22nd Summer School on PARALLEL COMPUTING

Compilers and optimization techniques

Gabriele Fatigati - g.fatigati@cineca.it
Supercomputing Group

CINECA

The compilation is the process by which a high-level code is converted to machine languages.

Born to avoid writing directly in machine code or Assembly.

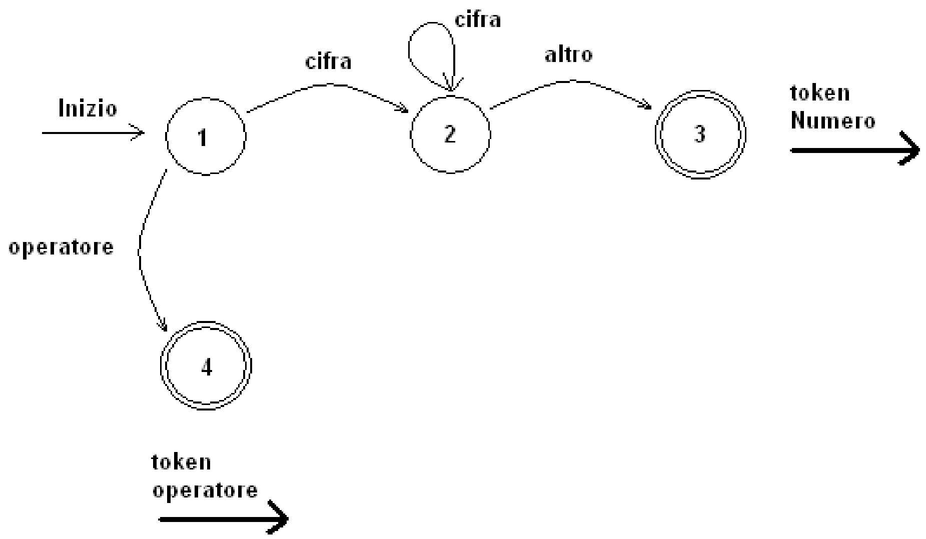
The most famous are the Intel Compiler, GCC (GNU Compiler Collection) and PGI for Linux.

- Preprocessing phase
- Compilation
- Linking

Lexical analysis: performed by a lexer or scanner, is responsible for analyzing a stream or characters and generate a stream of tokens.

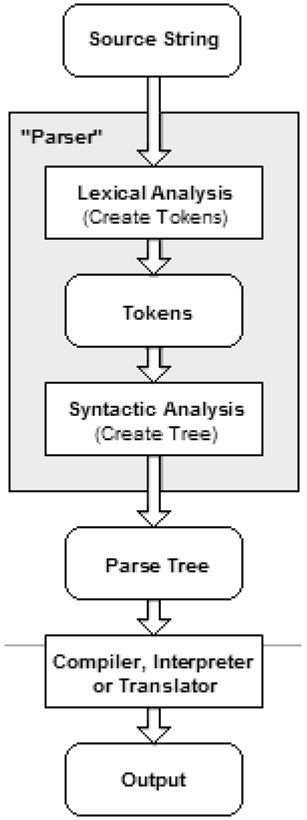
123 + 141 / 725

Type	Value
number	123
operator	+
number	141
operator	/
number	125



Syntactic analysis: analysis of a stream of characters according to the rules of formal grammar (language). Performed by a parser.

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


Source String
"Parser"
Lexical Analysis (Create Tokens)
Tokens
Syntactic Analysis (Create Tree)
Parse Tree
Compiler, Interpreter or Translator
Output

There may be a preprocessor. Example in C.

- #include directive

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

- #ifdef directive

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

- Define: directive

```
#define PI  3.14159
```

- **Macro:**

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

- **Pragma:** provides additional information to the compiler

```
#pragma unroll
```

- Forcing unroll a loop

```
#pragma intel optimization_level n
```

Compile a function with the optimizazion level n

Compilation: source code is translated into machine language according to the compilation flags. At this stage, objects are created.

-c option to manually create the object file. At this stage they are not looking for any external functions not present in the object.

Linking: integration of various modules, object files and libraries via a linker. This phase produces the executable.

Useful commands

Objdump: to explore the assembly of an object file

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
```

Useful commands

Ldd: displays the dynamic libraries used by an executable

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/lsf/7.0/linux2.6-glibc2.3-x86_64/lib/libbat.so (0x00002ae95364e000)
        liblsf.so => /cineca/sysprod/lsf/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)
```

The exadecimal value is the entry point (or load address) of the library into the executable, or the point which will be called

If you change the executable (eg: with the flags), the entry point can change.

Very useful if you have no a priori information about an executable.

Categories optimization

- Architecture
- Aliasing
- Interprocedural analysis
- Inlining
- Loop
- Intrinsic functions

Architecture

It is possible to enable specific optimizations for a given processor.

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

Why use them? The compiler should already know which processor is using.

All optimization quite often aren't enabled for a given processor

Both as a matter of compile time, both for the quality of results. The -O3 flag can intrinsically call up these flags. Refer to your compiler manual.

You can lose portability

If you are using a generic -march=i386 executable can potentially run on all i386.

If you are using -march=pentium4,the executable can not work on Pentium earlier.

The precompiled binaries are the most generic possible. Portability, but loss of performance.

Aliasing

It refers to a situation where the same memory location can be accessed through multiple symbolic names.

```
int vector[10];

int* punt = &vector[0];

int* punt2 = &vector[0];

vector[0] = 10;

punt[0] = 10;

punt2[0] = 10;
```

```
void func(int*vector){

vector[0] = 10;

}

int main(){

int a[10];
func(a);

}
```

Aliasing

The optimizer can make conservative assumptions in the presence of pointers

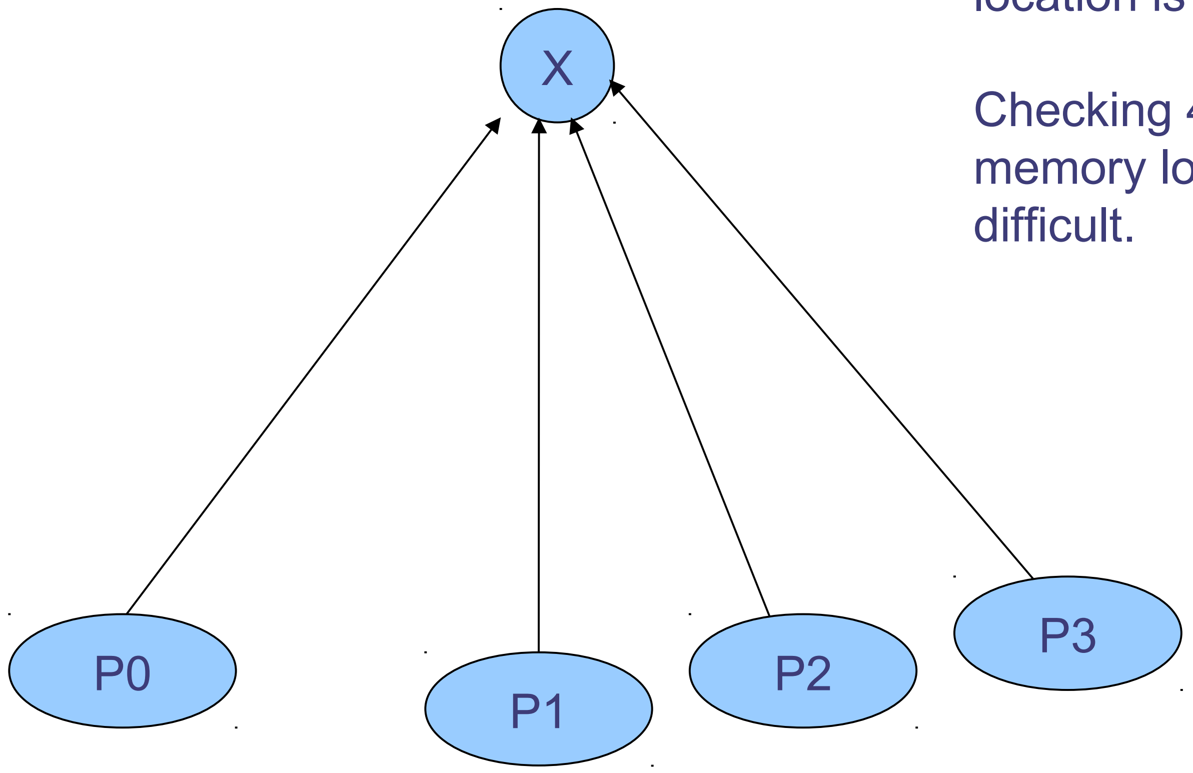
```
x = 5
.. codice...
int *y = &x
*y = 10
```

Can not propagate as well as the value 5, because y, (x alias) has changed it.

If y is not x aliases, the compiler may decide to reverse these instructions:

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

Aliasing



Cheching a single memory location is simple.

Checking 4 aliases to same memory location is more difficult.

Aliasing

If the compilers has information about pointers, it can perform optimizations.

Strict aliasing: C99 standard according to which pointers to object of different types do not ever refer to the same memory location

Flag: -fstrict-aliasing

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

The compiler assumes that foo and bar never refer to the same memory location

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

Flag: **-restrict.** Inform the compiler that vector is accessed exclusively within the function

Interprocedural analysis

By default, the compiler optimizes files for a time, without having a global vision, focusing on portions of code, loops, and/or functions

If a loop contains call to external function, the IPO can analyse whether or not it is convenient to inline it.

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

Interprocedural analisys

```
COMMON X,Y
                    ...
   DO I = 1, N

S0: CALL P
S1: X(I) = X(I) + Y(I)

   ENDDO
```

- Anybody use or change X?
- Anybody change Y?

Interprocedural analysis

In IPO is important to analyse whether a function has side-effects

A function has side-effects if a change was outside of their local scope.

- Changing global variables
- Changing static variables
- Changing one or more arguments
- Screen writing
- Writing/reading a file
- Throwing an exception
- Calling other side-effects functions

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

It might be more efficient to mantain different register X and Y and write X out of the loop
What happens if we call S(A,Y,N)?
 Y has X aliases
Any modification of X is reflected in Y

Inlining

The function call is an operation performed on the stack rather expensive

1) Create a stack frame on top of the stack
2) Writiting the return address
3) Writing any local variables
4) Writing any parameters passed (by value, reference)
3) Deleting of the stack frame and return to the caller

Inlining

PUSH: put a value on the stack
POP: read and remove a value on the stack
JSR: jump to subroutine, (saving the return address on the stack with PUSH)
RET: return from a subroutine to the caller (indentified by running a POP of return value from the stack)

Inlining

Inlining is a technique whereby a function call is replaced with its body

Benefits:

- Delete the cost of the function call and instruction return
- Delete statement executed branches and maintains the code locality

Disadvantages:

- Increase the executable size
- Could need additional variables (using multiple registers)

Inlining

Example::

```
int main(){
int x=10;
cout << " square value " << pow(x) << endl;

}

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

```
int main(){
int x=10;

cout << " square value " << x*x << endl;

}
```

Inlining

It is possible to make inlining by hand, but can be tedious and can lead to errors.

Modern compilers allow you to make automatic inlining:

Inline keyword in C/C++. In this case, a suggestion, it is said that the function is converted into inline

The compiler chooses whether to make an inline function or not according to the size of its body. You can not do inline parts of a function.

-finline-limit=n where n is the size of the function

Agrees to inlining functions “small” and frequently called.

Loop optimization

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

Loop interchange

```
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;
```

Allows you to reduce cache misses when access to non-contiguos memory locations.

You can not always do. It may not agree:

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

If you reverse the cycles, they are made useless store of “a” variable

Loop fusion

```
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;
 }
```

It eliminates a loop, but it extends the body loop. Need to find the right balance.

Loop unrolling

At the end of loop body, end-of-loop test is provided. This condition can be expensive, especially with many cycles iterations.

```
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;
 }
```

The new loop executes 1/5 of the control loop at the end than the original loop.

More instruction per iteration → better use of the pipeline. Potentially is 5 times faster.

If the unroll step is not a divisor of the number of iteration, you must handle the rest:

```
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;
 }
```

There is no method to find optimal unroll step.

Usually, a maximum of 2 or unroll 4 is enough.

If the loop is complex and has instruction dependencies, the compiler may fail to make the unroll.

If found the optimal unroll step, allows significant speedup.

Loop unswitching

Move internal loop condition outside, replicating the loop body in the if/else clauses:

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

```
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];
   }
 }
```

Used to optimize separately the cases

Loop fission

Unlike 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;
}
```

Allow to exploit better data and instruction locality

Performance of loop techniques are strongly affected by the number of the iterations of the loop under consideration.

It is often convenient try more than one technique, or even mix them

Usually, a loop is one of the portion more time expensive in a source code

Intrinsic functions

Modern compilers have built-in intrinsic functions highly optimized and tested.

Some are implemented directly in hardware (SSE, AVX)

Use them whenever possible instead of doing “by hand”

Refer to your manual compiler to the lists of functions available.

SSE instructions

Vector instructions that perform the same operations on multiple data.

Activated by the compiler, or by hand tuning (intrinsic)

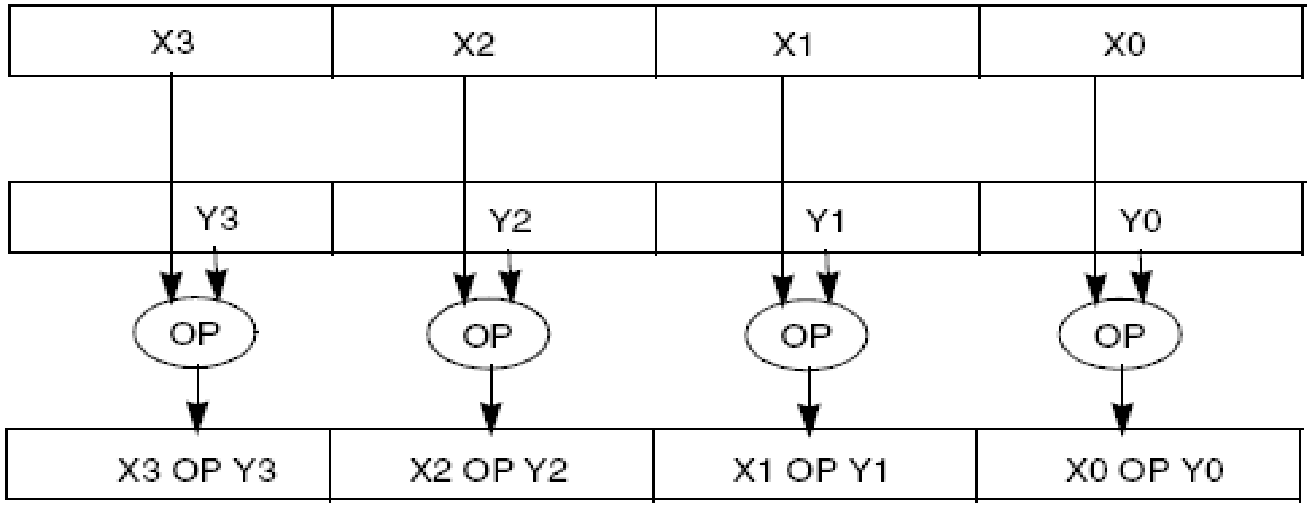
128-bit register integer/single precision floating point operations at a time, or 2 with double precision.

__m128 _mm_add_ps(__m128 a, __m128 b)

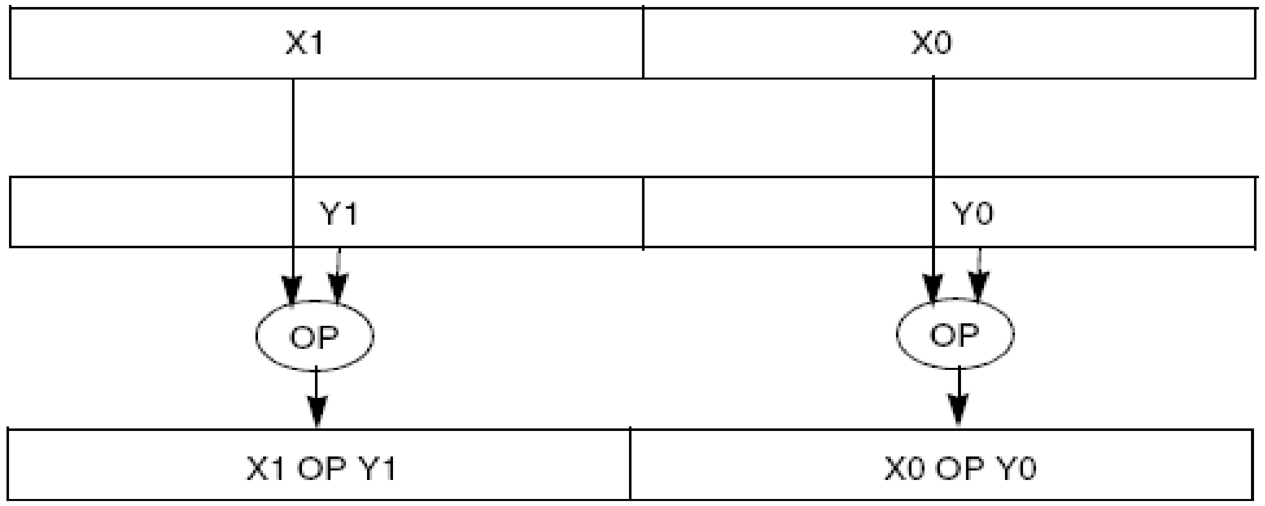
R0	R1	R2	R3
a0 +b0	a1 + b1	a2 + b2	a3 + b3

SSE Single precision

SSE instructions (Streaming SIMD Istruction)



SSE double precision


X1
X0
Y1
Y0
OP
OP
X1 OP Y1
X0 OP Y0