000)
      COMMON /C2/Y(1000)
C     Z(1)  contiene 1
      Y(1) = 2 * Z(1)
      .........
      .........
      SUBROUTINE SUB2
      COMMON /C1/Z(1000)
C     Z      ha un volore indeterminato quando SUB2 è chiamata
      Z(1) = 0
      Z(2) = 3
      .........
```

Figura 19: *Esempio di uso del parametro COPYINGO*

```
      .........
      COMMON /C1/Z(1000)
      Z(1) = 1
      Z(2) = 2
      .........
      SCHEDULE TASK T1,COPYINGO(C1),CALLING SUB1
            .........
      WAIT FOR ALL TASKS
C     Z(1)  contiene 0
C     Z(2)  ha un valore indeterminato
            .........
      SUBROUTINE SUB1
      COMMON /C1/Z(1000)
C     Z       ha un valore indeterminato quando SUB1 è chiamata
      Z(1) = 0
      .........
```

Figura 20: *Esempio di uso del parametro COPYINGO*

3.3. Estensioni ai servizi di libreria

Le estensioni ai servizi di libreria permettono la sincronizzazione degli accessi in scrittura a varibili in memoria comune e di sequenzializzare l'esecuzione di parti critiche di un programma. Prevedono inoltre servizi di *trace* per il *debug* del programma parallelo.La sincronizzazione è realizzata con l'uso dei meccanismi di *lock* e degli eventi.

La gestione dei *lock* è fatta con le seguenti routine di libreria:

PLORIG(*lockid*)

- crea un *parallel lock*
- memorizza in lockid l'identificatore del *lock*
- lockid variabile o elemento di un *array* (I*4).

PLTERM(*lockid*)

- cancella un *parallel lock*

PLLOCK(*lockid,var1*, ...)

- ottiene il *parallel lock*
- lockid contiene l'identificatore del *lock* richiesto
- **(...,var1, ...)** sono variabili, *array* o elementi di *array* per i quali si richiede la sincronizzazione degli accessi.

PLFREE(*lockid,var1*, ...)

rilascia un *parallel lock*

La figura 21 mostra l'utilizzo del meccanismo del lock per controllare l'incremento di uno stesso contatore, da parte di due subroutine in esecuzione parallela. Affinchè l'elaborazione risulti corretta l'accesso in scrittura alla variabile COUNT deve essere controllato dallo stesso *lock* il cui identificatore viene posto, al momento della sua creazione, nella variabile LOCK.

```
.........
INTEGER*4 LOCK,COUNT
COMMON /C1/LOCK,COUNT
Z(1) = 1
Z(2) = 2
.........
.........
CALL PLORIG(LOCK)
COUNT = 0
ORIGINATE ANY TASK IT1
ORIGINATE ANY TASK IT2
.........
SCHEDULE TASK IT1,SHARING(C1),CALLING SUB1
SCHEDULE TASK IT2,SHARING(C1),CALLING SUB2
    .........
WAIT FOR ALL TASKS
CALL PLTERM(LOCK)
.........
SUBROUTINE SUB1
INTEGER*4 LOCK,COUNT
COMMON /C1/LOCK,COUNT
CALL PLLOCK(LOCK,COUNT)
COUNT = COUNT + 1
CALL PLFREE (LOCK,COUNT)
.........
.........
SUBROUTINE SUB2
INTEGER*4 LOCK,COUNT
COMMON /C1/LOCK,COUNT
CALL PLLOCK(LOCK)
COUNT = COUNT + 1
CALL PLFREE (LOCK)
.........
```

Figura 21: *Esempio di utilizzo del meccanismo dei lock per serializzare l'accesso alla variabile COUNT*

Un *task* parallelo in esecuzione può segnalare il completamento di una certa sua parte e dopo continuare l'esecuzione, oppure mettersi in stato di *wait* in attesa del completamento di un certo evento, e riprendere l'esecuzione quando riceve la segnalazione che l'evento si è verificato.

In generale le due azioni sono descritte con le funzioni:

POST:	informa che un evento si è verificato
WAIT:	pone il *task* in attesa per un evento.

Per la gestione degli eventi sono previste le seguenti *routine* di libreria:

PEORIG(*eventid*)

- crea un *parallel event*

- eventid memorizza l'identificatore dell'evento creato
- imposta i contatori *postcount*, *waitcount* e *eventtype* a 1.

PETERM(*eventid*)

cancella un *parallel event*

PEPOST(*eventid*)

esegue un *post* di un *parallel event*

PEWAIT(*eventid*)

il *task* chiamante viene messo in attesa di un *parallel event*

PEINIT(*eventid,postcount,waitcount, eventtype*)

inizializza un *parallel event*

- *postcount*

  indica il numero di *post* richieste per soddisfare l'evento

- *waitcount*

  indica il numero di *wait* richieste per soddisfare l'evento

- *eventtype*

  può essere 0 o 1

  0 un *task* può eseguire più volte una post o una *wait*

  1 un *task* può eseguire una sola volta una post o una *wait*

Ad esempio l'evento inizializzato con: CALL PEINIT(IVENT1,5,1,1), necessita di almeno 5 *task* paralleli di cui uno esegua una WAIT e tutti gli altri eseguano una POST. Altri due esempi sono mostrati in figura 22 e 23.

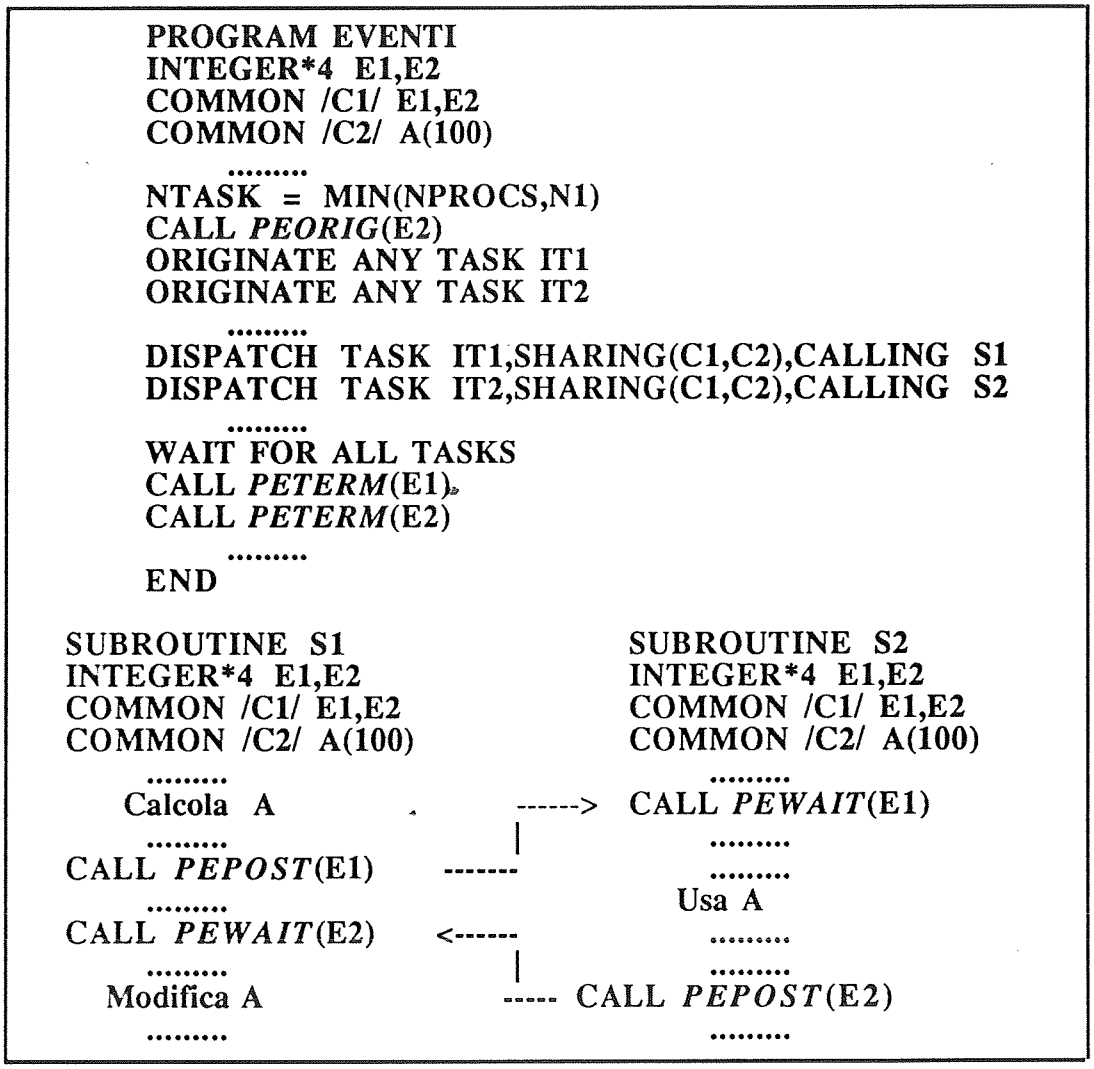


Figura 22: *Esempio di uso del meccanismo degli eventi*

```
      PROGRAM MAIN
      PARAMETER(N=6)
      DIMENSION IT(N)
      COMMON /AB/
            .....
            .....
      DO 10 I = 1,N
10    ORIGINATE ANY TASK IT(I)
            .....
C     crea e inizializza un evento che permette di sincronizzare
C     tutti i task ad un certo punto della loro esecuzione.
      CALL PEORIG(IEVNT)
      CALL PEINIT(IEVNT,N,N,1)
            .....
            .....
      DO 10 I = 1,N
10    DISPATCH TASK IT(I),SHARING(AB),
      CALLING SUB(IEVNT,A,B)
      WAIT FOR ALL TASKS
            .....
      CALL PETERM(IEVNT)
      DO 10 I = 1,N
10    TERMINATE TAS IT(I)
      END

      SUBROUTINE SUB(IEVNT,A,B)
      COMMON /AB/
            .....
            .....
C     la POST segnala che il task ha raggiunto un certo punto;
C     la Wait lo pone in stato di attesa fino a che tutti gli altri
C     task non abbiano raggiunto lo stesso punto.
      CALL PEPOST(IEVNT)
      CALL PEWAIT(IEVNT)
            .....
            .....
      RETURN
      END
```

Figura 23: *Esempio di uso del meccanismo degli eventi*

E' possibile conoscere a *run time* il numero dei processori Fortran richiamando l'esecuzione del sottoprogramma NPROCS nei seguenti modi:

CALL NPROCS(*n*)

memorizza in ***n*** il numero dei processori Fortran

n **= NPROCS()**

memorizza in ***n*** il numero dei processori Fortran

n1 **= NPROCS(*n2*)**

il numero dei processori Fortran è memorizzato in ***n1*** e ***n2*** .

3.4 Opzioni di Compilazione

La parallelizzazione è richiedibile con l'opzione di compilazione PARALLEL, che prevede a sua volta opzioni sia per il controllo della compilazione che per la produzione di *report* descriventi il tipo di codice generato.

PARALLEL([opzioni] ...) | PARALLEL | NOPARALLEL

opzioni per il controllo della compilazione:

AUTOMATIC | NOAUTOMATIC abbr.==> AUTO | NOAUTO

specifica se il compilatore deve o no analizzare i DO_loop per la generazione automatica del codice parallelo

LANGUAGE | NOLANGUAGE abbr.==> LANG | NOLANG

specifica se o no il compilatore deve generare codice parallelo per i costrutti paralleli (PARALLEL LOOP e PARALLEL CASES).

REPORT([*option*]) | NOREPORT abbr.==> REP([*opt.*]) | NOREP

specifica se o no il compilatore deve generare un *report* descrivente il codice generato per DO_loop, PARALLEL LOOP e PARALLEL CASES.

option:

LIST	il report è stampato
TERM	il report è mostrato su terminale

La figura 24 riporta un esempio di listing ottenuto con l'opzione REPORT. I *flag* stampati in corrispondenza del DO-loop analizzato possono avere i seguenti valori:

UNAN	il loop non è analizzabile
SCAL	il loop è stato analizzato ed stata scelta l'esecuzione scalare
VECT	il loop sarà eseguito in vettoriale
PARA	il loop sarà eseguito in parallelo
PAVE	il loop sarà eseguito in parallelo e vettoriale
SERI	il loop sarà eseguito in seriale.

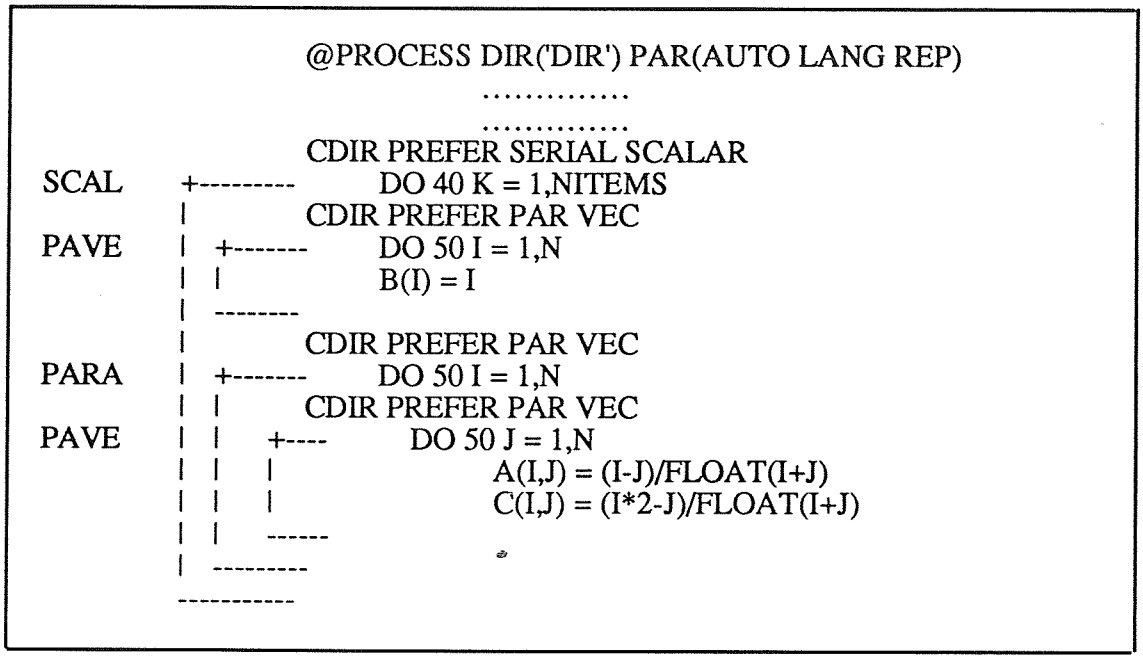


Figura 24: *Esempio di listing ottenuto con l'opzione REPORT*

L'opzione Parallel può essere specificata in vari modi, valgono le seguenti relazioni:

REPORT è equivalente a **REPORT(LIST)**.
PARALLEL() è equivalente a **PARALLEL(NOAUTO NOLANG NOREP)**.
PARALLEL è equivalente a **PARALLEL(AUTO LANG NOREP)**.
NOPARALLEL è equivalente a **PARALLEL()**.

Le opzioni di compilazione possono essere anche specificate sullo *statement* @PROCESS, che, se usato, deve costituire il primo *statement* del programma sorgente, sullo *statement* EXEC, oppure nel comando CMS con cui si richiede la compilazione:

- programma sorgente

  @PROCESS DIR('DIR') PAR(AUTO LANG REP)

- MVS

  // EXEC PFPCLG,PARM.COMP='PAR(AUTO,LANG,REP(LIST))'

- CMS

  PFPCOMP filename (PAR(AUTO LANG REP)

3.5 Opzioni di esecuzione

Una nuova opzione PARALLEL è disponibile per l'esecuzione ed il controllo dei *task*. Deve essere usata per richiedere l'esecuzione parallela di un programma e per specificare il numero di processori Fortran da usarsi nell'esecuzione stessa. Il numero di processori Fortran indica: in MVS il numero di *task* MVS che debbono essere generati all'inizio della esecuzione del programma parallelo, in CMS il numero di CPU virtuali che debbono essere definite e inizializzate all'inizio dell'esecuzione.
Le sottoopzioni sono:

PROCS(*n*)

definisce il numero dei processori Fortran usati per l'esecuzione del programma, *default* PROCS(1).

TRACE[(*opt1*[,*opt2*] ....)]

produce una traccia degli eventi descriventi l'evolversi dell'esecuzione del programma, *default* NOTRACE.

L'interpretazione dei dati prodotti dall'opzione TRACE non è semplice, pertanto per la sua descrizione si rimanda a [18].

Le opzioni di esecuzione possono essere definite: in MVS sullo *statement* EXEC ed in CMS nel comando con cui si richiede l'esecuzione.

- MVS

```
// EXEC  PFPCLG,PARM.COMP='PAR(AUTO)',
//            PARM.GO='PAR(PROCS(4)
```

- CMS

```
PFPEXEC  filename  (PARELLEL(PROC(2))
```

oppure

```
PFPEXEC  filename  2
```

4. Utilizzo PF

Il PF può eseguire programmi solo in *link mode*, l'intero programma, contenente anche tutte le routine di servizio, deve risiedere nei primi 16Mbyte della memoria virtuale. Solo i *common block* possono essere dinamicamente allocati ad indirizzi superiori ai 16Mbyte. Questi debbono essere specificati alla compilazione con il parametro DC.

Il modulo eseguibile può essere prodotto in due modi:

- un unico modulo eseguibile contenente tutto il programma (*task* radice più *task* paralleli
- due moduli eseguibili, uno contenente il *task* radice e l'altro contenente le subroutine da eseguirsi in parallelo.

Essendo il modulo oggetto generato dal compilatore non rientrante, nel primo caso ogni esecuzione dell'istruzione ORIGINATE provoca il caricamento di una nuova copia del programma, mentre nel secondo caso provoca il caricamento della sola parte parallela del programma. Il numero di *task* paralleli che possono essere creati è limitato dalla dimensione del programma e dalla quantità di memoria disponibile sotto i 16 Mbyte.

4.1 Operazioni di I/O

I nomi dei *file*, per le operazioni di I/O, seguono le convenzioni previste dai sistemi operativi MVS e VM/CMS.

Il *task* radice utilizza i *ddname* previsti dal Fortran standard, mentre i *task* espliciti utilizzano dei *ddname* del tipo **StttfFxx**, dove **tttt** è l'identificatore del *task* e **xx** il numero dell'unità logica. Ad esempio l'operazione di I/O eseguita dal *task* 1 sull'unità logica numero 4 ha la *ddname* S0001F04.

L'identificatore viene assegnato secondo l'ordine di generazione del *task* parallelo ed al momento della esecuzione della istruzione TERMINATE il *task* rilascia il suo identificatore che viene riusato all'esecuzione della successiva ORIGINATE. In tali casi è necessario porre attenzione che il nuovo *task* parallelo invii le proprie operazioni di I/O al file voluto.

Per quanto riguarda l'uso dell'opzione TRACE le *ddname* dei *file* in cui sono registrati gli eventi sono del tipo **S0000T00** per il *task* radice e **S000nT00** per i *task* espliciti.

4.2 Utilizzo in ambiente MVS

Un programa Fortran parallelo deve essere "linkeditato" in modo diverso dai normali programmi. Inoltre il *load module* deve deve avere un nome prefissato, pertanto ogni libreria di moduli eseguibili può contenere un solo programma in forma eseguibile.

Un programma Fortran parallelo può esere "linkeditato" in due modi:

- un unico *load module*. In questo caso ogni nuovo *task*, creato con l'esecuzione dell'istruzione ORIGINATE, implica il caricamento in memoria di una nuova copia dell'intero programma. Questo perchè il codice generato dal compilatore Fortran parallelo non è rientrante il che porta ad uno spreco di memoria.

- un *load module* per la parte seriale (*root task*) e un *load madule* per le subroutine che costituiranno i *task* paralleli In questo caso ogni nuovo *task* caricherà in memoria una nuova copia della sola parte parallela.

Le procedure catalogate per la compilazione ed esecuzione sono:

PFPC

per compilare programmi paralleli

PFPCL e PFPCCLL

per compilare e "linkeditare" programmi paralleli

PFPCL load module contiene sia il *task* radice che i *task* paralleli

PFPCCLL il root *task* e i *task* paralleli costituiscono due load module separati.

PFPCLG e PFPCCLLG

per compilare, "linkeditare" ed eseguire programmi paralleli

PFPG

per eseguire load module.

Le figure 25 e 26. mostrano due esempi di Job Control Language per la compilazione ed esecuzione di un programma Fortran parallelo usando le procedure catalogate PFPCLG e PFPCCLLG. In questi esempi vengono usati in esecuzione quattro processori Fortran (parmetro PROCS = 4). Le Data Definition GO.S0001F06, GO.S0002F06, GO.S0003F06, GO.S0004F06 specificano le unità dove verrano scritti eventuali messaggi di errore prodotti dai *task* creati con l'esecuzione della istruzione ORIGINATE.

```
//jobname JOB parametri
//   EXEC   PFPCLG,PARM.COMP=('PAR(LANG,AUTO,REP(LIST))',
//          'VECT(LEV(2)),OPT(3)'),PROCS=4
//FORT.SYSIN DD *

     (programma Fortran, root e task paralleli)

//GO.S0001F06 DD SYSOUT=*
//GO.S0002F06 DD SYSOUT=*
//GO.S0003F06 DD SYSOUT=*
//GO.S0004F06 DD SYSOUT=*
```

Figura 25: *Esempio di utilizzo della procedura catalogata PFPCLG*

```
//jobname JOB parametri
//   EXEC   PFPCCLLG,VFPPAR='PAR(LANG,AUTO,REP(LIST))',
//          VFPVEC=VEC,PROCS=4
//FORTMAIN.SYSIN DD *

      (programma Fortran, root task)

//FORTTASK.SYSIN DD *

      (programma Fortran, task paralleli)

//GO.S0001F06 DD SYSOUT=*
//GO.S0002F06 DD SYSOUT=*
//GO.S0003F06 DD SYSOUT=*
//GO.S0004F06 DD SYSOUT=*
```

Figura 26: *Esempio di utilizzo della procedura catalogata PFPCCLLG*

4.3 Utilizzo in ambiente CMS

Un programma Fortran parallelo deve essere "linkeditato" con una apposita procedura, non può essere eseguito con i soliti comandi CMS LOAD e START.
Come già descritto per il sistema MVS il load module può essere realizzato in due modi:

- un unico *load module*. In questo caso ogni nuovo *task*, creato con l'esecuzione dell'istruzione ORIGINATE, implica il caricamento in memoria di una nuova copia dell'intero programma. Questo perchè il codice generato dal compilatore Fortran parallelo non è rientrante il che porta ad uno spreco di memoria.

- un *load module* per la parte seriale (*root task*) e un *load madule* per le subroutine che costituiranno i *task* paralleli In questo caso ogni nuovo *task* caricherà in memoria una nuova copia della sola parte parallela.

I comandi usati per la compilazione il *linkedit* e l'esecuzione dei programmi in ambiente VM/CMS sono:

PFPCOMP ***filename*** **[(opzioni)]**

per compilare programmi paralleli (output: "filename TEXT A" ed eventuale "filename LISTING A")

Esempio:

PFPCOMP PROG1 (OPT(3) VEC(LEV(2)) PAR(AUTO LANG REP)

PFPCOPY ***outext intext1*** **[*intext2* [*intext3* ..]**

crea un *file* di nome "outext TEXT" contenete la combinazione dei *file* di *intext*. Questo passo è necessario solo se il programma principale e le *subroutine* non sono state compilate insieme.

PFPGLOB [*textlib1*] [*textlib2*] .....

definisce i nomi delle librerie utente usate nella fase di *link edit* per risolvere riferimenti esterni al programma, sostituendo il comando GLOBAL TXTLIB per i programmi fortran paralleli. Questo passo è necessario solo se si usano TXTLIB private.

PFPLINK *loadlib* [rtask | = ptask | =]
Link edit di un programma parallelo.
rtask indica il nome del *text* file contenente il codice per il *root task*. Il valore default è = che significa lo stesso nome di *loadlib*.
ptask indica il nome del *text* file contenente il codice per i *task* paralleli. Il valore default è = che significa lo stesso nome di *loadlib*.

In generale per "linkeditare" programmi paralleli (output "loadlib LOADLIB"), i passi necessari sono:

1. creare un solo *file* di *text* o un *file* di *text* per il *task* radice ed uno per i *task* paralleli; più *file* di *text* possono essere combinati in un unico *file* con la procedura PFPCOPY
2. specificare i nomi delle librerie di *text* usate nel passo di *link edit*, usando la procedura PFPGLOB (solo se sono usate librerie utente)
3. eseguire il *link edit*.

PFPEXEC *loadlib* [*procs* | (*opzioni*)]

esegue un programma preparato con la procedura PFPLINK. Loadlib è il nome del *file* di tipo LOADLIB contente il modulo eseguibile, *procs* è il numero di processori Fortran da usare per l'esecuzione (default *procs* = 1, il programma è eseguito in seriale). La sola opzione usabile è l'opzone PARALLEL. "*procs*" ed "*opzioni*" sono in alternativa.

Le figure 27, 28 e 29 riportano un esempio di sessione al terminale per la compilazione ed esecuzione di un programma Fortran parallelo creando un unico modulo eseguibile. Nelle figure sono riportati in corsivo i comandi CMS ed in maiuscolo i messaggi del sistema.

```
pfpcomp test1 (vec(lev(2) opt(3) par(auto lang rep(list))
PARALLEL VS FORTRAN ENTERED.     11:19:12

**MAIN** END OF COMPILATION 1 ******

**SUB1** END OF COMPILATION 1 ******

PARALLEL VS FORTRAN EXITED.                 11:19:12

Ready; T=0.11/0.19 11:19:12
```

Figura 27: *Esempio di compilazione di un programma Fortran parallelo creando un unico modulo oggetto*

```
pfplink test1

Ready; T=2.08/2.29 11:19:25
```

Figura 28: *Esempio di link edit per la creazione del modulo eseguibile*

```
pfpexec test1 2
00: CPU 01 defined
00: Vector Facility 01 defined
00: CPU 02 defined
00: Vector Facility 02 defined
01: HCPGSP2627I The virtual machine is placed in CP mode due to a SIGP initial
CPU reset from CPU 00.
02: HCPGSP2627I The virtual machine is placed in CP mode due to a SIGP initial
CPU reset from CPU 00.
NPROCS = 2
02: HCPGIR450W CP entered; disabled wait PSW 000A0000 FF0FADED
01: HCPGIR450W CP entered; disabled wait PSW 000A0000 FF0FADED
01: HCPGSP2629I The virtual machine is placed in CP mode due to a SIGP stop
from CPU 00.
02: HCPGSP2629I The virtual machine is placed in CP mode due to a SIGP stop
from CPU 00.
VIRT= 003:21.29 REAL= 003:22.09 WALL= 00:01:43
Ready; T=0.11/0.20 11:34:05
```

Figura 29: *Esempio di esecuzione del programma usando due processori*

I messaggi di figura 29 indicano la creazione dell'ambiente parallelo. Inizialmente sono definite 2 CPU virtuali ognuna con la sua vector facility. Esse sono messe in CP mode dalla CPU 0. Il messaggio NPROCS=2 viene scritto dal programma fortran.

I messaggi "CP entered; disabled wait PSW ..." indicano la fine dell'esecuzione dei task paralleli sulle due cpu virtuali. Successivamente le CPU virtuali sono messe in CP mode dalla CPU 0.
Il messaggio "VIRT=. . . REAL=. . . WALL=. . ." indica rispettivamente il tempo di CPU virtuale, reale e il tempo di elapsed usato dal programma in esecuzione. Notare il tempo di CPU riportato nel messaggio di Ready del CMS. Questo tempo è notevolmente inferiore al tempo di CPU usato dal programma, riportato nel messaggio precedente. Questa differenza è dovuta al fatto che il CMS conosce solo una CPU (CPU 0) di conseguenza il messaggio di ready riporta solo il tempo usato da questa CPU mentre l'esecuzione del programma avviene sulle CPU 1 e 2.

Le figure 30, 31, 32 e 33 riportano un esempio di sessione al terminale per la compilazione ed esecuzione di un programma Fortran parallelo creando un modulo per il *root task* e uno per la parte parallela. Il programma è costituito dal *root task* contenente il *main* PROVA ed una *subroutine* di nome SUB1 e dalla parte parallela contenete le *subroutine* di nome SUB2 e SUB3. In modo analogo a quello descritto in figura 30 vengono compilate le *subroutine*.SUB2 e SUB3

```
pfpcomp prova (vec(lev(2) opt(3) par(auto lang rep(list))
PARALLEL VS FORTRAN ENTERED.     09:20:15

**MAIN** END OF COMPILATION 1 ******

PARALLEL VS FORTRAN EXITED.                09:20:15

Ready; T=0.13/0.21 09:20:15
```

Figura 30: *Esempio di compilazione del programma principale*

```
pfpcopy prova  prova sub1

Ready; T=0.16/0.19 09:32:20



pfpcopy ptask sub2 sub3

Ready; T=0.15/0.18 09:34:10
```

Figura 31: *Esempio di unione del root task e delle subroutine parallele*

```
pfplink prova = ptask

Ready; T=3.03/3.11 09:37:22
```

Figura 32: *Esempio di link edit del root task e dei task paralleli in due moduli eseguibili distinti*

```
pfplexec prova (parallel(procs(4))

00: CPU 01 defined
00: Vector Facility 01 defined
00: CPU 02 defined
00: Vector Facility 02 defined
00: CPU 03 defined
00: Vector Facility 03 defined
00: CPU 04 defined
00: Vector Facility 04 defined
01: HCPGSP2627I The virtual machine is placed in CP mode due to a SIGP initial
CPU reset from CPU 00.
02: HCPGSP2627I The virtual machine is placed in CP mode due to a SIGP initial
CPU reset from CPU 00.
03: HCPGSP2627I The virtual machine is placed in CP mode due to a SIGP initial
CPU reset from CPU 00.
04: HCPGSP2627I The virtual machine is placed in CP mode due to a SIGP initial
CPU reset from CPU 00.
NPROCS = 4
04: HCPGIR450W CP entered; disabled wait PSW 000A0000 FF0FADED
02: HCPGIR450W CP entered; disabled wait PSW 000A0000 FF0FADED
01: HCPGIR450W CP entered; disabled wait PSW 000A0000 FF0FADED
03: HCPGIR450W CP entered; disabled wait PSW 000A0000 FF0FADED
01: HCPGSP2629I The virtual machine is placed in CP mode due to a SIGP stop
from CPU 00.
02: HCPGSP2629I The virtual machine is placed in CP mode due to a SIGP stop
from CPU 00.
03: HCPGSP2629I The virtual machine is placed in CP mode due to a SIGP stop
from CPU 00.
04: HCPGSP2629I The virtual machine is placed in CP mode due to a SIGP stop
from CPU 00.
VIRT= 004:11.01 REAL= 004:12.06 WALL= 01:12:00
Ready; T=0.12/0.22 10:50:08
```

Figura 33: *Esempio di esecuzione del programma usando quattro processori*

4.5 Conversione di programmi al PF

Le applicazioni che possono beneficiare dell'elaborazione parallela generalmente presentano un utilizzo massiccio del processore e necessità di memoria tali da limitare il livello di multiprogrammazione nel sistema.

L'elaborazione parallela permette di ridurre il tempo di esecuzione (*elapsed time*) dei programmi, quindi offre la possibilità di risolvere problemi che richiederebbero un tempo di cpu eccessivo per un singolo processore. Inoltre permette di sfruttare al meglio la capacità elaborativa di un computer multiprocessore quando non è possibile utilizzarla completamente con la sola multiprogrammazione.

Per contro richiede maggior tempo di cpu rispetto all'elaborazione seriale, la diminuzione del tempo di *elapsed* avviene a scapito di altri programmi in multiprogrammazione che potrebbero utilizzare i processori, ed inoltre rende più complessa la scrittura ed il *debug* dei programmi.

La conversione di un programma non è generalmente una operazione semplice, per essa non esistono metodologie generali. I programmi esistenti sono stati pensati "in seriale" e questo rende difficile la parallelizzazione senza consistenti modifiche. Si possono seguire due alternative: parallelizzazione automatica o manuale.

La prima riguarda la parallelizzazione dei soli DO-loop le cui iterazioni sono indipendenti e per il quali la stima del tempo di esecuzione è favorevole all'esecuzione parallela; le scelte del compilatore possono essere modificate con l'uso delle direttive. In questo caso il programma rimane trasportabile ma i vantaggi sono generalmente scarsi.

La seconda viene fatta dal programmatore utilizzando specifici costrutti paralleli. Ciò richiede la conoscenza del programma e la distribuzione dei tempi di cpu in modo da intervenire solo sulle parti del programma che consumano la maggior parte del tempo di cpu. Per conoscere la distribuzione del tempo di CPU all'interno del programma può esssere usato il VS Fortran Interactive Debug (IAD) [17].

Un altro strumento utile per la conversione dei programmi è costituito dall'opzione di compilazione ICA[3] (*Intercompilation Analysis*), che permette di rilevare le incompatibilità tra le unità di programma (numero e tipo degli argomenti, lunghezza dei *common block*, ecc.). Essa permette inoltre di generare una lista contenente per ogni:

- unità di programma ed i nomi delle subroutine e function chiamate
- sottoprogramma da chi e chiamato
- *common block* dove è usato
- variabile all'interno di un common block da quali unità di programma è referenziata e da chi è modificata
- ecc.

Con la parallelizzazione manuale si può ottenere una consistente diminuzione del tempo di *elapsed* ma il programma risultante non è più trasportabile. Occorre ricordare che ogni qualvolta si vogliano diminuire i tempi di elaborazione e necessario prima eseguire l'ottimizzazione e la vettorizzazione del programma. Queste due azioni riducono il tempo di CPU richiesto per l'esecuzione del programma, mentre la parallelizzazione distribuisce tale tempo tra più processori riducendo il tempo di *elapsed*, ma introducendo gli *overhead* insiti nel processo di parallelizzazione.

Nella scrittura di nuovi programmi paralleli o nella migrazione di programmi già esistenti è necessario controllare che la parallelizzazione dei loop non ne inibisca la vettorizzazione. Nell'elaborazione parallela, i vettori contenuti in loop parallelizzabili vengono di solito divisi in tante parti quanti sono i processori disponibili, è quindi necessario che la lunghezza dei vettori rimanga sufficiente per la vettorizzazione. Inoltre i

---

[3] E' utilizzabile solo con il compilatore VS Fortran almeno al livello 2.2

task paralleli dovrebbero usare aree di memoria distanti tra loro almeno 128 *byte*; questa è l'unità di trasferimento dati tra la cache e la memoria. L'architettura 3090 prevede una *cache memory* per ciascuna cpu; il trasferimento dati tra la *cache* e la memoria centrale avviene in multipli di 128 *byte* (*cache line*), per una più ampia descrizione della cache si rimanda a [13], [14].

Se due o più *task* in esecuzione parallela aggiornano frequentemente delle variabili anche diverse, ma appartenenti alla stessa *cache line* si ha un continuo trasferimento di dati tra un *cache* e l'altra. Questa attività viene fatta automaticamente dall'*hardware* con un aumento del tempo di cpu (*overhead*).

Altri *overhead* sono introdotti per la realizzazione della parallelizzazione: l'attivazione, la terminazione e la sincronizzazione dei *task* richiedono l'intervanto del sistema operativo, quindi è bene tener conto delle seguenti osservazioni:

- costriuire *task* paralleli con ampia granularità (*subroutine* o gruppi di *subroutine*)
- cercare di bilanciare il lavoro svolto da ciascun *task*
- se i *task* sono corti o non bilanciati fare in modo che ci siano sempre dei *task* in coda per l'esecuzione, i processori Fortran dovrebbero risultare sempre attivi (uso della DISPATCH)
- se possibile utilizzare gli eventi piuttosto che "schedulare" più volte i *task* con le istruzioni SCHEDULE o DISPATCH
- riusare gli stessi *task* piuttosto che crearne di nuovi, quindi usare l'istruzione ORIGINATE una sola volta all'inizio del programma per creare tutti i *task* Fortran necessari.

Se ad esempio un vettore viene calcolato da un certo numero di *task* paralleli (ogni *task* ne calcola una parte), successivamente gli stessi *task* possono utilizzare elementi del vettore calcolati da altri *task*. In questo caso è necessario che tutto il vettore sia calcolato prima di essere utilizzato.

Il debug del programma può risultare una operazione complessa poiché l'ordine in cui vengono eseguiti i *task* non è determinato, quindi gli errori possono essere diversi da una esecuzione all'altra. L'uso di istruzioni WRITE inserite in punti chiave del programma di solito non aiuta, in quanto ogni *task* parallelo scrive su propri file e quindi non è possibile conoscere l'ordine in cui sono state eseguite le istruzioni WRITE.

Il solo strumento attualmente usabile per il debug dei programmi scritti con il Parallel Fortran è costituito dalla opzione TRACE attivabile a *run time*. Questa durante l'esecuzione del programma registra i seguenti eventi:

- inizio e fine della esecuzione del programma
- creazione e cancellazione dei *task*
- esecuzione parallela dei *task*
- sincronizzazione
- allocazione e sharing dei COMMON
- inizio e fine dei costrutti paralleli PARALLEL LOOP e PARALLE CASES
- uso dei locks ed eventi.

4.5 Principali indici di prestazione per la valutazione dei programmi

Due indici di prestazione importanti per quantificare il guadagno, in termini di tempo di elapsed, ottenibile od ottenuto da un programma parallelo rispetto alla sua versione sequenziale, sono: *parallel speedup* ed efficienza. Si distinghe tra *speedup* calcolato tramite la legge di Amdhal e *speedup* misurato.

Speedup misurato:

$$\text{Speedup} = \frac{\text{tempo di elapsed della versione scalare}}{\text{tempo di elapsed della versione parallela}}$$

Legge di Amdahl:

$$\text{Speedup} = \frac{100}{100 - p + \frac{p}{n}}$$

Dove p è la percentuale di codice eseguibile in parallelo ed n il numero di processori su cui è eseguito il programma parallelo.

La legge di Amdahl permette di determinare il massimo fattore di speedup che si può ottenere dalla parallelizzazione di un programma conoscendo la percentuale del codice che potrà essere eseguita in parallelo.

A causa degli *overhead* dovuti alla gestione dell'elaborazione parallela lo *speedup* reale può essere molto diverso da quello che ci si potrebbe aspettare e può variare da esecuzione ad esecuzione. I vari *task* paralleli di un programma Fortran debbono essere allocati, per l'esecuzione alle cpu reali. Quest'attività è svolta dal sistema operativo il quale deve contemporaneamente servire anche gli altri lavori presenti in multiprogrammazione. Pertanto il tempo di esecuzione, in caso di multiprogrammazione, di un programma parallelo dipenderà fortemente dal carico dell'elaboratore

Il parametro efficienza è calcolato nel modo sequente:

$$\text{Efficienza} = \frac{\text{Speedup}}{\text{numero di processori}}$$

e può essere usato per confrontare piú versioni parallele di un stesso programma.
Il caso ideale di efficienza uguale ad 1 non è mai ottenibile; Ciò è imputabile alla schedulazione e sincronizzazione dei *task*.

Appendice A

Terminologia

Task

sezione logica di codice.

Task parallelo

- sezione di codice computazionalmente indipendente da altre sezioni di codice
- può essere eseguito contemporaneamente ad altri *task* paralleli.

Un *task* parallelo può essere costituito da:

- iterazioni di un DO-loop
- sequenze di istruzioni.
- sottoprogrammi

Granularità

quantità di lavoro associato ad un *task*:

- grana medio-grossa (*coarse-grained*)

  il *task* esegue un numero relativamente alto di operazioni (una o più subroutine)

- grana fine (*fine-grained*)

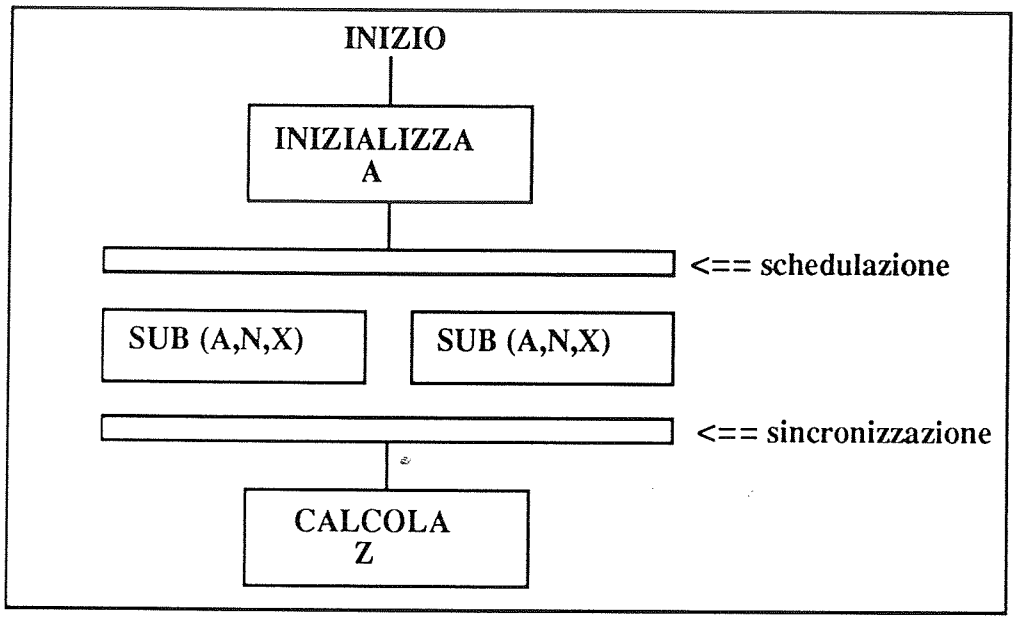
  il *task* esegue poche operazioni (gruppi di istruzioni)

Sincronizzazione

azione di coordinamento dell'esecuzione di *task* paralleli, un esempio è mostrato nella figura seguente.

```
      PROGRAM ESEMP                    SUBROUTINE SUB (T,N,S)
      PARAMETER (N = 1000)             REAL*4 T(N),S
      REAL*4  A(N,N),B(N,N),X,Y        S = 0
      DO 10 J = 1,N                    DO 10 J = 1,N
      DO 10 I = 1,N                    DO 10 I = 1,N
      A(I,J) = FLOAT(I+J)               S = S + T(I,J) * T(I,J)
      B(I,J) = FLOAT(I-J)           10 CONTINUE
10    CONTINUE                         RETURN
      CALL SUB(A,N,X)                  END
      CALL SUB(B,N,Y)
      Z = (SQRT(X) + SQRT(Y)) **2
      STOP
      END
```

Nel programma precedente le due chiamate alla subroutine SUB possono essere eseguite in parallelo per elaborare dati differenti. L'esecuzione parallela avviene secondo lo schema seguente:



Locking

stabilisce la modalità di accesso ad aree usate simultaneamente tra più *task* paralleli. Un semplice esempio che mostra la necessità della sincronizzazione è costituito dall'operazione N = N + 1 eseguita da due *task* concorrenti:

PROCESSORE 1	TEMPO	PROCESSORE 2
CARICA N IN UN REG. GENERALE	1	.....
SOMMA 1	2	CARICA N IN UN REG. GENERALE
MEMORIZZA IL NUOVO VALORE	3	SOMMA 1
.....	4	MEMORIZZA IL NUOVO VALORE
	5	
	6	

Al termine dei due aggiornammenti il valore memorizzato in N non sarà quello corretto, pertanto è necessario l'utilizzo di un meccanismo per il controllo degli accessi.

Data exchange

modalità con cui le aree dati (COMMON in PF) sono passate dal *root task* ai *task* da esso generati.

Variabili

si distinguono due tipi di variabili: variabili globali, quelle allocate al di fuori del *task* parallelo e variabili locali, quelle usate soltanto all'interno di uno specifico *task* parallelo. La modalità di accesso alle variabili globali è specificata per ciascun *task* dal *task* chiamante

Bibliografia

[1] Flynn, M.J.,"Some Computer Organizations and their effectiveness", IEEE Transact. Comput., C-21 948-60, 1972.

[2] Hockney, R. W. e Jessophe C. R. : Parallel Computers 2 (Adam Hilger, Bristol, 1988)

[3] Hwang K. e Briggs F. A., Computer architecture and parallel processing (McGraw-Hill, New York, 1989).

[4] Kuck D.J., Computers and Computations (John Wiley & Sons, New York, 1978).

[5] Computer Architecture concepts and systems, Veljko M. Milutinovic (ed.), North-Holland, 1988.

[6] Vanneschi M., "Architetture general purpouse", in AICA - Convegno Internazionale: Elaborazione Parallela, Novembre 1988.

[7] D. Laforenza, "Architetture avanzate per il calcolo scientifico", Informatica Oggi, anno 6, n. 13, Gennaio 1986.

[8] David A. Padua e Michael J. Wolfe, "Advanced Compiler Optimizations for Supercomputers", Comunications ACM vol. 29, n.29, December 1986.

[9] Buchholz W. , "The IBM System/370 vector architetture", IBM System Journal, vol. 25, n. 1, 1986 .

[10] Toomey L.J., Plachy E.C., Scarborough R.G., Sahlka R.J., Shaw J.F.e Shannon A.W., "IBM Parallel FORTRAN", in IBM System Journal, vol. 27, n.4, 1988.

[11] Gentzsch, W., Szelényi e Zecca V., " Use of Parallel Fortran for engineering problems on the IBM vector multiprocessor", in Parallel Computing 9 (1988).

[12] Gibson D. H., Rain D. W. e Walsh H. F., "Engineering and scientific processing on the IBM 3090", IBM System Journal , vol. 25, n. 1, 1986

[13] Singh Y., King G. M. e Anderson J. W., "IBM 3090 performance: a balanced system approach", IBM System Journal , vol. 25, n. 1, 1986.

[14] Tucker S. G., " The IBM 3090 system: an overview", IBM System Journal, vol. 25, n. 1, 1986.

[15] MVS/XA General Information Manual, IBM Order No.GC28-1118.

[16] VS FORTRAN Version 2 Language and Library Reference, IBM Order No. SC26-4221.

[17] VS FORTRAN Version 2 Interactive Debug Guide and Reference, IBM Order No. SC26-4223.

[18] Parallel FORTRAN Language and Library Reference, IBM Order No. SC23-0431

[19] VS FORTRAN Version 2 Programming Guide, IBM Order No. SC26-4222.

[20] D. Ferrari, G. Serazzi e S. Zeigner: Measurement and Tuning of Computer Systems, Prentice-Hall, 1983.

[21] John L. Gustafson, "Reevaluating Amdahl's Law", Comunications ACM vol. 31, n.5, May 1989.

[22] Calzarossa M. e Serazzi G., "Workload Characterization for Supercomputers",in: J. Martin (ed.) Performance Evaluation, North-Holland, 1988.

[23] G. Erbacci, "Prestazioni dei Programmi in Ambiente Parallelo", in AICA - 9° giornata di studio sulla: Valutazione delle prestazioni dei sistemi informatici, Maggio 1988.

[24] Laganà A., Gervasi O., Baraglia R. and Laforenza D., "Vector and Parallel Restructuring for Approssimate Quantum Reactive Scattering Computer Codes", in: European Symposium on High Performance Computing, North-Holland, 1989

[23] Baraglia R., Laforenza D. e Lazzareschi P. , "Tecniche di parallelizzazione di codici scientifici e valutazione delle prestazioni: il caso dell'elaboratore IBM

3090", in AICA - 10° giornata di studio sulla: Valutazione delle prestazioni dei sistemi informatici, Maggio 1989.

[25] Battle M., Carta S., Delhaye J.L., Journeau M., Richard P.G., Urbach G., Vectorization and Parallelization on IBM 3090/VF, Centre National Universitaire Sud de Calcul, August 1989