INTRODUZIONE AL CALCOLO VETTORIALE SULL'IBM 3090-VF

Sergio Barsocchi
Renato Ferrini
Pasquale Lazzareschi
Riccardo Medves
Paolo Pancrazi

Rapporto Interno C87-29

INTRODUZIONE AL CALCOLO VETTORIALE SULL'IBM 3090-VF

26 Ottobre 1987

Sergio Barsocchi
Renato Ferrini
Pasquale Lazzareschi
Riccardo Medves
Paolo Pancrazi

CNUCE - Istituto del CNR
via Santa Maria 36
56100 PISA (Italy)

Prefazione

Nei primi mesi dell'87, in previsione della istallazione di un elaboratore IBM 3090 con feature vettoriale al CNUCE, siamo stati ospiti del CNUSC (Centre National Universitaire Sud de Calcul) di Montpellier, presso il quale e' istallata un'analoga macchina.

La nostra esperienza su tale macchina e' stata molto produttiva, grazie anche alla accogliente ospitalita' e disponibilita' nei nostri confronti da parte del personale del suddetto Centro.

Sono stati in particolare presi accordi per una piu' stretta collaborazione reciproca, con scambi di esperienze personali, corsi e documentazione. Per quanto riguarda quest'ultimo punto, il CNUSC ci ha messo a disposizione il loro materiale, tra cui un manuale di introduzione al calcolo vettoriale, concedendosi gentilmente il anche permesso di riprodurlo per la nostra utenza; cosa che e' stata fatta riadattandolo, in parte, alle nostre esigenze.

Desideriamo quindi ringraziare vivamente il personale del CNUSC, dal direttore J.C. Ippolito, al direttore tecnico J.L. Delhaye, ai nostri colleghi M. Batlle, G. Hurbach e tutti gli operatori che ci hanno assistito durante la nostra permanenza presso il CNUSC.

Contenuto

Alcuni principi essenziali di programmazione 1
Leggibilita' dei programmi 1
Lo stile 1
Scelta dei nomi 1
Scrittura sequenziale 2
Rispetto dei caratteri 2
Utilizzo dei commenti 2
Analisi delle uscite del compilatore 2
Ottimizzazione dei programmi 2
Scrittura degli esponenti 3
Scrittura delle divisioni 3
Le correzioni provvisorie 3
Gli effetti perversi della ennesima precisione 3
Imprecisione dei risultati 3
L'importante e' l'argoritmo 3

Introduzione ai calcolatori vettoriali 4
Le quattro generazioni delle macchine ed il futuro 4
Sviluppo di nuove architetture 4
Il megaflops 5
Il bisogno di potenza 5
I calcolatori paralleli 5
I calcolatori vettoriali 6
Le componenti di un calcolatore vettoriale 8
L'unita' vettoriale 8
I registri vettoriali 8
Il concatenamento 8
I registri 'maschera'. 9
I registri indici 10
L'unita' scalare 10
Le istruzioni 11

Architettura e organizzazione dell'IBM 3090-VF 13
La domanda della SEAS e la risposta dell'IBM 13
Architettura IBM/370 estesa (370-XA) 14
Architettura dell'unita' di calcolo vettoriale 15
Istruzioni vettoriali 16
Accesso ai dati della memoria 18
Struttura scalare del 3090 19
Struttura della Vector Facility 22
L'unita' di esecuzione vettoriale 22
Software utilizzabile 24
Sistemi operativi 24
Linguaggi e librerie 24

VS Fortran versione 2 26
Il prodotto 26
Compilatore 26
Libreria 30
Debug interattivo 31
Generalita' 31
Comandi di debug 31

Controllo dell'esecuzione del programma: . . . 31
Controllo e modifica delle variabili: . . . 32
Gestione del video . . . 33
Informazioni statistiche . . . 33
Informazioni sulla natura delle variabili . . . 34
Manipolazioni dei files sequenziali . . . 34
Correzione degli errori . . . 35
Altri comandi: . . . 35
Riferimenti . . . 36
Esempi di utilizzo dei comandi . . . 37
Controllo dell'esecuzione del programma . . . 37
Controllo e modifica delle variabili . . . 38
Comandi full-screen esteso (in V.2) . . . 38
Informazioni sulla natura delle variabili (in V.2) . . . 38
Informazioni di traccia e di tempi . . . 39
Correzione degli errori . . . 39
Manipolazione di files sequenziali . . . 40
Altri comandi . . . 40

Le fasi della vettorizzazione . . . 41
Scrittura dell'applicazione . . . 41
Vettorizzazione automatica . . . 41
Utilizzo delle direttive . . . 42
Modifica del codice . . . 43

Gli ostacoli alla vettorizzazione: gli inibitori . . . 46
Gli inibitori "forti" . . . 46
Gli inibitori "deboli" . . . 49
La dimensione dei cicli DO . . . 49
L'ordine dei cicli DO . . . 49
Le istruzioni IF nei cicli DO . . . 51
I trasferimenti . . . 56

La vettorizzazione del VS Fortran . . . 58
L'analizzatore di esecuzione VS Fortran . . . 58
Generalita' . . . 58
Scopo . . . 58
Funzionamento . . . 58
Modalita' di utilizzo . . . 59
Descrizione delle uscite . . . 59
Compilatore vettorizzatore VS Fortran Versione 2 . . . 62
Scelta di un ciclo . . . 62
Vettorizzazione di un ciclo . . . 62
Vettorizzazione di un ciclo contenente un'istruzione IF . . . 62
Riorganizzazione delle istruzioni . . . 63
Frazionamento dei cicli . . . 63
Esempio di compilazione . . . 64

Le direttive di vettorizzazione . . . 69
Quando si usano le direttive . . . 69
Perche' si usano le direttive . . . 71
Come si usano le direttive . . . 71
Esempi di uso delle direttive . . . 72

Un programma di conversione di linguaggio: LCP . . . 74
La finalita' dell'LCP . . . 74
Principali conversioni effettuate dall'LCP . . . 75
Utilizzo dell'LCP . . . 75

Migrazione delle applicazioni . . . 76
Introduzione . . . 76
Strategie per la scelta dei programmi da vettorizzare . . . 76
Prima fase della vettorizzazione: ripulire . . . 77
Programma chiaro e ben strutturato . . . 77

Uso del Fortran 77 . . . 78
Ottimizzazione del codice . . . 78
Seconda fase della vettorizzazione: analizzare . . . 79
Terza Fase della vettorizzazione: vettorizzazione automatica . . . 79
Studio dei messaggi del compilatore . . . 79
Studio delle uscite dell'analizzatore di esecuzione. . . . 80
Quarta fase della vettorizzazione: modifiche semplici . . . 80
Utilizzo delle direttive . . . 80
Utilizzo della libreria ESSL . . . 80
Modifiche del codice . . . 81
Quinta fase della vettorizzazione: modifiche complesse . . . 83
Organizzazione dei dati . . . 83
Algoritmi . . . 84
Multitasking . . . 84

Errori di troncamento e di arrotondamento . . . 85
Modifica di algoritmi . . . 85
Riduzione vettoriale . . . 85
Underflow . . . 86

Figure

Figura 1. Funzionamento del cancatenamento 9
Figura 2. Comportamento dello scalare con il 70% di vettorizzazione. 11
Figura 3. Posizione del 3090-VF negli elaboratori scientifici 14
Figura 4. Schema dei registri vettoriali del 3090-VF 16
Figura 5. Elenco delle istruzioni vettoriali 17
Figura 6. Gerarchia della memoria. 20
Figura 7. Elementi funzionali del 3090 scalare 21
Figura 8. Elementi funzionali del 3090-VF 22
Figura 9. Unita' di esecuzione vettoriale 23
Figura 10. Comandi per il controllo dell'esecuzione del programma 32
Figura 11. Comandi per il controllo e modifica delle variabili 32
Figura 12. Comandi per la gestione del video 33
Figura 13. Comandi per le informazioni statistiche 34
Figura 14. Comandi per le informazioni sulla natura delle variabili 34
Figura 15. Comandi per la manipolazione dei file sequenziali 34
Figura 16. Comandi per la correzione degli errori 35
Figura 17. Altri comandi 36
Figura 18. Ciclo DO con calcolo di indice 81
Figura 19. Ciclo DO con calcolo di indice 81
Figura 20. Cicli DO prima e dopo la segmentazione 82
Figura 21. Ciclo non vettorizzabile 82
Figura 22. Cicli distribuiti 82
Figura 23. Organizzazione inadeguata dei dati 83
Figura 24. Migliore organizzazione dei dati 83
Figura 25. Cicli su matrici a 3 dimensioni 84
Figura 26. Cicli su matrici definiti con EQUIVALENCE 84

Alcuni principi essenziali di programmazione

Leggibilita' dei programmi

Per definizione, un programma leggibile e' un programma che puo' essere *letto* da chi non l'ha scritto. Benche' sia difficile, se non impossibile, far si' che lettura e comprensione vadano di pari passo, bisogna cercare di ridurre il piu' possibile lo scarto che li separa.

Questa finalita' si puo' ottenere seguendo alcuni principi di base.

Lo stile

Come in letteratura la grammatica da sola non fa lo stile, allo stesso modo in informatica le regole di codifica non sono di per se stesse sufficienti per una buona scrittura del programma, poiche':

- ci sono dei programmi troppo prolissi:
  una sequenza di piu' istruzioni quando una sola sarebbe stata sufficiente;
- ci sono dei programmi troppo concisi:
  una sola istruzione quando piu' istruzioni avrebbero facilitato la comprensione del testo.

Se e' difficile dare delle direttive sullo stile, e' pero' possibile affermare che un programma senza stile non ha nessuna possibilita' di evoluzione.

Si possono tuttavia dare alcune direttive per facilitare la lettura:

- Label sequenziali
- Identificazione dei blocchi
- Identificazione dell'inizio e della fine dei blocchi
- Scrittura delle proposizioni logiche nella loro forma diretta
- Segnalazione delle sequenze di istruzioni troppo complesse
- Documentazione dei programmi
- Uniformita' della scrittura

Scelta dei nomi

In tutti i casi la scelta dei nomi delle variabili deve essere fatta in maniera pertinente:

- Alcuni dicono che i nomi utilizzati nei programmi devono essere significativi. Tuttavia un nome mnemonico puo' ancora confondere perche' potrebbe rappresentare grandezze fisiche diverse, come una variabile chiamata TEMP puo' rappresentare una TEMPeratura o un TEMPo.

- Altri affermano il contrario.
  Non e' necessario che un nome di per se stesso sia significativo, ma solo una descrizione completa puo' dare tutto il suo senso ad una parola, tramite opportuni commenti.

Scrittura sequenziale

Questo puo' sembrare evidente ma bisogna scrivere un programma il piu' sequenzialmente possibile senza salti incondizionati. Tuttavia, se le esigenze del programma necessitano di simili salti, sarebbe opportuno che essi fossero eseguiti *in avanti.*

Rispetto dei caratteri

L'uso di alcuni caratteri e' vincolante nella maggior parte dei linguaggi di programmazione. In particolare il FORTRAN ed il PLI riconoscono il tipo di variabile numerica (intera o reale) dalla prima lettera del nome, per cui spetta al programmatore prendere tutte le precauzioni quando vuole usare dei nomi a piacere.

Inoltre il miscuglio dei tipi di variabile puo' inibire l'ottimizzazione fatta dal compilatore, e perfino, in alcuni casi curiosi, dare errori in esecuzione a causa delle conversioni piu' o meno fantasiose.

Utilizzo dei commenti

I commenti sono spesso inesatti poiche', quasi sempre, servono come appunti per il programmatore e non come descrizioni generali delle funzioni svolte.

Il commento e' un eccellente aiuto alla lettura di un programma, se e' esatto, conciso e appropriato.

Analisi delle uscite del compilatore

Tutti i compilatori moderni offrono molteplici mezzi d'analisi del programma come:

- liste e definizioni dei nomi utilizzati
- riferimenti incrociati
- messaggi d'errore o d'informazione
- ecc...

La semplice lettura di queste liste puo' permettere di correggere delle anomalie non palesate in fase di test.
Il famoso errore FORTRAN DO 10 I=1.3 invece di DO 10 I=1,3 non dovrebbe scappare ad un occhio attento.

Ottimizzazione dei programmi

In generale si puo' affermare che l'ottimizzazione manuale dei programmi e' difficile, ma con i compilatori moderni si superano quasi tutte queste difficolta'. Bisogna, sicuramente, non confondere ottimizzazione e la modifica di un programma in vista della sua vettorizzazione.

Ecco alcuni suggerimenti di codifiche da evitare nella scrittura di un programma FORTRAN per ottenere una migliore ottimizzazione.

Scrittura degli esponenti

- Non scrivere *REAL(I)* come esponente di una variabile
- Scrivere *A*A* piuttosto che *A**2*
- ecc...

Scrittura delle divisioni

La divisione e' gestita molto male, percio' e' bene farne un uso limitato. Quando e' possibile, e' meglio moltiplicare per l'inverso e quindi, per esempio, scrivere *X*0.5* al posto di *X/2*.

Le correzioni provvisorie

Bisogna essere molto prudenti nelle correzioni fatte a *volo;* queste sono sempre poco ponderate e, anche se partono come provvisorie, hanno spesso una facile tendenza a diventare definitive.

Gli effetti perversi della ennesima precisione

La scelta della precisione deve essere al momento dell'analisi del problema. Infatti raddoppiare o quadruplicare la precisione, perche', per esempio, una iterazione non converge, non produce, in generale, l'effetto desiderato.

Imprecisione dei risultati

E' un argomento delicato ma e' possibile enumerare i principali tipi d'imprecisione:

- Imprecisione dei calcoli dovuti alla *precisione* usata.
- Imprecisione dei dati.
- Scelta degli algoritmi.
- Impossibilita' totale o parziale di controllare i risultati.
- Errori di programmazione.

L'importante e' l'argoritmo

Bisogna persuadersi che un programma che *va male* non diventera' mai un programma che *va bene* con semplici modifiche del codice. Nella maggior parte dei casi, se *va male* e' per un difetto di analisi.

Introduzione ai calcolatori vettoriali

L'architettura di una macchina influisce sempre sulla programmazione. Tale influenza e' piu' o meno forte anche se spesso e' mascherata dai progressi della tecnologia e dalla maggiore potenza dei compilatori, che tengono sempre conto dell'architettura. Tuttavia, i compilatori vettoriali, essendo al loro inizio, peccano attualmente di inesperienza.

E' da prevedere, negli anni futuri, la diffusione sul mercato di super compilatori vettoriali che maschereranno totalmente o quasi le implicazioni dell'architettura vettoriale sulla programmazione. Si puo' quindi ritenere che gli argomenti trattati in questo documento saranno presto obsoleti in quanto e' facile prevedere che presto nasceranno dei sistemi esperti di compilazione vettoriale.

Le quattro generazioni delle macchine ed il futuro

Le differenti generazioni di calcolatori possono essere determinate dalla tecnologia, dall'architettura, dal sistema operativo e dai linguaggi utilizzati.

1. Prima generazione (1938-1954):
   questa generazione e' caratterizzata dall'utilizzo di rele' elettro-meccanici ed in seguito di valvole termoioniche. Si possono citare i nomi:
   ENIAC - 1938 (Elettronic Numerical Integrator Computer),
   EDVAC - 1950 (Elettronic Discrete Variable Automatic Computers);
   IBM 701 - 1952.

2. Seconda generazione (1954-1962):
   la seconda generazione e' caratterizzata dall'apparizione dei transistor.

3. Terza generazione (1962-1975):
   l'evoluzione delle tecnologie caratterizza la terza generazione con l'utilizzo di circuiti integrati SSI (Small Scale Integration).

4. Quarta generazione (1972 - oggi):
   da una parte la tecnologia dei circuiti integrati si rafforza col LSI (Large Scale Integration), dall'altra i linguaggi diventano sempre piu' potenti e vedono la luce le architetture vettoriali e parallele.

5. La prossima (1990- ......):
   la tecnologia dei circuiti integrati continua ad evolversi con il VLSI (Very Large Scale Integration) e l'apparizione dei calcolatori vettoriali e degli elaboratori con N processori dovrebbe permettere piu' di 1000 milioni di operazioni aritmetiche in virgola mobile al secondo.

Sviluppo di nuove architetture

I progressi tecnologici continueranno certamente ad evolversi, ma esistono dei limiti naturali a questi progressi, come per esempio la velocita' della luce. Per questo motivo la ricerca nella costruzione di elaboratori si sviluppera' su nuovi obiettivi basati essenzialmente sullo studio di nuove architetture dove, al momento attuale, non si conoscono limiti. Il calcolo numerico (aerodinamica, metereologia, trattamento della immagine) ha delle necessita' di potenza sempre piu' pressanti che hanno favorito l'apparizione di nuovi calcolatori. Per l'imme-

diato, questi calcolatori sono stati concepiti da una parte per risolvere essenzialmente il calcolo numerico (in modo particolare la sequenza addizione-moltiplicazione del tipo Y(I)=Y(I)+A(I)*Z(I,J)) e dall'altra per restituire il piu' rapidamente possibile i risultati dei cicli.

Il megaflops

Una unita' di potenza attualmente riconosciuta dai costruttori e' il megaflops (MFLOPS). La definizione di questa unita' consiste nel numero di milioni di operazioni aritmetiche a virgola mobile, fra numeri reali rappresentati su 64 bit, per secondo. Questa nozione non tiene pero' conto dei tempi d'accesso alle informazioni che hanno tuttavia un impatto importante sulla performance.

Il bisogno di potenza

Si puo' dare una idea della quantita' di potenza necessaria, allo stato attuale, in alcune discipline:

1. I calcoli di aerodinamica necessitano di 1000 megaflops
2. Alcune branche d'ingegneria nucleare richiedono piu' di 2000 megaflops.
3. Le previsioni metereologiche richiedono una potenza molto superiore a 1000 megaflops.
4. La ricerca petrolifera "si contenta" di circa 500 megaflops.

Da non dimenticare che i piu' potenti elaboratori scalari, attualmente sul mercato, arrivano ad una potenza di 10 megaflops!

I calcolatori paralleli

Il parallelismo si basa su un insieme di tecniche logiche e/o fisiche che permettono l'esecuzione simultanea di sequenze d'istruzioni indipendenti. Questo concetto non deve essere confuso con quello della multiprogrammazione che permette solo la distribuzione del tempo di accesso alla CPU ai vari programmi nel caso di monoprocessori; tuttavia dire che la multiprogrammazione su piu' processori e' una tecnica di parallelismo non e' del tutto sbagliato.

Per mostrare cos'e' il parallelismo nel senso stretto del termine, prendiamo in esame il seguente esempio:

```
      DO 10 J=1,2
      DO 20 I=1,10000
      A(I,J)=......
20    CONTINUE
10    CONTINUE

      DO 30 J=1,2
      .......
      ..=....A(N,J).....
      .....
30    CONTINUE
```

il loop

```
     DO 20 I=1,10000
     A(I,1)=......
20   CONTINUE
```

puo' essere preso in carico da una unita' di esecuzione

mentre il loop

```
     D= 20 I=1,10000
     A(I,2)=......
20   CONTINUE
```

puo' essere preso in carico da un'altra unita' di esecuzione

Il ciclo "DO 30" comincera' quando le 2 unita' di esecuzione avranno terminato il loro lavoro.

La parallelizzazione puo' essere gestita sia dal sistema, sia dall'utente e a tal fine si possono distinguere 3 tipi di parallelismo:

1. A livello di utilizzo: il sistema gestisce i lavori e li smista sui differenti processori.
2. A livello di unita' di programmazione: l'utente puo' forzare con l'aiuto di istruzioni CALL l'esecuzione simultanea di piu' programmi.
3. A livello di sequenze di istruzioni nel programma: l'utente puo', con ordini specifici, indicare che alcune istruzioni possono svolgersi simultaneamente.

Il calcolatore parallelo, anche se si basa sul funzionamento di monoprocessori che eseguono un solo lavoro alla volta, riesce a raggiungere grandi potenze di calcolo grazie a istruzioni che permettono di trattare, da una parte, vettori e matrici al posto di singole variabili e dall'altra l'esecuzione simultanea di gruppi di istruzioni.

I calcolatori vettoriali

I processori vettoriali si basano sulla combinazione di due architetture di base: il processore di tipo pipeline ed il processore di tipo SIMD (Single Istruction stream Multiple Data stream).

Consideriamo il seguente esempio:

```
     DO 10 I=1,N
     X(I)=Y(I)*Z(I)
10   CONTINUE
```

L'istruzione X(I)=Y(I)*Z(I) puo' essere grossolanamente scomposta nelle operazioni di:

1. Caricamento di Y(I)
2. Caricamento di Z(I)
3. Moltiplicazione
4. Memorizzazione di X(I)

In un calcolatore scalare, se T e' il tempo di esecuzione dell'istruzione X(I)=Y(I)*Z(I) ed N il numero di iterazioni, si puo' stimare il tempo di esecuzione dell'intero loop in N*T. Per un calcolatore di tipo pipeline (Pipelined Processor),

appena una operazione e' stata effettuata, il processore inizia l'iterazione seguente come e' mostrato nel succesivo esempio, dove la lettera C rappresenta l'abbreviazione di caricamento e M di memorizzazione.

```
Tempo  1   C Y(1)
Tempo  2   C Y(2) , C Z(1)
Tempo  3   C Y(3) , C Z(2) , Y(1)*Z(1)
Tempo  4   C Y(4) , C Z(3) , Y(2)*Z(2) , M X(1)
Tempo  5   C Y(5) , C Z(4) , Y(3)*Z(3) , M X(2)
Tempo  6   C Y(6) , C Z(5) , Y(4)*Z(4) , M X(3)
ecc ...
```

Come si puo' notare in un calcolatore di tipo pipeline si ottiene il primo risultato utile dopo quattro cicli, per cui, in generale, se K e' il numero di operazioni necessarie per avere il primo risultato utile, l'intero DO loop impieghera' un tempo totale di T+(N-1)*T/K. Per i loop corti il tempo T domina, allo stesso modo per i loop lunghi la velocita' di esecuzione sara' vicina al valore asintotico.

Noi abbiamo parlato di ciclo, ma la terminologia corrente adopera piu' volentieri il termine di vettore: un processore di tipo 'SIMD', avente M unita' centrali (Array Processor), si differenzia dal processore di tipo pipeline in quanto M iterazioni sono eseguite simultaneamente. In questo caso se M e' piu' grande del numero di iterazioni di un loop, il tempo necessario sara' T; se invece M e' inferiore a tale numero, la durata necessaria sara' di T*(N/M).

Ma torniamo al processore di tipo pipeline tentando di spiegarne il funzionamento nel caso di una operazione di moltiplicazione in virgola mobile. Questa si scompone in quattro sottooperazioni, ognuna delle quali richiede un ciclo di macchina:

1. Addizione delle caratteristiche
2. Normalizzazione delle mantisse e aggiustamento delle caratteristiche
3. Moltiplicazioni delle mantisse
4. Normalizzazione delle mantisse e aggiustamento delle caratteristiche

La normalizzazione e l'aggiustamento dell'esponente sono operazioni necessarie per rendere la prima cifra della mantissa significativa. Si puo' quindi rappresentare il funzionamento della pipeline per la moltiplicazione in virgola mobile con il seguente schema:

	Sottooper. 1	Sottooper. 2	Sottooper. 3	Sottooper. 4
Ciclo 1	Y(1),Z(1)			
Ciclo 2	Y(2),Z(2)	Y(1),Z(1)		
Ciclo 3	Y(3),Z(3)	Y(2),Z(2)	Y(1),Z(1)	
Ciclo 4	Y(4),Z(4)	Y(3),Z(3)	Y(2),Z(2)	Y(1),Z(1)
Ciclo 5	Y(5),Z(5)	Y(4),Z(4)	Y(3),Z(3)	Y(2),Z(2)
Ciclo 6	Y(6),Z(6)	Y(5),Z(5)	Y(4),Z(4)	Y(3),Z(3)
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

In questo esempio si vede che il risultato di Y(1)*Z(1) e' restituito dopo 4 cicli e Y(2)*Z(2) dopo 5 cicli, quando su un calcolatore scalare Y(2)*Z(2) sarebbe stato restituito dopo 8 cicli. La durata totale per ottenere i risultati della moltiplicazione vettoriale di N elementi e' percio' di 3+N cicli, mentre un calcolatore scalare avrebbe impiegato un tempo di 4*N cicli.

Questi tempi per il calcolo dei risultati inducono ad una osservazione: dopo i tempi di inizializzazione della pipeline (in questo caso 3 cicli) si ottiene un risul-

tato per ciclo; e' buono, ma bisogna che la pipeline sia alimentata in entrata con un carico sufficiente. E' uno dei punti deboli in un calcolatore vettoriale; infatti se la coppia Y(I),Z(I) fosse letta, ad ogni passo, da una periferica, l'alimentazione della pipeline sarebbe effettuata ad una velocita' troppo bassa, tanto piu' che le operazioni di I/O non sono vettoriali.

Le componenti di un calcolatore vettoriale

La CPU si compone di due elementi essenziali: l'unita' vettoriale e l'unita' scalare, le quali, una senza l'altra, non potrebbero formare una buona macchina vettoriale.

L'unita' vettoriale

L'unita' di esecuzione vettoriale e' composta di una pipeline alla quale e' demandata la gestione delle istruzioni vettoriali. Esistono delle macchine con 2 o piu' pipeline e questo puo' portare un notevole miglioramento delle performance, purche' esistano delle applicazioni e dei compilatori in grado di sfruttare questa possibilita'.

Su altre macchine, invece, si trovano due pipeline ognuna delle quali e' specializzata nell'esecuzione di una operazione, per esempio una esegue la moltiplicazione e l'altra l'addizione, e che possono funzionare in modo parallelo. Questa tecnica e' molto interessante perche' nel caso di un loop del tipo X(I)=X(I)+Y(I)*Z(I) si ha il primo risultato utile dopo 6 cicli, in quanto dopo aver calcolato il prodotto Y(I)*Z(I) in una pipeline con 3 cicli, il risultato passa nella pipeline che effettua la somma con altri 3 cicli. Quindi, se N e' il numero di iterazioni del loop, il tempo di esecuzione della suddetta istruzione e' stimato in 5+N cicli, quando senza parallelizzazione delle pipeline, il risultato si otterrebbe alla fine di 2*(3+N) cicli. Questa tecnica offre buone prestazioni, ma esistono seri problemi architetturali su una efficiente utilizzazione del parallelismo. Un'altra tecnica piu' largamente impiegata e' il concatenamento di cui parleremo piu' avanti.

I registri vettoriali

Un registro e' una zona di memoria situata nell'unita' centrale. Il suo ciclo base, cioe' il tempo che intercorre tra il momento in cui l'unita' centrale chiede una informazione ed il momento in cui la riceve, e' lo stesso dell'unita' centrale. Poiche' il ciclo base della memoria centrale e' circa 5 volte piu' grande del ciclo base della CPU, la pipeline non sarebbe alimentata con una sufficiente velocita' se i dati risiedessero nella memoria centrale e passerebbe quindi gran parte del suo tempo ad aspettare. E' necessario percio' che l'architettura di un calcolatore vettoriale preveda un certo numero di registri vettoriali in cui caricare i dati che rappresentano l'input della pipeline.

Purtroppo la tecnologia utilizzata per la realizzazione dei registri e' molto costosa e pertanto il numero dei registri e' sempre limitato. Per ovviare a questi problemi di costi, i costruttori hanno pensato di sostituire i registri vettoriali con memorie a cache o con memorie con piu' vie di accesso, abbassando cosi' drasticamente i tempi di trasferimento. Vi sono inoltre alcune macchine che utilizzano i registri vettoriali solo per le operazioni di concatenamento, essendo fornite di una memoria locale nella CPU per il caricamento temporaneo dei dati di ingresso alla pipeline.

Il concatenamento

Vediamo come funziona l'operazione di concatenamento analizzando il seguente esempio:

```
      DO 10 I=1,N
      X(I)=X(I)+Y(I)*Z(I)
   10 CONTINUE
```

In condizioni normali, un calcolatore vettoriale esegue in un primo tempo tutte le operazioni vettoriali Y(I)*Z(I) ed in secondo tempo tutte le addizioni vettoriali con X(I). Il concatenamento consiste da una parte nel conservare il risultato intermedio della moltiplicazione in un registro vettoriale e dall'altra nell'attivare la pipeline dell'addizione appena il primo risultato Y(I)*Z(I) esce dalla pipeline di moltiplicazione. In effetti tutto avviene quasi come se esistessero una pipeline moltiplicazione ed una di addizione.

Possiamo percio' schematizzare il funzionamento del concatenamento come segue:

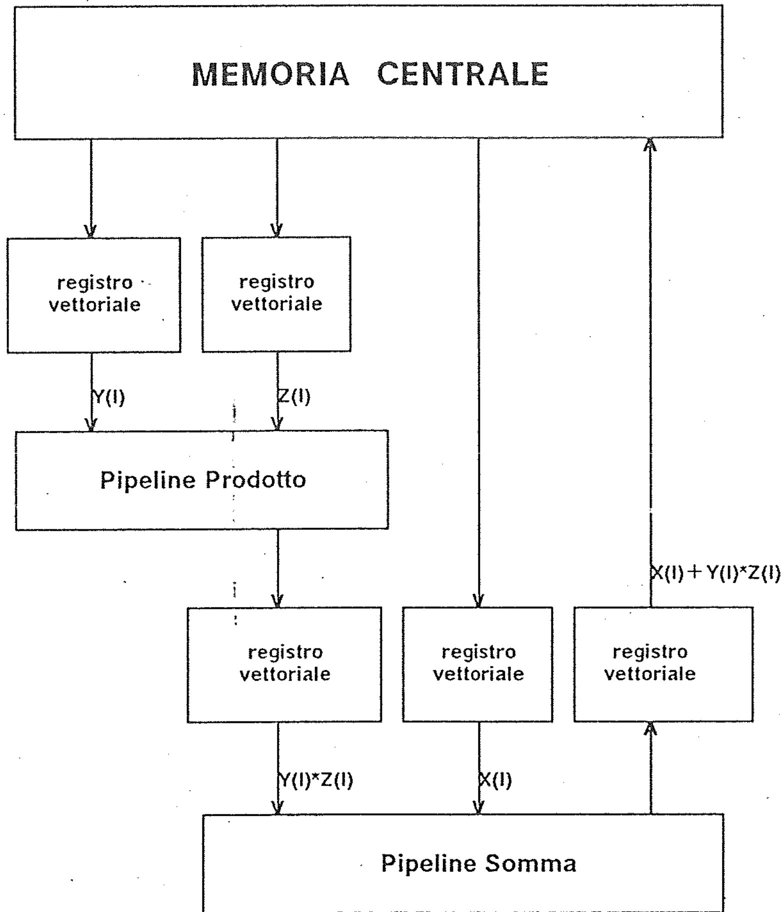

Figura 1. Funzionamento del cancatenamento

I registri 'maschera'.

I registri maschera permettono di elaborare l'operazione vettoriale solo su alcuni elementi della coppia di vettori in esame. Quindi in una situazione come la seguente:

Registro maschera	0	1	0	0	1	.
Registro vettoriale 1	A(1)	A(2)	A(3)	A(4)	A(5)	.....
Registro vettoriale 2	B(1)	B(2)	B(3)	B(4)	B(5)	.....

l'operazione vettoriale si verifichera' solo sulle coppie A(2)-B(2) , A(5)-B(5) , ecc...

Il registro maschera e' percio' utilizzato nelle funzioni di 'GATHER/SCATTER' dove le operazioni vettoriali interessano solo particolari elementi del vettore. Tutto cio' appare come se si lavorasse su due nuovi vettori A' e B' i cui elementi sono rappresentati dagli elementi dei vettori A e B che vengono selezionati. Nel caso precedente avremo percio':

Vettore A'	A(2)	A(5)	.....
Vettore B'	B(2)	B(5)	.....

L'operazione di Gather agisce quindi su tutti gli elementi dei nuovi vettori A' e B' e pone il risultato nel vettore R':

Vettore A'	A'(1)	A'(2)	.....
Vettore B'	B'(1)	B'(2)	.....
Vettore R'	R'(1)	R'(2)	.....

L'operazione di Scatter permette invece di ritornare alla dimensione dei vettori originali:

Registro maschera	0	1	0	0	1	.
Vettore R'		R'(1)			R'(2)	.....
Vettore R	R(1)	R(2)	R(3)	R(4)	R(5)	.....

Nello Scatter il contenuto degli altri elementi del vettore R non coinvolti nell'operazione vettoriale (R(1), R(3), R(4), ecc...) rimane invariato.

I registri indici

Il registro indice fornisce per ogni elemento sistemato nel registro vettoriale il suo ordinamento in memoria. Cosi' nella situazione:

Maschera	0	1	0	0	1	0	1
Memoria	a	b	c	d	e	f	g

il registro indice conterra' l'indirizzo degli elementi da caricare nel registro vettoriale, per cui avremo:

Registro indice	2	5	7
Registro vettoriale	b	e	g

I registri indice e maschera permettono quindi di trattare come operazioni vettoriali la maggioranza delle istruzioni 'IF'.

L'unita' scalare

Ogni calcolatore vettoriale possiede una unita' scalare. Questo puo' sembrare evidente, ma quello che lo e' meno, e' che questa unita' deve essere molto efficiente come si puo' notare in Figura 2 a pag. 11 dove e' riportato l'andamento dei megaflops composti in funzione del numero di megaflops scalari. Se, per esempio, si considera una macchina da 40 megaflops vettoriali, si puo' vedere che aumentando la potenza dell'unita' scalare di un megaflops, i megaflops composti aumentano di 2.

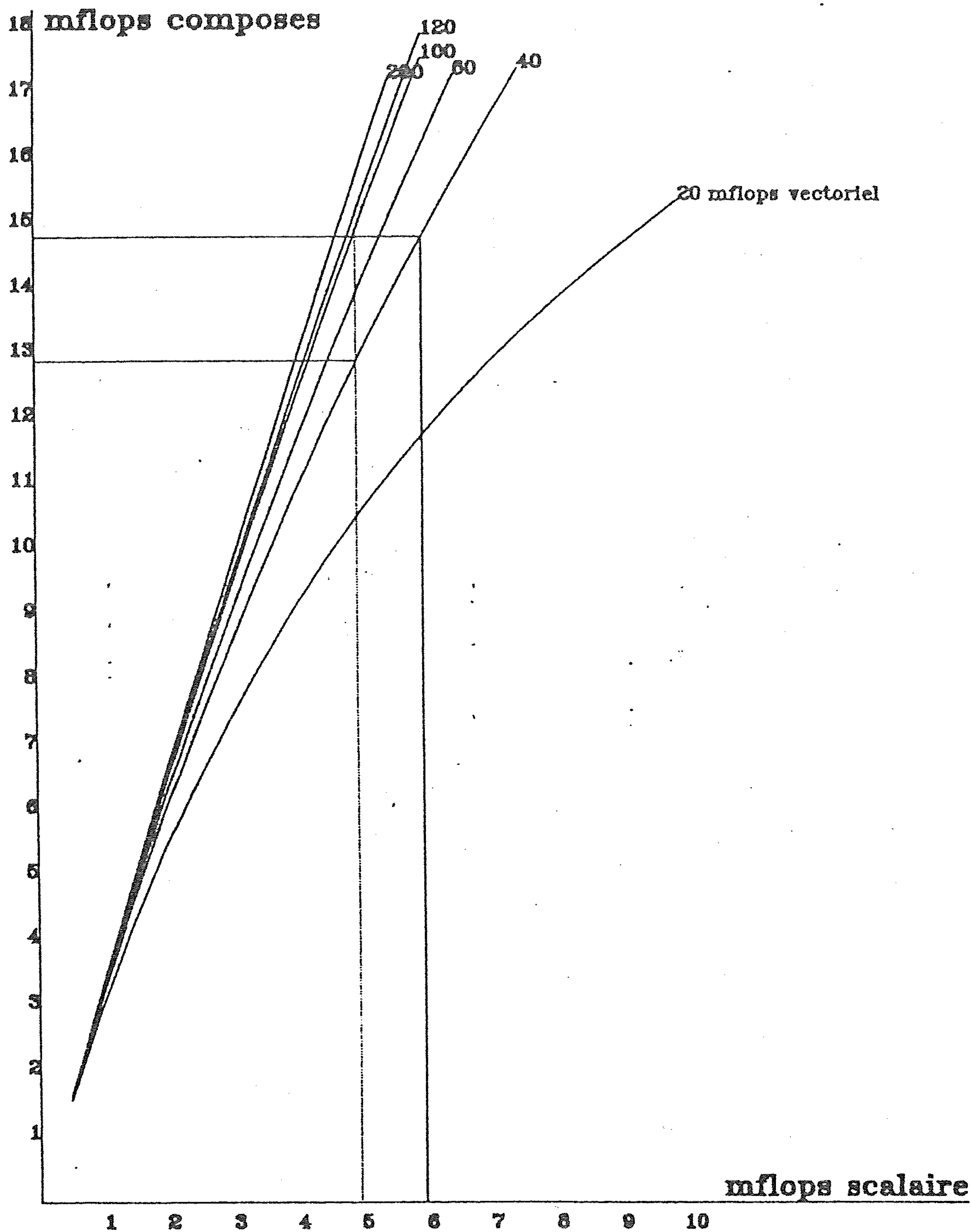

Figura 2. Comportamento dello scalare con il 70% di vettorizzazione.

Le istruzioni

Il numero di istruzioni per le macchine vettoriali e' circa il doppio del numero di istruzioni della corrispondente macchina scalare. Tra le istruzioni aggiuntive si possono citare:

1. l'addizione e la moltiplicazione,
2. istruzioni particolari per segnalare un salto incondizionato,
3. istruzione MIN(A,B) che permette la rappresentazione senza interruzione di una sequenza del tipo:

```
C=A
IF (A.GT.B) C=B
```

La comparsa degli elaboratori vettoriali ha costretto i costruttori a sviluppare dei compilatori specifici, ma poiche' al momento attuale questi compilatori non sanno generare un buon codice vettoriale, e' stato necessario creare degli 'strumenti di aiuto' alla vettorizzazione che il programmatore puo' utilizzare per condurre il compilatore ad una buona vettorizzazione del codice sorgente.

Tuttavia questi compilatori sono piu' paragonabili a dei sistemi esperti che a compilatori tradizionali, per il fatto che essi provano a operare delle scelte.

Architettura e organizzazione dell'IBM 3090-VF

L'architettura di un elaboratore e' cio' che si presenta ad un programmatore che lavora a livello di linguaggio macchina e quindi e' l'insieme di mezzi e metodi (tipo delle istruzioni, meccanismi di indirizzamento, modo di realizzare le operazioni di I/O, sistemi d'interruzione, ecc) che permettono l'utilizzo del calcolatore.
Si parla invece di 'design' per descrivere il modo in cui una architettura e' stata realizzata su un particolare elaboratore.

La domanda della SEAS e la risposta dell'IBM

La SEAS (Share European Association) e' l'associazione che raggruppa in Europa gli utenti dei sistemi IBM che operano in particolare nel campo ingegneristico/scientifico. Le universita' ed i grandi organismi di ricerca che lavorano con materiali IBM, si scambiano qui le loro esperienze e fanno delle domande (requirements) ufficiali all'IBM per il miglioramento e lo sviluppo di questo o quel prodotto.
Coscienti della rapida crescita dei bisogni di alte potenzedi calcolo, molti membri della SEAS hanno creato nel 1982 un gruppo di lavoro (High Performance Processing Project) con l'obiettivo non di definire un nuovo supercalcolatore, ma di stabilire le necessita' del calcolo scientifico e di proporre delle soluzioni ben integrate nell'ambiente IBM. Gli studi fatti da questo gruppo hanno portato a depositare una richiesta ufficiale nei confronti dell'IBM per la realizzazione, a costi contenuti, di un sistema che avesse le seguenti caratteristiche:

1. Performance reale di 30 megaflops effettivi al 50% di vettorizzazione
2. Precisione standard di 64 bit
3. Taglia minima di memoria di 32 Mb
4. Ambiente MVS integrato e trasparente
5. Costo ragionevole

Da sottolineare che la richiesta di una potenza di 30 megaflops deve essere raggiunta con applicazioni vettorizzabili al 50% poiche' queste rappresentano le applicazioni piu' comuni nel campo scientifico. Si tratta dunque di un processore che ha delle prestazioni medie (dello stesso ordine del CRAY-1), ma la cui utilizzazione deve essere la piu' trasparente possibile per i ricercatori e gli utenti abituati all'ambiente MVS.

L'IBM ha dato le seguenti risposte:

1. Marzo 1983: la richiesta e' messa allo studio
2. Aprile 1984: la richiesta e' accettata (questo significa che l'IBM propone una soluzione)
3. Ottobre 1985: e' annunciata la Vector Facility

Molte delle soluzioni rispondenti alle richieste della SEAS, erano state soddisfatte, come, per esempio, realizzare un processore utilizzabile da numerosi elaboratori IBM in 'front-end', ed avente eccellenti prestazioni scalari.

La strategia seguita dall'IBM e' dunque stata:

1. Prendere un processore scalare molto veloce (3090).
2. Aggiungere una unita' vettoriale poco costosa e con prestazioni medie.
3. Aggiungere del software per il supporto di questa unita' con:
   - Compilatore Fortran vettorizzante
   - Libreria Fortran vettorizzata
   - Biblioteca scientifica vettorizzata.
4. Permettere il trattamento parallelo attraverso l'utilizzo di architetture aventi piu' processori.

Un processore cosi' strutturato si pone percio' come soluzione intermedia tra elaboratori scalari provvisti di piu' processori ed i super-calcolatori capaci di raggiungere una potenza di diverse centinaia di megaflops. Quanto detto e' schematizzabile come segue:

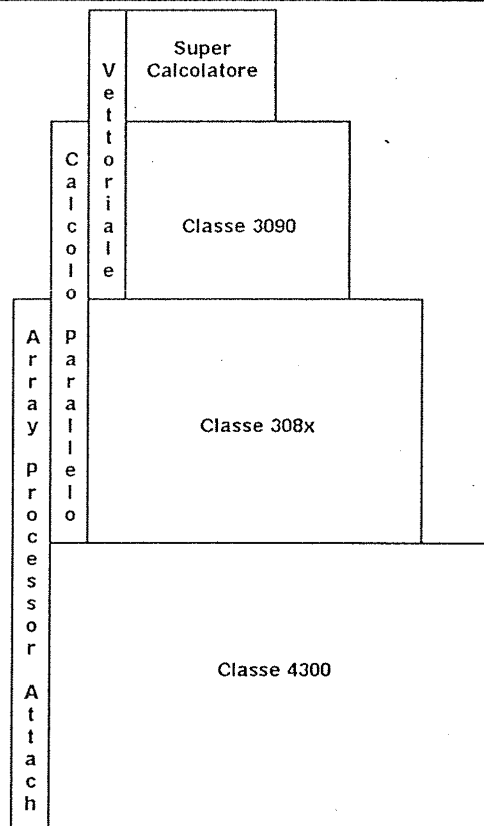

Figura 3. Posizione del 3090-VF negli elaboratori scientifici

Architettura IBM/370 estesa (370-XA)

Il 3090 puo' funzionare sia in architettura 370 che in architettura 370-XA. Il lettore desideroso di conoscere in dettaglio queste architetture, potra' far riferimento al manuale *IBM Principle of Operations;* daremo comunque qui un breve cenno alla architettura 370 estesa (370-XA).

L'architettura 370 estesa apporta un certo numero di importanti novita' rispetto alla precedente architettura 370, che sono:

1. L'indirizzamento di memoria esteso a 31 bit.

   Questa estensione permette di raggiungere una dimensione massima dello spazio indirizzabile di 2 GB contro i 16 MB della architettura 370. Poiche' il software (sistema e applicazione) ha bisogno di una memoria sempre piu' grande, le prestazioni globali dell'elaborazione possono essere migliorate se l'insieme programmi e dati puo' risiedere in memoria centrale. Il vantaggio e' ancora piu' evidente nelle elaborazioni di tipo scientifico dove vengono spesso usate matrici di grosse dimensioni.

2. Configurazioni unita' di I/O che possono arrivare fino a 4096 unita' fisiche.

3. Gestione dell'I/O piu' efficiente con l'introduzione di un processore specifico presente nella macchina, che svolge le seguenti funzioni:

   - sceglie le vie d'accesso all'unita' di I/O
   - gestisce la coda di attesa delle operazioni differite, cioe' nel caso in cui l'unita' o la *control unit* siano occupate
   - non richiede che la fine di una operazione di I/O venga gestita dalle stesse unita' (canale e processore) che l'hanno iniziata.

Architettura dell'unita' di calcolo vettoriale

L'architettura vettoriale, che e' descritta in dettaglio nel manuale *System/370 Vector Operations,* consta dei seguenti elementi:

1. Memoria e registri

   L'architettura vettoriale del 3090 dispone di 8 registri vettoriali in doppia precisione (64 bit), riconfigurabili anche come 16 registri in semplice precisone (32 bit). Ognuno dei registri puo' contenere 128 elementi aventi ciascuno una lunghezza di 32 o 64 bit. La scelta, per un modello particolare, della dimensione dei registri vettoriali e' in funzione del rapporto prezzo/prestazioni, in quanto, in generale, l'architettura vettoriale stabilisce che il numero di elementi contenuto in un registro vettoriale puo' essere una qualunque potenza di 2 compresa fra 8 e 512. E' percio' importante precisare che questa scelta di 128 elementi operata per il 3090-VF puo' essere cambiata su altri modelli.

2. Vector mask mode

   Il *Vector mask mode* e' un indicatore che segnala se l'elaborazione deve interessare tutti gli elementi del vettore oppure deve coinvolgere un sottoinsieme di essi selezionato mediante l'uso del registro maschera (*Vector Mask Register*).

3. Vector status register

   Il *Vector Status Register* contiene le informazioni necessarie per il controllo della *Vector Facility,* come:

   - l'indicazione se il *Vector Mask Mode* e' attivo o meno
   - le informazioni sulle istruzioni vettoriali interrotte
   - la lunghezza dell'attuale vettore da elaborare
   - ecc...

4. Vector Mask Register

   Il *Vector Mask Register* viene utilizzato per le operazioni condizionali (come l'istruzione COMPARE) ed e' costituito di 128 bit, ognuno dei quali corrisponde ad un elemento del vettore. L'istruzione corrente sara' eseguita in base allo stato del bit (ON o OFF) corrispondente all'elemento da elaborare.

Se per esempio si volesse evitare una divisione per zero, si puo' preventivamente operare una comparazione vettoriale degli elementi del vettore 'divisore' con il valore zero per impostare i bit del registro maschera; gli elementi cosi' mancanti non parteciperanno alla istruzione di divisione.

5. Vector activity count.

   Associato al vector status register, il vector activity count permette di distribuire e di controllare l'utilizzo delle risorse vettoriali (analisi dei lavori, tuning ecc..).

L'architettura degli elementi della architettura vettoriale del 3090-VF puo' essere cosi' raffigurata:

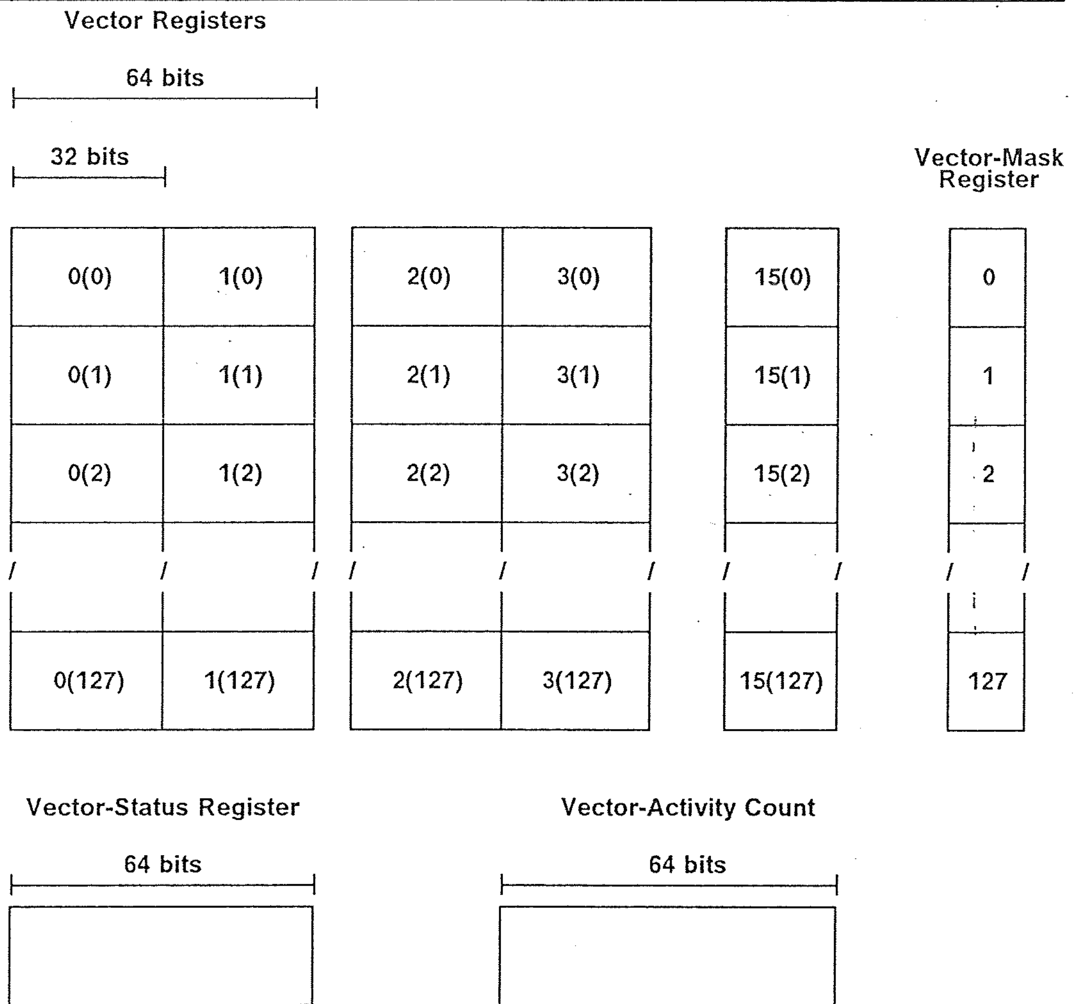

Figura 4. Schema dei registri vettoriali del 3090-VF

Istruzioni vettoriali

Il numero di istruzioni 370 e' stato esteso per il supporto della Vector Facility con l'aggiunta di ulteriori 171 nuove istruzioni di tipo vettoriale. Queste nuove istruzioni elaborano i dati sia in singola che in doppia precisione ed inoltre possono eseguire operazioni aritmetiche e logiche su vettori posti sia nella memoria centrale che nei registri vettoriali. Altre istruzioni permettono di realizzare la

funzione di concatenamento (MULTIPLY and ADD), mediante l'esecuzione di due operazioni aritmetiche per ciclo.

La tabella seguente contiene l'elenco delle istruzioni con il numero delle istruzioni di ogni tipo:

ISTRUZIONI	NUMERO
ADD, SUBTRACT AND, EXCLUSIVE OR, OR DIVIDE MULTIPLY	24 12 8 12
MULTIPLY AND ADD/SUBTRACT MULTIPLY AND ACCUMULATE ACCUMULATE LOAD COMPLEMENT LOAD NEGATIVE, LOAD POSITIVE SHIFT LEFT/RIGHT SINGLE LOGICAL MAXIMUM ABSOLUTE MAXIMUM/MINIMUM SIGNED	12 4 4 3 6 2 2 4
COMPARE LOAD, LOAD MATCHED STORE, STORE MATCHED LOAD EXPANDED, STORE COMPRESSD LOAD INTEGER VECTOR LOAD/STORE HALFWORD LOAD ZERO LOAD/STORE INDIRECT	12 14 4 4 1 2 2 4
LOAD BIT INDEX	1
SUM/ZERO PARTIAL SUMS	2
RESTORE/SAVE VR, SAVE CHANGED VR RESTORE VSR CLEAR VR	3 1 1
COUNT ONES/LEFT ZEROS IN VMR COMPLEMENT VMR, TEST VMR AND TO VMR, EXCLUSIVE OR TO VMR LOAD VMR, LOAD COMPLEMENT VMR OR TO VMR, STORE VMR	2 2 2 2 2
RESTORE VMR, SAVE VMR	2
EXTRACT ELEMENT, LOAD ELEMENT	6
EXTRACT VCT/VECTOR MASK MODE LOAD VCT AND UPDATE LOAD VCT FROM ADDRESS RESTORE VAC, SAVE VAC, SAVE VSR SET VECTOR MASK MODE STORE VECTOR PARAMETERS	6 1 1 3 1 1

Figura 5. Elenco delle istruzioni vettoriali

Alcune considerazioni su queste nuove istruzioni vettoriali:

- le istruzioni vettoriali possono avere fino a 3 operandi per referenziare sia i registri vettoriali, sia i registri scalari, sia l'indirizzo di memoria contenente il primo elemento di un vettore. In funzione del tipo d'istruzione, i risultati possono essere collocati o in un registro scalare, o in un registro vettoriale.
- Tutte le istruzioni vettoriali sono interrompibili ma vengono conservate tutte le informazioni relative all'elemento in cui e' avvenuta l'interruzione. In questo modo l'istruzione vettoriale potra' essere ripresa nel punto esatto dell'interruzione, senza necessariamente dover riprendere l'esecuzione dell'istruzione dall'inizio.
  Questo meccanismo consente inoltre di identificare facilmente gli elementi che causano un errore (per esempio, una divisione per zero), in quanto, in base all'interruzione, e' possibile risalire con precisione all'elemento del vettore che e' stato responsabile dell'errore.

Accesso ai dati della memoria

Il 3090-VF puo' accedere ai dati in due modi differenti:

- con indirizzamento sequenziale, quando gli elementi del vettore sono ordinati in posizioni contigue della memoria, o in posizioni separate distanti tra da loro da un *passo* costante. Questo passo costante viene chiamato *STRIDE* e vedremo in seguito che puo' avere un effetto importante sulle prestazioni.
- Con indirizzamento indiretto, sotto il controllo della maschera, quando gli elementi sono posizionati senza un ordine preciso.

Poiche' la dimensione dei registri vettoriali e' di 128 elementi, l'elaborazione vettoriale viene effettuata a gruppi di 128 (sezione del vettore) con esclusione dell'ultimo gruppo che puo' avere una dimensione o sezione inferiore.
Se abbiamo per esempio un ciclo DO che opera su di un vettore di lunghezza 200, il compilatore FORTRAN generera' una sequenza d'istruzioni comprendenti il caricamento nei registri vettoriali e le operazioni aritmetiche per l'esecuzione dei primi 128 elementi del vettore; in seguito le stesse operazioni saranno ripetute per i restanti 72 elementi del vettore.

Per gli esperti di linguaggio macchina, sono mostrate, nell'esempio seguente, le traduzioni operate dal compilatore sia in modo scalare e che in modo vettoriale:

```
Programma FORTRAN

              REAL*8 A(1000), B(1000), C(1000)
              DO 10 I=1,N
       10     C(I) = A(I) + B(I)


Istruzioni scalari

       L      G9,=F'1'      Carica '1' nel registro generale 9
       SLL    G9,3          Calcola dimensione di ogni elemento (8)
       L      G10,=F'8'     Carica incremento per BXLE
       L      G11,N         Carica numero degli elementi
       SLL    G11,3         Calcola indirizzo dell'ultimo elemento
                            (valore di comparazione per BXLE)
 LOOP  LD     F0,A(G9)      Carica A(I) nel FPR 0
       AD     F0,B(G9)      ADD B(I) nel FPR 0
       STD    F0,C(G9)      Registra il risultato nel C(I)
       BXLE   G9,G10,LOOP   Ripeti per tutti gli elementi del vettore
```

```
Istruzioni vettoriali

        L      G0,N        Carica lunghezza del vettore nel RG 0
        LA     G1,A        Carica indirizzo di A nel RG 1
        LA     G2,B        Carica indirizzo di B nel RG 2
        LA     G3,C        Carica indirizzo di C nel RG 3
  LOOP  VLVCU  G0          Carica numero di elementi da elaborare
        VLD    V0,G1       Carica la sezione di A
        VAD    V0,V0,G2    ADD la sezione di B
        VSTD   V0,G3       Registra sezione e risultati in C
        BP     LOOP        Continua se c'e' un' altra sezione
```

Struttura scalare del 3090

E' stato accennato in precedenza che la conoscenza dell'architettura e della struttura di un elaboratore e' importante ai fini di una buona programmazione. Non e' tuttavia nostra intenzione entrare nei dettagli per quanto riguarda la tecnologia costruttiva della serie 3090; diciamo semplicemente che l'unita' centrale e' a base di chips ECL (Emitter Coupled Logic) assemblati su TCM (Thermal Conduction Module) e che il ciclo di base e' di 17.2 ns, contro i 24 ns del 3081-KX. E' importante invece elencare gli elementi che fanno parte della struttura del 3090:

1. Unita' centrale del 3090

   La configurazione dell'unita' centrale del 3090 dipende dal modello ma in generale si compone di:

   - da 1 a 4 unita' di calcolo
   - un controllore di sistema
   - un processore di canali
   - una memoria centrale
   - una memoria estesa (opzionale).

   L'organizzazione generale e' detta 'diadica', cioe' le unita' centrali, se presenti, sono identiche e si dividono l'utilizzo della memoria centrale e dell'insieme dei canali di I/O. La gestione delle unita' centrali e del processore dei canali e' a carico del controllore di sistema che provvede anche ad un utilizzo ottimale di tali risorse.

2. Gerarchie della memoria.

   La struttura della memoria si compone di piu' livelli ognuno dei quali costituisce un elemento della gerarchia e quindi gioca un ruolo importante nelle prestazioni globali del sistema. I vari livelli formano una struttura piramidale nella quale gli elementi piu' alti rappresentano le memorie piu' veloci, ma anche le piu' costose e quindi le piu' limitate come dimensioni.

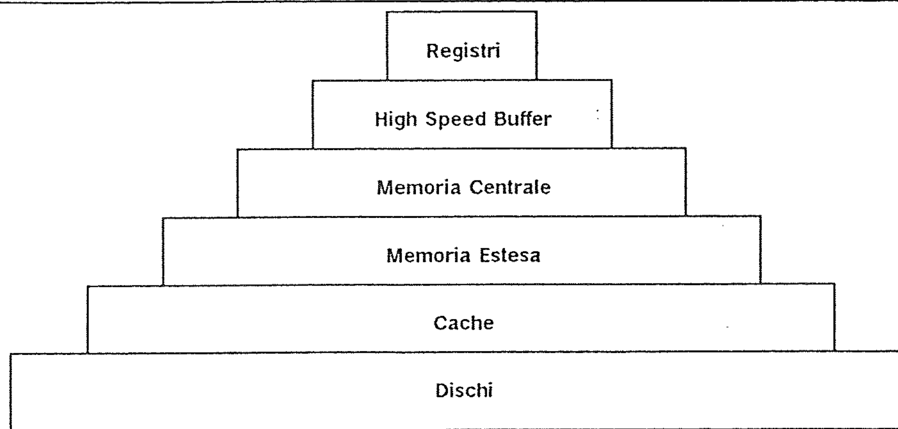

Figura 6. Gerarchia della memoria.

- Memoria centrale

  La programmazione scientifica e' facilitata dalla disponibilita' della memoria centrale di grandi dimensioni, che permette di ridurre, o di eliminare, le segmentazioni dei programmi e di aumentare la quantita' di dati da elaborare contemporaneamente, consentendo una notevole diminuzione delle operazioni di I/O.
  Le taglie della memoria centrale (reale) disponibili sul 3090 possono andare da 32 MB sul modello piu' piccolo fino a 256 MB sul modello piu' grande.

- Memoria di primo livello (o High Speed Buffer)

  Il tempo di esecuzione di molte istruzioni e' molto piu' breve del tempo di accesso alla memoria centrale dell'elaboratore, per cui quest'ultima rischia di diventare un freno per le unita' di calcolo. Essendo le tecnologie di costruzione di una memoria piu' veloce molto costose, e' stato trovato un compromesso con l'introduzione di una memoria di primo livello (o *High Speed Buffer*), che ha una piccola capacita' (64 KB per processore nel 3090), ma e' molto veloce e direttamente accessibile dall'unita' di calcolo. La sua gestione, che la fa sembrare come *una finestra di accelerazione* posta davanti alla memoria centrale, e' interamente a carico dall'hardware, rendendola dunque completamente trasparente al programmatore.

- Memoria estesa

  Come gia' detto il tempo d'accesso alle informazioni memorizzate sui dischi si esprime in decine di millisecondi, che costituisce un tempo considerevole in rapporto al ciclo base dell'unita' centrale (qualche nanosecondo). Sulla serie 3090 e' stata dunque annunciata una memoria estesa che ha un tempo d'accesso inferiore al microsecondo. Tale memoria viene gestita direttamente dal sistema operativo alla stessa stregua della memoria centrale, per cui, riducendo considerevolmente il carico di paginazione e di swap da e verso i dischi, si migliorano le prestazioni globali del sistema.
  Sui vari modelli della serie 3090 la memoria estesa puo' arrivare fino a 1024 MB.

3. Ricerca di informazioni

   Quando una istruzione necessita di una informazione, possono presentarsi tre casi:

   - l'informazione e' gia' nella memoria di primo livello (High Speed Buffer):

in questo caso il tempo d'accesso e' uguale al ciclo base dell'unita' centrale;

- l'informazione e' in memoria centrale:
  deve essere prelevata e scritta nella memoria di primo livello, quindi il tempo d'accesso e' uguale al ciclo della memoria centrale;

- l'informazione e' in memoria virtuale:
  in questo caso, il sistema operativo deve intervenire per trasferirla in memoria reale; il tempo d'accesso sara' quindi dell'ordine di decine di millisecondi.

Il ruolo della memoria di primo livello e' particolarmente efficace nei casi in cui si referenzino le stesse istruzioni e/o dati molto frequentemente, poiche' aumenta la probabilita' che questi si trovino gia' nell' High Speed Buffer. Per tale motivo i benefici maggiori si hanno nei programmi scientifici, dove non e' raro passare piu' dell' 80% del tempo su qualche decina di istruzioni.

4. I modelli 3090

	Mod.150E	Mod.180E	Mod.200E	Mod.400E
Numero di Unita' Centrali	1	1	2	4
Ciclo base (ns)	17.2	17.2	17.2	17.2
High Speed Buffer (KB)	64	64	128	256
Memoria centrale (MB)	32-64	32-64	64-128	128-256
Memoria estesa (MB)	0-128	0-256	0-512	0-1024
Numero di canali	16-24	16-32	32-64	64-128

A conclusione della struttura scalare del 3090, si possono riassumere tutti gli elementi architetturali, trattati in questo paragrafo, in uno schema che e' mostrato in Figura 7

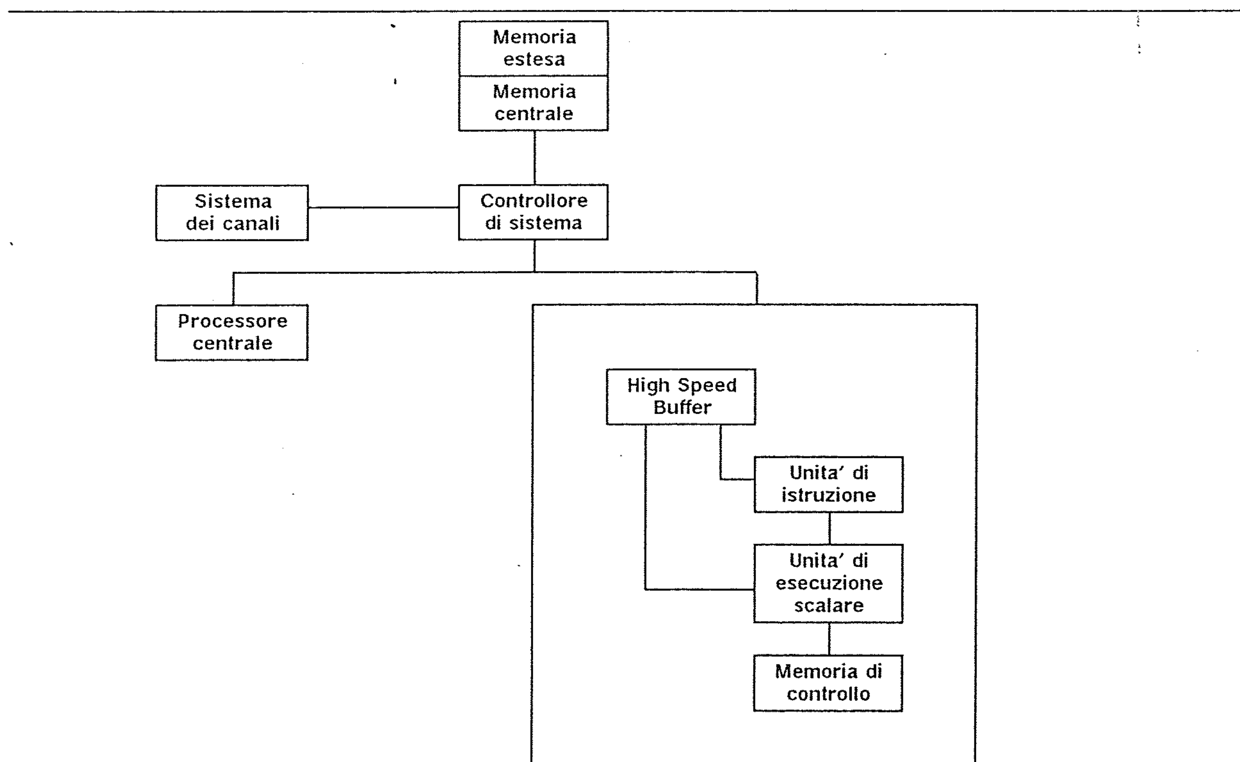

Figura 7. Elementi funzionali del 3090 scalare

Struttura della Vector Facility

L'unita' vettoriale e' integrata nel processore centrale, come e' mostrato nella figura seguente:

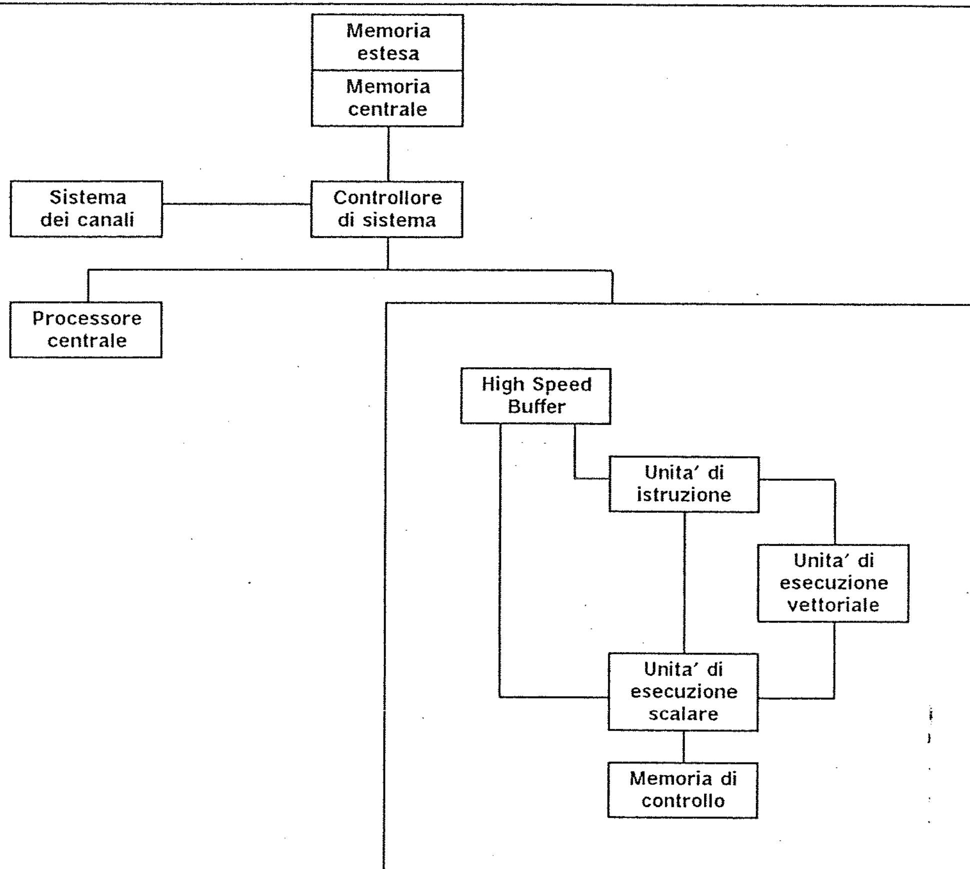

Figura 8. Elementi funzionali del 3090-VF

Il funzionamento del 3090-VF avviene come segue: ogni istruzione e' decodificata dall'unita' di istruzione che se la riconosce come una delle 171 nuove istruzioni vettoriali, la invia all'unita' di esecuzione vettoriale, in caso contrario la dirige all'unita' di esecuzione scalare.
Il flusso di istruzioni e' in genere un miscuglio di istruzioni scalari e vettoriali, ma, in un dato momento, c'e' una sola istruzione in esecuzione per ogni unita' di calcolo.

Se la configurazione e' asimmetrica, e cio' si verifica quando esiste una sola Vector Facility con 2 o piu' processori scalari, e' compito del dispatcher del sistema operativo far eseguire il programma sul processore che dispone della VF, non appena viene incontrata un'istruzione vettoriale.

Come il processore scalare, l'unita' vettoriale segue gli stessi modi per l'accesso ai dati, per cui sfrutta tutte le gerarchie di memoria del 3090 e le possibilita' offerte dall'indirizzamento virtuale.

L'unita' di esecuzione vettoriale

L'unita' di esecuzione vettoriale del 3090 e' composta dagli elementi rappresentati nella seguente figura:

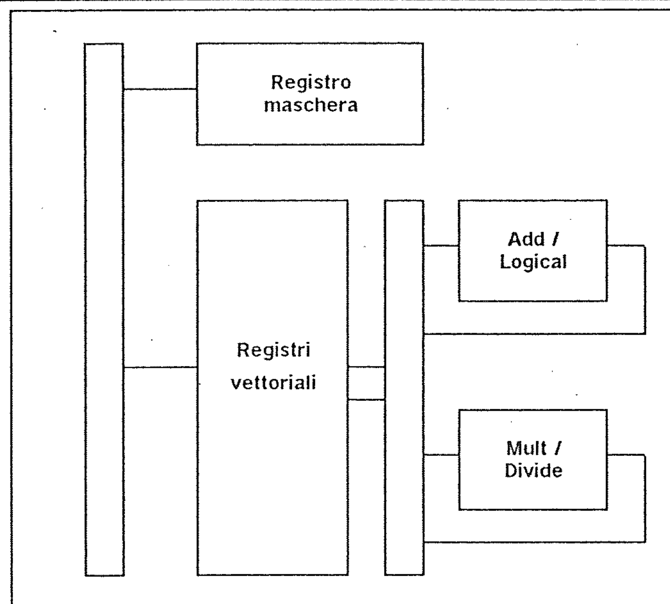

Figura 9. Unita' di esecuzione vettoriale

1. Unita' vettoriali di calcolo

   Le unita' aritmetiche e logiche dell'unita' di esecuzione vettoriale utilizzano la tecnica della pipeline. La Vector Facility del 3090 dispone di 2 pipeline, di cui la prima, chiamata *Add/Logical* e' utilizzata per le addizioni, le sottrazioni, i confronti e le operazioni logiche, mentre la seconda, chiamata *Mult/Divide* e' utilizzata per le operazioni di moltiplicazione e divisione.

   Come gia' detto, la caratteristica di una pipeline e' che, una volta inizializzata, fornisce un risultato ad ogni ciclo-macchina; tuttavia, delle due pipeline presenti nel 3090, una sola puo' essere in funzione in un dato momento, a meno che non sia in corso di esecuzione una istruzione *composta,* per cui avviene il concatenamento. Una istruzione composta, come il Multiply and Add, crea un flusso di dati tale che l'uscita della pipeline Mult/Divide corrisponde all'ingresso della pipeline Add/Logical per fornire un risultato ad ogni ciclo, benche' l'istruzione consista in due operazioni aritmetiche.

2. Registri vettoriali

   Per poter fornire un risultato per ciclo, la pipeline deve essere alimentata in ingresso ad una velocita' sufficiente, ma poiche' il ciclo base dell'unita' centrale e' piu' rapido di quello della memoria centrale, e' stato necessario fornire la macchina di registri vettoriali. Il ruolo dei registri, che sono delle memorie locali situate nella VF e che hanno lo stesso ciclo base della unita' di calcolo, e' dunque quello di alimentare la pipeline. Ricordiamo che l'unita' di calcolo vettoriale del 3090 consta di 16 registri vettoriali in precisione semplice o 8 registri in doppia precisione, ognuno dei quali comprende 128 elementi.

3. Potenza di calcolo vettoriale

   A questo punto, e' possibile calcolare la massima potenza istantanea della VF che si ottiene durante un operazione di concatenamento, in cui vengono eseguite 2 istruzioni per ciclo, e quando i 128 elementi sono gia' caricati nel registro vettoriale. Poiche' il ciclo base e' di 17.2 ns, ne deriva che la VF puo' operare ad una velocita' massima di 116 milioni di operazioni floating point per secondo quando esegue una istruzione composta (2 istruzioni/17.2 nanosecondi). Tale risultato e' pero' ottenibile per un tempo di 2.202

microsecondi che e' il tempo necessario per l'elaborazione dei 128 elementi presenti nel registro vettoriale.

E' evidente che ci sara' interesse a sfruttare le prestazioni date dal concatenamento, ogni volta che sara' possibile. Se si dispone di piu' unita' di calcolo vettoriale, che possono essere fino a 4, e si utilizzano le possibilita' del 'Multitasking', si potra' raggiungere una potenza *teorica* di 464 Megaflops.

Software utilizzabile

Sistemi operativi

L'unita' di calcolo vettoriale e' supportata dai seguenti sistemi operativi, a partire dalle versioni indicate tra parentesi:

- MVS (MVS/SP 2.1.3 VFE e MVS/XA DFP 1.2)
- VM (VM/SP HPO 4.2)

Linguaggi e librerie

Del software applicativo e' stato creato appositamente per l'utilizzo della VF e sara' percio' oggetto di analisi in seguito, per cui qui ne viene fornito solo un elenco:

1. VS Fortran versione 2

   Comprende:

   - il compilatore VS Fortran versione 2. Questo compilatore-vettorizzatore e' interamente compatibile con il VS Fortran versione 1 e la vettorizzazione e' collegata alla specifica di un livello di ottimizzazione determinato.
   - La libreria Fortran (livello 1.4.1), che comprende il Multitasking Facility (PMTF).
   - Il debug interattivo piu' esteso in confronto alla versione 1: supporto full-screen, finestre, ecc..

2. Libreria ESSL

   La 'Engineering and Scientific Subroutine Library' e' composta da un insieme di sottoprogrammi matematici ottimizzati per sfruttare le caratteristiche del processore 3090 e dell'unita' di calcolo vettoriale. Alcune versioni in semplice e doppia precisione sono fornite per i seguenti campi applicativi:

   - Algebra lineare
   - Operazioni con matrici
   - Equazioni simultanee di algebra lineare
   - Analisi di auto-valori
   - Trattamento dei segnali

3. Analizzatore dell'esecuzione Vs Fortran
   (Execution Analyzer - EA)

   L'analizzatore dell'esecuzione e' destinato ad aumentare l'ottimizzazione dei programmi scritti in VS Fortran, per mezzo dell'identificazione di quelle parti del programma che sono piu' frequentemente eseguite. E' dunque uno strumento molto importante nella fase di vettorizzazione delle applicazioni.

4. Programma di conversione dei linguaggi Fortran IBM (LCP) (Language Conversion Program - LCP)

   Questo programma converte le istruzioni dei programmi sorgente scritti in Fortran IBM livello 66 (Fortran G1, HX, Q) e del VS Fortran versione 1 in istruzioni equivalenti del VS Fortran livello 77, facilitando cosi' la standardizzazione al livello 77.

VS Fortran versione 2

Il prodotto

Il VS Fortran versione 2 e' stato annunciato nell'ottobre 1985. E' compatibile con il VS Fortran versione 1, livello 4.1, ed apporta nuove funzioni al livello di ogni suo componente:

1. Compilatore con opzione di vettorizzazione per ottenere un codice utilizzabile dall'unita' di calcolo vettoriale dell'IBM 3090.
2. Libreria aggiornata con:
   - funzioni matematiche, in singola e doppia precisione, che sono state ottimizzate sia per l'esecuzione scalare che vettoriale.
   - sottoprogrammi che permettono l'esecuzione parallela.
3. Debug interattivo.

   Il debug non solo e' un eccellente strumento di messa a punto, ma puo' essere considerato come un vero ambiente di lavoro sia per l'esecuzione dei programmi, che per la misura delle prestazioni.

Compilatore

Dopo il 1981, che segna l'anno della disponibilita' della sua versione 1, livello 1, il compilatore VS Fortran rimpiazza progressivamente i precedenti compilatori (G1 e H Extended) sul sistema IBM 370. Il compilatore Fortran versione 2 si distingue dalla versione 1 solo per l'opzione della vettorizzazione, per cui la descrizione delle caratteristiche che saranno qui elencate si trovano anche nella versione 1, livello 4.1.
Queste caratteristiche sono:

1. Trasportabilita' del Compilatore

   Il caricamento in memoria e l'esecuzione di un codice oggetto, che e' stato ottenuto da una compilazione sotto un sistema operativo, possono essere effettuati sotto un altro sistema operativo. In caso di codice vettorizzato, il sistema utilizzato deve essere MVS/SP2 o VM/HPO.

2. Rientranza del compilatore (sotto VM e MVS)

   Per limitare le operazioni di I/O sui dischi, e dunque migliorare le prestazioni, il compilatore puo' risiedere interamente in memoria reale. La rientranza permette inoltre di mantenere un'unica copia a disposizione di tutti gli utenti, evitando cosi' dei duplicati inutili e costosi per l'occupazione della memoria centrale.

3. Codice oggetto prodotto dal compilatore rientrante (opzione RENT)

   Per le stesse ragioni dette nel caso del compilatore, la creazione di un codice oggetto rientrante puo' essere vantaggiosa per produrre del software

scritto in Fortran che puo' essere contemporaneamente messo a disposizione di numerosi utenti sullo stesso sistema.

4. Ambiente architettura estesa

   I moduli oggetto, come la maggior parte dei sottoprogrammi della libreria possono beneficiare della capacita' di indirizzamento esteso fino a 2 GB.

5. *Common* dinamici

   Un'opzione del compilatore permette di realizzare l'allocazione dinamica di memoria, per mezzo di un *Common,* al momento dell'esecuzione di una unita' di programma che lo referenzia. Si ricorda che le unita' di programma che hanno definito delle zone comuni possono scambiarsi delle informazioni per mezzo dell'istruzione di *Common.*

6. Aumento automatico della precisione

   Con una semplice opzione di compilazione, e' possibile operare un raddoppio della precisione con o senza riempimento (padding). Con tale opzione si puo' ottenere:

```
Precisione semplice (REAL*4) ---> Precisione doppia (REAL*8)
Precisione doppia   (REAL*8) ---> Precisione estesa (REAL*16)
```

7. Compatibilita' ascendente.

   Il VS Fortran compila i programmi scritti in Fortran 77 o in Fortran 66 e la scelta del livello di linguaggio si effettua per ogni unita' di programma. In questo modo i moduli oggetto prodotti dal VS Fortran con l'opzione 66 o 77 possono essere eseguiti insieme ai moduli oggetto prodotti dai precedenti compilatori di tipo Fortran IV (Fortran G1, Fortran H esteso).

8. Estensione della sintassi del Fortran 77

   Il VS Fortran e' conforme alle norme del Fortran 77 ed in piu' presenta delle estensioni al linguaggio che hanno le seguenti caratteristiche:

   - Tre livelli di precisione per le variabili reali e complesse:

```
semplice   (REAL*4,  COMPLEX*8)
doppia     (REAL*8,  COMPLEX*16)
estesa     (REAL*16, COMPLEX*32)
```

   - Istruzione INCLUDE.
   - Programma sorgente in formato libero.
   - Supporto dei files VSAM (particolarmente l'accesso con le chiavi in KSDS).
   - In MVS, supporto di files sequenziali asincroni.
   - Lettura-scrittura di files interni con formato NAMELIST.
   - Funzioni di manipolazione dei bit.
   - Istruzioni di Debug statico.

9. Messa a punto del programma

   Per la messa a punto del programma il VS Fortran fornisce due strumenti:

   - Un debug statico che e' ottenuto con l'aggiunta di un 'pacchetto' di istruzioni specializzate in testa all'unita' di programma di cui si vuole

fare il debug. Tali istruzioni consentono la stampa di informazioni utili per la messa a punto del programma.

- Un debug interattivo che non necessita ne' di istruzioni supplementari, ne' di ricompilazioni con un'opzione speciale. Il debug interattivo permette di effettuare delle indagini lavorando direttamente da terminale.

Inoltre, il compilatore fornisce diverse informazioni utili alla messa a punto che possono essere selezionate sfruttando alcune opzioni di compilazione:

SOURCE	Lista del programma sorgente e dei messaggi di errore
MAP	Lista delle variabili di ogni programma con il loro indirizzo di memoria
XREF	Lista dei riferimenti incrociati
LIST	Lista del modulo oggetto tradotto in linguaggio assembler
SDUMP	Fornisce un'analisi del contenuto delle variabili, sia in caso di fine normale, che nel caso di una chiamata CALL SDUMP
VECREP	Lista di vettorizzazione

10. Ottimizzazione:

I quattro livelli di ottimizzazione permettono di passare da una compilazione veloce, ma senza ottimizzazione del codice oggetto, ad una piu' elaborata che porta ad un codice oggetto molto efficiente. I livelli di ottimizzazione sono:

OPT(0)	nessuna vettorizzazione
OPT(1)	ottimizzazione a livello di registri
OPT(2)	ottimizzazione parziale del testo
OPT(3)	ottimizzazione totale del testo

11. Vettorizzazione.

La vettorizzazione e' realizzata sotto il controllo dell'opzione di compilazione *VECTOR:*

VECTOR(LEVEL(0))	nessuna vettorizzazione
VECTOR(LEVEL(1))	vettorizzazione a livello di DO loop interamente vettorizzabili
VECTOR(LEVEL(2))	vettorizzazione istruzione per istruzione, cioe' di DO loop in cicli vettorizzabili e non.

Esiste un parametro dell'opzione VECTOR *(VECTOR(REPORT(XLIST)))* che fornisce una lista in cui sono evidenziati i DO loop vettorizzati. L'utente potra' trarne vantaggio per modificare il codice sorgente in prospettiva di una migliore vettorizzazione.

Per l'esame dei DO loop il compilatore opera come segue:

- ricerca gli eventuali *inibitori* della vettorizzazione
- aggiusta il codice per eliminare le dipendenze
- scambia i livelli di concatenamento dei DO loop
- sceglie il metodo di esecuzione piu' rapido

L'utilizzazione delle *direttive* permette poi all'utente di fornire delle informazioni supplementari al compilatore che possono portare alla vettorizzazione di alcuni segmenti del programma:

- imponendo la vettorizzazione o la non vettorizzazione di un DO loop senza far ricorso ad una stima della velocita' d'esecuzione
- fornendo una stima del numero di iterazioni di un DO loop
- chiedendo che non sia tenuto conto di una inibizione dovuta a dipendenze.

12. Parametri per il controllo dell'esecuzione

Per concludere la descrizione del prodotto VS Fortran, c'e' da sottolineare che nella versione 2 sono stati creati alcuni parametri per il controllo dell'esecuzione che ci sembra opportuno di ricordare:

ABSDUMP e NOABSDUMP	indica se deve essere prodotto o meno un *dump* della memoria in caso di fine anormale del programma.
AUTOTASK e NOAUTOTASK	utilizzo o meno dell'MTF (Multi-Tasking Facility).
DEBUG e NODEBUG	utilizzo o meno del debug interattivo.
DEBUNIT e NODEBUNIT	permette all'utente di specificare se certe unita' sono da considerarsi o meno dei terminali al momento del debug.
SPIE o NOSPIE	provoca l'esecuzione o meno di una macro (SPIE) in caso di errore. Se e' stata specificata l'opzione NOSTAE, alcune azioni vengono soppresse, come i messaggi, gli interventi correttivi in caso di overflow, ecc...
STAE e NOSTAE	provoca l'esecuzione o meno della macro STAE in caso di abend. Se e' specificato NOSTAE non si ottengono alcune informazioni come l'ultima istruzione eseguita, la traccia delle subroutines chiamate, ecc...
XUFLOW e NOXUFLOW	fa generare o meno una interruzione in caso di *exponent underflow*. Esiste comunque un sottoprogramma (XUFLOW) che produce lo stesso effetto di questa opzione.

Libreria

La libreria VS Fortran fornisce un insieme di sottoprogrammi che calcolano le funzioni di uso piu' comune:

1. Funzioni matematiche e trattamento dei caratteri

   Se per alcune funzioni, come, ad esempio, il valore assoluto, viene creato l'opportuno codice direttamente in fase di compilazione, le altre funzioni risiedono invece nella libreria. Per quanto riguarda le funzioni matematiche, la libreria del VS Fortran versione 2 e' stata generata con gli elementi della libreria EML (Elementary Mathematical Library) e dei quali e' stata creata sia la versione scalare che vettoriale. Gli algoritmi EML si applicano essenzialmente a calcoli, effettuati in semplice o doppia precisione, rivolti a:

   - radici quadrate
   - esponenziali
   - potenze
   - logaritmi
   - funzioni trigonometriche dirette
   - funzioni trigonometriche inverse.

2. Sottoprogrammi di I/O

   I sottoprogrammi di I/O del VS Fortran servono per creare un'interfaccia con il sistema per la gestione delle operazioni di ingresso/uscita.

3. Sottoprogrammi di servizio

   Alcuni di questi sottoprogrammi forniscono un aiuto per la messa a punto del programma, altri permettono di ottenere delle informazioni utili dal sistema, quali la data, l'ora e il codice di ritorno.

4. Sottoprogrammi di errore

   Questi sottoprogrammi consentono la gestione degli errori da parte del programmatore che puo' in modo particolare avvalersi della routine ERRSET.

5. Sottoprogrammi di trattamento MULTI-TASK (MTF)

   Questi sottoprogrammi, che devono essere eseguiti sotto il controllo dell'MVS, fanno si' che un programma Fortran possa girare simultaneamente su differenti processori dello stesso elaboratore, sia in modo scalare che vettoriale. Le chiamate CALL DSPTCH lanciano i sotto-task e possono essere seguite, nel programma, da una CALL SYNCHRO che serve a sincronizzare l'utilizzo dei differenti processori ed a permettere quindi la continuazione dell'elaborazione.

6. Sottoprogrammi di supporto del convertitore LCP

   Il convertitore LCP (Language Conversion Program) e' un prodotto che converte i programmi scritti in Fortran 66 in Fortran 77. Alcune funzioni non sono convertite ma sono fornite sotto forma di sottoprogrammi appartenenti alla libreria VS Fortran.

Debug interattivo

Generalita'

Il debug interattivo, che fa parte del prodotto VS Fortran versione 2, e' una estensione del debug del VS Fortran versione 1. Permette la messa a punto interattiva del codice oggetto, generato dal compilatore Fortran versione 2 sia in modo vettoriale che scalare. La sua principale prerogativa risiede nella ricchezza delle possibilita' di utilizzo, quali:

- Uso del 'full-screen' esteso (finestre, animazioni, ecc...) ottenuto mediante l'utilizzo di una finestra dove e' visualizzata l'esecuzione del codice sorgente e l'istruzione corrente e' segnalata con la doppia densita' luminosa o con un colore diverso colore.
- Uso intensivo di tasti funzionali o di quelli per il movimento del cursore.
- Debug gestito da istruzioni (modo 'batch').
- Estensione dei vecchi comandi.
- Aggiunta di nuovi comandi.
- Possibilita' di indicare le parti del codice su cui indagare a livello di unita' di programma o di gruppo di istruzioni.
- Comunicazioni con il sistema.
- Allocazione dinamica dei files.
- Manipolazione dei files sequenziali.

Il debug del VS Fortran e' disponibile sotto i sistemi VM/SP, con o senza HPO, e MVS/SP1 o MVS/SP2. Nel sistema MVS/SP2, puo' operare con indirizzamento di 31 bit, e quindi puo' analizzare i programmi che si estendono oltre i 16MB, ma comunque i moduli del debug devono risiedere sotto tale limite di memoria.

Prima di lanciare l'esecuzione, l'utente puo' indicare, mediante delle specifiche poste nel file *AFFON,* la lista delle parti del programma di cui si desidera fare il debug; questa lista puo' essere a due livelli di dettaglio: unita' di programma o gruppo di istruzioni. Il resto del programma sara' eseguito senza debug e quindi a velocita' normale.

Comandi di debug

Di seguito viene fornita una lista dei principali comandi che possono essere usati nell'ambiente di debug. Tali comandi sono stati suddivisi a seconda del tipo di operazione che l'utente intende effettuare sul programma sotto test.

Controllo dell'esecuzione del programma:

In questa fase l'utente puo' emettere comandi per espletare le seguenti funzioni:

- segnalazione dei punti di controllo;
- sospensione condizionale dell'esecuzione;
- arresto o ripresa dell'esecuzione;
- esecuzione istruzoine per istruzione;
- fine del debug.

I comandi di questa fase sono:

COMANDO	FUNZIONI	VSF V.1	VSF V.2
AT	Posizionamento di un punto di controllo (con lista comandi)	*	*,PF
GO	Rilancio dell'esecuzione	*	*
ENDDEBUG	Fermo del debug continuazione esecuzione	*	*
HALT	Fine esecuzione di una lista di programmi	*	*
LISTBRKS	Lista dei punti di controllo posizionati con AT e WHEN e dello stato di HALT	*	*
NEXT	Posizionamento di un punto di controllo temporaneo sulla prossima istruzione da eseguire	*	*
OFF	Cancellazione dei punti di controllo posizionati con AT	*	*,PF
OFFWN	Sospende l'indirizzamento stabilito con il comando WHEN	*	*
WHEN	Permette la sospensione dell'esecuzione in seguito ad una condizione data o ad un cambiamento di variabile	*	*
STEP	STEP n equivale a n coppie NEXT-GO	*	*

NOTE:

* indica che il comando e' supportato

PF indica che il comando puo' essere dato col tasto funzionale indicando gli operandi col cursore

Figura 10. Comandi per il controllo dell'esecuzione del programma

Controllo e modifica delle variabili:

I comandi di questa fase offrono la possibilita' di:

- visualizzare il contenuto delle variabili, prima dell'esecuzione di un'istruzione, e di modificarlo a comando o quando viene incontrato un punto di arresto;
- controllare il cambiamento del contenuto di una variabile;
- fornire, durante l'esecuzione, la lista dei sottoprogrammi da eseguire normalmente.

I comandi di questa fase sono:

COMANDO	FUNZIONI	VSF V.1	VSF V.2
IF	Esecuzione di un comando quando una espressione e' vera		*
LIST	Visualizzazione dei valori delle variabili, di elementi di tabelle o tabelle intere		*
AUTOLIST	Lista automatica (10 linee) dei contenuti delle variabili selezionate per essere visualizzate a ogni punto di controllo o sospensione del programma		*

NOTA:

* indica che il comando e' supportato

Figura 11. Comandi per il controllo e modifica delle variabili

Gestione del video

I comandi di questa fase servono per:

- selezionare i colori del pannello del debug;
- aprire o modificare le dimensioni della finestra;
- spostare il cursore fra la zona di comando e la finestra;
- ricercare nella finestra:
  - un numero di istruzioni ISN
  - una stringa di caratteri
- ricercare una riga nello *log* di console.

I comandi di questa fase sono:

COMANDO	FUNZIONI	VSF V.1	VSF V.2
COLOR	Visualizza un pannello di selezione del colore, luminosita' e intensita' per ogni campo del pannello di debug		*
MOVECURS	Muove il cursore fra la linea di comando e la finestra del sorgente		*
SEARCH	Ricerca di un numero di linee dello spool di console o di un ISN o una stringa di caratteri nella finestra sorgente		*
WINDOW	Apre una finestra dal suo angolo inferiore sinistro indicato con il cursore		*

NOTA:

* indica che il comando e' supportato

Figura 12. Comandi per la gestione del video

Informazioni statistiche

Con i comandi di questa fase si possono ottenere le seguenti informazioni:

- frequenze di utilizzo delle istruzioni;
- informazioni su una unita' di programma da porre sotto test;
- traccia delle chiamate ad altri programmi;
- visualizzazione del tempo di CPU totale;
- lista delle unita' di programma con il relativo tempo di CPU usato.

I comandi di questa fase sono:

COMANDO	FUNZIONI	VSF V.1	VSF V.2
LISTFREQ	Stampa la frequenza di utilizzo delle istruzioni di un sottoprogramma	*	*
LISTSUBS	Fornisce informazioni sulle unita' del programma da testare		*
LISTTIME	Fornisce informazioni sul tempo di CPU usato in ogni unita' di programma		*
TIMER	Controlla il consumo del tempo di CPU di una unita' di programma		*
TRACE	Fornisce la traccia dei trasferimenti di controllo nel programma o la traccia del passaggio in una unita' di programma	*	*

NOTA:

* indica che il comando e' supportato

Figura 13. Comandi per le informazioni statistiche

Informazioni sulla natura delle variabili

Questi comandi forniscono informazioni del tipo:

- tipo di una variabile;
- dimensione di una matrice.

I comandi di questa fase sono:

COMANDO	FUNZIONI	VSF V.1	VSF V.2
DESCRIBE	Fornisce il tipo di una variabile e la dimensione di una tabella		*

NOTA:

* indica che il comando e' supportato

Figura 14. Comandi per le informazioni sulla natura delle variabili

Manipolazioni dei files sequenziali

I comandi di questa fase servono per manipolare i file sequenziali e sono:

COMANDO	FUNZIONI	VSF V.1	VSF V.2
BACKSPACE	Riposiziona un file sequenziale all'inizio del record precedente	*	*
CLOSE	Chiude un file sequenziale	*	*
ENDFILE	Scrive una 'tape-mark'	*	*
REWIND	Posiziona un file sequenziale all'inizio del primo record	*	*

NOTA:

* indica che il comando e' supportato

Figura 15. Comandi per la manipolazione dei file sequenziali

Correzione degli errori

I comandi di questa fase servono per intervenire in caso di errore e rendono possibile:

- la correzione degli argomenti invalidi trasmessi ai sottoprogrammi della libreria Fortran;
- il controllo della correzione degli errori effettuata dal Fortran;
- la possibilita' di interagire con il sistema in un ambiente *subset;*
- l'allocazione dinamica dei files;

I comandi in questa fase sono:

COMANDO	FUNZIONI	VSF V.1	VSF V.2
ERROR	Determina se i messaggi sono ricevuti in seguito ad un errore di esecuzione e specifica la condizione di manipolazione degli errori	*	*
FIXUP	Assegna nuovi valori agli argomenti di una funzione di libreria	*	*

NOTA:

* indica che il comando e' supportato

Figura 16. Comandi per la correzione degli errori

Altri comandi:

Altri comandi utili in fase di debug servono per:

- chiedere informazioni online sull'utilizzo del debug;
- visualizzare l'ultimo pannello creato con il programma di utilizzo;
- emettere i comandi sistema attraverso la SYSCMD;
- ottenere la consultazione dei risultati su disco prima della fine dell'esecuzione;
- ottenere la sospensione di stampe a terminale, per emettere i comandi di debug.
- ecc..

Questi comandi sono:

COMANDO	FUNZIONI	VSF	VSF V.1 V.2
* o "	Commento		
HELP	Fornisce informazioni online sul debug	*	*
PURGE	Cancella l'uscita dopo il comando di debug in corso	*	*
PREVDISP	Visualizza l'ultimo pannello creato dal programma d'applicazione		*
QUALIFY	Specifica l'unita' di programma che deve essere interessata dal debug	*	*
QUIT	Ferma debug e da' il controllo al sistema	*	*
REFRESH	Controlla se il refresh del pannello del debug e' corretto		*
RESTART	Rilancia la sessione a partire dal file log		*
SYSCMD	Esegue comandi di sistema nel corso della sessione di debug	*	*
TERMIO	Specifica le routines di I/O da utilizzare	*	*

NOTA:

* indica che il comando e' supportato

Figura 17. Altri comandi

Riferimenti

Per ottenere maggiori informazioni sull'uso del prodotto VS Fortran versione 2 si rimanda alla documentazione IBM:

- VS Fortran Version 2 - Language Reference
- VS Fortran Version 2 - Programming Guide
- VS Fortran Version 2 - Interactive Debug

Esempi di utilizzo dei comandi

Controllo dell'esecuzione del programma

```
AT          at/65 count(10)
            at(180 220 10:/50 /100)
            at 140 (list a % set i=10 % go)

GO          go
            go 140

ENDDEBUG    enddebug

HALT        halt entry
            halt stmt
            at 10(if(a.gt.b) halt % go) nonotify

NEXT        AT : 10 IN MAIN
N           next
            go
            NEXT : 11 IN MAIN
            next
            go 40
            NEXT : 40 IN MAIN

NEXT        ERROR EXIT, ERROR 209 AT H IN MAIN
            next
            fixup
            STANDARD CORRECTIVE ACTION TAKEN
            NEXT : 12 IN MAIN

LISTBRKS    listbrks

OFF         off
            off (20:80 100/55)

OFFWN       offwn
            offwn

WHEN        when cond1 (sub25.t1.1t.sub2.t3)
            WHEN : COND1 IN 45/20 IN SUB5
            CURRENTLY AT : 46 IN MAIN
            when cond2 p

STEP        step 3 (equivale a : next
                                 go
                                 next
                                 go
                                 next
                                 go   )
```

Controllo e modifica delle variabili

```
IF          if(a.gt.b) halt
            if (main.a.lt.sub1.b) set sub1.b=main.a
            at/100 (if(over) set i=0 % go)
            at 10 (if(sub.a=0.0)halt % go)
            at 5 (if(a) go/10 % if(b) go/10 % go)

LIST        list ncount
            list a(1,1):a(7,10) print format
            list sub1.ctr
            list (i,j,p,q,r) dump

AUTOLIST    al(sub1.reall,sub2.tablo(i,j), chaine,i,j)
AL
```

Comandi full-screen esteso (in V.2)

```
COLOR       color

MOVECURS    mc

MC          (da destinare generalmente alla PF12)

SEARCH      search 9
            search /toto/
            search /toto
            search 'toto'

WINDOW      window  (da destinare ad un tasto funzionale)
```

Informazioni sulla natura delle variabili (in V.2)

```
DESCRIBE    de(i,k,dumchr,r8ary,r4dumy,l1aymn)
DE          I:          INTEGER*4
            K:          INTEGER*4    DUMMY
            DUMCHR:     CHARACTER*(*)DUMMY
            R8ARY:      REAL*8
              RANK=2,   SIZE=14 ELEMENTS
              DIM 1:    EXENT=7, LBOUND=1, UBOUBD=7
              DIM 2:    EXENT=7, LBOUND=1, UBOUND=7
            R4DUMY:     REAL*4       DUMMY
              RANK=3;   DUMMY ARRAY ARGUMENT OF INACTIVE SUBPROGR
                        OR ALTERNATE ENTRY POINT;
                        DIMENSION INFORMATION NOT AVAILABLE
            L1AYMN:     LOGICAL*1    DUMMY
              RANK=2;   SIZE=* ELEMENTS
              DIM 1:    EXTENT=5, LBOUND=-2, UBOUND=2
              DIM 2:    EXTENT=*, LBOUND=1, UBOUND=*
```

Informazioni di traccia e di tempi

```
LISTFREQ   listfreq 100
LF         listfreq 45:52
           listfreq
           STATEMENT IN MAIN        FREQUENCY
           9                          1
           10/40                      1
           11/50                      150
           13                         150
           14                         49
           ...                        ...

TRACE      t entry
T          TRACE: SUB1 ENTERED
           t goto
           TRACE: FROM 150/20 TO 210/40
           t goto print
           t off

LISTSUBS   ls print
LS         PROGRAM-UNIT   COMPILER    OPT   TIMING
               MAIN       VS 2.1.0     2      ON
               SUBA       VS 1.4.0     3      OFF
               SUBB       VS (TEST)    0      OFF
               SUBC       VS 1.3.1     1      ON

TIMER      timer reset
           timer sub1
           timer *
           timer (sub2,sub5) off

LISTTIME   lt print
LT         PROGRAM-UNIT   ENTRY-PT    EXEC-TIME   INVOCATIONS
                                        (MIC)
               MAIN        MAIN        43257          1
               SUBA        SUBA1        8515          5
                           SUBA2       63214         30
               SUBB        SUBB         1350          2
```

Correzione degli errori

```
ERROR      er 251 msg noexit
ER         er 241:285 noexit

FIXUP      f
F          f arg1(25)
```

Manipolazione di files sequenziali

```
BACKSPACE  backs 8
BACKSPA    backs n
BACKS

CLOSE      close 8
           sys filedef 8 disk output file a
           close 8
           sys allocate file (ft08f001) dataset(output,file)

ENDFILE    endf 8
ENDF       endf n

REWIND     rew 8
REW        rew n
```

Altri comandi

```
HELP       h if
H          h error (parm

PURGE      at 100(list a % set i=0 % list b % where % go)
              Ma a contiene 1000 elementi, allora si fa
              una istruzione e si purga. Si preme ENTER
              (o ATTN), poi:
           purge

QUALIFY    q subm
Q          q

QUIT       quit

SYSCMD     sys fi 8 disk toto listing a
SYS
CMS TSO

TERMIO     termio iad
           termio library

PREVDISP   prev
PREV

REFRESH    refresh on
           refresh off

RESTART    restart
```

Le fasi della vettorizzazione

Scrittura dell'applicazione

Come e' risultato ormai evidente dalla descrizione delle architetture vettoriali, il guadagno nelle prestazioni si ottiene con l'uso della pipeline. Cio' richiede quindi la presenza di cicli DO all'interno dell'applicazione nei quali vengano eseguite delle operazioni su vettori. Puo' sembrare ovvio, ma comunque e' bene sottolineare il fatto che nella programmazione vettoriale tutta la nostra attenzione deve essere rivolta ai cicli DO.

La scrittura di una applicazione vettoriale non differisce globalmente dalla scrittura di una applicazione tradizionale, tuttavia e' necessario, dove e' possibile, cercare di usare dei vettori al posto di variabili scalari, rendere la dimensione dei vettori piu' grande possibile e ridurre lo *stride*.

La metodologia piu' frequentemente usata da un *vecchio* programmatore e' quella di costruire l'algoritmo per un evento e poi ripeterlo tante volte quanti sono gli eventi da calcolare. Per scrivere invece un'applicazione vettoriale e' indispensabile prevedere un algoritmo che elabori N eventi contemporaneamente; cosi', per esempio, se si vuole calcolare la traiettoria di 1000 particelle, non bisogna ripetere 1000 volte il ciclo di istruzioni che calcola una sola traiettoria, che costringerebbe a lavorare con variabili scalari, ma scrivere le istruzioni che effettuano il calcolo contemporaneo di 100 traiettorie, che permette di lavorare con vettori, e poi ripetere il tutto 10 volte.

Vettorizzazione automatica

La compilazione di una applicazione in vettorizzazione automatica fornisce una lista dell'applicazione dove, per ogni 'DO' loop, esistono delle indicazioni sulla vettorizzazione realizzata. Queste indicazioni, che devono essere analizzate dal programmatore, sono:

1. Indicazione VECT

   Il ciclo e' stato vettorizzato; questo non significa necessariamente che la performance sara' superiore a quella scalare. Si puo' dare il seguente esempio:

   ```
         DO 10 I=1,N
         X(I)=B(I)+A*Y(I)
      10 CONTINUE
   ```

   Il compilatore, essendo N sconosciuto, assume per default N=20 e vettorizza il codice: se poi in esecuzione si avesse invece N=3, il vettore risulterebbe troppo corto ed il codice vettoriale si rivelerebbe meno efficiente di un codice non vettoriale.

2. Indicazione ELIG

Il ciclo 'DO' e' a priori vettorizzabile, ma il compilatore prevede che sara' eseguito piu' rapidamente in scalare, per cui ignora l'opzione VECTOR e compila questo ciclo in modo scalare.
Anche qui la scelta del compilatore puo' risultare errata, come e' mostrato nell'esempio seguente:

```
      DO 10 I=1,N
      X(I)=Y(N+1-I)
10    CONTINUE
```

Essendo N sconosciuto, il compilatore lo suppone uguale a 20 e, visti i tipi di operazioni da effettuare, giudica il ciclo piu' veloce in scalare, per cui non viene vettorizzato. Se pero' l'applicazione usa valori di N molto grandi l'esecuzione sarebbe stata piu' veloce se svolta in vettoriale.

Molti a questo punto si saranno chiesti perche' nel primo esempio, usato con l'indicatore VECT, il compilatore, pur prevedendo ugualmente un vettore di 20 elementi, decida di vettorizzare, quando nell'attuale esempio, rifiuti la vettorizzazione. Cio' si spiega con il fatto che la lunghezza minima del vettore, al di sotto della quale non conviene vettorizzare, e' in funzione del tipo di operazioni da elaborare. Infatti a parita' di tempo di inizializzazione, il numero di passi del ciclo e' piu' breve quando le operazioni in gioco sono piu' efficienti in rapporto allo scalare. Poiche' il guadagno ottenuto con operazioni aritmetiche e' molto superiore a quello ottenuto con i trasferimenti all'interno della memoria, nel primo esempio, dove e' presente una somma, e' sufficiente un vettore lungo 20 per indurre il compilatore alla vettorizzazione.

3. Indicazione RECR

   Questa indicazione segnala che il ciclo non e' stato vettorizzato. Le ragioni del rifiuto sono indicate dalla lista ed alcune di queste ragioni corrispondono a veri e propri inibitori come si puo' notare nel seguente esempio:

```
      DO 10 I=1,N
       X(I)=X(I-1)+A*X(I)
10    CONTINUE
```

   dove compare una ricorrenza a sinistra.

   In altri cicli puo' verificarsi un'interpretazione conflittuale del compilatore che viene annotata dalla RECR.

4. Idicazione UNAN

   Questa indicazione e' data quando il compilatore riconosce un inibitore nel ciclo.

   Una volta vettorizzata direttamente dal compilatore, senza modifica del codice ne' aggiunta di direttive, l'applicazione puo' essere eseguita e le sue prestazioni confrontate con quelle della sua versione scalare. In alcuni casi, il miglioramento cosi' ottenuto senza il minimo sforzo, e' tale che non si rende necessario spingere oltre la vettorizzazione.

Utilizzo delle direttive

Dopo le analisi delle indicazioni fornite dal compilatore, a volte il lavoro per la vettorizzazione del codice puo' continuare introducendo delle *direttive* per rimuovere alcune decisioni intraprese dal compilatore vettoriale e forzarne altre.

Con le direttive si puo' quindi favorire il ciclo

```
      DO 10 I=1,N
      X(I)=Y(N+1-I)
   10 CONTINUE
```

informando il compilatore che N e' superiore a 2000.

E' anche possibile inibire la vettorizzazione del ciclo

```
      DO 10 I=1,N
       X(I)=B(I)+A*Y(I)
   10 CONTINUE
```

segnalando che N e' uguale a 3.

C'e' poi il caso in cui non e' possibile giudicare la convenienza di avere un ciclo vettorizzato o meno, poiche' la dimensione del vettore, o meglio il numero degli elementi da analizzare, varia da una esecuzione all'altra. In questo caso, se abbiamo il ciclo:

```
      DO 10 I=1,N
      X(I)=B(I)+A*Y(I)
   10 CONTINUE
```

e' utile modificare il codice che tiene conto che N possa essere piccolo o grande:

```
      IF(N.LE.15)THEN
C     questo ciclo deve essere scalare
       DO 10 I=1,N
       X(I)=B(I)+A*Y(I)
  10   CONTINUE
      ELSE
C     questo ciclo deve essere vettorizzato
       DO 20 I=1,N
       X(I)=B(I)+A*Y(I)
  20   CONTINUE
      ENDIF
```

Cosi' l'applicazione, a seconda che N sia superiore o inferiore a 15, utilizzera' l'insieme di istruzioni che gli procurano le massime prestazioni. Questo tipo di scrittura si chiama DUAL PASS. Un altro tipo di direttiva consiste nell'imporre il modo 'vettore' al compilatore quando questo esita fra la compilazione scalare o vettoriale. Di contro, la presenza di un inibitore di tipo ricorrente, fa annullare l'effetto della direttiva di vettorizzazione e il ciclo e' compilato in scalare.

Modifica del codice

All'inizio e' stata brevemente descritta la metodologia da usare per la programmazione vettoriale, ma capita spesso di disporre di un software gia' esistente ed allora e' necessario apportare le opportune variazioni per renderlo vettorizzabile. Queste modifiche del codice possono essere semplici o complesse, possono riguardare l'aggiustamento di poche istruzioni o la revisione di tutto l'algoritmo del programma, ma in ogni caso e' difficile schematizzare i vari tipi di intervento, per cui ci limiteremo a considerare alcuni esempi.

Nell'esempio seguente:

```
      DO 10 I=2,N
      X(I)=B(I)+A*Y(I)
      Y(I)=X(I-1)+C*Z(I)
   10 CONTINUE
```

il compilatore non e' in grado di vettorizzare in quanto esiste una ricorrenza, determinata dal fatto che il calcolo di ogni elemento del vettore Y necessita degli elementi di X calcolati al passo precedente. Se questo ciclo viene riscritto come:

```
      DO 10 I=2,N
      X(I)=B(I)+A*Y(I)
10    CONTINUE
      DO 20 I=2,N
      Y(I)=X(I-1)+C*Z(I)
20    CONTINUE
```

viene a cadere ogni indeterminazione e i due cicli sono allora vettorizzati senza dover utilizzare alcuna direttiva.

Certe modifiche di codice possono essere piu' sottili e avere l'effetto di:

1. favorire una compilazione vettoriale che utilizza le operazioni composte.

   Analizziamo il seguente esempio:

   ```
         DO 10 I=1,N
         DO 20 J=1,N
         C(I)=C(I)+B(J)*A(I,J)
   20    CONTINUE
   10    CONTINUE
   ```

   Si raddoppieranno facilmente le prestazioni scrivendo:

   ```
         DO 10 I=1,N
         CTE=C(I)
         DO 20 J=1,N
         CTE=CTE+B(J)*A(I,J)
   20    CONTINUE
         C(I)=CTE
   10    CONTINUE
   ```

   Il compilatore vettorizzatore trattera' queste sequenze come segue, supponendo z la lunghezza dei registri vettoriali:

   ```
         DO 10 I=1,N,z
         CTE(k)=C(I+k-1)                   per k=1,......,z
         DO 20 J=1,N
         CTE(k)=CTE(k)+B(J)*A(I+k-1,J)     per k=1,......,z
   20    CONTINUE
         C(I+k-1)=CTE(k)                   per k=1,......,z
   10    CONTINUE
   ```

   Nel ciclo DO 20 l'istruzione vettoriale e' di tipo: VETTORE=VETTORE+SCALARE*VETTORE; il compilatore utilizza allora l'operazione composta (Mult,Add) e si avranno allora 2 risultati per ciclo.

2. minimizzare lo 'STRIDE'

   Per esempio il codice:

   ```
         DO 10 I=1,N
         DO 20 J=1,M
         X(I,J)=A+B*Y(I,J)
   20    CONTINUE
   10    CONTINUE
   ```

presenta l'incoveniente che l'operazione viene effettuata, nel ciclo piu' interno, per tutti gli elementi della stessa riga, cioe' facendo scorrere l'indice da una colonna all'altra. Poiche' in memoria le matrici Fortran sono tenute *per colonne,* l'effetto sara' quello di elaborare tutti gli elementi che distano tra loro la lunghezza della colonna. Per tale motivo sara' vantaggioso riscrivere le istruzioni sotto la forma:

```
      DO 10 J=1,M
      DO 20 I=1,N
      X(I,J)=A+B*Y(I,J)
20    CONTINUE
10    CONTINUE
```

Se non fosse possibile operare l'inversione degli indici e' un buon espediente quello di ridurre le matrici grosse in sottomatrici opportunamente dimensionate e questo per ridurre lo stride. Comunque, in generale, una matrice dimensionata X(10,5000) ha buone probabilita' di essere vettorizzata in maniera peggiore che non se fosse rappresentata come X(5000,10).

3. Rivedere l'organizzazione dei dati

   Questa operazione puo' essere utile sia per ridurre i tempi di accesso alla memoria e sia per favorire la vettorizzazione del codice. In generale i principi a cui e' necessario attenersi per raggiungere questo scopo sono:

   - evitare di mantenere i dati su files esterni;
   - strutturare il programma in modo da rendere possibile il riutilizzo dei dati stessi.

   Spesso pero' tali obiettivi costringono il programmatore a rivedere la logica e la struttura dell'applicazione.

4. Utilizzo delle librerie dei programmi vettorizzati

   Per una buona programmazione vettoriale e' vantaggioso l'utilizzo delle librerie, soprattutto quando queste sono state concepite specificatamente per i calcolatori vettoriali. Una di queste librerie fornita dalla IBM e' la ESSL ('Engineering & Scientific Subroutine Library'), che e' composta da un centinaio di sottoprogrammi e funzioni, in costante aggiornamento, riguardanti i seguenti campi:

   - Algebra lineare (operazioni fra scalari, vettori, matrici)
   - Equazioni algebriche lineari (risoluzioni di sistemi lineari)
   - Autovalori e autovettori
   - Trattamento del segnale
   - Generazione di numeri casuali.

   Questi sotto-programmi o funzioni possono essere usati dai programmi Fortran o Assembler sia per operazioni in semplice che in doppia precisione.

   La caratteristica principale di questa libreria e' di essere costituita di programmi scritti essenzialmente per il 3090-VF, che sfruttano quindi tutte le possibilita' dell'architettura di questa macchina, con l'uso di algoritmi tra i piu' efficienti ed esistenti attualmente. Tuttavia avendo la chiamata a un sottoprogramma delle ripercussioni sulle prestazioni, e' consigliabile utilizzare la ESSL solo su vettori relativamente lunghi.

   L'utilizzo di questa libreria non pone di solito grossi problemi, se non quello dovuto alla metodologia di scrittura dell'applicazione. Infatti spesso l'applicazione e' stata concepita come scalare e risulta difficile creare l'insieme dei vettori che si vogliono elaborare con una chiamata alla libreria.

Gli ostacoli alla vettorizzazione: gli inibitori

Oltre ai programmi che non contengono grossi cicli, alcuni tipi di codice possono inibire la vettorizzazione. I compilatori attuali danno prova di poca intelligenza o prendono, talvolta, delle decisioni arbitrarie, per cui e' necessario dunque ovviare a questa carenza con il nostro lavoro.

La lista seguente indica i casi che inibiscono la vettorizzazione

1. Le istruzioni IF

   Poiche' alcune istruzioni IF contenute all'interno di un ciclo interrompono la sequenzialita' delle istruzioni, l'intero ciclo non puo' essere vettorizzato.

2. Le ricorrenze

   La ricorrenza esiste quando in un ciclo DO c'e' un'istruzione che usa dei valori che devono essere calcolati in iterazioni precedenti.

3. Le dipendenze

   Si ha una dipendenza quando esiste una relazione tra le istruzioni del ciclo per cui l'ordine di esecuzione delle istruzioni diventa importante ai fini del risultato. Le dipendenze piu' comuni sono dovute o ad una ricorrenza, o ad un indirizzamento indiretto, oppure alla presenza di una istruzione di EQUIVALENCE.

4. Le interruzioni

   Un' istruzione di GOTO al di fuori del ciclo interrompe la sequenzialita' del flusso per cui viene inibita la vettorizzazione.

5. Le funzioni non vettorizzabili

   Il riferimento ad una funzione non vettorizzata inibisce la vettorizzazione poiche' causa il trasferimento del controllo.

6. Le chiamate a procedure

   La chiamata ad un sottoprogramma non vettorizzato inibisce la vettorizzazione poiche' causa il trasferimento del controllo.

7. Le operazioni di I/O

   In presenza di un'operazione di I/O il compilatore genera una chiamata a routines del sistema e quindi inibisce la vettorizzazione.

8. Le dichiarazioni

   Alcune dichiarazioni come INTEGER*2 inibiscono la vettorizzazione.

Gli inibitori "forti"

Gli inibitori appartenenti a questa categoria hanno l'effetto di inibire quasi sempre la vettorizzazione.
Questi inibitori possono essere cosi' riassunti:

1. Uso della precisione a 128 bits.
2. Uso delle dichiarative LOGICAL*1 oppure LOGICAL*2.
3. Uso della dichiarativa INTEGER*2.
4. Presenza di una ricorrenza del tipo:

```
              .....
              DO 10 I=2,N
                 .....
              X(I)=X(I-1)+A*Y(I)
                 .....
      10      CONTINUE
```

5. Uso del GO TO assegnato.

```
              .....
             ASSIGN 10 GO TO I
              .....
             GO TO I
      20     CONTINUE
              .....
      10     CONTINUE
```

6. Istruzione IF con salto fuori del ciclo.

```
              .....
             DO 10 I=1,N
              .....
             IF (C(I).EQ.0)) GO TO 20
              .....
             X(I)+C(I)+A*Y(I)
              .....
      10     CONTINUE
              .....
      20     CONTINUE
```

7. Istruzione IF con salto ad una istruzione precedente.

```
              .....
             DO 10 I=1,N
              .....
      20     CONTINUE
              .....
             IF (C(I).EQ.0)) GO TO 20
              .....
             X(I)=C(I)+A*Y(I)
              .....
      10     CONTINUE
              .....
```

L'eliminazione di questi inibitori puo', in alcuni casi, risultare impossibile, come l'eliminazione della precisione a 128 bits per un'applicazione che invece la richiede. Puo' invece spesso essere necessaria una revisione importante della logica del programma. Purtroppo non esiste una *ricetta miracolosa* per eliminare del tutto questi problemi, tuttavia e' facile regolarne alcuni.
Per esempio il ciclo:

```
      DO 10 I=1,N
       .....
      X(I)=X(I)+A*Y(I)
      IF (LOGIC) GO TO 10
      X(I)=X(I)+B*Z(I)
10    CONTINUE
```

puo' essere trasformato in:

```
      DO 10 I=1,N
       .....
      X(I)=X(I)+A*Y(I)
10    CONTINUE
      IF (LOGIC) GO TO 30
      DO 20 I=1,N
      X(I)=X(I)+B*Z(I)
20    CONTINUE
30    CONTINUE
```

Nel seguente esempio in cui la condizione dipende dall'indice del ciclo:

```
       .....
      DO 10 I=1,N
       .....
      X(I)=X(I)+A*Y(I)
      IF (LOGIC(I)) GO TO 10
      X(I)=X(I)+B*Z(I)
10    CONTINUE
```

il codice puo' essere modificato come segue:

```
       .....
      DO 10 I=1,N
       .....
      TEST(I)=1
      IF (LOGIC(I)) TEST(I)=0
10    CONTINUE
      DO 20 I=1,N
       .....
      IF (TEST(I).EQ.0)) THEN
        X(I)=X(I)+A*Y(I)
      ELSE
        X(I)=X(I)+A*Y(I)+B*Z(I)
      ENDIF
20    CONTINUE
```

In numerosi casi, la presenza di inibitori forti e degli inibitori leggeri e' un indice di una scrittura discutibile del programma. La modifica di un programma in vista della sua vettorizzazione risiede, in massima parte, nella eliminazione del maggior numero di inibitori.

Gli inibitori "deboli"

Questi piu' che inibitori, sono dei rallentatori dell'esecuzione.

La dimensione dei cicli DO

In generale, per il calcolo vettoriale, si puo' affermare che piu' il vettore e' grande e migliori saranno le prestazioni. Bisognera' dunque provare ad aumentare i cicli e per far cio' la tecnica classica e' la riduzione del numero di indici.
Per esmpio:

```
      SUBROUTINE MULTI(A,NX,NY,NZ,S)
      DIMENSION A(NX,NY,NZ)
      DO 10 IZ=1,NZ
      DO 20 IY=1,NY
      DO 30 IX=1,NX
      A(IX,IY,IZ)=S*A(IX,IY,IZ)
30    CONTINUE
20    CONTINUE
10    CONTINUE
      RETURN
      END
```

La sequenza precedente puo' essere modificata come segue:

```
      SUBROUTINE MULTI(A,NX,NY,NZ,S)
      DIMENSION A(NX*NY*NZ)
      DO 10 I=1,NZ*NY*NX
      A(I)=A(I)*S
10    CONTINUE
      RETURN
      END
```

Questo genere di modifica e' realistica solo se le dimensioni dichiarate della matrice A sono identiche alle dimensioni utilizzate.

L'ordine dei cicli DO

Le matrici sono ordinate in memoria seguendo inizialmente il primo indice, poi il secondo e cosi' via, per tale motivo l'ordine dei cicli avra' un impatto importante sull'effetto dello stride.
Nel seguente esempio:

```
          DO 10 I=1,N
          DO 20 J=1,N
          A(I,J)=A(I,J)+X(J)*Y(I)-X(I)*Y(J)
 20       CONTINUE
 10       CONTINUE
```

nel ciclo DO 20 lo stride del vettore sara' pari ad N, per cui si ha dunque tutto l'interesse ad invertire gli indici dei due cicli:

```
          DO 10 J=1,N
          DO 20 I=1,N
          A(I,J)=A(I,J)*X(J)*Y(I)-X(I)*Y(J)
 20       CONTINUE
 10       CONTINUE
```

Questo caso e' talmente evidente che il compilatore stesso operera' direttamente questa inversione.

Se invece i limiti del ciclo dipendono da uno degli indici sara' allora necessario modificare il codice:
Ad esempio:

```
          DO 10 I=1,N
          DO 20 J=1,I
          A(I,J)=A(I,J)+X(J)*Y(I)-X(I)*Y(J)
 20       CONTINUE
 10       CONTINUE
```

questo codice puo' essere cosi' modificato:

```
          DO 10 J=1,N
          DO 20 I=J,N
          A(I,J)=A(I,J)+X(J)*Y(I)-X(I)*Y(J)
 20       CONTINUE
 10       CONTINUE
```

Questo esempio merita alcune spiegazioni:

- per la manipolazione degli indici e' possibile avvalersi della regola: *"La manipolazione dei cicli DO sovrapposti e' identica a quella dei limiti di integrali multipli di cui si desidera invertire l'ordine" (Jean Claude Patau).* Tale manipolazione e' valida quindi se si descrive, in modo diverso, la stessa superficie della matrice da calcolare.
- nell'invertire l'ordine dei cicli DO, bisogna tener conto che si modifica l'ordine delle operazioni, per cui gli errori dovuti alla precisione saranno diversi.

Per quanto detto e' quindi necessario fare le seguenti considerazioni:

- Bisogna assicurarsi in tutti i casi che valga la pena trasformare l'ordine dei cicli.

- Queste trasformazioni sono generalmente sempre possibili ma non necessariamente semplici.
- Bisogna fare attenzione ai casi in cui la trasformazione e' falsa.

  Per esempio il codice:

```
      K=0
      DO 10 I=1,N
      DO 20 J=1,N
      K=K+1
      B(K)=A(I,J)
20    CONTINUE
10    CONTINUE
```

  non e' equivalente a:

```
      K=0
      DO 10 J=1,N
      DO 20 I=1,N
      K=K+1
      B(K)=A(I,J)
20    CONTINUE
10    CONTINUE
```

Le istruzioni IF nei cicli DO

Il compilatore vettorizza i cicli che comprendono le istruzioni IF non inibitrici.

La seguente sequenza:

```
      DO 10 I=1,N
      Y(I)=Y(I)+A*X(I)
      IF(Y(I).GT.0) Y(I)=Y(I)+B*X(I)
10    CONTINUE
```

e' trattata dal compilatore come segue:

```
1.      Y(I)=Y(I)+A*X(I)
2.      Y(I) > 0 ---> maschera
3.      TEMP(I)=Y(I)+B*X(I)
4.      Y(I)=TEMP(I) o Y(I) secondo la maschera
```

Un ciclo DO con un'istruzione IF puo' essere piu' lento in modo vettoriale che in scalare e questo se il numero dei passi e' elevato. La soluzione consiste nello spezzare i cicli per eliminare gli IF o, almeno, limitarne gli effetti.

Prendiamo in esame qualche caso:

1. Uscita di un IF inibitore

```
       .....
       S=....
       XMIN=....
       .....
       DO 10 I=1,N
       X(I)=X(I)+S*Z(I)
       IF (X(J) .LT. XMIN) GO TO 60
       Y(I)=Y(I)+S*X(I)
 10    CONTINUE
        ...
        ...
 60    CONTINUE
C     Trattamento in caso di errore (molto improbabile)
```

Essendo l'errore molto improbabile, si correra' il rischio di perdere piu' tempo nei pochi casi in cui la condizione X(J).LT.XMIN e' vera per guadagnarne pero' nei casi piu' frequenti.

```
       .....
       S=....
       XMIN=....
       ....
       DO 10 I=1,N
       X(I)=X(I)+S*Z(I)
 10    CONTINUE
 C     calcolo di un numero per il ciclo di Y
       IBR=0
       DO 20 I=1,N
       IF (X(I) .GE. XMIN) GO TO 20
       IBR=I
       GO TO 30
 20    CONTINUE

       JBR=N
       GO TO 40
 30    CONTINUE
       IF (IBR .EQ. 1) GO TO 60
       JBR=IBR-1
 40    CONTINUE
       DO 50 I=1,JBR
         Y(I)=Y(I)+S*X(I)
 50    CONTINUE
       IF (JBR .LT. N) GO TO 60
        .....
        .....
 60    CONTINUE
 C     Trattamento in caso di errore  (molto improbabile)
        .....
```

2. Caso semplice dell'uscita di un IF rallentatore.

```
      .....
      DO 10 I=1,N
      X(I)=X(I)+S*Z(I)
      IF (ABS(X(I)) .LT.1.) X(I)=1./X(I)
      Y(I)=Y(I)+S*X(I)
10    CONTINUE
      .....
```

Questo codice diventa:

```
      .....
      DO 10 I=1,N
      X(I)=X(I)+S*Z(I)
10    CONTINUE
C     il ciclo che sara' vettorizzato e' inutile
      DO 20 I=1,N
      IF (ABS(X(I)) .LT 1.) X(I)=1./X(I)
20    CONTINUE
      DO 30 I=1,N
      Y(I)=Y(I)+S*X(I)
30    CONTINUE
```

3. Uscita di un IF rallentatore con l'utilizzo dell'indirizzamento indiretto.

```
      .....
      DO 10 I=1,N
      IF (X(I)) .GE 0.) THEN
         Y(I)=Y(I)+X(I)/10.
      ELSE
         Y(I)=Y(I)+X(I)/20.
      ENDIF
10    CONTINUE
       .....
```

Se X(I) presenta frequenti cambi di segno, benche' l'indirizzamento sia poco apprezzato dal vettorizzatore, si puo' scrivere:

```
      KN=0
      KP=0
      DO 10 I=1,N
      IF (X(I)) .GE. 0.) THEN
         KP=KP+1
         JP(KP)=I
      ELSE
         KN=KN+1
         JN(KN)=I
      ENDIF
10    CONTINUE
      IF (KP .NE. 0) THEN
         DO 20 K=1,KP
         Y(JP(K))=Y(JP(K))+X(JP(K))/10.
20       CONTINUE
      ENDIF
      IF (KN .NE. 0) THEN
         DO 30 K=1,KN
         Y(JN(K))=Y(JN(K))+X(JN(K))/20.
30       CONTINUE
      ENDIF
```

4. Uscita di un IF rallentatore con l'utilizzo di tabelle

   In questo caso, consideriamo che gli elementi X(I) mantengano lo stesso segno per un numero elevato di iterazioni.

```
      JKN=1
      KN=0
      JKP=1
      KP=0
      DO 10 J=1,N
      IF (X(J)) .GE 0.) THEN
         JKN=1
         IF (JKP .LE. 1) THEN
            KP=KP+1
            JKP=JKP+1
            JP(1,KP)=J
            JP(2,KP)=J
         ELSE
            JP(2,KP)=J
         ENDIF
      ELSE
         JKP=1
         IF (JKP .LE. 1) THEN
            KN=KN+1
            JKN=JKN+1
            JN(1,KN)=J
            JN(2,KN)=J
         ELSE
            JN(2,KN)=J
         ENDIF
      ENDIF
10    CONTINUE
```

```
         IF (KP .NE. 0) THEN
            DO 20 K=1,KP
            IF ((JP(2,K)-JP(1,K)) .GE. LIM) THEN
               DO 20 J=JP(1,K),JP(2,K)
               Y(J)=Y(J)+X(J)/10.
20             CONTINUE
            ELSE
DIRECTIVE NOVEC
               DO 30 J=JP(1,K),JP(2,K)
               Y(J)=Y(J)+X(J)/10.
30             CONTINUE
            ENDIF
         ENDIF


         IF (KN .NE. 0) THEN
            DO 20 K=1,KN
            IF ((JN(2,K)-JN(1,K)) .GE. LIM) THEN
               DO 20 J=JN(1,K),JN(2,K)
               Y(J)=Y(J)+X(J)/20.
20             CONTINUE
            ELSE
DIRECTIVE NOVEC
               DO 30 J=JN(1,K),JN(2,K)
               Y(J)=Y(J)+X(J)/20.
30             CONTINUE
            ENDIF
         ENDIF
```

5. Esempio classico di uscita di un IF rallentatore

```
          .....
         DO 10 I=1,N
         IF (X(J) .GE. 0.) THEN
           Y(I)=Y(I)+SCAL1*Z(I)
         ELSE
           Y(I)=Y(I)+SCAL2*Z(I)
         ENDIF
10       CONTINUE
```

Il codice puo' essere riscritto:

```
         DO 10 I=1,N
         IF (X(J) .GE. 0.) THEN
            STAB(I)=SCAL1
         ELSE
            STAB(I)=SCAL2
         ENDIF
10       CONTINUE
         DO 20 I+1,N
         Y(I)=Y(I)+STAB(I)*Z(I)
 20      CONTINUE
```

A priori, non si e' mai certi che l'eliminazione degli inibitori portera' un guadagno sensibile nelle prestazioni, nonostante gli sforzi considerevoli spesi da parte del programmatore. Per cui e' buona norma che la ristrutturazione sia effettuata solo come ultimo tentativo e unicamente su cicli *pesanti* come consumo di tempo di CPU.

I trasferimenti

Come gia' detto, poiche' i tempi di accesso ai dati della memoria sono divenuti troppo elevati rispetto ai tempi di esecuzione delle operazioni aritmetiche, i trasferimenti, se sono troppo numerosi, possono costituire un collo di bottiglia per l'esecuzione vettoriale.

Questi trasferimenti avvengono nei due sensi tra:

- unita' di paginazione e memoria
- memoria e memoria
- memoria e *high speed buffer*
- *high speed buffer* e registri

Una tecnica per diminuire il numero di questi trasferimenti e' conosciuto come *unrolling* che andiamo ora a descrivere.

Consideriamo la seguente sequenza:

```
         DO 10 J=1,N2
         DO 20 I=1,N1
         Y(I)=Y(I)+X(J)*Z(I,J)
20       CONTINUE
10       CONTINUE
```

Si puo' rappresentare lo svolgimento (unroll) delle operazioni come segue:

```
      DO 10 J=1,N2
        Caricamento di X(J)              1 volta     'Trasferimento'
        Caricamento degli Y(I)          N1 volte     'Trasferimento'
        Caricamento degli Z(I,J)        N1 volte     'Trasferimento'
        Calcolo Z(I,J)*X(J)             N1 volte     'Operazione'
        Calcolo Y(I)+risultato
                       precedente       N1 volte     'Operazione'
        Memorizzazione degli Y(I)       N1 volte     'Trasferimento'
10    CONTINUE
```

Calcoliamo adesso il rapporto tra il numero delle operazioni in virgola mobile ed il numero dei trasferimenti:

- Numero di operazioni in virgola mobile: 2 * N1 * N2
- Numero di trasferimenti: (1 + 3 * N1) * N2
- Come si puo' notare il rapporto si avvicina a 2/3.

Se invece riscriviamo il codice come:

```
      DO 10 J=4,N2,4
      DO 20 I=1,N1
      Y(I)=(((Y(I)+X(J-3)*Z(I,J-3))+X(J-2)*Z(I,J-2))
     *                 +X(J-1)*Z(I,J-1))+X(J)*Z(I,J))
20    CONTINUE
10    CONTINUE
```

Si puo' rappresentare lo sviluppo delle operazioni come segue:

```
      DO 10 J=4,N2,4
        Caricamento di X(J-3),X(J-2),
                       X(J-1),X(J)       4 volte    'Trasferimento'
        Caricamento degli Y(I)          N1 volte    'Trasferimento'
        Caricamento degli Z(I,J-3),
           Z(I,J-2),Z(I,J-1),Z(I,J)   4*N1 volte    'Trasferimento'
        Calcolo  Z(I,J-3)*X(J-3)
                 Z(I,J-2)*X(J-2)
                 Z(I,J-1)*X(J-1)
                 Z(I,J)*X(J)          4*N1 volte    'Operazione'
        Calcolo Y(I)+risultato
                     precedente       4*N1 volte    'Operazione'
        Memorizzazione degli Y(I)       N1 volte    'Trasferimento'
10    CONTINUE
```

Ricalcoliamo adesso il precedente rapporto:

- Numero di operazioni in virgola mobile: 8 * N1 * (N2/4)
- Numero di trasferimenti: (4 + 6 * N1) * (N2/4)
- Questa volta il rapporto e' vicino a 4/3.

Con tale tecnica si e' quindi raddoppiato il numero di operazioni in virgola mobile a parita' di numero di trasferimenti effettuati. L'incremento del ciclo e' detto profondita' di *unrolling*.

La vettorizzazione del VS Fortran

L'analizzatore di esecuzione VS Fortran

Generalita'

Come e' stato precedentemente detto il problema piu' importante per rendere un'applicazione vettorizzabile risiede nell'analisi dei cicli DO. E' solo infatti l'esecuzione di tali istruzioni che innesca il meccanismo della pipeline. Si presenta quindi la necessita' di operare le opportune modifiche sul codice gia' esistente, ma spesso anche sulla prima stesura di un'applicazione scritta per una macchina vettoriale, per il raggiungimento di tali obiettivi.

Ma, per il fatto che la ristrutturazione e' quasi sempre difficoltosa, e' bene indirizzare gli sforzi solo su quelle zone del programma dove viene spesa la maggior parte del tempo di esecuzione.

Scopo

Una funzione di analisi dell'esecuzione e' svolta dall'analizzatore VS Fortran, che permette di identificare le parti del codice che contengono un calcolo intensivo e quindi attira l'attenzione dell'utente sulle parti del programma da cui si puo' trarre il massimo vantaggio, utilizzando delle:

- tecniche di vettorizzazione,
- tecniche di ottimizzazione.

Per fornire la conoscenza delle parti piu' eseguite di un programma, l'analizzatore stampa le seguenti informazioni:

- tempo trascorso in ogni unita' di programma
- lista delle istruzioni piu' eseguite ('istruzioni pesanti') con un istogramma.

Il sovraccarico in tempo di esecuzione prodotto dall'utilizzo dell'analizzatore, dipende dalla frequenza di campionatura. Quest'ultima puo' essere indicata dall'utente.

Funzionamento

Il funzionamento e' costituito di tre fasi:

1. Fase 'Probe'

   Il *Probe* si esegue mentre e' in esecuzione l'applicazione e determina, ad intervalli di tempo specificati dall'utente, quale istruzione e quale unita' di programma e' processata. Tutte queste informazioni sono quindi registrate in un file temporaneo.

2. Fase 'Sort'

Una volta che l'esecuzione dell'applicazione e' terminata, i dati contenuti nel file temporaneo sono ordinati per nome di sottoprogramma.

3. Fase 'Report'

   A partire dal file temporaneo e dalla lista di compilazione, l'analizzatore produce un report riassuntivo della distribuzione del tempo di esecuzione tra le unita' di programma e tra tutte le istruzioni.

Per ulteriori informazioni sull'analizzatore di esecuzione VS Fortran si rimanda al manuale:

- IBM VS Fortran Execution Analyser

Modalita' di utilizzo

L'analizzatore puo' essere usato con:

```
//EXEC nome-proc,PARM.GO='NAME(MAIN),INTERVAL(xxxx)'
//COMP.SYSIN DD *
      PROGRAM .....
        .
        .  programma sorgente VS Fortran
        .
      END
/*
```

Descrizione delle uscite

Nella pagine seguenti e' riportato un esempio di programma che e' stato esaminato con l' Analizzatore di esecuzione VS Fortran.

```
OPTIONS IN EFFECT: NOLIST NOMAP NOXREF NOGOSTMT NODECK SOURCE TERM OBJECT FIXED TRMFLG SRCFLG NOSYM NORENT
                   SDUMP(ISN) AUTODBL(NONE) NOSXM NOVECTOR IL NOTEST NODC NODIRECTIVE
                      OPT(0) LANGLVL(77) NOFIPS   FLAG(I)  NAME(MAIN)    LINECOUNT(60)      CHARLEN(500)
                                                                                                          ROUTINES
                                                                                                  % OF    TIME
 IF DO   ISN   *....*...1.........2.........3.........4.........5.........6.........7.*.......8      TOTAL   DISTRIBUTION

           1          DIMENSION X(50,50),Y(50,50),Z(50)
           2          DIMENSION  V(50),T(50,50)
           3          DIMENSION SOM(2500)
           4          EQUIVALENCE (Z(1),V(1))
           5          N=32
           6          M=50
           7          EPS=1.E-6
           8          DO2I=1,N
      1    9          DO2J=1,N/2
      2   10          Y(I,J)=33.*I**2.+27*J+1.
      2   11   2      X(I,J)=SQRT(I**2+J**2+1.)+EXP(1./(I+J+1))                          11.1   ***
          12          DO1I=1,N
      1   13          DO1J=N/2+1,N
      2   14          Y(I,J)=(33.*I**2+27*J+1.)/2.
      2   15   1      X(I,J)=(SQRT(I**2+J**2+1.)+EXP(1./(I+J+1)))/2.                       7.4   **
          16          DO6324J=1,N
      1   17          DO6324I=1,N
      2   18   6324   X(I,J)=1/Y(I,J)-X(I,J)
          19          DO324I=1,N
      1   20          DO324J=1,N
      2   21          T(I,J)=0                                                              7.4   **
      2   22          DO324K=1,N                                                           44.4   ************
      3   23   324    T(I,J)=T(I,J)+X(I,K)*Y(K,J)
          24          DO3240I=1,N
      1   25          DO3240K=1,N
      2   26          IF(Y(1,I).EQ.0)GOTO3241
      2   27   3240   T(K,I)=3.14*Y(K,I)+T(K,I)
          28   3241   CONTINUE
          29          DO3244I=1,N
      1   30          DO3243K=1,N
      2   31          IF(Y(1,I).NE.0)GOTO3244
      2   32   3243   T(K,I)=T(K,I)-3.14*Y(K,I)
      1   33   3244   CONTINUE
          34          DO234J=1,N
      1   35          IF(T(J,1).GT.0.)THEN
  1   1   36          Z(J)=T(J,J)+4*X(J,J)/5
  1   1   37          ELSE
  1   1   38          Z(J)=T(J,J)+5*X(J,J)/4
  1   1   39          ENDIF
      1   40   234    WRITE(6,432)Z(J)
          41   432    FORMAT(1X,F20.2)
          42   3547   FORMAT(1X,3F20.2)
          43          DO3457L=N+1,M
      1   44          DO3457K=N+1,M
      2   45          Z(L)=0
      2   46   3457   X(L,K)=0
          47          DO3456K=1,M
      1   48   3456   V(K)=X(MIN0(K,N),1)+Z(K)
          49          DO 456J=1,50
      1   50          DO456 I=1,50
      2   51          L=I+50*(J-1)
      2   52   456    SOM(L)=X(I,J)+I*J
```

```
LEVEL (0.B/0.B) VS FORTRAN EXECUTION ANALYZER

  SAMPLE INTERVAL         3000 MICROSECONDS

  DATE OF SAMPLING 86.266
  TIME OF SAMPLING 18:01:51

  NUMBER OF INTERRUPTS          27

                          % OF  TIME
PROGRAM NAME  PAGE        TOTAL DISTRIBUTION

UNKNOWN                   11.1  UUUUUUUUUUU
MAIN             1        88.9  ***************************************************************************UUUUUUUUUUUUUUU
```

Compilatore vettorizzatore VS Fortran Versione 2

La funzione principale della fase di vettorizzazione del compilatore vettorizzatore e' di ricercare tutti i cicli DO e di tradurli con un codice vettorizzato. Cio' pero' non avviene per tutti i cicli ma solo per quelli che traggono dei benefici dall'esecuzione vettoriale.

Per questo motivo il compilatore valuta, in prima istanza, il *peso* di ogni ciclo che puo' essere trattato con le operazioni della pipeline, e poi sceglie quelli piu' vantaggiosi. Tale scelta si basa su alcuni principi che ora andremo ad esaminare.

Scelta di un ciclo

La scelta se vettorizzare o meno un ciclo DO dipende essenzialmente dai seguenti fattori:

- Dimensione dei vettori.
- Tipo e numero di istruzioni all'interno del ciclo.

Nel caso di cicli multipli viene preferito quello piu' favorevole stabilito sulla base di elementi quali la dimensione dell'indice, lo stride minore, ecc...

Vettorizzazione di un ciclo

Prendendo in esame il seguente esempio, in cui sono presenti dei cicli multipli:

```
        DO 10 K=1,N
        DO 20 J=1,N
        DO 30 I=1,N
        A(K,J,I)=B(K,J,I) ...
30      CONTINUE
20      CONTINUE
10      CONTINUE
```

il compilatore opera la vettorizzazione prima scegliendo il ciclo migliore, che supponiamo essere il DO 10, e poi genenerando un codice del seguente tipo:

```
        DO 10 K=1,N,128
        DO 20 J=1,N
        DO 30 I=1,N

           DO 40 K1=K,K+MIN(N-K,127)
           A(K1,J,I)=B(K1,J,I) ...
40         CONTINUE

30      CONTINUE
20      CONTINUE
10      CONTINUE
```

Vettorizzazione di un ciclo contenente un'istruzione IF

Anche se la vettorizzazione di un ciclo con un IF e' gia' stata oggetto di analisi in precedenza, e' bene comunque ricordarla.
Se abbiamo, per esempio, il codice:

```
          DO 10 I=1,N
          Y(I)=Y(I)+A*X(I)
          IF (Y(I).GT.10.)) Y(I)=Y(I)+B*X(I)
10        CONTINUE
```

il compilatore trasforma la suddetta sequenza nel modo seguente:

```
1.        Y(I)=Y(I)+A*X(I)
2.        maschera(I)=Y(I).GT.10.
3.        TEMP(I)=Y(I)+B*X(I)
4.        Y(I)=TEMP(I) secondo maschera(I)
```

Si puo' notare, in questo esempio, che l'espressione Y(I) + B*X(I) e' calcolata per tutti i valori di Y(I), per cui se la condizione di Y(I).GT.10. e' spesso falsa, la vettorizzazione del ciclo non e' produttiva.

Riorganizzazione delle istruzioni

Questa tecnica e' utilizzata dal compilatore per permettere la vettorizzazione di alcuni tipi di cicli DO.

Per esempio il codice:

```
          DO 10 I=2,N
          Y(I)=Z(I-1)
          Z(I)=V(I)
10        CONTINUE
```

non sarebbe vettorizzabile per la presenza di una dipendenza, per cui e' trasformato dal compilatore in:

```
          DO 10 I=2,N
          Z(I)=V(I)
          Y(I)=Z(I-1)
10        CONTINUE
```

Frazionamento dei cicli

Questa tecnica e' utilizzata dal compilatore per permettere la vettorizzazione di una parte del ciclo.

Il codice:

```
          DO 10 I=2,N,2
          A(I)=A(I)+2
          B(I+2)=B(I)+2.
10        CONTINUE
```

e' trasformato dal compilatore come segue:

```
      DO 20 I=2,N,2
      A(I)=A(I)+2.
20    CONTINUE

      DO 30 I=2,N,2
      B(I+2)=B(I)+2.
30    CONTINUE
```

Dopo lo sdoppiamento il primo ciclo e' vettorizzato mentre il secondo viene eseguito in modo scalare per un inibitore di ricorrenza.

Esempio di compilazione

Le pagine che seguono mostrano una lista prodotta dal compilatore VS Fortran con le opzioni (VECTOR(LEV(2),REP(XLIST)).

```
REQUESTED OPTIONS (EXECUTE): OPT(3),VECTOR(LEV(2),REP(XLIST))

OPTIONS IN EFFECT: NOLIST NOMAP NOXREF NOGOSTMT NODECK SOURCE TERM OBJECT FIXED TRMFLG SRCFLG NOSYM NORENT
                   SDUMP(ISN) AUTODBL(NONE) NOSXM VECTOR IL NOTEST NODC NODIRECTIVE
                      OPT(3) LANGLVL(77) NOFIPS   FLAG(I)  NAME(MAIN)    LINECOUNT(60)      CHARLEN(500)
                      VECTOR (LEVEL(2)    INTRINSIC    REDUCTION  SIZE(ANY)    REPORT(XLIST))

 IF DO   ISN   *....*...1.........2.........3.........4.........5.........6.........7.*.......8

          1           DIMENSION X(50,50),Y(50,50),Z(50)
          2           DIMENSION  T(50,50)
          3           DIMENSION SOM(2500)
          4           N=43
          5           N=50
          6           DO 10 I=1,N
     1    7             DO 20 J=1,N/2
     2    8               Y(I,J)=FLOAT(33*I*I+27*J+1)
     2    9               X(I,J)=SQRT(FLOAT(I*I+J*J+1))+EXP(1./FLOAT(I+J+1))
     2   10     20      CONTINUE
     1   11     10    CONTINUE
         12           DO 30 I=1,N
     1   13             DO 40 J=N/2+1,N
     2   14               Y(I,J)=FLOAT(33*I*I+27*J+1)*0.5
     2   15               X(I,J)=(SQRT(FLOAT(I*I+J*J+1))+EXP(1./FLOAT(I+J+1)))*0.5
     2   16     40      CONTINUE
     1   17     30    CONTINUE
         18           CALL MULTI(T,X,Y,N)
         19           CALL MULTI(X,T,Y,N)
         20           DO 80 I=1,N
     1   21             Z(I)=T(I,I)+0.8*X(I,I)
     1   22     80    CONTINUE
         23           WRITE(6,1000 )(Z(I),I=1,N)
         24   1000    FORMAT(1X,F20.2)
         25           DO 90 I=N+1,M
     1   26             Z(I)=0.
     1   27             DO 100 J=N+1,M
     2   28               X(I,J)=0.
     2   29    100      CONTINUE
     1   30     90    CONTINUE
         31           DO 110 I=1,M
     1   32             Z(I)=X(MINO(I,N),1)+Z(I)
     1   33    110    CONTINUE
         34           DO 2000 I1=1,3
     1   35           DO 120 J=1,50
     2   36             DO 130 I=1,50
     3   37               L=I+50*(J-1)
     3   38               SOM(L)=X(I,J)+SQRT(FLOAT(I*J*I1))
     3   39    130      CONTINUE
     2   40    120    CONTINUE
     1   41           CALL SOMM(S,SOM)
     1   42           WRITE(6,1000 )S
     1   43   2000    CONTINUE
         44           WRITE(6,1000 )(Z(I),I=1,M)
         45                   STOP
         46                   END
```

REPORT(XLIST) VECTORIZATION ANALYSIS

```
ISN  FLAG NESTING *....*...1.........2.........3.........4.........5.........6.........7.*.......8 MESSAGES

0001                       DIMENSION X(50,50),Y(50,50),Z(50)
0002                       DIMENSION  T(50,50)
0003                       DIMENSION SOM(2500)
0004                       N=43
0005                       N=50                                                      NESTED NON-CONSTANT INDUCTION
0006 RECR +-------         DO 10 I=1,N
0007 VECT !+------           DO 20 J=1,N/2
0008      !!                   Y(I,J)=FLOAT(33*I*I+27*J+1)                           VECTOR INTRINSIC FUNCTION
0009      !!______             X(I,J)=SQRT(FLOAT(I*I+J*J+1))+EXP(1./FLOAT(I+J+1))
          !_______
                                                                                     NESTED NON-CONSTANT INDUCTION
0012 RECR +-------         DO 30 I=1,N
0013 VECT !+------           DO 40 J=N/2+1,N
0014      !!                   Y(I,J)=FLOAT(33*I*I+27*J+1)*0.5                       VECTOR INTRINSIC FUNCTION
0015      !!______             X(I,J)=(SQRT(FLOAT(I*I+J*J+1))+EXP(1./FLOAT(I+J+1)))*0.5
          !_______
0018                       CALL MULTI(T,X,Y,N)
0019                       CALL MULTI(X,T,Y,N)                                       SCALAR FASTER THAN VECTOR
0020 ELIG +-------         DO 80 I=1,N
0021      !_______           Z(I)=T(I,I)+0.8*X(I,I)
0023                       WRITE(6,1000 )(Z(I),I=1,N)
0024                  1000 FORMAT(1X,F20.2)
0025 VECT +-------         DO 90 I=N+1,M
0026      !                  Z(I)=0.
0027 ELIG !+------           DO 100 J=N+1,M
0028      !!______             X(I,J)=0.
          !_______

0031 UNSP +-------         DO 110 I=1,M                                              IN-LINE INTRINSIC FUNCTION
0032      !_______           Z(I)=X(MINO(I,N),1)+Z(I)
                                                                                     I/O OPERATION
0034 UNAN                  DO 2000 I1=1,3                                            USER FUNCTION OR SUBROUTINE
                                                                                     NESTED NON-CONSTANT INDUCTION
0035 RECR +-------         DO 120 J=1,50
0036 VECT !+------           DO 130 I=1,50                                           VECTOR INTRINSIC FUNCTION
0038      !!______             SOM(L)=X(I,J)+SQRT(FLOAT(I*J*I1))
          !_______
0041                       CALL SOMM(S,SOM)
0042                       WRITE(6,1000 )S
0043                  2000 CONTINUE
0044                       WRITE(6,1000 )(Z(I),I=1,M)
0045                               STOP
0046                               END
```

NUMBER	ISN	FLAG	VS FORTRAN VECTOR REPORT MESSAGES
ILX0129I	0006	RECR	THE LOOP VARIABLE OF THIS LOOP OR OF SOME NESTED LOOP AFFECTS THE LOOP VARIABLE OR AN AUXILIARY INDUCTION VARIABLE USED BY SOME OTHER NESTED LOOP.
ILX0151W	0009	VECT	THE INTRINSIC FUNCTION(S) "EXP" AND "SQRT" HAVE BEEN VECTORIZED.
ILX0129I	0012	RECR	THE LOOP VARIABLE OF THIS LOOP OR OF SOME NESTED LOOP AFFECTS THE LOOP VARIABLE OR AN AUXILIARY INDUCTION VARIABLE USED BY SOME OTHER NESTED LOOP.
ILX0151W	0015	VECT	THE INTRINSIC FUNCTION(S) "EXP" AND "SQRT" HAVE BEEN VECTORIZED.
ILX0148I	0020	ELIG	CODE THAT WAS ELIGIBLE TO EXECUTE IN VECTOR MODE WAS DETERMINED TO EXECUTE MORE EFFICIENTLY IN SCALAR.
ILX0140I	0032	UNSP	NO VECTOR SUPPORT EXISTS FOR THE IN-LINE INTRINSIC FUNCTION(S) "MIN".
ILX0104I	0034	UNAN	ONE OR MORE I/O STATEMENTS OCCUR AT ISN(S) "42".
ILX0114I	0034	UNAN	THE USER FUNCTION(S) OR SUBROUTINE(S) "SOMM" AT ISN "41" ARE NOT ANALYZABLE.
ILX0129I	0035	RECR	THE LOOP VARIABLE OF THIS LOOP OR OF SOME NESTED LOOP AFFECTS THE LOOP VARIABLE OR AN AUXILIARY INDUCTION VARIABLE USED BY SOME OTHER NESTED LOOP.
ILX0151W	0038	VECT	THE INTRINSIC FUNCTION(S) "SQRT" HAVE BEEN VECTORIZED.

```
*STATISTICS*   SOURCE STATEMENTS = 46, PROGRAM SIZE = 43356 BYTES, PROGRAM NAME = MAIN    PAGE:    1.
*STATISTICS*    NO DIAGNOSTICS GENERATED.
**MAIN** END OF COMPILATION 1 ******
```

Le direttive di vettorizzazione

Una direttiva e' un ordine dato al compilatore per forzare o meno la vettorizzazione di un ciclo da parte del programmatore. Bisogna pero' tenere presente che l'uso di alcune direttive puo' essere pericoloso.

Quando si usano le direttive

L'utilizzo delle direttive puo' essere utile nei seguenti otto casi:

1. Il numero di iterazioni e' sconosciuto dal compilatore

   Ad esempio il ciclo:

```
         DO 10 I=L,M,N
         .........
   10    CONTINUE
```

   a seconda dei valori assunti in esecuzione dalle variabili L, M e N potrebbe essere piu' veloce in scalare che in vettoriale.

2. Il limite superiore dell'indice del ciclo e' sconosciuto dal compilatore

   Cosi' nel ciclo:

```
         DO 10 I=1,N
         A(I+133)=A(I)*B(I)
   10    CONTINUE
```

   c'e' una ricorrenza se il valore di N e' superiore a 133.

3. L'incremento del ciclo DO e' sconosciuto dal compilatore.

```
         DO 10 I=L,N,M
         A(I)=A(I+1)*B(I)
   10    CONTINUE
```

   non e' vettorizzabile perche', a seconda del segno di M, c'e' una ricorrenza.

4. L'incremento delle variabili indice e' sconosciuto dal compilatore.

   Ad esempio il ciclo:

```
      J=20
      DO 10 I=1,N
      A(I)=A(J)*B(I)
      J=J+M
10    CONTINUE
```

non e' vettorizzabile perche', a seconda del segno e del valore di M, c'e' una ricorrenza.

5. Lo spostamento dell'indice e' sconosciuto dal compilatore.

   Nel ciclo:

```
      DO 10 I=1,N
      A(I+M)=A(I)*B(I)
10    CONTINUE
```

   a seconda del segno e del valore di M c'e' una ricorrenza.

6. La dipendenza e' dovuta ad una istruzione di EQUIVALENCE.

   Nell'esempio seguente:

```
      EQUIVALENCE (A(1),B(1))
      .............
      DO 10 I=1,N
      A(I)=B(I+50)
10    CONTINUE
```

   c'e' una ricorrenza per la sovrapposizione dei vettori A e B.

7. La dipendenza e' dovuta ad un indirizzamento indiretto.

   Nel codice seguente:

```
      .......
      DO 10 I=1,N
      A(M1(I))=A(M2(I))+B(I)
10    CONTINUE
```

   gli elementi del vettore A non sono indipendenti.

8. Scelta del ciclo DO

   Nel codice seguente:

```
        .......
        DO 10 I=1,N
        .......
        DO 20 I=1,N
        .......
20      CONTINUE
        ....
10      CONTINUE
```

e' possibile scegliere il ciclo migliore.

Perche' si usano le direttive

L'utilizzo delle direttive da parte dell'utente puo' portare una soluzione agli 8 problemi precedenti. E' pero' necessario un uso accorto di tale strumento in quanto puo' indurre il compilatore all'errore. Per tale motivo e' bene sottolineare le seguenti situazioni:

1. I risultati del programma saranno indipendenti dall'uso delle direttive per i casi:
   - (1). Numero di iterazioni sconosciuto
   - (8) Scelta ciclo
2. I risultati del programma possono essere errati, a causa del cattivo uso delle direttive, per risolvere i seguenti cassi:
   - (2) Limite superiore dell'indice del ciclo sconosciuto
   - (3) Incremento del ciclo DO sconosciuto
   - (4) Incremento della variabile induce sconosciuto
   - (5) Spostamento di indice sconosciuto
   - (6) Dipendenza dovuta a EQUIVALENCE sconosciuta
   - (7) Dipendenza dovuta ad indirizzamento indiretto sconosciuta

Come si usano le direttive

L'utilizzo delle direttive e' controllato a livello di unita' di programma dall'istruzione:

@PROCESS DIRECTIVE('stringa di caratteri')

Se, per esempio, si codifica @PROCESS DIRECTIVE('*CNUCE*') in un programma Fortran, la direttiva specifica va poi scritta prima del ciclo DO interessato sotto la forma:

C*CNUCE* direttiva

dove direttiva puo' essere:

ASSUME COUNT (n)	Specifica una stima di numero di iterazioni
PREFER {SCALAR/VECTOR}	Richiede l'esecuzione del ciclo nel modo specificato:

- SCALAR: in modo scalare
- VECTOR: in modo vettoriale

Il modo VECTOR e' inibito esiste un inibitore all'-interno del ciclo.

IGNORE EQUDEPS Specifica di ignorare la dipendenza dovuta ad una equivalenza.

IGNORE RECRDEPS (nome di array) Specifica di ignorare una dipendenza.

A questo punto e' pero' necessario fare alcune considerazioni sull'uso delle suddette direttive:

- ASSUME COUNT e' la direttiva meno pericolosa, per cui puo' essere usata frequentemente.
- IGNORE deve essere usata con precauzione.
- PREFER va usata solo dopo misure ed esperimenti.

Quindi le direttive possono, se mal utilizzate, deteriorare le prestazioni e, nel peggiore dei casi, rendere sbagliato un programma corretto. Per tali ragioni e' bene attenersi sempre a queste regole:

- Utilizzarle le direttive solo in caso di necessita' e se non e' possibile riscrivere il codice.
- Utilizzare solo quelle direttive strettamente indispensabili.
- Utilizzare le direttive dove se ne tragga un vantaggio immediato.
- Diffidare del nostro *intuito.*

Esempi di uso delle direttive

Riportiamo di seguito alcuni esempi di uso delle direttive:

1. Esempio per fornire delle stime del numero di iterazioni:

```
@PROCESS DIR ('*CNUCE*')
      SUBROUTINE
        ............
C*CNUCE* ASSUME COUNT(10)
      DO 10 I=1,N
        .....
C*CNUCE* ASSUME COUNT(500)
      DO 20 I=1,M
        .....
20    CONTINUE
10    CONTINUE
```

2. Esempio per imporre l''esecuzione vettoriale

   Il compilatore non vettorizza il caso seguente stimando che l'esecuzione scalare sia piu' veloce, ma, confrontando i tempi di esecuzione ottenuti con l'opzione SCALAR e VECTOR, l'esecuzione vettoriale risulta essere piu' veloce.

```
C*CNUCE* PREFER VECTOR
      DO 10 I=1,20
         .....
10    CONTINUE
```

3. Esempio in cui il ciclo esterno e' preferibile:

```
    ................
C*CNUCE* PREFER VECTOR
      DO 10 I=1,N
         .....
C*CNUCE* PREFER SCALAR
      DO 20 I=1,M
20    CONTINUE
         .....
10    CONTINUE
```

   In questo caso poteva essere usata anche la direttiva ASSUME COUNT, che per quanto detto prima e' da preferirsi.

4. Esempio in cui il compilatore vede una dipendenza, che e' *immaginaria,* in quanto il programmatore sa con certezza che M e' superiore a 128:

```
C*CNUCE* IGNORE RECRDEPS(A)
      DO 10 I=1,N
      A(I+M)=A(I)*B(I)
10    CONTINUE
```

Un programma di conversione di linguaggio: LCP

Come gia' detto il prodotto LCP (Language Conversion Program) serve per convertire i programmi scritti in linguaggio Fortran dal livello 66 al livello 77. Questa conversione puo' essere vantaggiosa per chi vuole sfruttare una delle seguenti possibilita':

- utilizzo dell'indirizzamento esteso a 31 bits di con la possibilita' di usare memorie virtuali con una taglia superiore a 16MB;
- uso di Common dinamici e superiori a 16Mb;
- produzione di codice rientrante;
- accesso a sottoprogrammi delle nuove librerie disponibili con VS Fortran;
- utilizzo, soprattutto, dell'unita' di calcolo vettoriale con il VS Fortran Versione 2;

Un altro vantaggio importante offerto dall'uso dell'LCP e' quello di mantenere tutto il software prodotto in modo omogeneo, evitando quindi di gestire i vecchi ed i nuovi programmi con compilatori e librerie diversi.

La finalita' dell'LCP

Il programma di conversione di linguaggio e' uno strumento creato per facilitare la conversione di programmi dal linguaggio Fortran IV in Fortran 77.

L' input dell'LCP e' costituito dai programmi sorgenti scritti per i seguenti compilatori IBM Fortran 66:

- Fortran G1
- Fortran H
- Fortran H esteso
- VS Fortran livello 66

Le conversioni fatte dall'LCP consistono in:

- conversione sintattica
- conversione semantica
- eliminazione di messaggi WARNING

Le conversioni semantiche sono particolarmente importanti perche' una differenza semantica puo' non essere considerata come errore dal compilatore VS Fortran, ma dare risultati differenti nell'esecuzione.

L'uscita di LCP e' un programma sorgente scritto in linguaggio VS Fortran livello 77 ed una lista che indica le modifiche apportate.

Principali conversioni effettuate dall'LCP

I principali cambiamenti del codice effettuati dall'LCP possono essere riassunti nei seguenti punti:

1. Esecuzione di un ciclo DO

   In Fortran 77, le istruzioni contenute in un ciclo DO possono non essere mai eseguite nel caso in cui il limite dell'indice sia inferiore al valore di partenza, mentre con il Fortran 66, un ciclo DO e' sempre eseguito almeno una volta. L'LCP deve percio' modificare l'istruzione DO affinche' il ciclo sia eseguito almeno una volta.

2. Argomenti fittizzi

   In Fortran IV, gli argomenti fittizzi possono essere passati sia come valore che come riferimento. In Fortran 77, si possono passare solo riferimenti.

3. Istruzione DEFINE FILE

   L'istruzione DEFINE FILE non esiste nel livello 77, per cui viene rimpiazzata da un'istruzione di OPEN. La conversione realizzata da LCP tiene conto delle differenze esistenti tra queste due istruzioni.

4. Sintassi di READ e WRITE

   La sintassi delle istruzioni di READ e WRITE presenta alcune differenze in Fortran 77 rispetto al Fortran 66. L'LCP risolve questa situazione con l'aggiunta di una chiamata ad un sottoprogramma che opera la modifica della sintassi.

5. Istruzione IMPLICIT

   Il Fortran H, H esteso e VS Fortran 66 possono ammettere piu' specifiche di IMPLICIT per lo stesso carattere alfabetico, mentre il VS Fortran 77 ed il Fortran G1 ne accettano una sola. Nel programma convertito l'LCP sopprimera' quindi le specifiche multiple e conservera' solo l'ultima dichiarazione IMPLICIT.

Utilizzo dell'LCP

Il prodotto LCP funziona sia sotto MVS che sotto VM

I programmi sorgente che costituiscono l'input possono essere files sequenziali o membri di un file partitioned e si possono elaborare piu' programmi con la stessa esecuzione dell'LCP.

L'LCP genera in uscita tre files particolarmente interessanti:

- il source convertito, che contiene le istruzioni del VS Fortran 77;
- il listing di comparazione, in cui sono descritti, in parallelo, sia il source originale (Fortran IV) che il source generato da LCP (Fortran 77);
- la lista della diagnostica che contiene l'insieme dei messaggi prodotti da LCP

Migrazione delle applicazioni

Introduzione

In questo capitolo si vuole dare una sintesi di tutto cio' che e' stato detto sulle architetture vettoriali, sui problemi della vettorizzazione e quindi sulla possibilita' di apportare piccole o grandi modifiche al codice. Non e' nostra intenzione riprendere gli argomenti gia' ampiamente discussi, ma di fornire alcuni consigli e di ponderare certe possibilita' attualmente esistenti che possono agevolare il lavoro del programmatore.

L'evoluzione continua delle architetture degli elaboratori e lo sviluppo del calcolo vettoriale permettono oggi l'uso di tale tecnica anche da utenti non specificamente informatici, come ingegneri, chimici, matematici, ecc... Un errore pero' che viene spesso commesso e' quello di avvicinarsi al calcolo vettoriale come se fosse uguale al calcolo scalare. Il passaggio richiede infatti un'analisi preliminare necessaria se si vogliono aumentare le prestazioni dell'applicazione. Non e' pensabile fare del calcolo vettoriale se non esistono i presupposti di un utilizzo pieno ed intensivo delle nuove architetture su cui andiamo ad elaborare.

Per cui e' indispensabile valutare se l'applicazione e':

- suscettibile di avere un elevato tasso di vettorizzazione;
- adattabile, in modo molto facile, all'architettura dell'elaboratore utilizzato.

Questo lavoro pero' deve essere fatto intelligentemente, in modo da minimizzare il tempo speso per la valutazione necessaria, con la prospettiva di ottenere la massima efficacia del risultato. Percio' lo scopo della migrazione delle applicazioni e' di avere degli strumenti per:

- fare una classifica dei programmi candidati
- ottimizzare ogni programma in funzione delle caratteristiche della Vector Facility che si intende utilizzare.

Strategie per la scelta dei programmi da vettorizzare

La migrazione delle applicazioni e' normalmente limitata ad un insieme circoscritto di lavori che possono essere convertiti in un tempo ragionevole. Si prensenta quindi la necessita' di operare una selezione tra tutte le applicazioni che vengono spesso elaborate. La seconda fase consistera' poi in una classifica, per mettere in testa quei programmi sui quali si concentrera' maggiormente il nostro sforzo. Per questo tipo di analisi ci si puo' attenere ai seguenti criteri:

- Tempo di esecuzione e frequenza di elaborazione

  E' evidente che i programmi che hanno un tempo di esecuzione breve o che sono elaborati molto raramente, costituiscono dei candidati poco interessanti. E' piu' redditizio dedicare il proprio tempo per migrare programmi lunghi e frequentemente utilizzati.

- Utilizzo della CPU

In generale, il miglioramento delle prestazioni sara' piu' sensibile sui programmi che hanno un altissimo tasso di utilizzo dell'unita' centrale. Ma non bisogna escludere a priori i programmi che hanno una grossa attivita' di I/O, perche' la loro riorganizzazione, legata alla vettorizzazione, puo' ridurre sensibilmente questa attivita' di I/O.

- Virgola mobile

  La VF puo' apportare miglioramenti significativi su vettori interi o logici; tuttavia, le prestazioni saranno in generale migliori su operazioni in virgola mobile.

Anche altri elementi giocano un ruolo importante nella strategia della migrazione:

- Il modo in cui l'applicazione utilizza la memoria poiche' e' preferibile che i dati utilizzati nei cicli DO si trovino in indirizzi contigui di memoria.
- La conoscenza degli algoritmi utilizzati nell'applicazione, in quanto e' piu' difficile ottimizzare un programma scritto da un'altra persona o, piu' in generale, un programma di cui non si conosce ne' la scrittura ne' gli algoritmi usati.
- La possibilita' di rimpiazzare delle parti del codice o delle chiamate a sottoprogrammi di librerie diverse, con l'uso di sottoprogrammi ottimizzati per la VF ed appartenenti alla libreria ESSL.

La combinazione di questi differenti elementi, che comprendono sia considerazioni globali, sia considerazioni molto contingenti, induce a chiedersi come si possa creare una metodologia generale per realizzare la migrazione di una applicazione scalare verso l'unita' di calcolo vettoriale. Cercheremo di rispondere in parte a questa domanda nel seguito del capitolo, in cui sono descritte le differenti fasi riguardanti la migrazione dell'applicazione.

Prima fase della vettorizzazione: ripulire

Una applicazione utilizzata in modo scalre ha generalmente una storia molto lunga. Essa e' stata scritta da una o piu' persone e per un calcolatore avente una architettura ben precisa; spesso in seguito e' stata completata, o modificata, da altre persone per i piu' svariati motivi.

Questa storia provoca quasi sempre almeno due conseguenze molto nefaste:

- La lettura e la comprensione del codice risultano difficili;
- Le prestazioni in modo scalare sono lontane dall'essere quelle ottimali.

Cominciare il lavoro di vettorizzazione con questi presupposti e' senza dubbio un errore, per cui la prima fase consistera' nel *ripulire* l'applicazione.

Lo scopo da raggiungere e' di ottenere un programma che abbia le caratteristiche che analizzeremo di seguito.

Programma chiaro e ben strutturato

Nel capitolo *Alcuni principi essenziali della programmazione* sono gia' stati elencati alcuni principi a cui attenersi per una scrittura chiara del programma.

La programmazione strutturata e' ancora troppo poco conosciuta e quindi utilizzata nell'ambiente scientifico e tecnico, tuttavia deve essere compiuto un minimo sforzo per:

- Rendere il programma piu' chiaro, scomponendolo in in blocchi logici. Questa operazione puo' tornare anche utile qualora si decida di utilizzare il multitasking.

- Eliminare la maggior parte di GOTO a favore dell'utilizzo dei cicli DO, che sono candidati alla vettorizzazione.
- Togliere le istruzioni inutili.
- Eliminare, dove e' possibile, le istruzioni dei cicli DO che possono costituire delle inibizioni alla vettorizzazione, in particolare le istruzioni di READ/WRITE.

Uso del Fortran 77

Spesso le applicazioni sono rimaste a livello di una versione di Fortran molto vecchia. Poiche' solo il compilatore VS Fortran versione 2 puo' generare codice oggetto per l'unita' di calcolo vettoriale, e' dunque consigliabile mantenere i programmi scritti in Fortran 77. E' percio' utile una loro conversione che apporta anche i seguenti vantaggi:

- possibilita' di indirizzamento a 31 bits;
- possibilita' di usare la 'dynamic common area' e nella parte superiore ai 16MB;
- la rientranza del codice;
- l'accesso alle nuove librerie di sottoprogrammi, disponibili con il VS Fortran;

La sola compilazione di un codice scritto in Fortran 66, con il VS Fortran 2 gia' puo' segnalare:

- errori di sintassi, che saranno rilevati al momento della compilazione;
- errori di semantica, non rilevati alla compilazione, ma molto pericolosi perche' possono portare a risultati diversi in fase di esecuzione.

L'LCP e' uno strumento essenziale per facilitare queste operazioni di conversione.

Ottimizzazione del codice

Ci sono pochissime possibilita' che una applicazione non ottimizzata in modo scalare divenga una applicazione molto efficiente in vettoriale.

In vista di una migliore ottimizzazione e' necessario porre la nostra attenzione su alcuni elementi che possono rappresentare un ostacolo alla ottimizzazione del codice:

- rimpiazzare alcune istruzioni *costose* come:

```
A**2    con    A*A
A/2.    con    A*0.5
```

- valutare le operazioni di I/O in funzione del loro numero, della loro organizzazione, della loro collocazione nell'interno dell'applicazione ecc ...
- utilizzare maggiormente la memoria virtuale, ricordandosi che l'architettura 370-XA permette di avere a disposizione uno spazio di 2GB. Questo permette di ridurre considerevolmente le operazioni di I/O e dunque di aumentare le possibilita' di vettorizzazione dell'applicazione.
- analizzare l'organizzazione dei dati in memoria per diminuire i tempi di accesso.

Seconda fase della vettorizzazione: analizzare

Il tempo dell'unita' centrale consumato da un programma e' spesso concentrato in alcune parti del codice. Questo e' soprattutto vero nell'ambiente scientifico e tecnico, dove spesso il 90% del tempo di calcolo e' imputatabile a uno o due sottoprogrammi. E' quindi necessario impegnare i propri sforzi su quelle parti del codice che consumano la maggior quantita' del tempo di calcolo. Questo tipo di analisi e' anche importante per una conoscenza piu' approfondita dell'applicazione e delle prestazioni che si possono raggiungere.

La seconda fase consiste dunque nell'analizzare il programma per poter determinare:

- i sottoprogrammi che utilizzano maggiormente l'unita' centrale

  Se il tempo di calcolo e' concentrato su una piccola parte del codice, identificabile in uno o due sottoprogrammi, allora esistono buone probabilita' che il lavoro di vettorizzazione sia rapido ed efficace

- La distribuzione del tempo di CPU relativa a tutte le istruzioni presenti in ogni unita' di programma, ed in particolar modo quella relativa ai cicli DO.

Questa fase consente quindi di raccogliere i seguenti elementi:

- le routines *pesanti*
- le istruzioni *pesanti*
- il tempo di esecuzione scalare che possiamo indicare *T0*

Lo strumento piu' idoneo da utilizzare in questa fase e' *l'Analizzatore di esecuzione* che e' stato descritto in precedenza.

Terza Fase della vettorizzazione: vettorizzazione automatica

Dopo le fasi precedenti e' ora possibile compilare il programma in modo vettoriale con l'opzione VECTOR(LEVEL(2)) ed effettuare un esecuzione vettoriale. Se il tempo di esecuzione del programma, che indichiamo con *T1*, corrisponde all'obiettivo che si era prefissato, non e' necessario andare oltre poiche' il lavoro operato sul programma scalare e quello fatto dal compilatore sono stati efficaci.

In caso contrario e' opportuno ripetere l'esecuzione vettoriale sotto il controllo dell' *Analizzatore di esecuzione* ed effettuare un'analisi dei seguenti fattori:

- Le uscite del compilatore, con le indicazioni sulla vettorizzazione o meno dei cicli DO.
- Le uscite dell'analizzatore di esecuzione con i pesi dei sottoprogrammi e istruzioni.

Studio dei messaggi del compilatore

Il compilatore VS Fortran V2 riporta da una parte la lista classica del codice, e dall'altra una lista piu' semplificata, ma che fornisce per ogni ciclo DO, delle indicazioni sulla vettorizzazione.

Come gia' visto, si possono avere le seguenti situazioni:

1. VECT: il ciclo e' stato vettorizzato.

   E' in generale, il risultato migliore, anche se e' bene controllare, nel caso di indice variabile del ciclo, che non sia preferibile l'esecuzione scalare.

2. ELIG: il ciclo e' in teoria vettorizzabile, ma non viene vettorizzato.

Nonostante che il compilatore riconosca la possibilita' di vettorizzare il ciclo, decide di non farlo in quanto ritiene migliore l'esecuzione scalare. In ogni caso il compilatore specifica nella lista il motivo di tale scelta. Anche in questo caso e' utile controllare per forzare eventualmente la vettorizzazione mediante l'uso di una direttiva.

3. UNAN: il ciclo non e' vettorizzabile.

   Questa condizione e' dovuta alla presenza di un inibitore, che viene precisato dal compilatore nella lista prodotta. Se il ciclo in esame ha un peso importante nell'applicazione e' necessario apportare delle modifiche al codice.

4. RECR: il ciclo non e' vettorizzabile per ragioni diverse.

   Queste ragioni sono indicate nella lista prodotta dal compilatore. Anche in questo caso, saranno necessarie delle modifiche al codice se il ciclo ha un peso significativo.

Studio delle uscite dell'analizzatore di esecuzione.

Sono identiche alle uscite del modo scalare, e permettono dunque di avere:

- le routines pesanti
- le istruzioni pesanti

Con gli strumenti sopra indicati e' quindi possibile avere un'indicazione abbastanza precisa del tasso di vettorizzazione e del numero di interventi da fare sul codice. Prima di continuare bisogna pero' considerare a fondo quale guadagno di prestazioni si riuscira' ad ottenere e quanto tempo sara' necessario per tali modifiche. In altre parole e' indispensabile valutare attentamente che il rapporto costo/prestazioni sia favorevole.

Bisogna pero' sottolineare che, almeno a livello della prima stesura in vista dell'ottimizzazione dell'applicazione, alcune modifiche molto semplici, come l'inversione degli indici del ciclo, comportano un significativo guadagno nell'esecuzione.

Quarta fase della vettorizzazione: modifiche semplici

Tali modifiche possono essere effettuate a diversi livelli che adesso andiamo ad esaminare.

Utilizzo delle direttive

In base all'analisi delle uscite del compilatore ed alla conoscenza del codice, con le direttive di compilazione e' possibile imporre al compilatore delle scelte, secondo i seguenti fattori:

- lunghezza del ciclo
- vettorizzazione imposta o interdetta
- ricorrenze vere o false.

Utilizzo della libreria ESSL

Il sostituire alcuni blocchi del codice con un sottoprogramma della libreria ESSL comporta un significante guadagno del tempo di esecuzione. Allo stesso modo, bisognera' rimpiazzare, se possibile, tutte le chiamate a sottoprogrammi di librerie diverse con le equivalenti chiamate ai sottoprogrammi della ESSL.

Modifiche del codice

Le modifiche del codice sono legate sia alle indicazioni UNAN e RECR del compilatore, che a uno studio particolareggiato dei cicli molto pesanti sui quali si intravedono utili modifiche.

A tale proposito ricordiamo alcuni punti importanti:

1. Problemi di indici

   Gli indici dei vettori non sempre coincidono con gli indici del ciclo e quando gli indici dei vettori sono calcolati all'interno del ciclo il compilatore non sempre riconosce se il ciclo e' vettorizzabile.

```
C     Ciclo originale          C     Ciclo ottimizzato

      DO 10 I=1,N                    DO 10 I=1,N
      J=I+5                          R(I+5)=1.0
      R(J)=1.0                  10   CONTINUE
10    CONTINUE
```

Figura 18. Ciclo DO con calcolo di indice

In Figura 18 l'eliminazione del calcolo di J evita un'istruzione inutile, inoltre il ciclo ciclo puo' essere vettorizzato.

Come regola generale, si puo' affermare che piu' il calcolo degli indici sara' semplice, e migliore sara' l'esecuzione, come e' mostrato in Figura 19

```
C     Ciclo originale          C     Ciclo ottimizzato

      DO 10 J=1,M                    DO 10 J=1,M
      ...                            ...
      DO 11 I=1,N                    INC=(J-1)*N
      K=(J-1)*N+I                    DO 11 I=1,N
      A(I)=B(K)                      A(I)=B(I+INC)
11    CONTINUE                 11    CONTINUE
10    CONTINUE                 10    CONTINUE
```

Figura 19. Ciclo DO con calcolo di indice

2. Segmentazione dei cicli

   Si possono frazionare certi cicli, in vista di una loro migliore esecuzione vettoriale, come si puo' vedere in Figura 20 a pag. 82

```
      DO 10 J=1,M                   DO 10 J=1,M
      A(J)=S*A(J)                   A(J)=S*A(J)
      B(J)=S*B(J)                   B(J)=S*B(J)
      C(J)=A(J)+B(J)                C(J)=A(J)+B(J)
      D(J)=S*C(J)                   D(J)=S*C(J)
                              10    CONTINUE
                                    DO 20 J=1,M
      DO 30 I=1,N                   DO 30 I=1,N
      E(I,J)=E(I,J)+D(J)            E(I,J)=E(I,J)+D(J)
30    CONTINUE                30    CONTINUE
10    CONTINUE                20    CONTINUE
```

Figura 20. Cicli DO prima e dopo la segmentazione

3. Distribuzione dei cicli

   La figura seguente mostra un ciclo non vettorizzabile a causa di una ricorrenza:

```
      DO 15 I=2,N
      AA(I)=AA(I)+B(I)*B(I)
      X(I)=X(I-1)+Y(I)
15    CONTINUE
```

Figura 21. Ciclo non vettorizzabile

La distribuzione in due cicli permette di vettorizzare il ciclo su AA, mentre rimane non vettorizzato il ciclo su X.

```
      DO 15 I=2,N
      AA(I)=AA(I)+B(I)*B(I)
15    CONTINUE
      DO 16 I=2,N
      X(I)=X(I-1)+Y(I)
16    CONTINUE
```

Figura 22. Cicli distribuiti

4. Utilizzo del concatenamento

   Le istruzioni composte sono molto piu' efficienti per la possibilita' di ottenere due operazioni per ciclo e devono quindi essere utilizzate il piu' spesso possibile.

5. Lo stride e l'ordine dei cicli DO

   E' opportuno ricordare di porre particolare attenzione all'ordine dei cicli DO, poiche' puo' comportare l'esecuzione con una dimensione dello stride molto diverso.

Dopo le eventuali modifiche al codice e' necessario ripetere la compilazione e l'esecuzione del programma previste nella terza fase.

Quinta fase della vettorizzazione: modifiche complesse

Questa fase si rende necessaria se i risultati delle precedenti fasi non sono stati soddisfacenti. A questo punto le modifiche non verteranno piu' su delle modifiche locali, ma sulla struttura dell'applicazione e sugli algoritmi utilizzati. Le indicazioni fornite dal compilatore hanno ora una minore importanza, mentre quelle fornite dall'analizzatore di esecuzione restano essenziali, poiche' segnalano le parti del codice che consumano maggiormente il tempo di CPU e che quindi bisogna rivedere.

Organizzazione dei dati

L'organizzazione dei dati e' forse tale che e' impossibile trarre un vantaggio reale dalle possibilita' della VF. E' un campo da esplorare se si vogliono migliorare le prestazioni e l'indagine va soprattutto rivolta a fattori come la lunghezza dei vettori, lo stride, ecc ...

Analizziamo adesso alcuni punti particolari:

1. Organizzazione dei dati e riduzione dello stride

   Anche se i cicli DO sono vettorizzati, l'organizzazione dei dati puo' non permettere una reale ottimizzazione dell'applicazione. L'utente ha tutto l'interesse ad organizzare i dati in modo da avere i vettori piu' lunghi con uno stride piu' piccolo possibile.

   Supponiamo di avere una applicazione che tratta 10.000 matrici 5X5.

```
      DIMENSION A(5,5,10000)
        ...
      DO 99 K=1,10000
      DO 99 ICOL=1,5
      DO 99 IRIG=1,5
      A(IRIG,ICOL,K)=A(IRIG,ICOL,K)+ ...
 99   CONTINUE
```

Figura 23. Organizzazione inadeguata dei dati

I cicli su ICOL e IRIG sono troppo corti per essere vettorizzati, allora verra' vettorizzato solo il ciclo su K, ma il suo stride e' di 25. Mentre l'organizzazione del ciclo riportato in Figura 24 fornira' uno stride di 1 e dunque delle migliori prestazioni.

```
      DIMENSION A(10000,5,5)
        ...
      DO 99 ICOL=1,5
      DO 99 IRIG=1,5
      DO 99 K=1,10000
      A(K,IRIG,ICOL)=A(K,IRIG,ICOL)+ ...
 99   CONTINUE
```

Figura 24. Migliore organizzazione dei dati

2. Utilizzo di EQUIVALENCE per migliorare la lunghezza dei vettori.

Con l'istruzione di EQUIVALENCE e' possibile sfruttare la ridefinizione di parti di matrici per aggiustare la lunghezza dei vettori e quindi migliorare le prestazioni.

Si consideri il seguente esempio:

```
      DIMENSION A(N,M,L), B(N,M,L)
         ...
      DO 99 K=1,L
      DO 99 J=1,M
      DO 99 I=1,N
      A(I,J,K)=A(I,J,K)+B(I,J,K)
99    CONTINUE
```

Figura 25. Cicli su matrici a 3 dimensioni

Si puo' osservare che, nelle prime due dimensioni delle matrici, le operazioni sono indipendenti e quindi vettorizzabili. Se si utilizza la tecnica di allocazione in memoria effettuata dal Fortran mediante l'istruzione EQUIVALENCE, si ottiene:

```
      DIMENSION A(N,M,L), B(N,M,L), AA(NM,L), BB(NM,L)
      EQUIVALENCE (A(1,1,1),AA(1,1)), (B(1,1,1),BB(1,1))
         ...
      DO 99 K=1,1
      DO 99 IJ=1,NM
      AA(IJ,K)=AA(IJ,K)+BB(IJ,K)
99    CONTINUE
```

Figura 26. Cicli su matrici definiti con EQUIVALENCE

Algoritmi

Gli algoritmi sui quali si basa l'applicazione non sono forse adatti al calcolo vettoriale e si rende quindi necessario rivedere completamente l'applicazione o, almeno, le parti contenenti il calcolo intensivo.
Questo tipo di soluzione e' generalmente molto impegnativo e costoso, per cui deve essere intrapreso solo se la posta in palio lo giustifica.

Multitasking

Da non dimenticare che l'esecuzione in modo *multitask,* usata sugli elaboratori che, avendo piu' processori, offrono tale possibilita', puo' essere una soluzione per ridurre *l'elapsed time.*
Rappresenta quindi un tipo di soluzione che non deve essere scartata a priori, soprattutto se il tipo di applicazione si presta ad essere suddivisa in piu' blocchi logici indipendenti tra loro.

Dopo le eventuali modifiche al codice e' necessario ripetere la compilazione e l'esecuzione del programma previste nella terza fase.

Errori di troncamento e di arrotondamento

E' necessario a questo punto fare alcune considerazioni sugli errori che possono essere indotti con la vettorizzazione. Com'e' noto la rappresentazione di un numero reale in una configurazione binaria di 32 o 64 bit provoca un troncamento delle cifre meno significative. Tale fenomeno, noto nel calcolo numerico, produce anche degli effetti piu' o meno vistosi durante l'esecuzione, poiche' lo svolgimento di un'operazione aritmetica provoca un arrotondamento ad ogni passo.

Cambiare quindi l'ordine degli elementi che partecipano ad una somma od un prodotto determina un risultato finale diverso e tale caratteristica fa si' che il calcolo numerico non sia commutativo.

Modifica di algoritmi

Come si e' visto in precedenza la fase di vettorizzazione di un programma comporta molto spesso delle modifiche all'algoritmo che possono essere operate sia dall'utente che dal compilatore stesso. Queste operazioni, che talvolta possono essere molto semplici, tendono a scrivere l'applicazione sotto una forma piu' favorevole alla vettorizzazione mediante:

- l'aumento della dimensione dei vettori
- la diminuzione dello stride
- la diminuzione dei trasferimenti da e verso la memoria
- l'utilizzo piu' frequente possibile delle operazioni composte
- ecc...

Questi interventi pero' alterano quasi sempre l'ordine con cui gli elementi vengono elaborati e per questo motivo i risultati ottenuti con una elaborazione vettoriale *possono essere diversi* da quelli ricavati in modo scalare.

Riduzione vettoriale

Quando il compilatore VS Fortran versione 2 riconosce nel codice un'operazione di riduzione del tipo:

```
      S=0
      DO 10 I=1,N
      S=S+X(I)
10    CONTINUE
```

la trasforma in una forma piu' vantaggiosa per la vettorizzazione mediante l'inserimento di quattro addizionatori, come e' mostrato in seguito:

```
        S1=0
        S2=0
        S3=0
        S4=0
        DO 10 I=1,N,4
        S1=S1+X(I)
        S2=S2+X(I+1)
        S3=S3+X(I+2)
        S4=S4+X(I+3)
10      CONTINUE
        S=S1+S2+S3+S4
```

Ovviamente anche questa modifica dell'algoritmo operata dal compilatore provoca un cambiamento dei risultati dell'esecuzione vettoriale rispetto a quella scalare. L'utente puo' comunque costringere il compilatore ad usare l'algoritmo scalare mediante l'opzione NORED, da specificare in fase di compilazione.

Underflow

Anche se non strettamente legato al processo di vettorizzazione, e' bene pero' accennare qui alla gestione degli *underflow,* in quanto estremamente significativi per il peso *rilevante* che hanno sui tempi di esecuzione del programma. Poiche' il verificarsi di tale evento provoca un'interruzione del programma e il passaggio del controllo alle routine del Fortran che gestiscono gli errori, una condizione di underflow molto frequente, durante l'esecuzione, puo' aumentare i tempi di calcolo piu' di 100 volte.

Se invece l'utente non e' interessato alla segnalazione di tale tipo di errore, puo' disabilitarlo in due modi:

1. Passando il parametro NOXUFLOW in fase di esecuzione nel campo PARM della EXEC.

   Il parametro ha in questo caso effetto per tutta la durata del programma.

2. Usando la routine XUFLOW della libreria del VS Fortran Versione 2.

   Tale routine, la cui chiamata deve essere inserita all'interno del programma, ammette il passaggio di un parametro che puo' avere il valore di 0 o 1. Si ha quindi:

   - CALL XUFLOW(0)
     In questo caso, tutte le condizioni di underflow che si verificano dopo la chiamata alla routine XUFLOW, non provocano piu' interruzioni di programma e non causano la segnalazione di un messaggio di errore.

   - CALL XUFLOW(1)
     Con il parametro uguale a 1 si ha invece un'interruzione del programma e la scrittura di un messaggio di errore tutte le volte che si verifica un underflow.