

Compilatori e tecniche di ottimizzazione



www.cineca.it

Gabriele Fatigati
CINECA System & Technologies Department
Supercomputing Group
g.fatigati@cineca.it

La compilazione è il processo mediante il quale un codice di alto livello viene convertito in linguaggio macchina.

Nato per evitare di scrivere direttamente in codice macchina o Assembly.

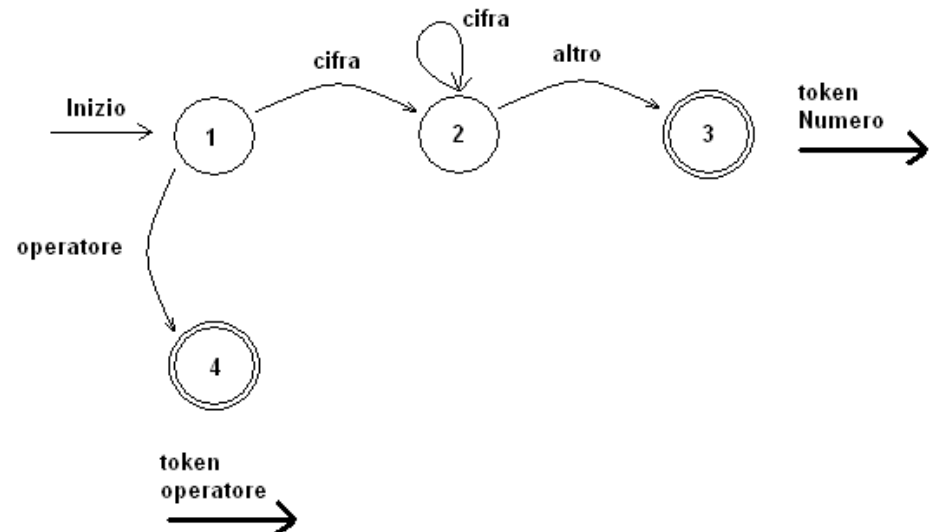
I più famosi sono Intel Compiler, GCC (GNU Compiler Collection) e PGI per Linux.

- Fase di preprocessing
- Compilazione
- Linking

Analisi lessicale: effettuato da uno scanner o lexer, si occupa di analizzare uno stream di caratteri e generare uno stream di tokens:

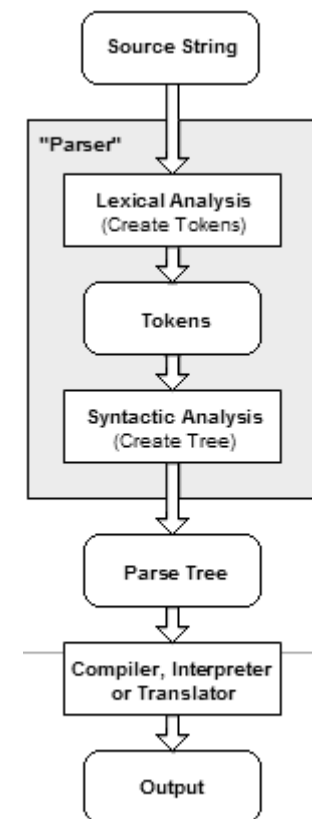
123 + 141 / 725

Tipo	Valore
numero	123
operatore	+
numero	141
operatore	/



Analisi sintattica: analisi di uno stream di caratteri secondo le regole di una grammatica formale (linguaggio). Effettuata da un parser.

```
int a = 0  << scorretta  
int a=0; OK
```



Può essere presente un preprocessore. Es in C.

- Direttiva #include

```
#include <stdio.h>
```

- Direttiva #ifdef

```
#ifdef DEBUG  
printf( "versione debug \n");  
#else  
printf( "versione release \n");  
#endif
```

- Direttiva define:

```
#define PI 3.14159
```

- **Macro:**

```
#define RADTODEG(x) ((x) * 57.29578)
```

- **Pragma:** Fornisce informazioni aggiuntive al compilatore

```
#pragma unroll
```

- Forzare l'unroll di un loop.

```
#pragma intel optimization_level n
```

Compila una funzione con il livello di ottimizzazione n

Compilazione: Il sorgente viene tradotto in linguaggio macchina secondo i flags di compilazione. In questa fase si creano i file objects.

Opzione **-c** per creare manualmente il file object. In questa fase non vengono cercate eventuali funzioni esterne non presenti nell'object.

Linking: integrazione dei vari moduli, file object e librerie mediante un **linker**. Tale fase produce l'eseguibile.

Objdump: per visualizzare l'assembly di un file object

`objdump -D object.o`

00000000 <.comment>:

0:	00 47 43	add	%al,0x43(%edi)
3:	43	inc	%ebx
4:	3a 20	cmp	(%eax),%ah
6:	28 55 62	sub	%dl,0x62(%ebp)
9:	75 6e	jne	79 <s+0x69>
b:	74 75	je	82 <s+0x72>
d:	20 34 2e	and	%dh,(%esi,%ebp,1)
10:	34 2e	xor	\$0x2e,%al
12:	33 2d 34 75 62 75	xor	0x75627534,%ebp
18:	6e	outsb	%ds:(%esi),(%dx)
19:	74 75	je	90 <s+0x80>
1b:	35 29 20 34 2e	xor	\$0x2e342029,%eax
20:	34 2e	xor	\$0x2e,%al
22:	33 00	xor	(%eax),%eax

Ldd: visualizza le librerie dinamiche utilizzate da un eseguibile:
`ldd <executable>:`

```
libmpi_f90.so.0 => /cineca/prod/opt/compilers/openmpi/1.3.3/intel--11.1--binary/lib/libmpi_f90.so.0 (0x00002ae9526f4000)
    libmpi_f77.so.0 => /cineca/prod/opt/compilers/openmpi/1.3.3/intel--11.1--binary/lib/libmpi_f77.so.0 (0x00002ae952a2d000)
    libmpi.so.0 => /cineca/prod/opt/compilers/openmpi/1.3.3/intel--11.1--binary/lib/libmpi.so.0 (0x00002ae952c64000)
    libopen-rte.so.0 => /cineca/prod/opt/compilers/openmpi/1.3.3/intel--11.1--binary/lib/libopen-rte.so.0 (0x00002ae9530f4000)
    libopen-pal.so.0 => /cineca/prod/opt/compilers/openmpi/1.3.3/intel--11.1--binary/lib/libopen-pal.so.0 (0x00002ae9533a0000)
    librdmacm.so.1 => /usr/lib64/librdmacm.so.1 (0x0000003cd0800000)
    libibverbs.so.1 => /usr/lib64/libibverbs.so.1 (0x0000003ccf800000)
    libbat.so => /cineca/sysprod/lfs/7.0/linux2.6-glibc2.3-x86_64/lib/libbat.so (0x00002ae95364e000)
    liblsf.so => /cineca/sysprod/lfs/7.0/linux2.6-glibc2.3-x86_64/lib/liblsf.so (0x00002ae95390d000)
    libnsl.so.1 => /lib64/libnsl.so.1 (0x0000003cd6800000)
    libutil.so.1 => /lib64/libutil.so.1 (0x0000003cdde00000)
    libm.so.6 => /lib64/libm.so.6 (0x00002ae953c06000)
    libpthread.so.0 => /lib64/libpthread.so.0 (0x0000003cd0000000)
    libc.so.6 => /lib64/libc.so.6 (0x0000003ccf400000)
```

Il valore in esadecimale è l'entry point (o load address) della libreria nell'eseguibile, ovvero il punto in cui verrà richiamata.

Se cambia l'eseguibile (es: con i flags), l'entry point può cambiare.

Molto utile se non abbiamo informazioni a priori su di un eseguibile.

- Architettura
- Aliasing
- Analisi interprocedurale
- Inlining
- Loop
- Funzioni intrinseche

E' possibile abilitare ottimizzazioni specifiche per un dato processore.

- -march=pentium4
- -mtune=pentium2 | pentium3 | pentium4 | core2 | atom | athlon

Perché usarle? Il compilatore già dovrebbe sapere quale processore sta utilizzando.

Non è detto che vengano abilitate tutte le ottimizzazioni per un dato processore.

Sia per una questione di tempo di compilazione, sia per la bontà dei risultati. Il -O3 può intrinsecamente richiamare questi flags ==> Consultare il manuale del compilatore.

Si può perdere di portabilità.

Se si utilizza un `-march=i386` generico l'eseguibile può potenzialmente girare su tutte le architetture i386.

Se si utilizza `-march=pentium4`, l'eseguibile può non funzionare su Pentium precedenti.

I binari precompilati sono il più generici possibile. Portabilità ma perdita di performances.

Per aliasing si intende una situazione in cui una stessa locazione di memoria può essere accessibile tramite più nomi simbolici.

```
int vector[10];  
  
int* punt = &vector[0];  
  
int* punt2 = &vector[0];  
  
vector[0] = 10;  
  
punt[0] = 10;  
  
punt2[0] = 10;
```

```
void modifica(int*vector){  
  
vector[0] = 10;  
  
}  
  
int main(){  
  
int a[10];  
modifica(a);  
  
}
```

L'ottimizzatore può fare assunzioni conservative in presenza di puntatori.

Si supponga di avere

```
x = 5  
.. codice...  
int *y = &x  
*y = 10
```

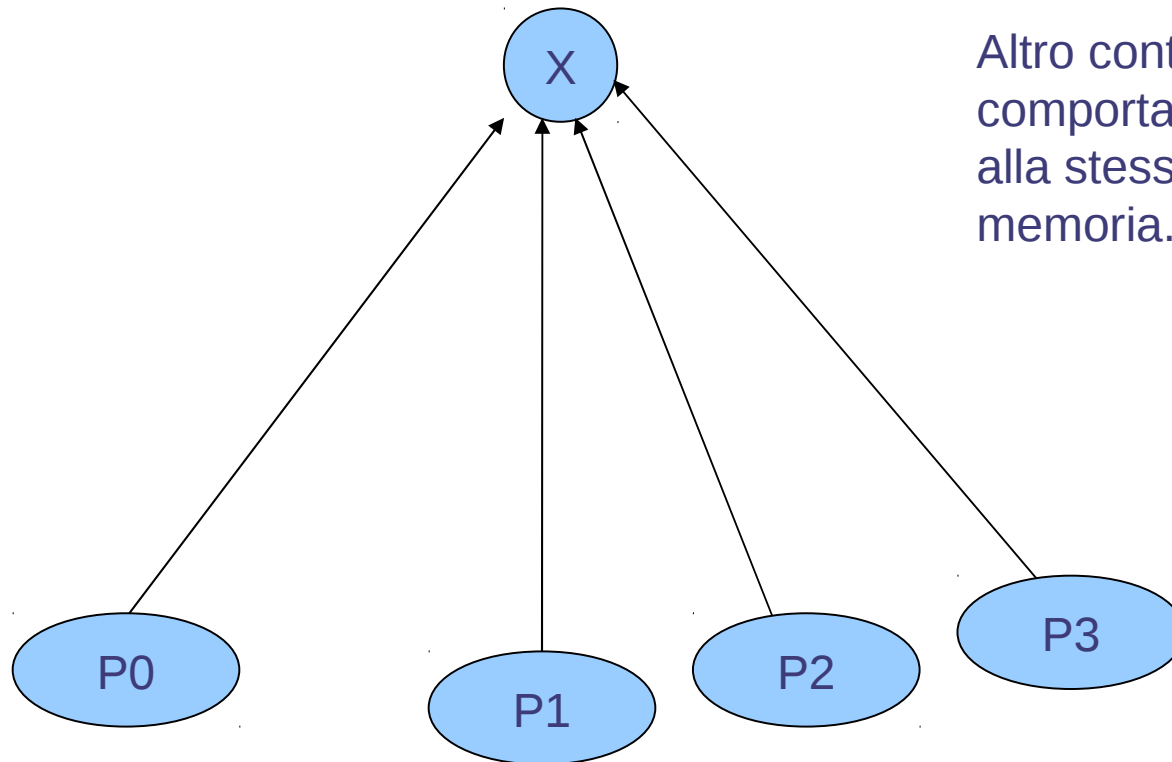
Non si può propagare oltre il valore 5 in quanto y, alias di x lo ha cambiato.

Se y non è alias di x, il compilatore può decidere di invertire le istruzioni:

```
*y = 10  
x = 5
```


Un conto è controllare solo x.

Altro conto è controllare il comportamento di 4 puntatori alla stessa locazione di memoria.



Se il compilatore ha informazioni riguardo i puntatori, può effettuare ottimizzazioni.

Strict aliasing: assunzione in C99 secondo la quale puntatori ad oggetti di tipi differenti non riferiscono mai alla stessa zona di memoria.

Flag: `-fstrict-aliasing`

```
int16_t* foo;  
int32_t* bar;
```

Il compilatore assume che `foo` e `bar` non si riferiranno mai alla stessa locazione di memoria.

```
funzione(int* restrict vector)
```

Flag: **-restrict**. informo il compilatore che `vector` è acceduto in maniera esclusiva all'interno della funzione.

Di default il compilatore ottimizza un file object per volta, senza avere una visione globale., concentrandosi su porzioni di codice, loops, e/o funzioni

Se in un loop è presenta una chiamata a funzione esterna, la IPO può analizzare se è conveniente o meno farla inline. Può anche riordinarla per un migliore memory layout.

Flags: **-ip -ipo (o -ipa)**

```
COMMON X,Y  
    ...  
    DO I = 1, N  
  
S0: CALL P  
S1: X(I) = X(I) + Y(I)  
  
ENDDO
```

Può essere vettorizzata se in P:

- Nessuno modifica o usa X
- Non si modifica Y

Nella IPO è importante analizzare se una funzione ha side-effect.

Una funzione ha side-effect se modifica uno stato al di fuori del proprio scoping locale.

- Modifica variabili globali
- Modifica variabili statiche
- Modifica di uno o più argomenti
- Scrittura a schermo
- Scrittura/lettura su file
- Lanciare un'eccezione.
- Chiamare altre funzioni side-effecting

```
SUBROUTINE S(A,X,N)
COMMON Y      /* Y is global variable */
      DO I = 1, N
          S0:   X = X + Y*A(I)
      ENDDO
END
```

Potrebbe essere più efficiente mantenere X e Y in registri differenti e scrivere X fuori del loop

Cosa succede se chiamo S(A,Y,N)?

Y ha un alias su X

Qualsiasi cosa fatta su X si riflette in Y

La chiamata a funzione è un'operazione piuttosto costosa eseguita sullo stack:

- 1) Creazione di uno stack frame sul top dello stack
- 2) Scrittura dell'indirizzo di ritorno
- 3) Scrittura di eventuali variabili locali
- 4) Scrittura di eventuali parametri passati (per valore, per riferimento)
- 3) Cancellazione dello stack frame e ritorno al chiamante.

PUSH: metti un valore sullo stack

POP: leggi e togli dallo stack un valore

JSR: jump to subroutine, ovvero salta ad una subroutine (salvando l'indirizzo di ritorno sullo stack con PUSH)

RET: ritorno da una subroutine al chiamante (identificato eseguendo una POP dell'indirizzo di ritorno dallo stack)

L'inlining è la tecnica mediante la quale una chiamata a funzione viene sostituita con il corpo stesso della funzione

Vantaggi:

- Elimina il costo della chiamata a funzione e istruzione di return.
- Elimina branches e mantiene le istruzioni eseguite “locali”

Svantaggi:

- Aumenta al dimensione dell'eseguibile
- Possono servire variabili addizionali (utilizzo di più registri)

Esempio:

```
int main(){  
    int x=10;  
    cout << " valore al quadrato: " << pot(x) << endl;  
  
}
```

```
coid pot(int value){  
    return value*value;  
}
```

```
int main(){  
    int x=10;  
  
    cout << " valore al quadrato: " << x*x <<  
    endl;  
  
}
```

E' possibile fare inlining a mano, ma può essere noioso e può indurre errori.

I moderni compilatori permettono di fare inlining automatico:

Keyword `inline` in C/C++. In questo caso è un suggerimento, non è detto che la funzione sia trasformata in inline

Il compilatore sceglie se trasformare una funzione inline o meno in base alla dimensione del suo corpo. Non si può fare inline di parti di una funzione.

-finline-limit=n dove n è la size della funzione

Conviene fare inlining con funzioni “piccole” chiamate frequentemente.

- Loop interchange
- Loop fusion
- Loop unrolling
- Loop unswitching
- Loop fission

```
for( int i = 0; i < N; i++)  
    for( int j=0; j<N; j++)  
        matrix[i][j] = i*j;
```

```
for( int j=0; j<N; j++)  
    for( int i = 0; i < N; i++)  
        matrix[i][j] = i*j;
```

Permette di diminuire i cache miss nel caso di accessi a posizioni di memoria non contigue

Non sempre si può fare. E può non convenire:

```
do i = 1, 10000
  do j = 1, 1000
    a(i) = a(i) + b(j,i) * c(i)
  end do
end do
```

Se si invertono i cicli, vengono fatti inutili store di a

```
int i, a[100], b[100];  
.....  
for (i = 0; i < 100; i++){  
    a[i] = a[i] + 1;  
    x+=a[i];  
}  
for (i = 0; i < 100; i++){  
    a[i] = a[i] + 2;
```

```
int i, a[100], b[100];  
.....  
for (i = 0; i < 100; i++){  
    {  
        a[i] = a[i]+1;  
        x+=a[i];  
        a[i] = a[i]+2;  
    }
```

Si elimina un loop, ma si aumenta il corpo del loop stesso. Necessità di trovare il giusto bilanciamento.

Alla fine del corpo di un loop, c'è il test sulla condizione di fine ciclo. Questa condizione può pesare molto, soprattutto su cicli con molte iterazioni.

```
int x;  
for (x = 0; x < 100; x++)  
{  
    a[i] = a[i]+1;  
}
```

```
int x;  
for (x = 0; x < 100; x += 5)  
{  
    a[i] = a[i]+1;  
    a[i+1] = a[i+1]+1;  
    a[i+2] = a[i+2]+1;  
    a[i+3] = a[i+3]+1;  
    a[i+4] = a[i+4]+1;  
}
```


Il nuovo loop esegue 1/5 dei controlli di fine loop rispetto al loop originale.

Più istruzioni per iterazione -> miglior utilizzo della pipeline.
Potenzialmente è 5 volte più veloce.

Se il passo di unroll non è divisore del numero di iterazioni, è necessario gestire il resto:

```
int x;  
for (x = 0; x < 11; x++)  
{  
    a[i] = a[i]+1;  
}
```

```
int x;  
a[0] = a[0] + 1  
for (x = 1; x < 11; x += 2)  
{  
    a[i] = a[i]+1;  
    a[i+1] = a[i+1]+1;  
}
```

Non esiste un metodo per trovare il passo di unroll ottimale.

Generalmente un unroll di 2 o max 4 è più che sufficiente.

Se il loop è complesso e pieno di dipendenze, il compilatore può non riuscire a fare l'unroll.

Se viene trovato il passo ottimale, permette speedup notevoli.

Muove condizione dentro il loop esternamente, replicando il corpo del loop in clausole if/else.

```
int i, w, x[1000], y[1000];  
for (i = 0; i < 1000; i++) {  
    x[i] = x[i] + y[i];  
    if (w)  
        y[i] = 0;  
}
```

Permette di ottimizzare separatamente i casi.

```
int i, w, x[1000], y[1000];  
if (w) {  
    for (i = 0; i < 1000; i++) {  
        x[i] = x[i] + y[i];  
        y[i] = 0;  
    }  
} else {  
    for (i = 0; i < 1000; i++) {  
        x[i] = x[i] + y[i];  
    }  
}
```

Contrario di loop fusion

```
int i, a[100], b[100];  
for (i = 0; i < 100; i++) {  
    a[i] = 1;  
    b[i] = 2;  
}
```

```
int i, a[100], b[100];  
for (i = 0; i < 100; i++) {  
    a[i] = 1;  
}  
for (i = 0; i < 100; i++) {  
    b[i] = 2;  
}
```

Permette di sfruttare maggiormente il principio di località.

Tutte le tecniche da attuare sui loop sono fortemente influenzate dal numero di iterazioni del ciclo in esame

Spesso conviene provare più di una tecnica, o addirittura mescolarle.

Generalmente, un loop è una delle porzioni di codice più dispendiose in un'applicazione.

I moderni compilatori hanno delle funzioni built-in altamente testate ed ottimizzate.

Alcune sono implementate direttamente in hardware. (istruzioni SSE, AVX).

Utilizzarle quando possibile anziché fare “a mano”.

Consultare il manuale del compilatore per lista di funzioni disponibili.

Istruzioni vettoriali che eseguono la stessa operazione su più dati

Attivabili da compilatore, o tramite hand tuning (intrinsic)

Registri a 128 bit = 4 operazioni intere/floating point singola precisione per volta, o 2 con doppia precisione

`__m128 _mm_add_ps(__m128 a, __m128 b)`

R0	R1	R2	R3
a0 +b0	a1 + b1	a2 + b2	a3 + b3

Istruzioni SSE (Streaming SIMD Instruction)

