Consiglio Nazionale delle Ricerche

Ottimizzazione e Errori

R. Medves

198
CNUCE

A cura di : Riccardo Medves

Copyright - Luglio 1985
by - CNUCE - Pisa
Istituto del Consiglio Nazionale delle Ricerche

Ottimizzazione e Errori

R.Medves

ZC-198
CNUCE

ARTE della programmazione

ovvero

programmare COSCIENTEMENTE

Le PULCI nell'orecchio

ovvero

avvicinarsi con UMILTA'
alla programmazione

INTRODUZIONE

Perche' i sottotitoli della pagina precedente?

ARTE della programmazione: programmare e' facile, programmare bene e' invece un'arte che richiede, oltre una perfetta conoscenza del linguaggio utilizzato, anche una perfetta conoscenza degli algoritmi da utilizzare;

programmare COSCIENTEMENTE: la programmazione non va affrontata con leggerezza e superficialita', se si desidera che i risultati ottenuti siano attendibili;

le PULCI nell'orecchio: questo manuale non e' un trattato completo sull'ottimizzazione; da un lato esistono numerosi libri teorico-pratici sull'argomento, dall'altro ogni calcolatore e ogni prodotto software ha le sue regole ben precise e pretendere di descriverle tutte nei dettagli e' al di fuori della portata delle mie conoscenze; e' utile pero' che siano chiariti e messi in luce alcuni aspetti fondamentali relativi alla ottimizzazione dei programmi e ad una buona programmazione; una raccolta di suggerimenti pertanto, di avvertenze e di puntualizzazioni;

avvicinarsi con UMILTA' alla programmazione: si incontrano spesso persone che credono di sapere tutto sulla programmazione e che spesso invece compiono errori rilevanti che li portano ad ottenere risultati completamente errati; attenzione quindi a cercare di evidenziare e studiare con cura le possibili fonti di errore, senza paura di essere estremamente prudenti.

Argomento centrale sono dunque le regole di ottimizzazione e gli accorgimenti da adottare e le iniziative _da non seguire_ per poter sfruttare nella dovuta maniera il processo di ottimizzazione automatica degli attuali compilatori ottimizzanti.

Il manuale fornisce regole note ed altre meno note per dare un'idea di quali siano i problemi legati alla ottimizzazione dei programmi, troppo spesso ignorati e causa pertanto di inutili appesantimenti nelle elaborazioni.

La conoscenza di tali processi e oggi ancor piu' indispensabile con l'avvento del calcolo vettoriale, dove e' necessario saper sfruttare al massimo la potenza degli elaboratori per poter essere in grado di ottimizzare tempo e risorse in processi che si ripercuotono anche sul piano economico.

Il manuale contiene anche alcune considerazioni sulla precisione dei calcoli e sugli errori che si possono involontariamente ottenere durante le elaborazioni, e mette in guardia verso una passiva accettazione non critica dei risultati ottenuti.

Il manuale vuole quindi essere una veloce carrellata su quanto possa essere pericoloso:

SOTTOVALUTARE la conoscenza delle regole di ottimizzazione e la teoria degli errori di calcolo, ovvero

SOPRAVALUTARE le proprie capacita' di scrittura dei programmi senza delle adeguate basi di conoscenza di hardware e software.

Per quanto riguarda i problemi di ottimizzazione mi sono essenzialmente basato sulle seguenti pubblicazioni:

- Fortran optimization - Michael Metcalf
  Academic Press - London (1982)
- Writing optimizable Fortran - R.G.Scarborough
  IBM Scientific Center - Palo Alto California

e per la parte vettoriale, oltre allo stesso ottimo libro di Metcalf, anche del volume:

- Efficient Fortran Techniques for Vector Processing
  Pacific Sierra Research Corporation - (1981)

Per quanto riguarda invece i problemi relativi agli errori e alla precisione, mi sono basato sul primo capitolo del libro:

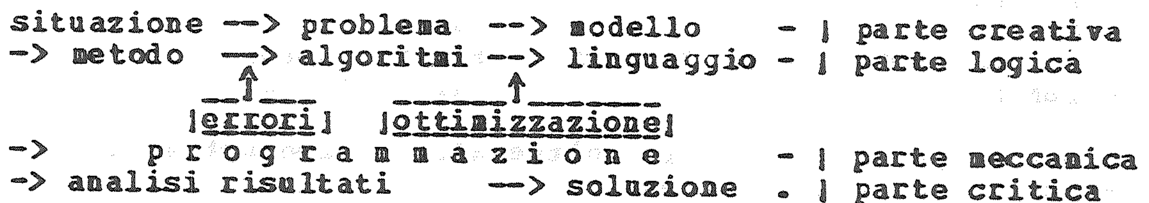
- Introduzione al Calcolo Numerico - R.Bevilacqua,O.Menchi
  ETS - Pisa (1980)

Il presente manuale si basa su una serie di appunti e di 'lucidi' da proiettare utilizzati per un seminario interno al CNUCE e ottenuti rielaborando il materiale contenuto nelle pubblicazioni precedenti assieme ad altro materiale reperito qua e la' su altre riviste e pubblicazioni: per quanto riguarda l'aspetto piu' propriamente tecnico per la discussione dei problemi di ottimizzazione assume come base i compilatori IBM Fortran H-esteso e VS-Fortran

DIDATTICA NELLA SCUOLA

E prima di passare nel vivo dell'argomento, ritengo utile spendere due parole sull'utilizzo del calcolatore nella scuola, argomento che potrebbe sembrare del tutto fuori tema, ma che invece ritengo strettamente legato al contenuto di questo manuale, in quanto e' proprio nella scuola che il calcolatore dovrebbe essere presentato come una 'macchina', con tutti i suoi pregi e difetti: non il toccasana a tutti i problemi, ma uno strumento reale, con grandi potenzialita', ma anche limiti che bisogna imparare a conoscere bene per poterlo sfruttare nella sua interezza e per evitare di incorrere in quei subdoli errori di cui abbiamo brevemente accennato in precedenza.

In genere, quando si presenta un calcolatore, si pensa subito alla programmazione: quello che bisognerebbe cominciare a insegnare invece e' che la programmazione non e' che una piccolissima parte di tutto quel processo che ci porta da un problema alla sua soluzione. Cerchiamo in qualche modo di schematizzare i passi di questo processo:



Si parte da una certa situazione reale, dalla quale si viene a creare il problema da risolvere; e qui e' in gioco la parte creativa del processo risolutivo: dal problema ci si deve creare un modello, un qualcosa cioe' che rispecchia e schematizza il problema stesso, per poi passare alla parte logica, cioe' alla scelta ragionata di un metodo di risoluzione, alla conseguente scelta dell'algoritmo piu' opportuno e del linguaggio di programmazione piu' adatto.
E' qui che entrano in gioco e sono stati per questo evidenziati due aspetti della massima importanza: il controllo degli ERRORI e la conoscenza delle regole di OTTIMIZZAZIONE; ed e' proprio in questa fase pertanto che si deve porre la maggiore attenzione.
A questo punto si abbandonano temporaneamente le fasi intelligenti per passare ad una fase puramente meccanica: quella appunto della programmazione, che era inizialmente al centro dell'attenzione.
Ma vi renderete ora conto che questa fase e' veramente puramente meccanica e che le fasi fondamentali da temere e quindi da conoscere a menadito sono le altre.
La fase successiva diviene di nuovo intelligente: e' la parte critica dell'analisi dei risultati, della determinazione della loro correttezza e affidabilita': un buon programmatore sa anche come sia importante sapersi

creare dei modelli semplificati del problema che conducano a dei risultati noti o comunque facilmente verificabili, in modo da poter stimare la correttezza di tutto il procedimento adottato. L'analisi dei risultati si riallaccia quindi in maniera molto stretta alla creazione del modello, alla scelta del metodo e degli algoritmi, a quelle fasi 'intelligenti' che costituivano i primi passi fondamentali del procedimento.

E ritorniamo ora per un momento alla didattica nelle scuole:

- e' negativa, e' positiva?
- provoca una diminuzione delle capacita' di memorizzazione o una maggiore disciplina nell'attivita' mentale?
- si riduce ad una supina accettazione dei risultati o sviluppa una maggiore rapidita' di riflessi nell'apprendimento di nuove situazioni?
- certo e' negativa se si limita all'utilizzo di algoritmi pre-preparati acquistati a scatola chiusa dalle case fornitrici, ma sarebbe estremamente positiva se ponesse l'accento su elementi concreti dei calcolatori, quali ad esempio le dimensioni limitate delle 'voci' di memoria e quindi l'esistenza di errori indotti nei calcoli della macchina (troncamento, arrotondamento, underflow, overflow, ecc.)
- e' negativa se l'attenzione dello studente si concentra troppo su elementi esteriori, quali ad esempio i diversi aspetti formali dei linguaggi, l'aspetto grafico fine a se stesso, la casistica degli errori rilevati dai compilatori, ecc.; e' positiva invece se la spinta avviene verso la sollecitazione al controllo dei risultati, alla ricerca di errori non formali e quindi ad un conseguente esame critico di metodi e algoritmi.

Non tentiamo certo qui di dare una risposta: cerchiamo invece con questo manuale di portare un contributo positivo al problema, ponendo l'accento sulla necessita' di farsi delle buone conoscenze delle macchine hardware e del relativo software, in modo da programmare in maniera cosciente e ottenere i risultati voluti.

CORSI E RICORSI
OVVERO
UN PASSO INDIETRO?

"Ieri" i calcolatori presentavano delle caratteristiche HW-SW abbastanza primitive; risalendo, senza andare troppo lontano, ai primi tempi del CNUCE, si ricordera' certo come un grosso elaboratore tipo la 7090 avesse una memoria centrale veramente ridotta (dell'ordine dei 128KB odierni) rispetto agli ordini di grandezza a cui i recenti grandi elaboratori ci hanno abituato (16MB e addirittura 2GB con l'attuale architettura estesa, in termini tecnici XA).

Cio' costringeva i programmatori di allora ad essere in buona parte anche "sistemisti", cioe' a conoscere nei dettagli la struttura della macchina e le sue caratteristiche tecniche, in modo da poterle sfruttare al meglio per ottenere gli scopi desiderati dai programmi.

Cosi' i programmatori erano abituati (o costretti) a programmare in maniera accorta rosicchiando byte su byte della memoria in modo che questa potesse contenere il programma da eseguire e ricorrendo in molti casi a tecniche di overlay per programmi di grosse dimensioni.

Analogamente nel campo SW i compilatori erano relativamente primitivi e non offrivano tutte le "facilities" di quelli attuali (ottimizzazione di tempo, di memoria, ecc.) costringendo ancor piu' i programmatori a ricoprire il ruolo di sistemisti per poter sopperire manualmente alle carenze tecniche: ed ecco quindi che si tramandavano l'un l'altro regole di ottimizzazione da applicare ai programmi per una esecuzione piu' rapida: evitare le divisioni...... limitare i loop.... attenzione alle chiamate di subroutine, ecc. ecc........

Poi, col passar del tempo e l'affinarsi della tecnica, il programmatore si e' trovato al centro di una vera e propria rivoluzione, espressamente diretta ad alleviare le sue fatiche: Memorie virtuali! finalmente non piu' vincoli da rispettare! Compilatori ottimizzanti! finalmente liberi di pensare al nocciolo del problema e non al modo di scrivere le istruzioni!

Il programmatore si sente finalmente spinto verso il ruolo di analista: la macchina e' una scatola nera in cui si immettono delle richieste e da cui si ottengono le risposte cercate, senza doversi impegnare troppo pesantemente in lavori di contorno inutili sul come scrivere il programma o sugli accorgimenti da adottare a causa delle restrizioni imposte dalla macchina stessa......

Ma e' proprio cosi'?

"Oggi" e' vero che i moderni elaboratori presentano tali e tanti accorgimenti tecnici HW (paginazione, parallelismo

nel trasferimento dei dati tra CPU e memoria, buffer intermedi per velocizzare l'esecuzione delle operazioni di I/O, ecc.) e SW (compilatori altamente ottimizzanti, linguaggi strutturati, ecc.) per velocizzare l'esecuzione ed aiutare il programmatore nel suo lavoro, togliendo molti dei primitivi colli di bottiglia, ma e' pur vero che il lavoro del sistema operativo e' enormemente aumentato, che il livello di multiprogrammazione e' estremamente cresciuto, che i costi delle elaborazioni sono lievitati e che pertanto, per non incorrere in catastrofici appesantimenti dell'esecuzione dei propri programmi, che si ripercuotono anche sul piano economico, non basta affidarsi ciecamente alla macchina.

Tutte le caratteristiche di cui abbiamo parlato (paginazione, parallelismo, buffer, compilatori ecc.) hanno delle ben precise particolarita': la paginazione avviene a blocchi di 4KB e il riferimento a dati che stanno in blocchi diversi comporta un notevole lavoro di I/O che si ritrova poi sotto forma di spesa per la memoria utilizzata o per numero di pagine trasferite;

il parallelismo e' associato ad una suddivisione logica della memoria in gruppi di byte e il non adeguato dimensionamento di variabili in un programma puo' portare uno sparpagliamento dei dati in memoria, tale da far lievitare il costo della esecuzione del programma in seguito ad un inefficiente metodo di trasferimento di tali dati tra memoria e CPU;

il buffer consente e' vero di velocizzare operazioni di ricerca dei dati, tramite un precaricamento automatico degli stessi, ma anche qui una cattiva gestione, come ad esempio operazioni di IF con salti non predeterminabili dalla macchina o il riferimento a matrici di rilevanti dimensioni possono causare aumenti notevoli nel tempo di elaborazione;

bisogna inoltre tenere anche ben conto dei supporti ausiliari, quali le unita' disco, e conoscerne a fondo le caratteristiche (numero dei byte per traccia, formato dei records e delle tracce, ecc.) per ottimizzare in maniera adeguata i fattori di bloccaggio ed evitare costosi sprechi di spazio;

analogamente in campo SW l'utilizzo di compilatori ottimizzanti riduce teoricamente la fatica del programmatore, ma nella pratica accade che tali compilatori "fanno troppo", cioe' in molti casi prendono iniziative tali - in base a come e' scritto il programma - da causare piu' danni che benefici (sicuramente c'e' chi ha sperimentato questo sulla propria pelle utilizzando i livelli piu' alti di ottimizzazione permessi, tipo OPT=3): invece di tramandare regole su come ottimizzare un programma occorre ora tramandarsi regole su come scrivere le istruzioni perche' il compilatore le ottimizzi correttamente.

Il risultato di tutto cio' e' uno solo: il programmatore si e' solo virtualmente liberato dell'abito del sistemista, perche' in realta' e' ancora circondato da regole, limitazioni e norme da conoscere, rispettare e seguire.

E ancor piu' grave e' che, al contrario di ieri, oggi questi limiti e queste regole sono piu' numerose, piu' sottili e piu' nascoste: inoltre, con lo sviluppo accelerato della tecnica, cambiano piu' frequentemente che in passato, costringendo il programmatore ad un continuo adeguamento ad ogni cambio di sistema o di macchina (cambiano le dimensioni dei buffer, cambiano i byte trasferiti nel parallelismo, cambiano le caratteristiche dei compilatori e di conseguenza le relative regole di ottimizzazione); infine, cosa ancora piu' grave, mentre "ieri" il non seguire certe regole aveva come sola conseguenza un appesantimento delle elaborazioni, "oggi" puo' anche condurre a risultati errati delle stesse.

E che si puo' dire poi dei personal?

Salutati come il toccasana per tutti, come la bacchetta magica con cui ottenere in casa propria a poco prezzo la soluzione dei propri problemi elaborativi non sono altro, in realta', che copie in scala ridotta degli elaboratori di "ieri", relativamente lenti, con poca memoria e scarso spazio disco.

Non fraintendetemi: rapportati ai costi sono evidentemente un balzo in avanti pauroso, ma dal punto di vista della programmazione costringono il programmatore a quei salti mortali ai quali non era piu' abituato.

Scendiamo ancora un gradino piu' in basso nella scala dei valori: provate a programmare seriamente un home computer tipo Commodore64, Sinclair e simili: roba veramente da far drizzare i capelli con tutte quelle PEEK e POKE con riferimento ad indirizzi di memoria che solo vostro figlio riesce a ricordare perfettamente (alla barba dell'universalita' e della trasportabilita' del SW), o con quelle istruzioni DATA seguite da interminabili righe di numeri (separati da virgole per fortuna!) nei quali solo l'amico degli amici di vostro figlio riesce ad individuare quel 2 sbagliato al posto di un 3 nella 20ma riga dal basso.....

Eppure mentre a noi programmatori di "ieri" (per non dire vecchi) i listati di questi programmi fanno venire i brividi, i programmatori di domani (giovani) sembrano trovarsi perfettamente a loro agio; e la differenza di valutazione consiste essenzialmente nel diverso lato di approccio: chi e' abituato ad usare macchine piu' potenti e flessibili nota tutte le limitazioni e le ristrettezze, chi parte dal basso invece apprezza di trovarsi tra le mani uno strumento docile e maneggevole, che dietro precisi ordini esegue le funzioni piu' meravigliose: dai calcoli alla scrittura, dalla grafica al suono.....

Dai nani ai giganti.

Cose analoghe stanno avvenendo con i cosiddetti supercomputer, calcolatori con potenze superiori dell'ordine delle decine di volte rispetto agli attuali calcolatori general purpose e che raggiungono queste elevate capacita' elaborative con l'uso di tecniche "vettoriali" e/o "parallele".

Non tutti sanno pero' che tali elaboratori richiedono, per poter raggiungere le potenze dichiarate, di una scrittura dei programmi e spesso di una impostazione dei problemi molto particolare.

La conoscenza delle macchine (HW + SW) in questi casi e' ancora piu' stringente e necessaria, dato il rilevante costo delle elaborazioni. E non e' quindi certo sufficiente "metter dentro" i programmi cosi' come sono, lasciando alle macchine e ai compilatori l'opera di ottimizzazione: in questi casi si rischia veramente di veder calare le potenze teoriche a valori paragonabili, se non addirittura piu' bassi, di un analogo elaboratore scalare general purpose.

Qui l'opera di scrittura del programma e impostazione del problema diviene davvero rilevante; qui ciascuna macchina ha le sue caratteristiche e di conseguenza le sue "esigenze" di voler presentati i programmi in un certo modo; qui la semplice inversione di due variabili o di due costanti puo' arrivare a dare risultati catastrofici.

Per lavorare su queste macchine occorre essere preparati; preparati e aperti a cambiare persino la tradizionale impostazione dei problemi, e quindi a cambiare i propri schemi logici di pensiero, per ottenere i risultati voluti delle elaborazioni.

Ma e' veramente questa la strada del futuro? Il programmatore che nell'evolversi della tecnica, nel calo dei prezzi dell'HW, nell'aumento delle prestazioni e delle dimensioni delle memorie dei calcolatori, nella nascita di sistemi operativi e compilatori sempre piu' sofisticati, aveva intravisto la concreta possibilita' di veder alleviate le sue fatiche nel campo della programmazione, si vede invece ora sopraffatto e schiacciato da questa tecnica che lo costringe ad entrare approfonditamente fino nei piu' reconditi recessi della macchina, perche' il suo problema venga correttamente interpretato e risolto.

O non si dovra' invece andare in una direzione in cui la macchina stessa si adatti al problema, in cui noti i termini di questo, la tecnica sia in grado di fornire un prodotto capace di afferrarne i concetti e di impostare un'architettura capace di risolverlo nella maniera piu' ottimizzata possibile. Fantascienza? Chi sa.......

COMPILATORI E REGOLE DI OTTIMIZZAZIONE

Abbiamo quindi visto come per i compilatori primitivi di un tempo fosse fondamentale un processo di ottimizzazione manuale dei programmi, secondo regole pratiche che si tramandavano oralmente... di padre in figlio.

Ma per i compilatori ottimizzanti di oggi e' ugualmente produttiva una pre-ottimizzazione manuale dei programmi?

IN GENERE NO!

Spesso il lavoro di ottimizzazione manuale e' inutilmente pesante rispetto a quello che il compilatore puo' fare automaticamente e con maggiore efficienza.

E INOLTRE

spesso i provvedimenti adottati manualmente sono ADDIRITTURA CONTROPRODUCENTI ai fini dell'ottimizzazione automatica da parte del compilatore

QUINDI

per tali compilatori, piu' che cercare di ottimizzare il codice a mano, si dovrebbe conoscere a fondo come strutturare determinate parti di codice per facilitare il lavoro del compilatore e quello che invece si deve evitare di fare per non incorrere nell'errore di inibire il processo automatico di ottimizzazione.

Attenzione quindi: rileggete accuratamente quanto detto qui sopra, perche' questo e' il nocciolo fondamentale della questione, se si vogliono sfruttare a pieno le capacita' dei moderni compilatori e non incorrere invece in rilevanti appesantimenti dell'esecuzione.

Quando e dove ottimizzare:
spesso si riporta una semplice regola d'oro, detta del 10-90, che piu' o meno suona cosi':
"Nel 10% del codice si consuma il 90% del tempo".

Cosa significa questo?
Significa che non e' assolutamente necessario ottimizzare un programma a tappeto in tutte le sue parti: e' invece molto piu' produttivo individuare le parti che pesano di piu' e agire solo su di esse per ottenere dei miglioramenti rilevanti.
Per inciso, a spanne, queste parti includono in genere i loop (o cicli che dir si voglia) e le operazioni di I/O.

Per quanto riguarda poi in maniera specifica le regole di ottimizzazione, dividiamole per la trattazione nei seguenti gruppi: regole scontate, regole meno note, loop, regole nascoste, regole di nuova concezione (elaborazioni vettoriali).

REGOLE SCONTATE

Si tratta in genere delle regole "manuali" di cui abbiamo parlato all'inizio; come pero' abbiamo detto si tratta di quelle regole che andavano applicate al tempo dei compilatori NON ottimizzanti, mentre sono in genere proprio <u>da evitare</u> con gli attuali compilatori ottimizzanti.
E vediamo di elencarne alcune tra le piu' comuni:

- rimpiazzare divisioni con moltiplicazioni:

```
B = A/2     -->     B = A*0.5
```

- rimpiazzare moltiplicazioni con somme:

```
B = 2*A     -->     B = A+A
```

- rimpiazzare potenze con moltiplicazioni:

```
B = 2**5    -->     B = A*A*A*A*A
```

- IF con condizioni multiple (... .AND. ... .OR. ...): mettere prima le condizioni che si verificano con maggiore probabilita' (molti compilatori non eseguono in tal caso il codice successivo);

  ATTENZIONE: in alcuni casi questo puo' addirittura causare errori: IF (A.EQ.0 .OR. 1/A.LT.5) ... nel caso A=0 si ha o meno errore a seconda che la seconda condizione venga ugualmente calcolata o no;

- ecc.

REGOLE MENO NOTE

Entriamo invece con questo gruppo nelle norme da seguire per facilitare o addirittura permettere il processo di ottimizzazione del compilatore: non si tratta cioe' in genere di provvedimenti del tutto evidenti, ma che tendono a strutturare il codice di programma in modo tale che il compilatore riconosca facilmente le parti da ottimizzare e operi in maniera opportuna su di esse.
E ricordiamoci le avvertenze di alcune pagine prima: con i compilatori di oggi e' importante e molto piu' produttivo aiutare il compilatore a capire <u>dove</u> poter ottimizzare, piuttosto che ottimizzare direttamente il codice a mano:

- evitare le conversioni dovute al "modo misto" dei dati (sia costanti che variabili):

```
A = I                   -->   A = FLOAT(I)
```

- seguire un determinato ordine dei dati all'interno della istruzione: prima le costanti, poi gli invarianti (cioe' le variabili che non cambiano) e poi le parti variabili:

```
A = 2.0*PI*X(I)

A(I) = (B(I)-C(I))/S    -->   A = (1./S)*(B(I)-C(I))
```

- eseguire operazioni multiple direttamente senza risultati intermedi, in quanto questi causano accessi ulteriori alla memoria rallentando in maniera notevole l'esecuzione:

```
A = B+C+D+E             non   T1 = B+C  :  T2 = D+E
                              A = T1+T2
```

- cercare di raggruppare parti comuni di codice facendo uso di parentesi, e inserirle all'inizio delle istruzioni:

```
A = B*C+2.0+C*B         -->   A = (B*C)+2.0+(B*C)

A = B-C                  |    A = (B-C)
....                  -->|    ....
D = C-B                  |    D = -(B-C)

A = B*C/D                |    A = (B/D)*C
....                  -->|    ....
E = B*F/D                |    E = (B/D)*F
```

ma ATTENZIONE alle label delle istruzioni:

```
   A = (B/D)*C
10 E = (B/D)*F
```

in tal caso la possibilita' di saltare all'istruzione 10 da altra parte del programma inibisce l'ottimizzazione della sequenza (B/D);

- nelle operazioni di I/O cercare di utilizzare alti fattori di bloccaggio;

- nelle operazioni di I/O su vettori utilizzare la forma compressa WRITE (...) A piuttosto che il DO implicito WRITE (...) (A(I,J),I=...);

- attenzione alle interferenze tra i dati: il compilatore deve essere sempre in grado di sapere se i dati che compaiono in due istruzioni distinte sono o no la stessa quantita': attenzione quindi ad altre istruzioni intermedie che possono mascherare questo fatto:

```
      B = A(2)
      A(I) = 1.
      C = A(2)
```

in questo caso nella prima istruzione si ha un accesso alla memoria per il prelievo della quantita' A(2); dato che nella terza istruzione si fa di nuovo riferimento ad A(2), il compilatore potrebbe evitare un nuovo accesso alla memoria, ma la presenza dell'istruzione intermedia crea un'interferenza: in questo caso il compilatore NON SA se l'istruzione intermedia cambiera' in fase di esecuzione il valore di A(2) e quindi nella istruzione successiva e' costretto a fare un nuovo accesso alla memoria.

```
      K = N+1
      DO 1 I = 2,N
      A(I) = A(K)/A(1)
    1 CONTINUE
```

dato che K e' sempre maggiore di N, la quantita' A(K)/A(1) e' invariante e potrebbe ad esempio utilmente essere estratta da loop: ma il compilatore non puo' riconoscere la relazione iniziale tra K ed N e pertanto non puo' effettuare l'ottimizzazione del codice.

LOOP

Siccome i loop costituiscono, insieme all'I/O, le parti fondamentali di un programma nelle quali viene spesa una parte preponderante del tempo (si ricordi la regola del 10-90 data all'inizio) la descrizione relativa e' stata evidenziata con uno spazio a se', anche se negli altri paragrafi del manuale compaiono a volte esempi che coinvolgono i loop:

- evitare IF, CALL, FUNCTION e GOTO all'interno di un loop particolarmente pesante: eventualmente sostituire la chiamata ad una subroutine col codice esplicito in linea, oppure cercare di scambiare tra di loro CALL e loop;

- sostituire loop corti con espansione esplicita in linea:

```
      DO 1 I = 1,2            |
      A(I) = 0.               |-->      A(1) = 0.   :  A(2) = 0.
    1 CONTINUE                |
```

- estrarre le parti invarianti:

```
      SUM = 0.                |         SUM = 0.
      DO 1 I = 1,N            |         DO 1 I = 1,N
      SUM = SUM+B*A(I)        |-->      SUM = SUM+A(I)
    1 CONTINUE                |       1 CONTINUE
                              |         SUM = SUM*B
```

ma attenzione:

```
      DO 1 J = 1,10           |         DO 1 J = 1,10
      DO 1 I = 1,10           |         T = C(J)
      A(I,J) = C(J)*B(I,J)    |-->      DO 1 I = 1,10
    1 CONTINUE                |         A(I,J) = T*B(I,J)
                              |       1 CONTINUE
```

C(J) e' invariante rispetto al loop interno: il compilatore lo capisce e usera' un registro per mantenere il valore di C(J); il tentativo manuale di ottimizzazione (uso dello scalare T invece del vettore C(J) all'interno del loop) comporta invece una superflua operazione di riferimento ad una cella di memoria;

- e ora fate bene attenzione: spesso (molto piu' spesso di quanto non si immagini) il compilatore fa piu' del necessario!
  In tal caso puo' essere necessario usare un valore di ottimizzazione (OPT=...) minore:

```
      DO 1 I = 1,N
      DO 2 J = 1,M
      IF (...) GOTO 2
      A(J,I) = SQRT(FLOAT(I))+B(J,I)
    2 CONTINUE
    1 CONTINUE
```

Il compilatore riconosce SQRT come invariante rispetto al loop interno e lo tira fuori, senza tener conto che l'IF potrebbe limitare il numero di chiamate;
possibile soluzione: usare un valore di ottimizzazione inferiore o scrivere una function esplicita che si sostituisca alla SQRT, tipo:

```
                                  | FUNCTION FUNC(I,B)
     A(J,I)=FUNC(I,B(J,I))   con  | FUNC = SQRT(...)
                                  | END
```

Altro esempio:

```
     DO 1 I = 1,N
     B(I) = 0.
     IF (A .GT. 0.) B(I)=C(I)/A
   1 CONTINUE
```

il compilatore riconosce 1/A come invariante e lo tira fuori, togliendolo pero' cosi' dal controllo dell'IF: come si intuisce, si puo' in questo modo avere una condizione di errore per overflow del tutto inaspettata e a prima vista incomprensibile;
possibile soluzione: estrarre manualmente 1/A e IF dal loop;

- nel caso di piu' loop uno interno all'altro (nested loops) si abbia l'avvertenza di inserire quali loop interni quelli piu' grossi:

  1) i compilatori tentano infatti in genere di ottimizzare maggiormente i loop interni: quindi questi devono anche essere i piu' complessi come tipo di istruzioni;

  2) inoltre il numero di inizializzazioni e test varia in misura notevole:

```
                              INIZ. | INIZ.
                              ------|-----
DO 1 I = 1,100                    1 | 1              DO 1 I = 1,2
DO 2 J = 1,10                   100 | 2              DO 2 J = 1,10
DO 3 K = 1,2         10*100=   1000 | 20   =2*10     DO 3 K = 1,100
                              ------|-----
                               1001 | 23

                               TEST | TEST
                              ------|-----
3 CONTINUE       2*10*100=     2000 | 2000 =2*10*100  3 CONTINUE
2 CONTINUE         10*100=     1000 | 20   =2*10      2 CONTINUE
1 CONTINUE                      100 | 2               1 CONTINUE
                              ------|-----
                               3100 | 2022
```

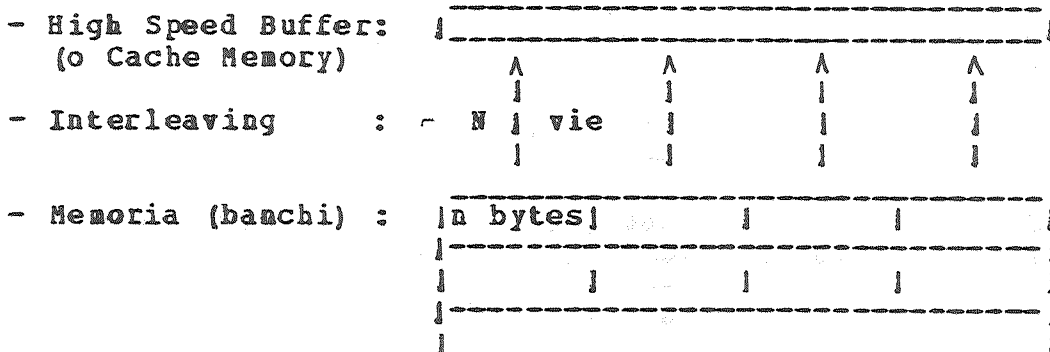
REGOLE NASCOSTE

Si tratta di regole non proprie del compilatore, ma che derivano in genere dalla struttura hardware e dall'architettura della macchina; nel capitolo "Corsi e ricorsi" abbiamo gia' accennato all'esistenza di questi problemi.

Naturalmente, dipendendo in stretta misura dalla macchina piu' che dal compilatore, questa sarebbe una parte da tenere costantemente sotto controllo al cambiare dell'elaboratore su cui vengono eseguiti i programmi: ho detto volutamente "sarebbe", perche' non e' sempre facile da un lato essere costantemente aggiornati su tutte le nuove caratteristiche di una macchina, da un altro rispettare in ogni caso i mille vincoli e le mille limitazioni che tali caratteristiche impongono.

- Paginazione: nei sistemi operativi VM e MVS la paginazione avviene con pagine di 4KB; pertanto si deve tener conto che programmi o matrici che superano tale dimensione possono dar luogo a paginazione, e questo in maniera tanto maggiore quanto piu' numerosi sono riferimenti a parti distanti di codice;

- Indirizzamento: per programmi di dimensioni > 8192 bytes viene usato un registro ulteriore che non partecipa al processo di ottimizzazione.
  Analogamente COMMON > 4096 bytes richiedono piu' registri per essere indirizzati: cio' suggerisce di mettere prima variabili corte, poi vettori;

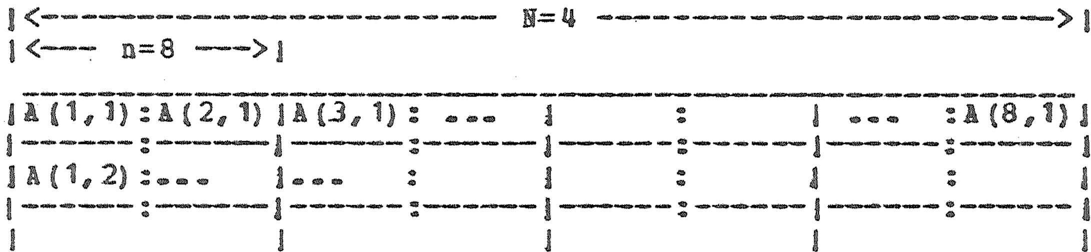


La memoria e' suddivisa logicamente in banchi, con accessi paralleli per ogni banco: supponiamo di avere N banchi con n bytes prelevati in parallelo da ciascuno di essi:

Alcune macchine	H.S.B	N	n
IBM 168	16K	4	8
IBM 3033 N	16K	4	8
IBM 3033 U	32K	8	8
IBM 3081	32/64K	8/16	8

Si deve tener presente che voci consecutive sono mantenute in banchi consecutivi, piuttosto che in parole consecutive dello stesso banco: pertanto ATTENZIONE nel referenziare i vettori:

```
   DIMENSION A(8,100)
   DO 1 I = 1,100
   A(1,I) = FLOAT(I)
 1 CONTINUE
```



Nell'esempio precedente nascono due problemi:
il primo e' che le istruzioni nel loop fanno sempre riferimento allo stesso banco, non sfruttando quindi il parallelismo e sequenzializzando tutti i riferimenti alla memoria con conseguente rallentamento di tutto il processo;
il secondo e' uno spreco ulteriore che deriva dal fatto che la macchina opera su una doppia voce, trasferendo quindi due variabili tra memoria e CPU: delle due pero' quella di destra non viene mai usata.

Una possibile soluzione, seppur strana, e' quella di dimensionare diversamente la variabile: DIMENSION A(9,100)

si spreca cosi' memoria, ma il disallineamento delle singole voci che ne consegue permette di guadagnare tempo.

Ed ancora:
nel dimensionare le variabili si cerchi di far si' che la prima dimensione sia la maggiore: DIMENSION A(100,10)

e che il primo indice vari piu' rapidamente del secondo in caso di loop (confrontare con: nested loops):

```
      DO 1 I = 1,10
      DO 1 J = 1,100
      A(J,I) = ...
    1 CONTINUE
```

REGOLE DI NUOVA CONCEZIONE
OVVERO
ELABORAZIONI VETTORIALI

E' forse improprio parlare di "ottimizzazione" in questo caso: si tratta infatti piu' propriamente di "vettorizzazione" di espressioni, cioe' di far si' che il compilatore sia in grado di trattare tutti gli elementi del vettore in una sola volta.

Si possono distinguere tre diversi metodi di vettorizzazione:

- vettorizzazione del codice sorgente: il programma viene cioe' strutturato in maniera opportuna (analogamente al caso della ottimizzazione trattato nei precedenti capitoli) in modo da facilitare il compilatore nella sua opera; naturalmente e' questo solo un primo livello di vettorizzazione che non portera' il massimo dei benefici, ma che consentira' comunque di mantenere il programma in una forma compatibile anche con un qualsiasi compilatore non vettoriale e quindi garantendo la massima trasportabilita' del software;

- uso di istruzioni e funzioni "vettoriali": ricorso cioe' a particolari istruzioni Fortran che costituiscono una estensione rispetto ad un normale compilatore scalare e sono invece proprie di un particolare compilatore vettoriale; si guadagna naturalmente in velocita', anche se si perde in trasportabilita';

- uso di parti di codice in linguaggio macchina vettoriale: e' la forma piu' elevata di ottimizzazione, ma come si puo' ben capire anche la piu' deleteria per la trasportabilita'; inoltre l'uso di un linguaggio assembler non facilita certo l'effettuazione di possibili modifiche e aggiornamenti al programma: questa soluzione e' pertanto da adottare solo per programmi ormai stabili e che richiedono veramente una velocita' notevole.

Un programma su cui siano applicate le regole di vettorizzazione precedenti e' fino a 20 volte piu' veloce dello stesso programma non "trattato"; nonostante questo da stime fatte risulta che in tal modo si riesce a vettorizzare al massimo un 50% del codice: per ottenere di piu' occorre una accurata revisione degli algoritmi utilizzati; in ogni caso anche per la vettorizzazione vale ancora la regola del 10-90 come per l'ottimizzazione scalare dei programmi.

Ma passiamo ora brevemente all'esame di alcune regole di vettorizzazione.
Si tratta solo di pochi esempi; e' infatti impensabile di fare qui una casistica completa sulle regole di vettorizzazione, tenendo anche conto che ciascun elaboratore vettoriale ha le sue ben precise regole, e quello che risulta ben codificato e ottimizzato per l'uno, non e' detto lo sia anche per l'altro: anche in questo caso la struttura hardware delle macchine gioca un ruolo fondamentale e i libri delle case produttrici o di altre ditte indipendenti

che operano in tale campo e che danno per tutti i casi regole dettagliate specifiche sono veramente di dimensioni rilevanti.
Qui mi basta "mettere una pulce nell'orecchio": predisporre un programma per una elaborazione vettoriale non e' banale e richiede una appropriata conoscenza del compilatore e della macchina su cui si va ad operare, pena un degrado notevole delle prestazioni.

- Evitare IF, CALL, function (fa in genere eccezione la funzione SQRT che viene riconosciuta e trattata in modo particolare), GOTO dentro un loop:

```
   DO 1 I = 1,100                   |   DO 1 I = 1,100
   IF (A(I).LT.0) A(I)=0            |   A(I) = AMAX1(A(I),0.)
   B(I) = SQRT(A(I))+1.             |   B(I) = SQRT(A(I))+1.
 1 CONTINUE                         | 1 CONTINUE
```

- Evitare una interdipendenza tra le variabili:

```
   J1 = J-1                         |
   DO 1 I = 2,100                   |   DO 1 I = 2,100
   A(I,J) = A(I-1,J1)               |   A(I,J) = A(I-1,J-1)
 1 CONTINUE                         | 1 CONTINUE
```

nel primo caso il compilatore non puo' sapere se J1 sara' mai uguale a J e quindi non puo' vettorizzare il loop.

- Variabili "ricorsive": due parti di un programma all'apparenza simili (anche se non hanno lo stesso scopo):

```
   DO 1 I = 2,100                   |   DO 1 I = 2,100
   X(I) = X(I-1)+1                  |   X(I) = X(I+1)+1
 1 CONTINUE                         | 1 CONTINUE
```

MA: la prima <u>NON</u> viene vettorizzata in quanto il valore di X(I) dipende dal valore di X(I-1) che non e' noto a priori (viene calcolato nella iterazione precedente); la seconda invece <u>VIENE</u> vettorizzata in quanto tutti i valori da assegnare a X(I) sono gia' noti a priori.

```
   DO 1 I = 1,N                     |   DO 1 I = 1,N
   DO 1 J = 1,M                     |   DO 2 J = 1,M
   A(I,J) = 0.                      |   A(I,J) = 0.
                                    | 2 CONTINUE
   DO 1 K = 1,L                     |   DO 1 K = 1,L
                                    |   DO 1 J = 1,M
   A(I,J) = A(I,J)+B(I,K)*C(K,J)    |   A(I,J) = ......
 1 CONTINUE                         | 1 CONTINUE
```

nel primo caso A(I,J) e' indipendente dal loop interno e pertanto costituisce una variabile ricorsiva che inibisce la vettorizzazione del loop; la seconda versione e' di gran lunga piu' veloce. Un metodo empirico per la verifica di questo consiste nell'espandere i loop per piccoli valori degli indici e verificare che il compilatore possa agire contemporaneamente su tutti gli elementi del vettore.

ALGORITMI LEGATI ALLA OTTIMIZZAZIONE

All'inizio del manuale abbiamo insistito sull'importanza della scelta dell'algoritmo e non ci stancheremo mai di ripeterlo.
E per ribadirlo ancora una volta, riportiamo un esempio tratto dal libro di Metcalf.

Supponiamo di dover programmare la formula:

$$S = \sum_{k=1}^{n} (-1)^{k} K$$

```
 S = 0.                 |  S = 0.           | S = N/2
 DO 1 K = 1,N           |  DO 1 K = 1,N,2   | IF (MOD(N,2).EQ.1)
 S = S + (-1)**K * K    |  S = S-K          |     S = S-FLOAT(N)
1 CONTINUE              | 1 CONTINUE        |
                        |  DO 2 K = 2,N,2   |
                        |  S = S+K          |
                        | 2 CONTINUE        |
```

Il primo algoritmo ricalca pedissequamente la formula; il secondo e' venti volte piu' veloce del primo, ma se supponiamo che il problema iniziale si possa affrontare anche da altri punti di vista (e in questo caso cio' puo' essere rappresentato dalla conoscenza della formula risolutiva) il programma diviene pressoche' banale e di gran lunga piu' veloce degli altri due.

Per di piu' il terzo programma:

- non ha problemi di conversione integer <--> real;
- non ha risultati intermedi maggiori di N, e quindi ha minori possibilita' di overflow;
- e' meno affetto da imprecisioni dovute alla propagazione degli errori in operazioni consecutive in grossi loop.

E questa ultima nota ci porta alla successiva parte del manuale: quella sugli errori e la precisione.

ERRORI E PRECISIONE

Una cosa mi ha fatto sempre innervorire da parte della stampa e della televisione: il voler affidare ad un elaboratore doti pressoche' sovrannaturali: il "cervellone" ha stabilito questo, il "cervellone" ha deciso quest'altro, ecc., ecc.
Del "cervellone", ma in senso negativo, lo darei a chi afferma queste cose e contribuisce cosi' negativamente a creare dei calcolatori l'immagine di macchine con una propria autonomia decisionale, ultraprecise e che non sbagliano mai.

C'e' intanto da dire che dietro a ciascuna macchina c'e' sempre l'uomo e che anche un calcolatore viene programmato a compiere certe operazioni dall'uomo stesso, ma anche trascurando questo, un calcolatore e' pur sempre una macchina fisica, con una circuiteria e un numero grande quanto si vuole, ma pur sempre limitato, di componenti elettronici.
E nell'eseguire le operazioni un calcolatore deve fare i conti proprio con queste componenti fisiche e limitate.

Siamo pertanto ben lontani dalla macchina ideale e ultraperfetta: un calcolatore attuale deve memorizzare i dati da qualche parte e questa "qualche parte" e' una cella di memoria o un registro di una determinata dimensione: 8, 16, 32, 64 bit, ma pur sempre finita.

E' fondamentalmente da questo che nasce il problema degli errori di calcolo durante l'esecuzione di programmi: il calcolatore, tutt'altro che essere infallibile, e' soggetto a una serie di errori dovuti appunto alla necessita' di memorizzare i dati in voci di memoria limitate (troncamento, arrotondamento) o alla necessita' di normalizzare e allineare tra di loro gli operandi durante le operazioni aritmetiche, che ne invalidano in maniera piu' a meno rilevante i risultati: da un lato quindi si pone ancora qui l'accento sull'importanza dell'analisi critica dei risultati, dall'altro sulla scelta dell'algoritmo piu' opportuno.

E per la scelta dell'algoritmo piu' opportuno si devono pertanto conoscere le cause degli errori indotti dal calcolatore e il loro modo di combinarsi, propagarsi e inficiare i risultati.

1) Troncamento/Arrotondamento di un numero.
E' dovuto alla dimensione finita della cella di memoria che deve ospitare il numero da memorizzare e alla rappresentazione interna binaria o esadecimale utilizzata.

Se si indicano con a1 e b1 i numeri veri e con **a2, b2** i numeri di macchina (troncati), cioe' i numeri che effettivamente possono essere contenuti nella cella di memoria, si ha:

$$a2 = a1 * (1+ea)$$
$$b2 = b1 * (1+eb)$$

dove **ea** e **eb** sono gli errori relativi di troncamento o arrotondamento.

2) Normalizzazione/Allineamento tra due numeri di macchina.
I numeri vengono in genere mantenuti in una cella di memoria in forma normalizzata (cioe' con la prima cifra dopo la virgola diversa da zero) in binario o in esadecimale.

Durante le operazioni di addizione e sottrazione tra due numeri inoltre gli operandi devono venire allineati l'uno con l'altro: si puo' pertanto deteminare la possibile perdita di cifre del minore di essi, col successivo troncamento del risultato.

Anche in questo caso, indicando con op1 l'operazione vera e con **op2** quella di macchina, si ha:

$$a2 \ op2 \ b2 = (a2 \ op1 \ b2) * (1+eop)$$

dove con **eop** si e' indicato l'errore relativo dovuto all'operazione.

3) Nel caso di operazioni su numeri veri, cioe' non di macchina, si ha naturalmente la combinazione dei due errori:

$$\begin{aligned} a2 \ op2 \ b2 &= (a2 \ op1 \ b2) * (1+eop) = \\ &= [a1(1+ea) \ op1 \ b1(1+eb)] * (1+eop) = \\ &= (a1 \ op1 \ b1) * (1+Eop) \end{aligned}$$

dove:

$Eop = eop + (a1/(a1+b1))*ea + (b1/(a1+b1))*eb$
per operazioni di addizione

$Eop = eop + (a1/(a1-b1))*ea - (b1/(a1-b1))*eb$
per operazioni di sottrazione

$Eop = eop + ea + eb$
per operazioni di moltiplicazione

$Eop = eop + ea - eb$
per operazioni di divisione

L'ampliamento dell'effetto di **ea** e **eb** costituisce il fenomeno della <u>propagazione</u> dell'errore nei calcoli.
Si vede che grossi errori nei calcoli si possono produrre solo nel caso dell'operazione di <u>sottrazione</u> tra due numeri **a1** e **b1** di grandezza paragonabile e sono dovuti alla cancellazione delle cifre piu' significative.

E' pertanto sbagliata l'opinione, abbastanza generalizzata, che sia la divisione a produrre i danni maggiori: l'operazione piu' pericolosa e' invece la <u>sottrazione</u>!

4) Si esamini infine una espressione completa, combinazione cioe' di piu' operazioni elementari, eseguibile con piu' algoritmi diversi.
Si puo' dimostrare che l'errore relativo dell'espressione completa si puo' scomporre in due termini distinti:

```
                                | alg1 (a1 op1 b1) --> ealg1+ei
Expr (a1 op1 b1)  --->          | ....
                                | algn (a1 op1 b1) --> ealgn+ei
```

dove:

ei = e' l'errore inerente ai dati, funzione <u>solo</u> dei dati e degli errori iniziali e <u>indipendente</u> dall'algoritmo prescelto;

ealg= e' l'errore algoritmico, funzione <u>solo</u> dei dati, delle operazioni dell'algoritmo scelto e degli errori sui dati temporanei intermedi, e quindi <u>dipendente</u> dall'algoritmo adottato.

Pertanto la valutazione del confronto tra due algoritmi viene fatta semplicemente dall'analisi dei rispettivi errori algoritmici **ealg**.
Senza entrare nei dettagli delle dimostrazioni (argomento ostico anche per me), diamo delle regole intuitive che si basano su quanto espresso finora per valutare alcuni diversi algoritmi.

ALGORITMI LEGATI ALLA PRECISIONE

1) (a^2-b^2) ---> | a*a - b*b alg1
| (a-b)*(a+b) alg2

Ricordiamo che nella valutazione di un algoritmo quello che conta e' l'errore algoritmico ealg che e' indipendente dagli errori sui dati **a** e **b** : cioe' ai fini del calcolo di ealg e' come se **a** e **b** fossero numeri di macchina.

alg1: la moltiplicazione introduce un errore sui dati intermedi e la sottrazione puo' amplificarlo <u>anche notevolmente</u>;
alg2: la sottrazione e la somma <u>tra</u> <u>due</u> <u>numeri</u> <u>di</u> <u>macchina</u> introducono solo un piccolo errore e la moltiplicazione, dal canto suo, non lo amplifica in maniera eccessiva.

Ne consegue che il secondo algoritmo e' in genere piu' affidabile del primo.

2) (x^2+bx+c) ---> | (x*x)+(b*x)+c alg1
| (x+b)*x+c alg2

alg1: le due moltiplicazioni introducono dei piccoli errori nei calcoli intermedi, che possono venire amplificati dalla somma;
alg2: solo il fattore (x+b) delle operazioni e' affetto da errore, pertanto questo algoritmo (che tra l'altro ha una operazione in meno) e' leggermente migliore del precedente.

3) Qual e' l'algoritmo migliore per effettuare la somma di n numeri?

alg1: in ordine decrescente | ealg maggiore
alg2: in ordine crescente |
alg3: a coppie ↓ ealg minore

4) Vediamo ora il risultato pratico della scelta di un algoritmo nel calcolo di due espressioni matematiche equivalenti:

X = 97865431

1° Caso	alg1 SQRT(X+1.)-SQRT(X)	= alg2 1./(SQRT(X+1.)+SQRT(X))
Singola precisione	.0	.505423377 * 10-4
Doppia precisione	.5054234225 * 10-4	.5054234201 * 10-4

2° Caso	alg1 99-70*SQRT(2.)	= alg2 1./(SQRT(2.)+1.)**6
Singola precisione	.508117676 * 10-2	.505063310 * 10-2
Doppia precisione	.5050633883342 * 10-2	.5050633883346 * 10-2

5) Alcuni casi in PL/1

```
DCL A FIXED DEC(15,10) INIT(1);
A = A+0.1;
DISPLAY (A);
```

Il risultato, sorprendente, e' 1.0625 invece di 1.1 !!!

```
DCL A FIXED DEC(5,2);
A = 25+1/3;
DISPLAY (A);
```

Questa volta il risultato e' piu' drammatico ancora: 5.3333... con la perdita totale della prima cifra; in altrnativa si puo' avere un bellissimo errore di FIXED OVFL !!!

Se pero' l'istruzione viene cambiata in:

```
A = 25+1/03;
```

cioe' con l'aggiunta di unc zero <u>NON SIGNIFICATIVO</u> allora il risultato e' corretto: 25.3333....

INDICE

Introduzione. 2

Didattica nella scuola 4

Corsi e ricorsi. 6

Compilatori e regole di ottimizzazione 10

Regole scontate 11

Regole meno note 12

Loop 14

Regole nascoste 16

Elaborazioni vettoriali. 18

Algoritmi legati alla ottimizzazione. 20

Errori e precisione 21

Algoritmi legati alla precisione 24