Debugging and Optimization of Scientific Applications

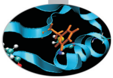
P. Lanucara V. Ruggiero

CINECA Rome - SCAI Department

Rome, 19-21 April 2017

SCAI
SuperComputing Applications and Innovation

AGENDA



19th April 2017

9.00-9.30 Registration
9.30-10.30 Architectures
10.30-13.00 Cache and Memory System
14.00-15.00 Pipelines
15.00-17.00 Profilers

20th april 2017

9.30-13.00 Compilers
14.00-15.30 Scientific Libraries
15.00-17.00 Makefile

21st april 2017

9.30-13.00 Debugging
14.00-17.00 Final hands-on

CINECA

Outline

Compilers and Code optimization

Scientific Libraries

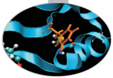
Makefile

Programming languages

- Many programming languages were defined...
- http://foldoc.org/contents/language.html

```
20-GATE; 2.PAK; 473L Query; 51forth; A#; A-0; a1; a56;
Abbreviated Test Language for Avionics Systems; ABC;
ABC ALGOL; ABCL/1; ABCL/c+; ABCL/R; ABCL/R2; ABLE;
ABSET; abstract machine; Abstract Machine Notation;
abstract syntax; Abstract Syntax Notation 1;
Abstract-Type and Scheme-Definition Language; ABSYS;
Accent; Acceptance, Test Or Launch Language; Access;
ACOM; ACOS; ACT++; Act1; Act2; Act3; Actalk; ACT ONE;
Actor; Actra; Actus; Ada; Ada++; Ada 83; Ada 95; Ada 9X;
Ada/Ed; Ada-O; Adaplan; Adaplex; ADAPT; Adaptive Simulated
Annealing; Ada Semantic Interface Specification;
Ada Software Repository; ADD 1 TO COBOL GIVING COBOL;
ADELE; ADES; ADL; AdLog; ADM; Advanced Function Presentation;
Advantage Gen; Adventure Definition Language; ADVSYS; Aeolus;
AFAC; AFP; AGORA; A Hardware Programming Language; AIDA;
AIr MAterial COmmand compiler; ALADIN; ALAM; A-language;
A Language Encouraging Program Hierarchy; A Language for Attributed ...
```

Programming languages



- Interpreted:
  - statement by statement translation during code execution
  - no way to perform optimization between different statements
  - easy to find semantic errors
  - e.g. scritping languages, Java (bytecode),. . .
- Compiled:
  - code is translated by the compiler before the execution
  - possibility to perform optimization between different statements
  - e.g. Fortran, C, C++

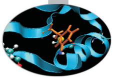
CPU/1

- It is composed by (first approximation):
  - Registers: hold instruction operands
  - Functional units: performs instructions
- Functional units
  - logical operations (bitwise)
  - integer arithmetic
  - floating-point arithmetic
  - computing address
  - load & store operation
  - branch prediction and branch execution

CPU/2

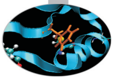
- RISC: Reduced Instruction Set CPU
  - simple "basic" instructions
  - one statement $\rightarrow$ many istructions
  - simple decode and execution
- CISC: Complex Instruction Set CPU
  - many "complex" instructions
  - one statement $\rightarrow$ few istructions
  - complex decode and execution

Architecture vs. Implementation



- Architecture:
  - instruction set (ISA)
  - registers (integer, floating point, . . .)
- Implementation:
  - physical registers
  - clock & latency
  - # of functional units
  - Cache’s size & features
  - Out Of Order execution, Simultaneous Multi-Threading, ...
- Same architecture, different implementations:
  - Power: Power3, Power4, . . ., Power8
  - x86: Pentium III, Pentium 4, Xeon, Pentium M, Pentium D, Core, Core2, Athlon, Opteron, . . .
  - different performances
  - different way to improve performance

The Compiler



- "Translate" source code in an executable
- Rejects code with syntax errors
- Warns (sometimes) about "semantic" problems
- Try (if allowed) to optimize the code
  - code independent optimization
  - code dependent optimization
  - CPU dependent optimization
  - Cache & Memory oriented optimization
  - Hint to the CPU (branch prediction)
- It is:
  - powerfull: can save programmer’s time
  - complex: can perform "complex" optimization
  - limited: it is an expert system but can be fooled by the way you write the code . . .

Building an executable

A three-step process:

1. Pre-processing:
   - every source code is analyzed by the pre-processor
     - MACROs substitution (**`#define`**)
     - code insertion for **`#include`** statements
     - code insertion or code removal (**`#ifdef`** . . .)
     - removing comments . . .
2. Compiling:
   - each code is translated in object files
     - object files is a collection of "symbols" that refere to variables/function defined in the program
3. Linking:
   - All the object files are put together to build the finale executable
   - Any symbol in the program must be resolved
     - the symbols can be defined inside your object files
     - you can use other object file (e.g. external libraries)

Example: gfortran compilation

- With the command:

```
user@caspur$> gfortran dsp.f90 dsp_test.f90 -o dsp.x
```

  all the three steps (preprocessing, compiling, linking) are performed at the same time

- Pre-processing

```
user@caspur$> gfortran -E -cpp dsp.f90
user@caspur$> gfortran -E -cpp dsp_test.f90
```

  - `-E -cpp` options force `gfortran` to stop after pre-processing
  - no need to use `-cpp` if file extension is `*.F90`

- Compiling

```
user@caspur$> gfortran -c dsp.f90
user@caspur$> gfortran -c dsp_test.f90
```

  - `-c` option force `gfortran` only to pre-processing and compile
  - from every source file an object file `*.o` is created

Example: gfortran linking

- Linking: we must use object files

```
user@caspur$> gfortran dsp.o dsp_test.o -o dsp.x
```

- To solve symbols from external libraries
  - suggest the libraries to use with option `-l`
  - suggest the directory where looking for libraries with option `-L`
- e.g.: link `libdsp.a` library located in `/opt/lib`

```
user@caspur$> gfortran file1.o file2.o -L/opt/lib -ldsp -o dsp.x
```

- How create and link a static library

```
user@caspur$> gfortran -c dsp.f90
user@caspur$> ar curv libdsp.a dsp.o
user@caspur$> ranlib libdsp.a
user@caspur$> gfortran test_dsp.f90 -L. -ldsp
```

  - `ar` creates the archive `libdsp.a` containing `dsp.o`
  - `ranlib` builds the library

Compiler: what it can do

- It performs many code modifications
  - Register allocation
  - Register spilling
  - Copy propagation
  - Code motion
  - Dead and redundant code removal
  - Common subexpression elimination
  - Strength reduction
  - Inlining
  - Index reordering
  - Loop pipelining , unrolling, merging
  - Cache blocking
  - . . .
- Everything is done to maximize performances!!!

Compiler: what it cannot do

- Global optimization of "big" source code, unless switch on interprocedural analisys (IPO) but it is very time consuming ...
- Understand and resolve complex indirect addressing
- Strenght reduction (with non-integer values)
- Common subexpression elimination through function calls
- Unrolling, Merging, Blocking with:
  - functions/subroutine calls
  - I/O statement
- Implicit function inlining
- Knowing at run-time variabile's values

Optimizations: levels

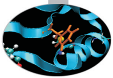
- All compilers have “predefined” optimization levels **`-O<n>`**
  - with **`n`** from 0 a 3 (IBM compiler up to 5)
- Usually :
  - **`-O0`**: no optimization is performed, simple translation (tu use with **`-g`** for debugging)
  - **`-O`**: default value (each compiler has it’s own default)
  - **`-O1`**: basic optimizations
  - **`-O2`**: memory-intensive optimizations
  - **`-O3`**: more aggressive optimizations, it can alter the instruction order (see floating point section)
- Some compilers have **`-fast/-Ofast`** option (**`-O3`** plus more options)

Intel compiler: –O3 option

```
icc (or ifort) -O3
```

- Automatic vectorization (use of packed SIMD instructions)
- Loop interchange (for more efficient memory access)
- Loop unrolling (more instruction level parallelism)
- Prefetching (for patterns not recognized by h/w prefetcher)
- Cache blocking (for more reuse of data in cache)
- Loop peeling (allow for misalignment)
- Loop versioning (for loop count; data alignment; runtime dependency tests)
- Memcpy recognition (call Intel's fast memcpy, memset)
- Loop splitting (facilitate vectorization)
- Loop fusion (more efficient vectorization)
- Scalar replacement (reduce array accesses by scalar temps)
- Loop rerolling (enable vectorization)
- Loop reversal (handle dependencies)

From source code to executable



- Executable (i.e. istructions performed by CPU) is very very different from what you think writing a code
- Example: matrix-matrix production

```
do j = 1, n
   do k = 1, n
      do i = 1, n
         c(i,j) = c(i,j) + a(i,k)*b(k,j)
      end do
   end do
end do
```

- Computational kernel
  - load from memory three numbers
  - perform one product and one sum
  - store back the result

Hands-on: download code

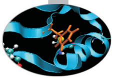
- Exercises

```
https://hpc-forge.cineca.it/files/CoursesDev/public/2016/...
...Debugging_and_Optimization_of_Scientific_Applications/Rome/

Compilers_codes.tar

Libraries_codes.tar

FloatingPoints_codes.tar

Make_codes.tar (tomorrow)
```

- To expand archive

```
tar -xvf Compilers_codes.tar
```

Hands-on: available modules for desktop



- Sintax;

```
module av
------ /usr/local/Modules/3.2.10/modulefiles ---------------
autoload                          hdf5/intel-serial/1.8.16
gcc/5.2                           intel/compilers/pe-xe-2016
grace/5.1                         intel/mkl/11.3
gromacs/5.0.4                     intel/vtune/16.1
hdf5/gnu-api16-serial/1.8.16 openmpi/1.10.1/gcc-5.2
hdf5/gnu-parallel/1.8.16      openmpi/1.8.5/gcc-4.8
hdf5/gnu-serial/1.8.16        paraview/4.4.

module li

module load intel/compilers/pe-xe-2016

module purge
```

Hands-on: compiler optimization flags

- Matrix-Matrix product, 1024×1024, double precision
- Cache friendly loop
- The Code is in **`matrixmul`** directory (both C & Fortran)
- to load compiler: (**`module load profile/advanced`**):
  - GNU –> **`gfortran, gcc`** : **`module load gcc/5.2`**
  - Intel –> **`ifort, icc`** : **`module load intel/compilers/pe-xe-2016`**
  - You can load one compiler at time, **`module purge`** to remove previous compiler

	GNU	Intel	GNU	Intel
flags	seconds	seconds	GFlops	GFlops
-O0				
-O1				
-O2				
-O3				
-O3 -funroll-loops		——		——
-Ofast	——		—-	

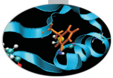
Hands-on: Solution

- Matrix-Matrix product, 1024×1024, double precision
- 2 esa-core XEON 5645 Westmere CPUs@2.40GHz
- Fortran results

	GNU	Intel	PGI	GNU	Intel	PGI
flags	seconds	seconds	seconds	GFlops	GFlops	GFlops
default	7.78	0.76	3.49	0.27	2.82	0.61
-O0	7.82	8.87	3.43	0.27	0.24	0.62
-O1	1.86	1.45	3.42	1.16	1.49	0.63
-O2	1.31	0.73	0.72	1.55	2.94	2.99
-O3	0.79	0.34	0.71	2.70	6.31	3.00
-O3 -funroll-loops	0.65	——	——	3.29	——	——
-fast	——	0.33	0.70	——	6.46	3.04

- Open question:
  - Why this behaviour?
  - Which is the best compiler?
  - http://www.epcc.ed.ac.uk/blog/2016/03/30/array-index-order-matters-right

Matmul: performance



- Size 1024×1024, duoble precision
- Fortran core, cache friendly loop
  - FERMI: IBM Blue Gene/Q system, single-socket PowerA2 with 1.6 GHz, 16 core
  - PLX: 2 esa-core XEON 5650 Westmere CPUs 2.40 GHz

FERMI - xlf

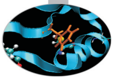
Option	seconds	Mflops
-O0	65.78	32.6
-O2	7.13	301
-O3	0.78	2735
-O4	55.52	38.7
-O5	0.65	3311

PLX - ifort

Option	seconds	MFlops
-O0	8.94	240
-O1	1.41	1514
-O2	0.72	2955
-O3	0.33	6392
-fast	0.32	6623

- Why ?

Compiler: report



- What happens at different optimization level?
  - Why performance degradation using **`-O4`**?
- Hint: use report flags to investigate
- Using IBM **`-qreport`** flag for **`-O4`** level shows that:
  - The compiler understant matrix-matrix pattern (it is smart) ad perform a substitution with external BLAS function (**`__xl_dgemm`**)
  - But it is slow because it doesn't belong to IBM optimized BLAS library (ESSL)
  - At **`-O5`** level it decides not to use external library
- As general rule of thumb performance increase as the optimization level increase . . .
  - . . . but it's bettet to check!!!

Who does the dirty work?

- option **`-fast`** (ifort on PLX) produce a $\simeq$ 30x speed-up respect to option **`-O0`**
  - many different (and complex) optimizations are done ...
- **Hand-made optimizations?**
- The compiler is able to do
  - Dead code removal: removing branch

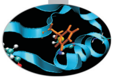
```
b = a + 5.0;
if ((a>0.0) && (b<0.0)) {
    ......
}
```

  - Redudant code removal

```
integer, parameter :: c=1.0
f=c*f
```

- **But coding style can fool the compiler**

Loop counters



- Always use the correct data type
- If you use as loop index a real type means to perform a implicit casting real → integer every time
- I should be an error according to standard, but compilers are (sometimes) sloppy...

```
real :: i,j,k
....
do j=1,n
do k=1,n
do i=1,n
c(i,j)=c(i,j)+a(i,k)*b(k,j)
enddo
enddo
enddo
```

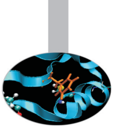
Time in seconds

compiler/level	integer	real
(PLX) gfortran -O0	9.96	8.37
(PLX) gfortran -O3	0.75	2.63
(PLX) ifort -O0	6.72	8.28
(PLX) ifort -fast	0.33	1.74
(PLX) pgif90 -O0	4.73	4.85
(PLX) pgif90 -fast	0.68	2.30
(FERMI) bgxlf -O0	64.78	104.10
(FERMI) bgxlf -O5	0.64	12.38

Compilers limitations

- A compiler can do a lot of work . . . but it is a program
- It is easy to fool it!
  - loop body too complex
  - loop values not defined a compile time
  - to much nested **`if`** structure
  - complicate indirect addressing/pointers

index reordering



- For simple loops there's no problem
  - . . . using appropriate optimization level

```
do i=1,n
   do k=1,n
      do j=1,n
         c(i,j) = c(i,j) + a(i,k)*b(k,j)
      end do
   end do
end do
```

- Time in seconds

	j-k-i	i-k-j
(PLX) ifort -O0	6.72	21.8
(PLX) ifort -fast	0.34	0.33

index reordering/2

- For more complicated loop nesting could be a problem . . .
  - also at higher optimization levels
  - solution: always write cache friendly loops, if possible

```
do jj = 1, n, step
   do kk = 1, n, step
      do ii = 1, n, step
         do j = jj, jj+step-1
            do k = kk, kk+step-1
               do i = ii, ii+step-1
                  c(i,j) = c(i,j) + a(i,k)*b(k,j)
               enddo
            enddo
         enddo
      enddo
   enddo
enddo
```

- Time in seconds

Otimization level	j-k-i	i-k-j
(PLX) ifort -O0	10	11.5
(PLX) ifort -fast	1.	2.4

Cache & subroutine

```
do i=1,nwax+1
    do k=1,2*nwaz+1
       call diffus (u_1,invRe,qv,rv,sv,K2,i,k,Lu_1)
       call diffus (u_2,invRe,qv,rv,sv,K2,i,k,Lu_2)
....
    end do
 end do


 subroutine diffus (u_n,invRe,qv,rv,sv,K2,i,k,Lu_n)
   do j=2,Ny-1
     Lu_n(i,j,k)=invRe*(2.d0*qv(j-1)*u_n(i,j-1,k)-(2.d0*rv(j-1)
               +K2(i,k))*u_n(i,j,k)+2.d0*sv(j-1)*u_n(i,j+1,k))
   end do
 end subroutine
```

- non unitary access (stride MUST be $\simeq$ 1)

Cache & subroutine/2

```
call diffus (u_1,invRe,qv,rv,sv,K2,Lu_1)
call diffus (u_2,invRe,qv,rv,sv,K2,Lu_2)
....

subroutine diffus (u_n,invRe,qv,rv,sv,K2,i,k,Lu_n)
 do k=1,2*nwaz+1
  do j=2,Ny-1
   do i=1,nwax+1
    Lu_n(i,j,k)=invRe*(2.d0*qv(j-1)*u_n(i,j-1,k)-(2.d0*rv(j-1)
              +K2(i,k))*u_n(i,j,k)+2.d0*sv(j-1)*u_n(i,j+1,k))
   end do
  end do
 end do
end subroutine
```

- "same" results as the the previous one
- stride = 1
- Sometimes compiler can perform the transformations, but `inlining` option must be activated

Inlining

- means to substitue the functon call with all the instruction
  - no more jump in the program
  - help to perform interpocedural analysis
- the keyword **`inline`** for C and C++ is a “hint” for compiler
- Intel (n: 0=disable, 1=inline functions declared, 2=inline any function, at the compiler’s discretion)

```
-inline-level=n
```

- GNU (n: size, default is 600):

```
-finline-functions
-finline-limit=n
```

- It varies from compiler to compiler, read the manpage . . .

Common Subexpression Elimination

- Using Common Subexpression for intermediate results:
  A= B+C+D
  E= B+F+C
- ask for: 4 load, 2 store, 4 sums
  A=(B+C) + D
  E=(B+C) + F
- ask for 4 load, 2 store, 3 sums, 1 intermediate result.
- WARNING: with floating point arithmetics results can be different
- "Scalar replacement" if you access to a vector location many times
  - compilers can do that (at some optimization level)

Functions & Side Effects

- Functions returns a values but
  - sometimes global variables are modified
  - I/O operations can produce side effects
- side effects can “stop” compiler to perform inlining
- Example (no side effect):

```
function f(x)
   f=x+dx
end
```

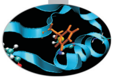
so `f(x)+f(x)+f(x)` it is equivalent to `3*f(x)`

- Example (side effect):

```
function f(x)
   x=x+dx
   f=x
end
```

so `f(x)+f(x)+f(x)` it is different from `3*f(x)`

CSE & function



- reordering function calls can produce different results
- It is hard for a compiler understand is there are side effects
- Example: 5 calls to functons, 5 products:

```
x=r*sin(a)*cos(b);
y=r*sin(a)*sin(b);
z=r*cos(a);
```

- Example: 4 calls to functons, 4 products, 1 temporary variable:

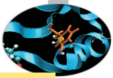
```
temp=r*sin(a)
x=temp*cos(b);
y=temp*sin(b);
z=r*cos(a);
```

- Correct if there’s no side effect!

CSE: limitations

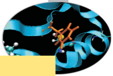
- Core loop too wide:
  - Compiler is able to handle a fixed number of lines: it could not realize that there's room for improvement
- Functions:
  - there is a side effect?
- CSE mean to alter order of operations
  - enabled at "high" optimization level (**`-qnostrict`** per IBM)
  - use parentheis to "inhibit" CSE
- "register spilling": when too much intermediate values are used

What can do a compiler?



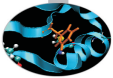
```
do k=1,n3m
   do j=n2i,n2do
      jj=my_node*n2do+j
      do i=1,n1m
         acc =1./(1.-coe*aciv(i)*(1.-int(forclo(nve,i,j,k))))
         aci(jj,i)= 1.
         api(jj,i)=-coe*apiv(i)*acc*(1.-int(forclo(nve,i,j,k)))
         ami(jj,i)=-coe*amiv(i)*acc*(1.-int(forclo(nve,i,j,k)))
         fi(jj,i)=qcap(i,j,k)*acc
     enddo
   enddo
enddo
...
...
do i=1,n1m
   do j=n2i,n2do
      jj=my_node*n2do+j
      do k=1,n3m
         acc =1./(1.-coe*ackv(k)*(1.-int(forclo(nve,i,j,k))))
         ack(jj,k)= 1.
         apk(jj,k)=-coe*apkv(k)*acc*(1.-int(forclo(nve,i,j,k)))
         amk(jj,k)=-coe*amkv(k)*acc*(1.-int(forclo(nve,i,j,k)))
         fk(jj,k)=qcap(i,j,k)*acc
      enddo
   enddo
enddo
```

. . . this . . .



```
do k=1,n3m
   do j=n2i,n2do
      jj=my_node*n2do+j
      do i=1,n1m
         temp = 1.-int(forclo(nve,i,j,k))
         acc =1./(1.-coe*aciv(i)*temp)
         aci(jj,i)= 1.
         api(jj,i)=-coe*apiv(i)*acc*temp
         ami(jj,i)=-coe*amiv(i)*acc*temp
         fi(jj,i)=qcap(i,j,k)*acc
      enddo
   enddo
enddo
...
...
do i=1,n1m
   do j=n2i,n2do
      jj=my_node*n2do+j
      do k=1,n3m
         temp = 1.-int(forclo(nve,i,j,k))
         acc =1./(1.-coe*ackv(k)*temp)
         ack(jj,k)= 1.
         apk(jj,k)=-coe*apkv(k)*acc*temp
         amk(jj,k)=-coe*amkv(k)*acc*temp
         fk(jj,k)=qcap(i,j,k)*acc
      enddo
   enddo
enddo
```

. . . but not that!!! (20% faster)



```
do k=1,n3m
   do j=n2i,n2do
      do i=1,n1m
         temp_fact(i,j,k) = 1.-int(forclo(nve,i,j,k))
      enddo
   enddo
enddo
...
...
do i=1,n1m
   do j=n2i,n2do
      jj=my_node*n2do+j
      do k=1,n3m
         temp = temp_fact(i,j,k)
         acc =1./(1.-coe*ackv(k)*temp)
         ack(jj,k)= 1.
         apk(jj,k)=-coe*apkv(k)*acc*temp
         amk(jj,k)=-coe*amkv(k)*acc*temp
         fk(jj,k)=qcap(i,j,k)*acc
      enddo
   enddo
enddo
...
...
!  the same for the other loop
```

Array Syntax

- in place 3D-array translation (512^3)
- Explixcit loop (Fortran77): 0.19 seconds
  - CAVEAT: the loop order is “inverse” in order not to overwirte data

```
do k = nd, 1, -1
  do j = nd, 1, -1
    do i = nd, 1, -1
      a03(i,j,k) = a03(i-1,j-1,k)
    enddo
  enddo
enddo
```

- Array Syntax (Fortran90): 0.75 seconds
  - According to the Standard → store in an intermediate array to avoid to overwrite data

```
a03(1:nd, 1:nd, 1:nd) = a03(0:nd-1, 0:nd-1, 1:nd)
```

- Array Syntax with hint: 0.19 seconds

```
a03(nd:1:-1,nd:1:-1,nd:1:-1) = a03(nd-1:0:-1, nd-1:0:-1, nd:1:-1)
```

Ottimizzazione Report/1

- A report of optimization performed can help to find “problems”
- Intel

```
-opt-report[n]      n=0(none),1(min),2(med),3(max)
-opt-report-file<file>
-opt-report-phase<phase>
-opt-report-routine<routine>
```

- one or more `*.optrpt` file are generated

```
...
Loop at line:64 memcopy generated
...
```

- Is this `memcopy` necessary?

Ottimizzazione Report/2

- There's no equivalent flag for GNU compilers
  - Best solution:

```
-fdump-tree-all
```

  - dump all compiler operations
  - very hard to understand

- PGI compilers

```
-Minfo
-Minfo=accel,inline,ipa,loop,opt,par,vect
```

Info at standard output

Give hints to compiler

- Loop size known at compile-time o run-time
  - Some optimizations (like unrolling) can be inhibited

```
real a(1:1024,1:1024)
real b(1:1024,1:1024)
real c(1:1024,1:1024)
...
read(*,*) i1,i2
read(*,*) j1,j2
read(*,*) k1,k2
...
do j = j1, j2
do k = k1, k2
do i = i1, i2
c(i,j)=c(i,j)+a(i,k)*b(k,j)
enddo
enddo
enddo
```

- Time in seconds (Loop Bounds Compile-Time o Run-Time)

flag	LB-CT	LB-RT
(PLX) ifort -O0	6.72	9
(PLX) ifort -fast	0.34	0.75

- WARNING: compiler dependent...

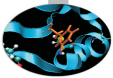
Static vs. Dynamic allocation

- Static allocation gives more information to compilers
  - but the code is less flexible
  - recompile every time is really boring

```
integer :: n
parameter(n=1024)
real a(1:n,1:n)
real b(1:n,1:n)
real c(1:n,1:n)
```

```
real, allocatable, dimension(:,:) :: a
real, allocatable, dimension(:,:) :: b
real, allocatable, dimension(:,:) :: c
print*,'Enter matrix size'
read(*,*) n
allocate(a(n,n),b(n,n),c(n,n))
```

Static vs. Dynamic allocation/2



- for today compilers there's no big difference
  - Matrix-Matrix Multiplication (time in seconds)

	static	dynamic
(PLX) ifort -O0	6.72	18.26
(PLX) ifort -fast	0.34	0.35

- With static allocation data are put in the "stack"
  - at run-time take care of stacksize (e.g. segmentation fault)
  - bash: to check

```
ulimit -a
```

  - bash: to modify

```
ulimit -s unlimited
```

Dynamic allocation using C/1

- Using C matrix → arrays of array
  - with static allocation data are contiguos (columnwise)

```
    double A[nrows][ncols];
```

- with dynamic allocation ....
  - “the wrong way”

```
/* Allocate a double matrix with many malloc */
double** allocate_matrix(int nrows, int ncols) {
    double **A;
    /* Allocate space for row pointers */
    A = (double**) malloc(nrows*sizeof(double*) );
    /* Allocate space for each row */
    for (int ii=1; ii<nrows; ++ii) {
        A[ii] = (double*) malloc(ncols*sizeof(double));
    }
    return A;
}
```

Dynamic allocation using C/2

- allocate a linear array

```
/* Allocate a double matrix with one malloc */
double* allocate_matrix_as_array(int nrows, int ncols) {
   double *arr_A;
   /* Allocate enough raw space */
   arr_A = (double*) malloc(nrows*ncols*sizeof(double));
   return arr_A;
}
```

- using as a matrix (with index linearization)

```
arr_A[i*ncols+j]
```

- MACROs can help
- also use pointers

```
/* Allocate a double matrix with one malloc */
double** allocate_matrix(int nrows, int ncols, double* arr_A) {
   double **A;
   /* Prepare pointers for each matrix row */
   A = new double*[nrows];
   /* Initialize the pointers */
   for (int ii=0; ii<nrows; ++ii) {
      A[ii] = &(arr_A[ii*ncols]);
   }
   return A;
}
```

Aliasing & Restrict

- Aliasing: when two pointers point at the same area
- Aliasing can inhibit optimization
  - you cannot alter order of operations
- C99 standard introduce **`restrict`** keyword to point out that aliasing is not allowed

```
void saxpy(int n, float a, float *x, float* restrict y)
```

- C++: aliasing not allowed between pointer to different type (strict aliasing)

Different operations, different latencies

For a CPU different operations present very different latencies

- sum: few clock cycles
- product: few clock cycles
- sum+product: few clock cycles
- division: many clock cycle ($O(10)$)
- sin,sos: many many clock cycle ($O(100)$)
- exp,pow: many many clock cycle ($O(100)$)
- I/O operations: many many many clock cycles ($O(1000-10000)$)

Input/Output

- Handled by the OS:
  - many system calls
  - pipeline goes dry
  - cache coerency can be destroyed
  - it is very slow (HW limitation)

- Golden Rule #1: NEVER mix computing with I/O operations
- Golden Rule #2: NEVER read/write a single data, pack them in a block

Different I/O

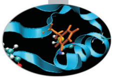
```
do k=1,n  ;  do j=1,n  ;  do i=1,n
write(69,*) a(i,j,k)                            ! formatted I/O
enddo     ;  enddo     ;  enddo

do k=1,n  ;  do j=1,n  ;  do i=1,n
write(69) a(i,j,k)                              ! binary I/O
enddo     ;  enddo     ;  enddo

do k=1,n  ;  do j=1,n
write(69) (a(i,j,k),i=1,n)                      ! by colomn
enddo     ;  enddo

do k=1,n
write(69) ((a(i,j,k),i=1),n,j=1,n)              ! by matrix
enddo

write(69) (((a(i,j,k),i=1,n),j=1,n),k=1,n)      ! dump (1)

write(69) a                                     ! dump (2)
```

Different I/O: some figures

	seconds	Kbyte
formatted	81.6	419430
binary	81.1	419430
by colunm	60.1	268435
by matrix	0.66	134742
dump (1)	0.94	134219
dump (2)	0.66	134217

- WARNING: the filesystem used could affect performance (e.g. RAID)...

I/O



- read/write operations are slow
- read/write format data are very very slow
- ALWAYS read/write binary data

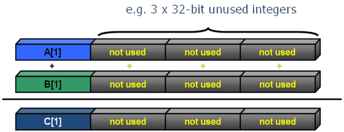
- Golden Rule #1: NEVER mix computing with I/O operations
- Golden Rule #2: NEVER read/write a single data, pack them in a block
- For HPC is possibile use:
  - I/O libraries: MPI-I/O, HDF5, NetCDF,...

Vector units

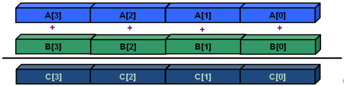
- We are not talking of vector machine
- Vector Units performs parallel floating/integer point operations on dedicate units (SIMD)
  - Intel: MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2
- i.e.: summing 2 arrays of 4 elements in one single instruction

```
c(0) = a(0) + b(0)
c(1) = a(1) + b(1)
c(2) = a(2) + b(2)
c(3) = a(3) + b(3)
```

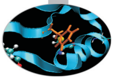
no vectorization



vectorization



SIMD Parallelism



- Vector instructions are handled by an additional unit in the CPU core, called something like a vector arithmetic unit.
- If used to their potential, they can allow you to perform the same operation on multiple pieces of data in a single instruction.
  - Single-Instruction, Multiple Data parallelism.
  - Your algorithm may not be amenable to this...
  - ... But lots are. (Spatially-local inner loops over arrays are a classic.)
- It has traditionally been hard for the compiler to vectorise code efficiently, except in trivial cases.
  - It would suck to have to write in assembly to use vector instructions...

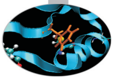
SIMD - evolution

- SSE: 128 bit register (from Intel Core/AMD Opteron)
  - 4 floating/integer operations in single precision
  - 2 floating/integer operations in double precision
- AVX: 256 bit register (from Sandy Bridge/AMD Bulldozer)
  - 8 floating/integer operations in single precision
  - 4 floating/integer operations in double precision
- MIC: 512 bit register (Intel Knights Corner)
  - 16 floating/integer operations in single precision
  - 8 floating/integer operations in double precision

Vectorization issues

- Vectorization is a key issue for performance
- To be vectorized a single loop iteration must be independent: no data dependence
- Coding style can inhibit vectorization
- Some issues for vectorization:
  - Countable at runtime
    - Number of loop iterations is known before loop executes
    - No conditional termination (break statements)
  - Have single control flow
    - No Switch statements
    - 'if' statements are allowable when they can be implemented as masked assignments
  - Must be the innermost loop if nested
    - Compiler may reverse loop order as an optimization!
  - No function calls
    - Basic math is allowed: pow(), sqrt(), sin(), etc
    - Some inline functions allowed
- WARNING: due to floating point arithmetic results could differ …

When vectorization fails



- Not Inner Loop: only the inner loop of a nested loop may be vectorized, unless some previous optimization has produced a reduced nest level. On some occasions the compiler can vectorize an outer loop, but obviously this message will not then be generated.
- Low trip count:The loop does not have sufficient iterations for vectorization to be worthwhile.
- Vectorization possible but seems inefficient:the compiler has concluded that vectorizing the loop would not improve performance. You can override this by placing
  `#pragma vector always` (C C++) or
  `!dir$ vector always` (Fortran) before the loop in question
- Contains unvectorizable statement: certain statements, such as those involving switch and printf , cannot be vectorized

When vectorization fails

- Subscript too complex: an array subscript may be too complicated for the compiler to handle. You should always try to use simplified subscript expressions
- Condition may protect exception: when the compiler tries to vectorize a loop containing an if statement, it typically evaluates the RHS expressions for all values of the loop index, but only makes the final assignment in those cases where the conditional evaluates to TRUE. In some cases, the compiler may not vectorize because the condition may be protecting against accessing an illegal memory address. You can use the **`#pragma ivdep`** to reassure the compiler that the conditional is not protecting against a memory exception in such cases.
- Unsupported loop Structure: loops that do not fulfill the requirements of countability, single entry and exit, and so on, may generate error messages

When vectorization fails

- Operator unsuited for vectorization: Certain operators, such as the % (modulus) operator, cannot be vectorized
- Non-unit stride used: non-contiguous memory access.
- Existence of vector dependence: vectorization entails changes in the order of operations within a loop, since each SIMD instruction operates on several data elements at once. Vectorization is only possible if this change of order does not change the results of the calculation

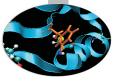
Vectorized loops? (intel compiler)

- Vectorization is enabled by the flag -vec and by default at -O2.

```
-vec-report[N] (deprecated)
-qopt-report[=N] -qopt-report-phase=vec
```

```
N (Optional) Indicates the level of detail in the report.
You can specify values 0 through 5. If you specify zero,
no report is generated. For levels N=1 through N=5,
each level includes all the information of the previous level,
as well as potentially some additional information. Level 5
produces the greatest level of detail. If you do not specify N,
the default is level 2, which produces a medium level of detail
```

Vectorized loops?



gnu compiler

- Vectorization is enabled by the flag -ftree-vectorize and by default at -O3.

```
-ftree-vectorizer-verbose=[N] (deprecated)
-fopt-info-vec
```

pgi compiler

- Vectorization is enabled by the flag -Mvec and by default at -fast or -fastsse .

```
-Minfo-vec
```

When vectorization fails

- Programmers need to provide the necessary information
- Programmers need to transform the code

- Add compiler directives
- Transform the code
- Program using vector intrinsics

Algorithm & Vectorization

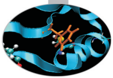
- Different algorithm, for the same problem, could be vectorized or not
  - Gauss-Seidel: data dependencies, cannot be vectorized

```
for( i = 1; i < n-1; ++i )
  for( j = 1; j < m-1; ++j )
    a[i][j] = w0 * a[i][j] +
      w1*(a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]);
```

  - Jacobi: no data dependence, can be vectorized

```
for( i = 1; i < n-1; ++i )
  for( j = 1; j < m-1; ++j )
     b[i][j] = w0*a[i][j] +
       w1*(a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1]);
for( i = 1; i < n-1; ++i )
  for( j = 1; j < m-1; ++j )
     a[i][j] = b[i][j];
```

Optimization & Vectorization



- "coding tricks" can inhibit vectorization
  - can be vectorized

```
for( i = 0; i < n-1; ++i ){
   b[i] = a[i] + a[i+1];
}
```

  - cannot be vectorized

```
x = a[0];
for( i = 0; i < n-1; ++i ){
  y = a[i+1];
  b[i] = x + y;
  x = y;
}
```

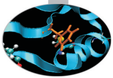
- You can help compiler's work
  - removing unnecessary data dependencies
  - using directives for forcing vectorization

Directives

- You can force to vectorize when the compiler doesn't want using directive
- they are "compiler dependent"
  - Intel Fortran: **!DIR$ simd**
  - Intel C: **#pragma simd**
- Example: data dependency found by the compiler is apparent, cause every time step **inow** is different from **inew**

```
     do k = 1,n
!DIR$ simd
        do i = 1,l
...
          x02 = a02(i-1,k+1,inow)
          x04 = a04(i-1,k-1,inow)
          x05 = a05(i-1,k  ,inow)
          x06 = a06(i  ,k-1,inow)
          x11 = a11(i+1,k+1,inow)
          x13 = a13(i+1,k-1,inow)
          x14 = a14(i+1,k  ,inow)
          x15 = a15(i  ,k+1,inow)
          x19 = a19(i  ,k  ,inow)

          rho =+x02+x04+x05+x06+x11+x13+x14+x15+x19
...
          a05(i,k,inew) = x05 - omega*(x05-e05) + force
          a06(i,k,inew) = x06 - omega*(x06-e06)
...
```

Hands-on: Vectorization



- Compare performances w/o vectorization **`simple_loop.f90`** using Intel compiler
  - **`-Ofast`**, to inhibit vectorization use **`-no-vec`** (Intel)
- Program **`vectorization_test.f90`** contains 18 different loops
  - Which can be vectorized?
  - check with Intel compiler with reporting flag **`-Ofast -opt-report3 -vec-report3`**
  - check with GNU compiler with reporting flag **`-ftree-vectorizer-verbose=n / -fopt-info-all`**
  - Any idea to force vectorization?
  - (using PGI compiler with reporting flag **`-fast -Minfo`**, **`-Mnovect`** to inhibit vectorization use)

Hands-on: Vectorization/2

	Intel
Vectorized time	
Non-Vectorized time	

# Loop	# Description	Vect/Not
1	Simple	
2	Short	
3	Previous	
4	Next	
5	Double write	
6	Reduction	
7	Function bound	
8	Mixed	
9	Branching	
10	Branching-II	
11	Modulus	
12	Index	
13	Exit	
14	Cycle	
15	Nested-I	
16	Nested-II	
17	Function	
18	Math-Function	

Hands-on: Vectorization Results

	PGI	Intel
Vectorized time	0.79	0.52
Non-Vectorized time	1.58	0.75

# Loop	Description	PGI	Intel
1	Simple	yes	yes
2	Short	no: unrolled	yes
3	Previous	no: data dep.	no: data dep.
4	Next	yes	yes: how?
5	Double write	no: data dep.	no: data dep.
6	Reduction	yes	? ignored
7	Function bound	yes	yes
8	Mixed	yes	yes
9	Branching	yes	yes
10	Branching-II	ignored	yes
11	Modulus	no: mixed type	no: inefficient
12	Index	no: mixed type	yes
13	Exit	no: exits	no: exits
14	Cycle	? ignored	yes
15	Nested-I	yes	yes
16	Nested-II	yes	yes
17	Function	no: function call	yes
18	Math-Function	yes	yes

Handmade Vectorization

- It is possible to insert inside the code vectorized function
- You have to rewrite the loop making 4 iteration in parallel ...

```
void scalar(float* restrict result,
            const float* restrict v,
            unsigned length)
{
  for (unsigned i = 0; i < length; ++i)
  {
    float val = v[i];
    if (val >= 0.f)
      result[i] = sqrt(val);
    else
      result[i] = val;
  }
}
```

```
void sse(float* restrict result,
         const float* restrict v,
         unsigned length)
{
  __m128 zero = _mm_set1_ps(0.f);

  for (unsigned i = 0; i <= length - 4; i += 4)
  {
    __m128 vec  = _mm_load_ps(v + i);
    __m128 mask = _mm_cmpge_ps(vec, zero);
    __m128 sqrt = _mm_sqrt_ps(vec);
    __m128 res  =
         _mm_or_ps(_mm_and_ps(mask, sqrt),
         _mm_andnot_ps(mask, vec));
    _mm_store_ps(result + i, res);
  }
}
```

- Non-portable tecnique...

Automatic parallelization

- Some compilers are able to exploit parallelism in an automatic way
- Shared Memory Parallelism
- Similar to OpenMP Paradigm without directives
  - Usually performance are not good . . .
- Intel:

```
-parallel
-par-threshold[n] - set loop count threshold
-par-report{0|1|2|3}
```

- IBM:

```
-qsmp                   automatic parallelization
-qsmp=openmp:noauto     no automatic parallelization
```

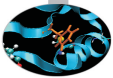
Outline

Compilers and Code optimization

Scientific Libraries

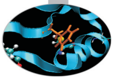
Makefile

Scientific Libraries



- A (complete?) set of function implementing different numeric algorithms
- A set of basic function (e.g. Fast Fourier Transform, . . .)
- A set of low level function (e.g. scalar products or random number generator), or more complex algorithms (Fourier Transform or Matrix diagonalization)
- (Usually) Faster than hand made code (i.e. sometimes it is written in assembler)
- Proprietary or Open Source
- Sometimes developed for a particular compiler/architecture . . .

Pros & Cons



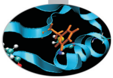
- Pros:
  - Helps to modularize the code
  - Portability
  - Efficient
  - Ready to use
- Cons:
  - Some details are hidden (e.g. Memory requirements)
  - You don’t have the complete control
  - You have to read carefully the documentation (complex syntax, error prone....)
  - …

Which library?

- It is hard to have a complete overview of Scientific libraries
  - many different libraries
  - still evolving . . .
  - . . . especially for “new architectures” (e.g GPU, Intel Xeon PHI...)

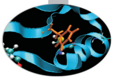
- Main libraries used in HPC
  - Linear Algebra
  - FFT
  - I/O libraries
  - Message Passing
  - Mesh decomposition
  - . . .

Linear Algebra



- Different parallelization paradigm
  - Shared memory (i.e. multi-threaded) or/and Distributed Memory
- Shared memory
  - BLAS
  - GOTOBLAS
  - LAPACK/CLAPACK/LAPACK++
  - ATLAS
  - PLASMA
  - SuiteSparse
  - ...
- Distributed Memory
  - Blacs (only decomposition)
  - ScaLAPACK
  - PSBLAS
  - Elemental
  - ...

BLAS

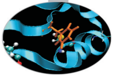


- BLAS: Basic Linear Algebra Subprograms
  - it is one of the first library developed for HPC (1979, vector machine)
  - it includes basic operations between vectors, matrix and vector, matrix and matrix
  - it is used by many other high level libraries
- It is divided into 3 different levels
  - BLAS lev. 1: basic subroutines for scalar-vector operations (1977-79, vector machine)
  - BLAS lev. 2: basic subroutines for vector-matrix operations (1984-86)
  - BLAS lev. 3: subroutines for matrix-matrix operations (1988)

BLAS/2

- It apply to real/complex data, in single/double precision
- Old Fortran77 style
- Level 1: scalar-vector operations (O(n))
  - *SWAP vector swap
  - *COPY vector copy
  - *SCAL scaling
  - *NRM2 L2-norm
  - *AXPY sum: a*X+Y (X,Y are vectors)
- Level 2: vector-matrix operations ($O(n^2)$)
  - *GEMV product vector/matrix (generic)
  - *HEMV product vector/matrix (hermitian)
  - *SYMV product vector/matrix (simmetric)

BLAS/3



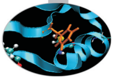
- Level 3: matrix-matrix operations ($O(n^3)$)
  - *GEMM product matrix/matrix (generic)
  - *HEMM product matrix/matrix (hermitian)
  - *SYMM product matrix/matrix (simmetric)

- GOTOBLAS
  - optimized (using assembler) BLAS library for different supercomputers. Develped by Kazushige Goto, now at Texas Advanced Computing Center, University of Texas at Austin.

LAPACK & Co.

- LAPACK: Linear Algebra PACKage
  - Linear algebral solvers (linear systems of equations, Ordinary Least Square, eigenvalues, ...)
  - evolution of LINPACK e EISPACK
- ATLAS: Automatically Tuned Linear Algebra Software
  - BLAS and LAPACK (but only some subroutine) implementations
  - Automatic optization of Software paradigm
- PLASMA: Parallel Linear Algebra Software for Multi-core Architectures
  - Similare to LAPACK (less subroutines) developed to be efficent on multicore systems.
- SuiteSparse
  - Sparse Matrix

Linear Algebra/2



- Eigenvalues/Eigenvectors
  - EISPACK: with specialized version for matrix for different kind (real/complex, hermitia, simmetrich, tridiagonal, ...)
  - ARPACK: eigenvalues for big size problems. Parallel version use BLACS and MPI libraries.
- Distributed Memory Linear Algebra
  - BLACS: linear algebra oriented message passing interface
  - ScaLAPACK: Scalable Linear Algebra PACKage
  - Elemental: framework for dense linear algebra
  - PSBLAS: Parallel Sparse Basic Linear Algebra Subroutines
  - ...

Input/Output Libraries

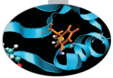
- I/O Libraries are extremely important for
  - Interoperability: C/Fortran, Little Endian/Big Endian, . . .
  - Visualization
  - Sub-set data analysis
  - Metadata
  - Parallel I/O

- HDF5: “is a data model, library, and file format for storing and managing data”
- NetCDF: “NetCDF is a set of software libraries and self-describing, machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data”
- VTK: “open-source, freely available software system for 3D computer graphics, image processing and visualization”

Other Libraries

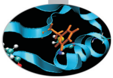
- MPI: Message Passing Interface
  - De facto standard for Distributed Memory Parallelization (MPICH/OpenMPI or vendor (IntelMPI))
- Mesh decomposition
  - METIS and ParMETIS: “can partition a graph, partition a finite element mesh, or reorder a sparse matrix”
  - Scotch and PT-Scotch: “sequential and parallel graph partitioning, static mapping and clustering, sequential mesh and hypergraph partitioning, and sequential and parallel sparse matrix block ordering”

Other Scientific computing libraries



- Trilinos
  - object oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific problems
  - A two-level software structure designed around collections of packages
  - A package is an integral unit developed by a team of experts in a particular algorithms area
- PETSc
  - It is a suite of data structures and routines for the (parallel) solution of applications modeled by partial differential equations.
  - It supports MPI, shared memory pthreads, and GPUs through CUDA or OpenCL, as well as hybrid MPI-shared memory pthreads or MPI-GPU parallelism.

Specialized Libraries



- MKL: Intel Math Kernel Library
  - Major functional categories include Linear Algebra, Fast Fourier Transforms (FFT), Vector Math and Statistics. Cluster-based versions of LAPACK and FFT are also included to support MPI-based distributed memory computing.
- ACML: AMD Core Math Library
  - Optimized functions for AMD processors. It includes BLAS, LAPACK, FFT, Random Generators . . .
- GSL: GNU Scientific Library
  - The library provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. There are over 1000 functions in total with an extensive test suite.
- ESSL (IBM): Engineering and Scientific Subroutine library
  - BLAS, LAPACK, ScaLAPACK, Sparse Solvers, FFT....others libraries.... The Parallel versions are mainly MPI-based

How to call a library

- first of all the syntax should be correct
- always check for the right version
- sometimes for proprietary libraries linking could be "complicated"
- e.g. Intel ScaLAPACK

```
mpif77 <program> -L$MKLROOT/lib/intel64 \
   -lmkl_scalapack_lp64  -lmkl_blacs_openmpi \
   -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core \
   -liomp5 -lpthread
```

Static and Dynamic libraries

- you have to link with
  -L<library_directory> -l<library_name>

- Static library:
  - ***.a**
  - all symbols are included in the executable at linking
  - if you built a new library that use an other external library it doesn’t contains the other symbols: you have to explicitly linking the library
- Dynamic Library:
  - ***.so**
  - Symbols are resolved at run-time
  - you have to set-up where find the requested library at run-time (i.e. setting **LD_LIBRARY_PATH** environment variable)
  - **ldd <exe_name>** gives you info about dynamic library needed

Interoperability

- Many libraries are written using C, many others using Fortran
- This can produce some problems when calling C (Fortran) libraries from Fortran (C) source
  - type matching: C `int` is not granted to be the same with Fortran `integer`
  - symbols matching: Fortran and C++ may “alter” symbol’s name producing object file (e.g. Fortran put an extra _)
- Brute force approach:
  - hand-made match all types and add _ to match all libraries objects.
  - `nm <object_file>` lists all symbols
- Standard Fortran 2003 (module `iso_c_binding`)
  - The most important libraries should provide a Fortran2003 interface
- In C++ command `extern "C"`

Interoperability/2

- To call libraries from C to Fortran and viceversa
- Example: **`MPI library`** written using C/C++:
  - old style: **`include "mpif.h"`**
  - new style: **`use mpi`**
  - the two approach are not fully equivalent: using the module implies also a compile-time type check!
- Example: **`FFTW library`** written using C
  - legacy : **`include "fftw3.f"`**
  - modern:

```
use iso_c_binding
include 'fftw3.f03'
```

- Example: **`BLAS`** written using Fortran
  - legacy: call **`dgemm_`** instead of **`dgemm`**
  - modern: call **`cblas_dgemm`**
- Standardization still lacking...
  - Read the manual . . .

BLAS: Interoperability/1

- Take a look at “netlib” web site

```
http://www.netlib.org/blas/
```

- BLAS was written in Fortran 77, some compiler may gives you some interfaces (types check, F95 features)
  - Using Intel and MKL

```
use mkl95_blas
```

BLAS:Interoperability/2

- C (legacy):
  - add underscore to function's name
  - Fortran: argoments by reference, it is mandatory to pass pointers
  - Type matching (compiler dependent): probably **`double`**, **`int`**, **`char`** $\rightarrow$ **`double precision`**, **`integer`**, **`character`**
- C (modern)
  - use interface **`cblas`**: GSL (GNU) or MKL (Intel)
  - include header file **`#include <gsl.h>`** or **`#include<mkl.h>`**

`http://www.gnu.org/software/gsl/manual/html_node/GSL-CBLAS-Library.htm`

Hands-on: BLAS

- make an explicit call to **DGEMM** routine (BLAS).
- **DGEMM:** It perform double precision matrix-matrix multiplication
- **DGEMM: http://www.netlib.org/blas/dgemm.f**

```
C := alpha*op( A )*op( B ) + beta*C
```

- Fortran: Intel, use **mkl**:
  - **sequential** (serial)
  - **parallel** (multi-threaded)

```
module load intel/cs-xe-2015--binary
module load intel/mkl/mkl/11.2--binary
ifort -O3 -mkl=sequential matrixmulblas.dgemm.F90
```

Hands-on: BLAS/2

- C: Intel MKL
  - include header file `#include<mkl.h>`
  - try `-mkl=sequential` and `-mkl=parallel`

```
module load intel/cs-xe-2015--binary
module load intel/mkl/mkl/11.2--binary
icc -O3 -mkl=sequential matrixmulblas.dgemm.c
```

- C: GNU (GSL with cblas)
  - include header file `#include <gsl/gsl_cblas.h>`

```
module load profile/advanced
module load gnu/4.9.2
module load gsl/1.16--gnu--4.9.2
gcc -O3 -L$GSL_HOME/lib -lgslcblas matrixmulblas.cblas.c -I$GSL_INC
```

  - Compare with performance obtained with baseline `-O3`/`-O3 -parallel`
- Write the measured GFlops for a matrix of size 4096x4096

Intel -O3	Intel -O3 -parallel	GNU-GSL seq	Intel-MKL seq	Intel-MKL par

Hands-on: BLAS/3

- Fortran:

```
call DGEMM('n','n',N,N,N,1.d0,a,N,b,N,0.d0,c,N)
```

- C (cblas):

```
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
            nn, nn, nn, 1., (double*)a, nn, (double*)b,
            nn, 0., (double*)c, nn);
```

- C (legacy):

```
dgemm_(transpose1, transpose2, &n, &n, &n, &alfa,
       (double*)a, &n, (double*)b, &n, &beta, (double*)c, &n);
```

C

Intel -O3	Intel -O3 -parallel	GNU-GSL seq	Intel-MKL seq	Intel-MKL par

Fortran

Intel -O3	Intel -O3 -parallel	GNU-GSL seq	Intel-MKL seq	Intel-MKL par
		N.A.		

SuperComputing Applications and Innovation

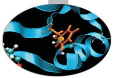
Outline

Compilers and Code optimization

Scientific Libraries

Makefile

HPC development tools



- What do I need to develop my HPC application? At least:
  - A compiler (e.g. GNU, Intel, PGI, PathScale, . . .)
  - A code editor
- Several tools may help you (even for non HPC applications)
  - Debugger: e.g. gdb, TotalView, DDD
  - Profiler: e.g. gprof, Scalasca, Tau, Vampir
  - Project management: e.g. make, projects
  - Revision control: e.g. svn, git, cvs, mercurial
  - Generating documentation: e.g. doxygen
  - Source code repository: e.g. sourceforge, github, google.code
  - Data repository, currently under significant increase
  - . . .

Code editor

- You can select the code editor among a very wide range
  - from the light and fast text editors (e.g. VIM, emacs, . . .)
  - to the more sophisticated Integrated development environment (IDE), (e.g. Ecplise)
  - or you have intermediate options (e.g. Geany)
- The choice obviously depends on the complexity and on the software tasks
- ... but also on your personal taste

Project management

- Non trivial programs are hosted in several source files and link libraries
- Different types of files require different compilation
  - different optimization flags
  - different languages may be mixed, too
  - compilation and linking require different flags
  - and the code could work on different platforms
- During development (and debugging) several recompilations are needed, and we do not want to recompile all the source files but only the modified ones
- How to deal with it?
  - use the IDE (with plug-ins) and their project files to manage the content (e.g. Eclipse)
  - use language-specific compiler features
  - use external utilities, e.g. Make!

GNU Make

- "Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files"
- Make gets its knowledge from a file called the makefile, which lists each of the non-source files and how to compute it from other files
- When you write a program, you should write a makefile for it, so that it is possible to use Make to build and install the program and more . . .
- GNU Make has some powerful features for use in makefiles, beyond what other Make versions have

Preparing and Running Make

- To prepare to use make, you have to write a file that describes:
  - the relationships among files in your program
  - commands for updating each file
- Typically, the executable file is updated from object files, which are created by compiling source files
- Once a suitable makefile exists, each time you change some source files, the shell command

```
make -f <makefile_name>
```

  performs all necessary recompilations
- If **`-f`** option is missing, the default names **`makefile`** or **`Makefile`** are used

Rules

- A simple makefile consists of “rules”:

```
    target ... : prerequisites ...
            recipe
            ...
            ...
```

  - a **`target`** is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as clean
  - a **`prerequisite`** is a file that is used as input to create the target. A target often depends on several files.
  - a **`recipe`** is an action that make carries out. A recipe may have more than one command, either on the same line or each on its own line. Recipe commands must be preceded by a tab character.
- By default, make starts with the first target (default goal)

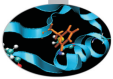
My first rule

- A simple rule:

```
foo.o : foo.c defs.h
	gcc -c -g foo.c
```

- This rule says two things
  - how to decide whether foo.o is out of date: it is out of date if it does not exist, or if either foo.c or defs.h is more recent than it
  - how to update the file foo.o: by running gcc as stated. The recipe does not explicitly mention defs.h, but we presume that foo.c includes it, and that that is why defs.h was added to the prerequisites.
- WARNING: Remember the tab character before starting the recipe lines!

A simple example in C



- The main program is in laplace2d.c file
  - includes two header files: timing.h and size.h
  - calls functions in two source files: update_A.c and copy_A.c
- update_A.c and copy_A.c includes two header files: laplace2d.h and size.h
- A possible (naive) Makefile

```
laplace2d_exe: laplace2d.o update_A.o copy_A.o
        gcc -o laplace2d_exe laplace2d.o update_A.o copy_A.o

laplace2d.o: laplace2d.c timing.h size.h
        gcc -c laplace2d.c

update_A.o: update_A.c laplace2d.h size.h
        gcc -c update_A.c

copy_A.o: copy_A.c laplace2d.h size.h
        gcc -c copy_A.c

.PHONY: clean
clean:
        rm -f laplace2d_exe *.o
```

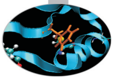
How it works

- The default goal is (re-)linking laplace2d_exe
- Before make can fully process this rule, it must process the rules for the files that edit depends on, which in this case are the object files
- The object files, according to their own rules, are recompiled if the source files, or any of the header files named as prerequisites, is more recent than the object file, or if the object file does not exist
- Note: in this makefile .c and .h are not the targets of any rules, but this could happen it they are automatically generated
- After recompiling whichever object files need it, make decides whether to relink edit according to the same “updating” rules.
- Try to follow the path: what happens if, e.g., laplace2d.h is modified?

A simple example in Fortran

- The main program is in laplace2d.f90 file
  - uses two modules named prec and timing
  - calls subroutines in two source files: update_A.f90 and copy_A.f90
- update_A.f90 and copy_A.f90 use only prec module
- sources of prec and timing modules are in the prec.f90 and timing.f90 files
- Beware of the Fortran modules:
  - program units using modules require the mod files to exist
  - a target may be a list of files: e.g., both timing.o and timing.mod depend on timing.f90 and are produced compiling timing.f90
- Remember: the order of rules is not significant, except for determining the default goal

A simple example in Fortran / 2



```
laplace2d_exe: laplace2d.o update_A.o copy_A.o prec.o timing.o
        gfortran -o laplace2d_exe prec.o timing.o laplace2d.o update_A.o copy_A.o

prec.o prec.mod: prec.f90
        gfortran -c prec.f90

timing.o timing.mod: timing.f90
        gfortran -c timing.f90

laplace2d.o: laplace2d.f90 prec.mod timing.mod
        gfortran -c laplace2d.f90

update_A.o: update_A.f90 prec.mod
        gfortran -c update_A.f90

copy_A.o: copy_A.f90 prec.mod
        gfortran -c copy_A.f90

.PHONY: clean
clean:
        rm -f laplace2d_exe *.o *.mod
```

Phony Targets and clean

- A phony target is one that is not really the name of a file; rather it is just a name for a recipe to be executed when you make an explicit request.
  - avoid target name clash
  - improve performance
- `clean`: an ubiquitous target

```
.PHONY: clean
clean:
        rm *.o temp
```

- Another common solution: since `FORCE` has no prerequisite, recipe and no corresponding file, make imagines this target to have been updated whenever its rule is run

```
clean: FORCE
        rm *.o temp
FORCE:
```

Variables

- The previous makefiles have several duplications
  - error-prone and not expressive
- Use variables!
  - define
    `objects = laplace2d.o update_A.o copy_A.o`
  - and use as **`$(objects)`**

```
objects := laplace2d.o update_A.o copy_A.o

laplace2d_exe: $(objects)
        gcc -o laplace2d_exe $(objects)

laplace2d.o: laplace2d.c timing.h size.h
        gcc -c laplace2d.c

update_A.o: update_A.c laplace2d.h size.h
        gcc -c update_A.c

copy_A.o: copy_A.c laplace2d.h size.h
        gcc -c copy_A.c

.PHONY: clean
clean:
        rm -f laplace2d_exe *.o
```

More Variables

- Use more variables to enhance readability and generality
- Modifying the first four lines it is easy to modify compilers and flags

```
CC       := gcc
CFLAGS   := -O2
CPPFLAGS :=
LDFLAGS  :=

objects  := laplace2d.o update_A.o copy_A.o

laplace2d_exe: $(objects)
        $(CC) $(CFLAGS) $(CPPFLAGS) -o laplace2d_exe $(objects) $(LDFLAGS)

laplace2d.o: laplace2d.c timing.h size.h
        $(CC) $(CFLAGS) $(CPPFLAGS) -c laplace2d.c

update_A.o: update_A.c laplace2d.h size.h
        $(CC) $(CFLAGS) $(CPPFLAGS) -c update_A.c

copy_A.o: copy_A.c laplace2d.h size.h
        $(CC) $(CFLAGS) $(CPPFLAGS) -c copy_A.c

.PHONY: clean
clean:
        rm -f laplace2d_exe *.o
```

Implicit rules

- There are still duplications: each compilation needs the same command except for the file name
  - immagine what happens with hundred/thousand of files!
- What happens if Make does not find a rule to produce one or more prerequisite (e.g., and object file)?
- Make searches for an implicit rule, defining default recipes depending on the processed type
  - C programs: n.o is made automatically from n.c with a recipe of the form

```
$(CC) $(CPPFLAGS) $(CFLAGS) -c
```

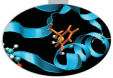
  - C++ programs: n.o is made automatically from n.cc, n.cpp or n.C with a recipe of the form

```
$(CXX) $(CPPFLAGS) $(CXXFLAGS) -c
```

  - Fortran programs: n.o is made automatically from n.f, n.F (`$(CPPFLAGS)` only for .F)

```
$(FC) $(FFLAGS) $(CPPFLAGS) -c
```

Pattern rules



- Implicit rules allow for saving many recipe lines
  - but what happens is not clear reading the Makefile
  - and you are forced to use a predefined structure and variables
  - to clarify the types to be processed, you may define **.SUFFIXES** variable at the beginning of Makefile

```
.SUFFIXES:
.SUFFIXES: .o .f
```

- You may use re-define an implicit rule by writing a pattern rule
  - a pattern rule looks like an ordinary rule, except that its target contains one character `%`
  - usually written as first target, does not become the default target

```
%.o : %.c
	$(CC) -c $(OPT_FLAGS) $(DEB_FLAGS) $(CPP_FLAGS) $< -o $@
```

- Automatic variables are usually exploited
  - `$@` is the target
  - `$<` is the first prerequisite (usually the source code)
  - `$^` is the list of prerequisites (useful in linking stage)

Running make targets

- It is possible to select a specific target to be updated, instead of the default goal (remember **`clean`**)

```
make copy_A.o
```

  - of course, it will update the chain of its prerequisite
  - useful during development when the full target has not been programmed, yet
- And it is possible to set target-specific variables as (repeated) target prerequisites
- Consider you want to write a Makefile considering both GNU and Intel compilers
- Use a default goal which is a help to inform that compiler must be specified as target

C example

```
CPPFLAGS  :=
LDFLAGS   :=

objects   := laplace2d.o update_A.o copy_A.o

.SUFFIXES :=
.SUFFIXES := .c .o

%.o: %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $<

help:
        @echo "Please select gnu or intel compilers as targets"

gnu: CC       := gcc
gnu: CFLAGS   := -O3
gnu: $(objects)
        $(CC) $(CFLAGS) $(CPPFLAGS) -o laplace2d_gnu $(objects) $(LDFLAGS)

intel: CC     := icc
intel: CFLAGS := -fast
intel: $(objects)
        $(CC) $(CFLAGS) $(CPPFLAGS) -o laplace2d_intel $(objects) $(LDFLAGS)

laplace2d.o: laplace2d.c timing.h size.h
update_A.o : update_A.c laplace2d.h size.h
copy_A.o   : copy_A.c laplace2d.h size.h

.PHONY: clean
clean:
        rm -f laplace2d_gnu laplace2d_intel $(objects)
```

SuperComputing Applications and Innovation

Fortran example

```
LDFLAGS :=
objects := prec.o timing.o laplace2d.o update_A.o copy_A.o
.SUFFIXES:
.SUFFIXES: .f90 .o .mod

%.o: %.f90
	$(FC) $(FFLAGS) -c $<
%.o %.mod: %.f90
	$(FC) $(FFLAGS) -c $<

help:
	@echo "Please select gnu or intel compilers as targets"
gnu: FC       := gfortran
gnu: FCFLAGS  := -O3
gnu: $(objects)
	$(FC) $(FCFLAGS) -o laplace2d_gnu $^ $(LDFLAGS)
intel: FC      := ifort
intel: FCFLAGS := -fast
intel: $(objects)
	$(FC) $(CFLAGS) -o laplace2d_intel $^ $(LDFLAGS)

prec.o prec.mod:     prec.f90
timing.o timing.mod: timing.f90
laplace2d.o:         laplace2d.f90 prec.mod timing.mod
update_A.o:          update_A.f90 prec.mod
copy_A.o:            copy_A.f90 prec.mod

.PHONY: clean
clean:
	rm -f laplace2d_gnu laplace2d_intel $(objects) *.mod
```

Defining variables

- Another way to support different compilers or platforms is to include a platform specific file (e.g., make.inc) containing the needed definition of variables

```
include make.inc
```

- Common applications feature many make.inc.<platform_name> which you have to select and copy to make.inc before compiling

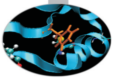
- When invoking `make`, it is also possible to set a variable

```
make OPTFLAGS=-O3
```

  - this value will override the value inside the Makefile
  - unless `override` directive is used
  - but override is useful when you want to add options to the user defined options, e.g.

```
override CFLAGS += -g
```

Variable Flavours



- The variables considered until now are called *simply expanded* variables, are assigned using `:=` and work like variables in most programming languages.
- The other flavour of variables is called *recursively expanded*, and is assigned by the simple `=`
  - recursive expansion allows to make the next assignments working as expected

```
CFLAGS       = $(include_dirs) -O
include_dirs = -Ifoo -Ibar
```

  - but may lead to unpredictable substitutions or even impossibile circular dependencies

```
CFLAGS       = $(CFLAGS) -g
```

- You may use **+=** to add more text to the value of a variable
  - acts just like normal **=** if the variable in still undefined
  - otherwise, exactly what **+=** does depends on what flavor of variable you defined originally
- Use recursive variables only if needed

Wildcards

- A single file name can specify many files using wildcard characters: `*`, `?` and `[...]`
- Wildcard expansion depends on the context
  - performed by make automatically in targets and in prerequisites
  - in recipes, the shell is responsible for
  - what happens typing **`make print`** in the example below? (The automatic variable **`$?`** stands for files that have changed)

```
print: *.c
       lpr -p $?
       touch print
```

  - if you define

```
objects = *.o
foo : $(objects)
       gcc -o foo $(objects)
```

  it is expanded only when is used and it is not expanded if no .o file exists: in that case, foo depends on a oddly-named `.o` file
  - use instead the `wildcard` function:

```
objects := $(wildcard *.o)
```

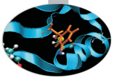
Conditional parts of Makefile

- Environment variables are automatically transformed into make variables
- Variables could be not enough to generalize rules
  - e.g., you may need non-trivial variable dependencies
- Immagine your application has to be compiled using GNU on your local machine mac_loc, and Intel on the cluster mac_clus
- You can catch the hostname from shell and use a conditional statement (**`$SHELL`** is not exported)

```
SHELL := /bin/sh
HOST  := $(shell hostname)
ifeq ($(HOST),mac_loc)
   CC     := gcc
   CFLAGS := -O3
endif
ifeq ($(HOST),mac_clus)
   CC     := icc
   CFLAGS := -fast
endif
```

- Be careful on Windows systems!

Directories for Prerequisites



- For large systems, it is often desirable to put sources and headers in separate directories from the binaries
- Using Make, you do not need to change the individual prerequisites, just the search paths
- A **vpath** pattern is a string containing a % character.
  - **%.h** matches files that end in **.h**

```
vpath %.c foo
vpath %   blish
vpath %.c bar
```

will look for a file ending in .c in foo, then blish, then bar

- using **vpath** without specifying directory clears all search paths associated with patterns

Directories for Prerequisites / 2

- When using directory searching, recipe generalizing is mandatory

```
vpath %.c src
vpath %.h ../headers
foo.o : foo.c defs.h hack.h
        gcc -c $< -o $@
```

- Again, automatic variables solve the problem
- And implicit or pattern rules may be used, too
- Directory search also works for linking libraries using prerequisites of the form `-lname`
- `make` will search for the file libname.so and, if not found, for libname.a first searching in vpath and then in system directory

```
foo : foo.c -lcurses
      gcc $^ -o $@
```

Advanced topics

- Functions, also user-defined
  - e.g., define objects as the list of file which will be produced from all .c files in the directory

```
objects := $(patsubst %.c,%.o,$(wildcard *.c))
```

  - e.g., sorts the words of list in lexical order, removing duplicate words

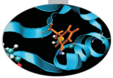
```
headers := $(sort math.h stdio.h timer.h math.h)
```

- Recursive make, i.e. make calling chains of makes
  - MAKELEVEL variable keeps the level of invocation

Standard Targets (good practice)

- all → Compile the entire program. This should be the default target
- install → Compile the program and copy the executables, libraries, and so on to the file names where they should reside for actual use.
- uninstall → Delete all the installed files
- clean → Delete all files in the current directory that are normally created by building the program.
- distclean → Delete all files in the current directory (or created by this makefile) that are created by configuring or building the program.
- check → Perform self-tests (if any).
- install-html/install-dvi/install-pdf/install-ps → Install documentation
- html/dvi/pdf/ps → Create documentation

Much more...



- Compiling a large application may require several hours
- Running make in parallel can be very helpful, e.g. to use 8 processes

```
make -j8
```

  - but not completely safe (e.g., recursive make compilation)
- There is much more you could know about **`make`**
  - this should be enough for your in-house application
  - but probably not enough for understanding large projects you could encounter

```
http://www.gnu.org/software/make/manual/make.html
```