1 Compilation and Linking Instructions

- C/C++ and fortran code needs to be compiled before it can be run. The compilation takes two steps: producing object (machine) code from the source code, and linking objects into executable. The commands are
  - compile: g++ [options] -c <source1>.cc
  - compile: g++ [options] -c <source2>.cc
  - link: g++ [options] -o <executable> <source1>.o <source2>.o
  - execute: ./<executable>

  If compiling a single source file, we can achieve boths steps with one command
  - compile&link: g++ [options] -o <executable> <source>.cc
  - execute: ./<executable>

  Options can be omitted, but we will many times use options for optimization (-O, or -O3) adding debugging information (-g), or additin profiling information (-p)
- For fotran, code we can use identical process, except ”g++” is replaced by fortran compiler, i.e., either gnu-fortram ”gfortran” or intel’s ”ifort”.

- Python is interpreter. The code does not need explicit compilation. By invoking Python interpreter, the code is compiled on the fly and executed at the same time
  - compile&execute: python <script>.py

  If we want to avoid invoking python interpreter explicitely, we need to do the following.
  - change script permission: chmod +x <script>.py
  - the first line needs to read:

    ```
    #!/usr/bin/env python
    ```
  - execute: ./<script>.py

Writing makefiles

It is a good practice to write a makefile for every project. Makefile typically contains information about the default compilers, location of necessary include files and necessary libraries to link to the executable.

There are many nice tutorials available on the Web including `http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/` or `http://www.tutorialspoint.com/makefile/index.htm` or `http://www.gnu.org/software/make/manual/`

We will briefly describe the steps in writing simple makefiles.

- The name of the makefile can be ”Makefile” or ”makefile” and is typically located in the same directory as other source files.
- User types ”make” in the source directory and makefile is executed producing the executable file.

Lets call our project `manc`. The C++ source file is `manc.cc`. The simplest makefile contains the following two lines

```
manc : manc.cc
        g++ -o manc manc.cc
```

Note: *Each line in the commands list must begin with a TAB character!*

- The dependency rule defines under what conditions a given file needs to be recompiled, and how to compile it.

The above rule states that the executable `manc` has to be recompiled whenever `manc.cc` is modified. The rule tells us that `manc` can be obtained by the command `g++ -o manc manc.cc`.

We can have multiple rules, which are executed recursively. By default, make always executes the first rule. The other rules are executed, if they are called by some other rule (starting from the first rule). If we give an argument to the make, make will start at the rule with such name.

Here is such example with multiple rules

```
all : manc manf     # if all does not exists, manc and manf are envoked

manc : manc.cc      # target : dependencies    // time1 > time2 -> execute
        g++ -o manc manc.cc    # commands

manf : manf.f90                    # target : dependencies
        gfortran -o manf manf.f90   # command
```

The first rule is `all`, and make will start evaluating it.

The first lines says that `all` depends on `manc` and `manf`. If the two files do not exist, make will create them by finding and executing rules for `manc` and `manf`. Even if the two files (`manc` & `manf`) exist, make will check if they are up to date, otherwise it will evaluate the rules. Up to date means that dependencies (on the right) are older than targets (on the left). For example, if `manc.cc` is newer than `manc`, the rule for `manc` will be evaluated even though `manc` exists. We could say that if the file does not exists, it is equivalent to be very old for the purpose of makefile rules evaluation.

Next we could define some constants for compiler names and compiler flags (optimization). For example

```
C++ = g++          # define variable C++
FORT = gfortran    # define variable F90
CFLAGS = -O3
FFLAGS = -O3

# rules below
all : manc manf    # target : dependencies

manc : manc.cc      # target : dependencies    // time1 > time2 -> execute
        $(C++) $(CFLAGS) -o manc manc.cc    # commands

manf : manf.f90                    # target : dependencies
        $(FORT) $(FFLAGS) -o manf manf.f90   # command
```

This is useful for porting makefiles to different computer, as only a few variables needs to be changed on different system.

Most makefiles have a rule named `clean`. This will remove all object files and all executables, so that a fresh compilation can be started after clean is invoked. We would add a rule like that

```
clean :
        rm -f manc manf
```

Notice that the dependency list is empty, hence the rule is always executed when invoked.

To invoke the `clean` rule, we need to call make with the argument: `make clean`

make also defined many special variables, such as `$@, $<, $*`. The variable `$@` stands for the target on the left hand side, and `$<` is the first item in the dependency list. We could rewrite the `manc` and `manf` rules in the following way:

```
manc : manc.cc      # target : dependencies    // time1 > time2 -> execute
        $(C++) $(CFLAGS) -o $@ $<

manf : manf.f90                    # target : dependencies
        $(FORT) $(FFLAGS) -o $@ $<
```

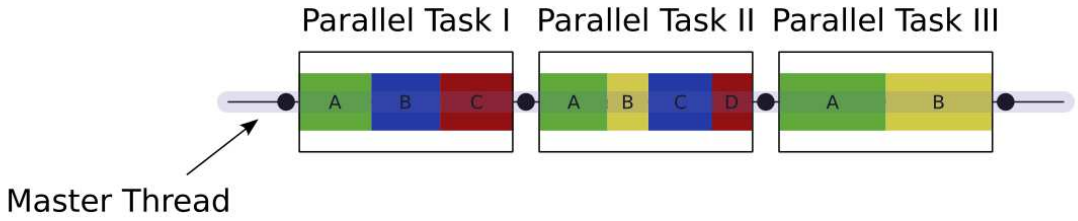
Finally, often we have many C++ and fortran files, which need to be compiled in a very similar way. It is useful to write generic rules to obtain a file named `xxx.o` from a corresponding `xxx.f90` file. We can achieve that by so called pattern rules, which can be added at the end of the makefile.

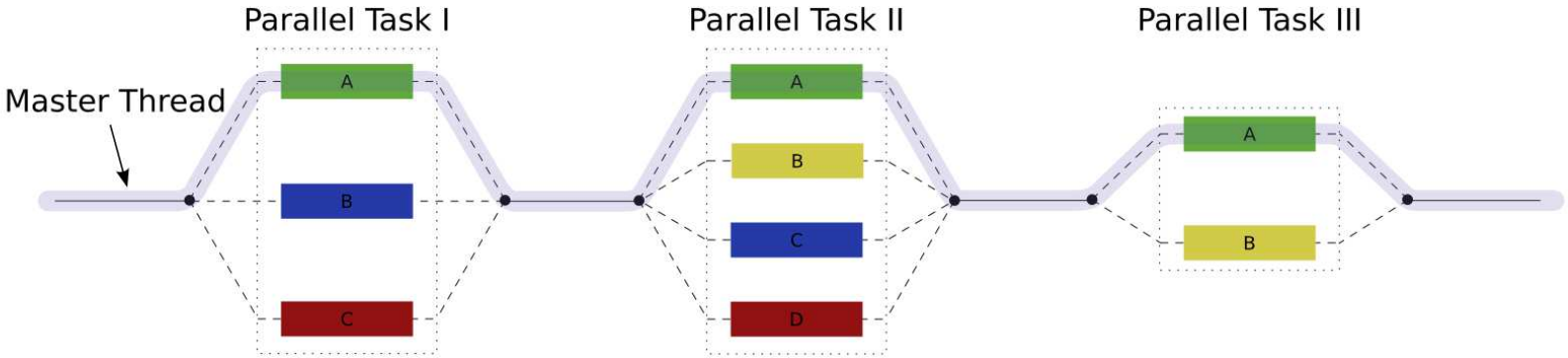
```
#.....................
# Pattern rules
#.....................
%.o : %.f90
        $(FORT) $(FFLAGS)  -c $< -o $@

%.o : %.cc
        $(CC) $(CFLAGS) -c $< -o $@
```

Now we do not need to write a rule for obtaining `xxx.o` from `xxx.f90` or `xxx.cc`.

open_mp to speed up C++ and Python code





- OpenMP is designed for multi-processor/core to run a program on several cores (using several ”threads”)
- OpenMP programs accomplish parallelism exclusively through the use of threads. Typically, the number of threads match the number of machine processors/cores.

However, the actual use of threads is up to the application.

- OpenMP is a shared memory programming model, most variables in OpenMP code are visible to all threads by default.
- But sometimes private variables are necessary to avoid race conditions. and there is a need to pass values between the sequential part and the parallel region.
- OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
- Parallelization can be as simple as taking a serial program and inserting compiler directives.... Or as complex as inserting subroutines to set multiple levels of parallelism, locks and even nested locks.

The simplest case of parallel mandelbrot calculation:

```
#pragma omp parallel for
for (int i=0; i<Nx; i++){
  for (int j=0; j<Ny; j++){
    double x = -2 + 3.*i/(Nx-1.);
    double y = -1 + 2.*j/(Ny-1.);
    mand[i*Ny+j] = Mandelb( complex<double>(x,y), max_iterations);
  }
}
```

The loop over i is parallelized. Each core is calculating different i term.

Note that `mand` is shared across all cores, because each core is changing its own slice of the array.

Note that `x` and `y` must be different on each core. As they are declared inside the loop, compiler makes them private to each core.

In more general case, the `omp_parallel` statement is

```
#pragma omp parallel shared(x,y) private(beta,pi)
```

By default all variables are shared, hence `shared` statement is not really needed.

The same loop in fortran would look like

```
!$OMP PARALLEL DO  PRIVATE(j,x,y,z0)
do i=1,Nx
   do j=1,Ny
      x = -2.+3.*(i-1.)/(Nx-1.)
      y = -1.+2.*(j-1.)/(Ny-1.)
      z0 = dcmplx(x,y)
      mand(i,j) = Mandelb(z0, max_steps)
   enddo
enddo
!$OMP END PARALLEL DO
```

Note that in fortran all variables are declared at the top of the program, hence x, y, $z0$, j need to be declared private. Also i is private, but the first loop counter does not need to be added to the private list, as compiler will add it.

The code is compiled by adding a flag `-fopenmp`:

```
g++ -fopenmp -O3 -o mandc mandc.cc
```

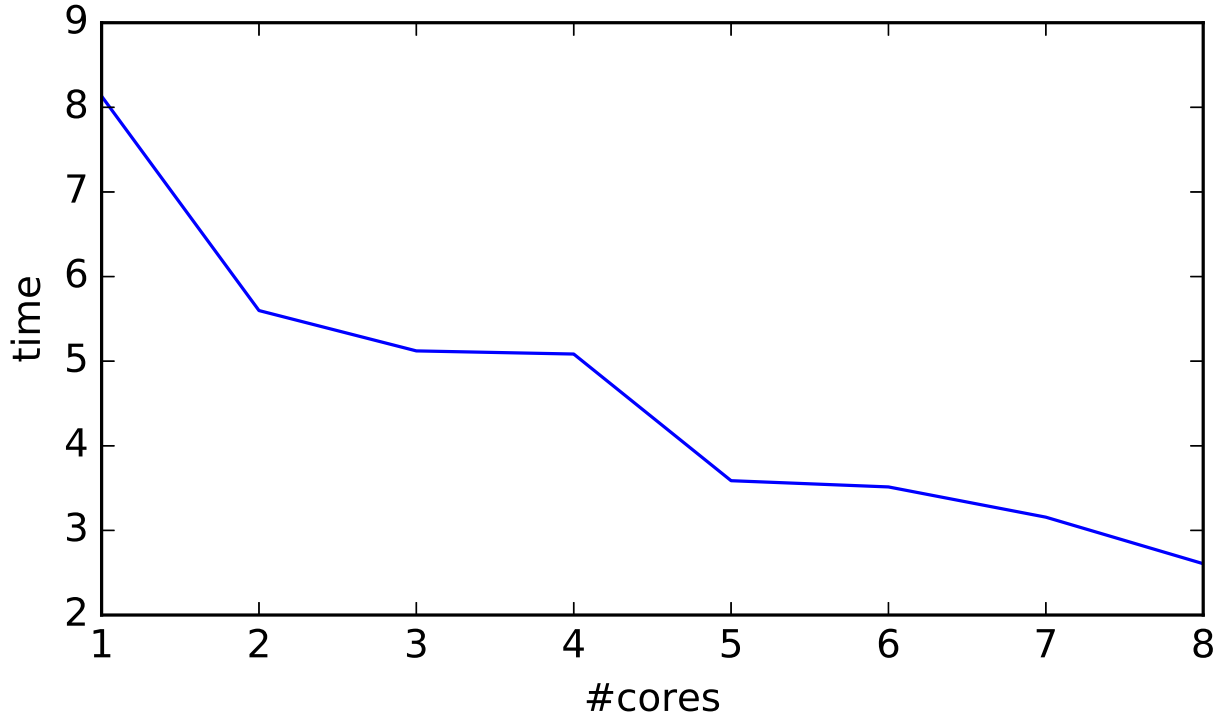
and

```
gfortran -fopenmp -O3 -o mandf mandf.f90
```

Also the environment variable `OMP_NUM_THREADS` should be set to the number of cores (threads) we want to use. We can issue a command

```
export OMP_NUM_THREAS=4
```

Example of time for mandelbrot set on multiple cores for `Intel Core i7` processor:



If you want to learn more about `open_mp`, you should read examples at

`http://openmp.org/mp-documents/OpenMP4.0.0.Examples.pdf`

GPU acceleration should be supported in `open_mp`.4.0, but current examples do not yet contain it. Most of compilers have limited support for GPU acceleration at this point, but this could change very shortly.

Python tricks to speedup the code

With **numpy** and **scipy** package, Python is one of the best languages for numerics.

But, it is slow!

Not, if combined with C++/Fortran!

The idea: *Write most of the code in Python. Allocate all arrays in Python, to avoid annoying bookeeping of allocation/deallocation of memory. Speed-up the innermost loop by fortran/C++.*

Great tools to ”glue” fotran code with Python: **f2py**.

Many tools to ”glue” C++ with Python. The simplest to use comes with **scipy**: **weave**. We will also discuss **pybind11**, which is very powerful, but somewhat harder to use.

Others:

- weave : used to be part of scipy, and is still among standard Python packages. Very simple to use. But code is a string, which is very clumsy for writting more than 10 lines of code.
- Pybind11 :`https://github.com/pybind/pybind11`
  very powerful for C++ to Python-library conversion. Needs newer C++-11 compiler. It requires only a few header files, and no libraires or compilation. Efficient, and not too hard to use.
- Swig : very general. It can glue almost everything with everything. It is demanding to master.
- python-cxx : smaller, intented only for C/C++ $<-> $ Python conversion. Looks quite simple, but very limited numpy support.
- Cython : `http://cython.org/`
  like a new compiler for python. We do not write real C++ code, but code similar to C++, which is being compiled. However, it needs separate installation. Some (but limited) support for numpy/scipy. (This might have improved recently.)

2 f2py

- Step 1: Create fortran subroutine, and compile it with fortran compiler.
- Step 2: Add special f2py directives to fortran code for more user-friendly modules.
- Step 3: Create Python module by f2py:

```
f2py -c <source-name> -m <module-name> <libraries>
```

- Step 4: Include module in Python, and use it as python function.

2.1 Mandelbrot

```
SUBROUTINE Mandelb(data, ext, Nx, Ny, max_iterations)
  IMPLICIT NONE ! Don't use any implicit names of variables!
  ! Function arguments
  REAL*8, intent(out) :: data(Nx,Ny)
  REAL*8, intent(in) :: ext(4)
  INTEGER, intent(in) :: max_iterations
  INTEGER, intent(in) :: Nx, Ny
  !f2py integer optional, intent(in)          :: max_iterations=1000
  !  !f2py integer intent(hide), depend(data) :: Nx=shape(data,0)
  !  !f2py integer intent(hide), depend(data) :: Ny=shape(data,1)
  ! Local variables
  INTEGER    :: i, j, itt
  COMPLEX*16 :: z0, z
  data(:,:) = 0.0
  DO i=1,Nx
     DO j=1,Ny
        z0 = dcmplx( ext(1) + (ext(2)-ext(1))*(i-1)/(Nx-1.), ext(3) + (ext(4)-ext(3))*(j-1)/(Ny-1.) )
        z=0
        DO itt=1,max_iterations
           IF (abs(z)>2.) THEN
              data(i,j) = 1./itt            ! result is number of iterations
              EXIT
           ENDIF
           z = z**2 + z0              ! f(z) = z**2+z0 -> z
        ENDDO
     ENDDO
  ENDDO
  RETURN
END SUBROUTINE Mandelb
```

Note `!f2py` directives. They can be used to specify optional parameters, sizes of arrays,...

To check if the code is free from grammatical errors, do

```
ifort -c mandel.f90
```

If successful, use f2py to get pythom module, mandel.so:

```
f2py -c mandel.f90 -m mandel
```

If your f2py does not find the right fortran compiler, you might need to add option `--fcompiler=intel` or `--fcompiler=intelem`

Open python interpreter `ipython` and check the module:

```
import mandel
print mandel.__doc__
```

This should give you help on how the fortran subroutine was converted. Should be something like:

```
This module 'mandel' is auto-generated with f2py (version:2_4
Functions:
  data = mandelb(ext,nx,ny,max_iterations=1000)
```

Now we have python function, which can be used to plot mandelbrot set:

```
from scipy import *
from pylab import *
import mandel  # importing module created by f2py

# The range of the mandelbrot plot [x0,x1,y0,y1]
#ext=[-2,1,-1,1]
ext=[-1.8,-1.72,-0.05,0.05]

data = mandel.mandelb(ext,400,400).transpose()

# Using python's pylab, we display pixels to the screen!
imshow(data, interpolation='bilinear', cmap=cm.hot, origin='lower', extent=ext, aspect=1.)
colorbar()
show()
```

3 weave

The idea of weave is to write C/C++ code directly inside python script (much like we used to insert assembler code inside C code).

Optimal use of hardware & software

```
from pylab import *
from scipy import weave

# The C++ code is written in a multiline string.
# It will be compiler, when run for the first time. Next time,
# it will just use the "old" executable.
code="""
     using namespace std;

     return_val=0;
     for (int i=0; i<max_iterations; i++){
         if (norm(z)>4.) { return_val= 1./i; break; }
         z = z*z + z0;     // if |z|>2 the point is not part of mandelbrot set
     }
     """

Nx = 400
Ny = 400
data = zeros((Nx,Ny)) # allocate all arrays in Python!
max_iterations=1000
ext=[-1.8,-1.72,-0.05,0.05] # The range of the mandelbrot plot [x0,x1,y0,y1]


# The double loop is written in Python, which takes some time.
# Not the most optimal code.
for i in range(Nx):
    for j in range(Ny):
        z0 = ext[0] + (ext[1]-ext[0])*i/(Nx-1.) + 1j*(ext[2] + (ext[3]-ext[2])*j/(Ny-1.) )
        z = 0j
        # This line compiles and executed the code.
        # First argument - the code, second - all necessary variables, the rest - options.
        data[i,j] = weave.inline(code, ['max_iterations', 'z0', 'z'],
                                 type_converters=weave.converters.blitz, compiler = 'gcc')

data = data.transpose() # for plotting, need to transpose

# Using python's pylab, we display pixels to the screen!
imshow(data, interpolation='bilinear', cmap=cm.hot, origin='lower', extent=ext, aspect=1.)
colorbar()
show()
```

This is faster then Python, but the double loop is still expensive in Python. We can further optimized by coding the double loop in C++:

```
from pylab import *
from scipy import weave

# The C++ code
code="""
     using namespace std; // for using cout when debugging
     for (int i=0; i<Nx; i++){
         for (int j=0; j<Ny; j++){
             complex<double> z0( ext(0)+(ext(1)-ext(0))*i/(Nx-1.), ext(2)+(ext(3)-ext(2))*j/(Ny-1.) );
             complex<double> z=0.0;
             for (int itt=0; itt<max_iterations; itt++){
                 if (norm(z)>4.) { data(i,j)=1./itt; break; }
                 z = z*z + z0;     // if |z|>2 the point is not part of mandelbrot set
             }
         }
     }
     """

Nx = 400
Ny = 400
data = zeros((Nx,Ny), 'd')
max_iterations=1000
ext=array([-1.8,-1.72,-0.05,0.05]) # The range of the mandelbrot plot [x0,x1,y0,y1]

weave.inline(code, ['data', 'Nx', 'Ny', 'max_iterations', 'ext'],
             type_converters=waeve.converters.blitz, compiler = 'gcc')

# Using python's pylab, we display pixels to the screen!
data = data.transpose()
imshow(data, interpolation='bilinear', cmap=cm.hot, origin='lower', extent=ext, aspect=1.)
colorbar()
show()
```

4 pybind11

Pybind11 is just a collection of header files (*.h files to be included in your C++ code). You can downloaded them from `https://github.com/pybind/pybind11`. You might want to read the manual :
`https://media.readthedocs.org/pdf/pybind11/master/pybind11.p`

One can produce stand-alone C++ code, in which the “main” function is replaced by the pybind11 wrapper code (a few lines of code), and then one can compile the code into Python module.

The code can have a form like:

Optimal use of hardware & software

```cpp
#include "pybind11/pybind11.h"
#include "pybind11/numpy.h"
#include "pybind11/stl.h"
#include <cstdint>

namespace py = pybind11;
using namespace std;

void mand(py::array_t<double>& data, int Nx, int Ny, int max_steps, const vector<int>& ext)
{
  auto dat = data.mutable_unchecked<2>();
  #pragma omp parallel for
  for (int i=0; i<Nx; i++){
    for (int j=0; j<Ny; j++){
      double x = ext[0] + (ext[1]-ext[0])*i/(Nx-1.);
      double y = ext[2] + (ext[3]-ext[2])*j/(Ny-1.);
      complex<double> z0(x,y);
      complex<double> z=0;
      for (int itr=0; itr<max_steps; itr++){
        if (norm(z)>4.){
          dat(j,i) = itr;
          break;
        }
        z = z*z + z0;
      }
      if (norm(z)<4.) dat(j,i) = max_steps;
    }
  }
}

PYBIND11_PLUGIN(imanc){
  py::module m("imanc", "pybind11 wrap for mandelbroat class");
  m.def("mand", &mand);
}
```

To compile the code, and produce python model “imanc.so” you can create makefile, which will have the form

```
CC = g++-7
PYBIND = -I<Python_path>/include/python2.7
PYLIBS = -L<Python_path>/lib/python2.7/config -lpython2.7 \
         -ldl -framework CoreFoundation
imanc.so : imanc.cc
        $(CC) $(PYBIND) -O3 -fopenmp -shared -std=c++11 \
              imanc.cc -o imanc.so $(PYLIBS)
```

Note that we were able to use numpy array for data, which was accessed in C++. It is also quite straighforward to convert blitz arrays into numpy arrays with this tool.

You should be able to get the correct paths for your platform by typing:

```
python-config --includes
python-config --libs
```

These instructions might be slightly outdated, as pybind11 is evloving very rapidly. Consult their manual, if these instructions do not work for you.

Hardware

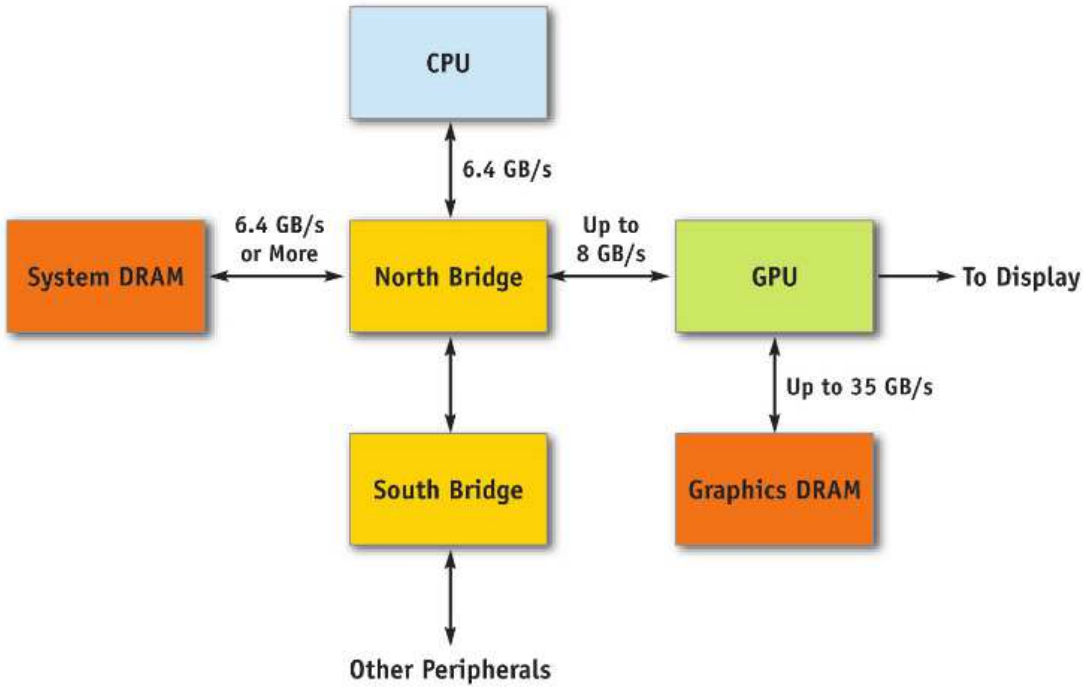
- The central unit of a computer is CPU (Central processing unit). It consists of a few very high speed memory units called registers. Nowadays CPU has many cores - even 16.
- Closest to the CPU is a small part of memory, called Cache. Nowadays, typical size of the Cache is 1MB. The Cache is very important for performance (difference between Pentium and Celeron).
- RAM is the cental memory unit which can be accessed randomly. It is fast, but orders of magnitude slower than cache, which is orders of magnitude slower than registers.
- Hard disc is the slowest memory unit where the data can be (more or less) permanently stored.

ADVICE: *Use the fastest memory you can*

5 GPU

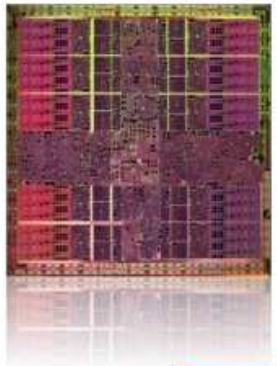
Recently, a new type of high performance computing is emerging - computing by Graphical Processing Unit (GPU). Nvidia developed the TESLA unit, and ATI and AMD developed FireStream.

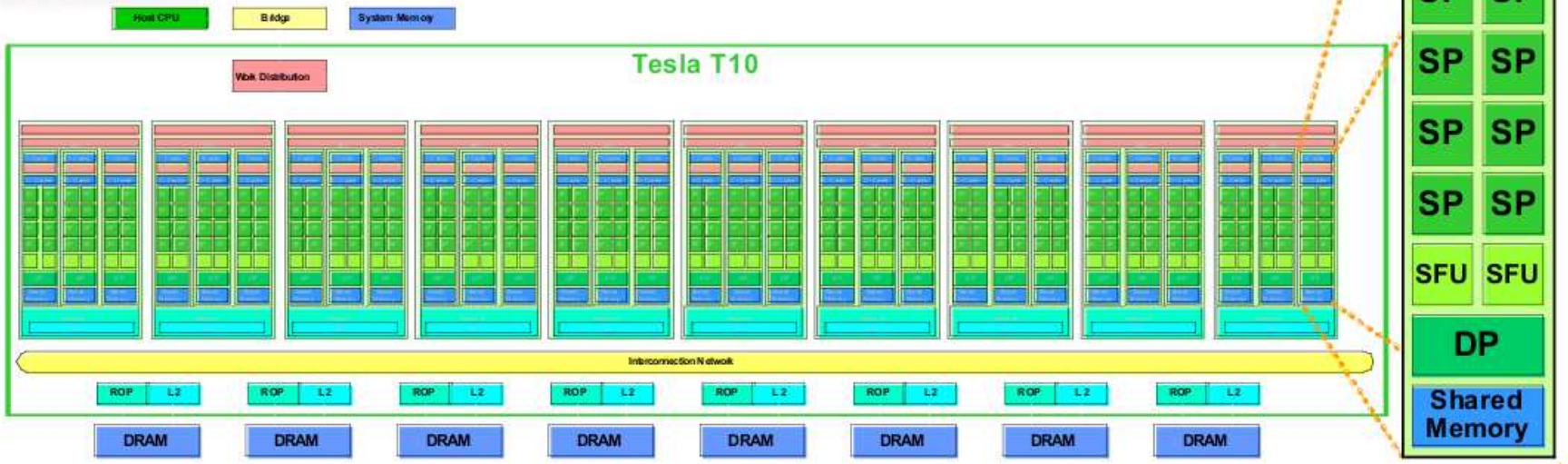
The GPU's have outperformed CPU's in terms of speed. The speed of each unit (CPU/GPU) is not increasing so much with time. But the number of processing units is increasing. GPU contains > 200 processors.



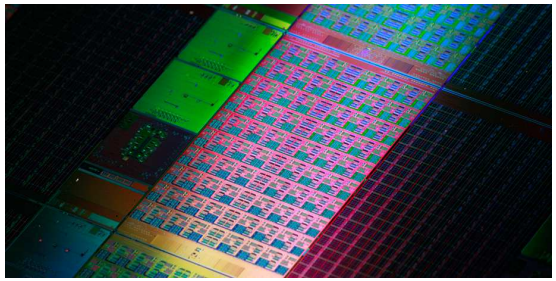
CUDA Computing with Tesla T10

- 240 SP processors at 1.45 GHz: 1 TFLOPS peak
- 30 DP processors at 1.45Ghz: 86 GFLOPS peak
- 128 threads per processor: 30,720 threads total





6 Intel's Many Integrated Core (MIC) Architecture/Xeon Phi



New/Old name : Xeon Phi/MIC

Comparison between MIC & GPU:

- MIC cores are general purpose cores just like CPUs, while GPU cores have limited instruction set [use of openMP versus CUDA]
- MIC cores have more memory per core, i.e., Knights Ferry (version released in 2010) has 32 KB of L1 cache, 256 KB of L2 cache, and 64MB of GDDR5 memory per core. GPU's typically have 1KB of registry and 6MB of additional memory per core.
- GPUs have more cores than MICs. Tesla typically includes 512 cores while Knights Ferry contains 32 and Knights Corner 50 cores.

7 Optimization

- Do not use hard-disc for data manipulation. Keep data in RAM. If you need a lot of RAM, estimate weather it fits into RAM. Rethink your algorithm before you start writing data to hard-disc.
- Try avoiding random access of data in RAM to reduce cache misses.
- The data which you need in the innermost loop should be stored in a way that the access is maximally continuous.

Typical example is a matrix manipulation.

In C or C++, one needs to access multidimensional arrays in the following order

```
for (int i=0; i<size; i++)
  for (int j=0; j<size; j++)
    A[i][j] = .....
```

since the data is stored in a row major order. In Fortran, the same loop should be written in the following way

```
do i=1, size
  do j=1, size
    A(j,i) = .....
  enddo
enddo
```

This is because Fortran uses column major storage. The figure explains it all.

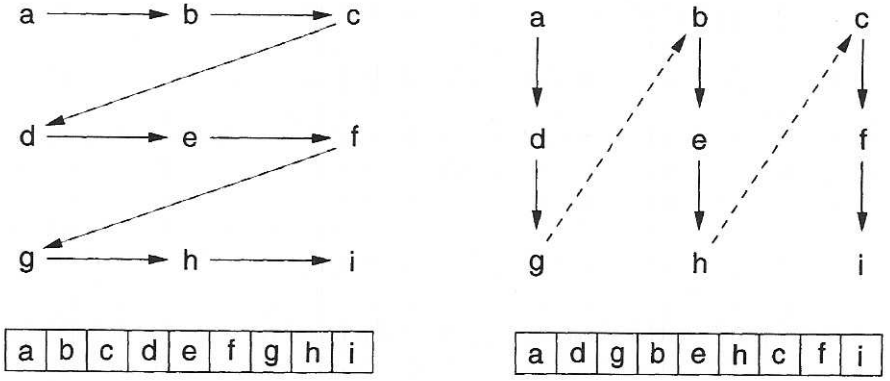


Fig. 15.2 (*Left*) Row-major order used for matrix storage in C and Pascal; (*right*) column-major order used for matrix storage in Fortran. On the bottom is shown how successive matrix elements are stored in a linear fashion in memory.

Remember: *The innermost index j runs:*

in C++ A[]...[j]

in Fortran A(j,....)

Scientific programs are usually very tuned for performance. This usually goes in expense of portability and readability.

ADVICE: *Newer optimize those parts of the program which are not very crucial for speed. In typical applications, only 20% of the code spends 80% of the time. Optimize only those 20% of the code. Make the rest of the code more readable.*

Remember: *80/20 rule*

Available numeric software

Numerous libraries are available on the Web. Sometimes it is hard to decide which one to use.

The current situation: Most of the best numeric algorithms are still written in Fortran $\rightarrow$ need to learn how to use them.

One of the most comprehensive archives is GAMS at `http://gams.nist.gov/`:

GAMS info:

Public access repository NETLIB is located at Oak Ridge National Laboratory, Knoxville, TN and AT&T Bell Laboratories, Murray Hill, NJ

It includes 111 packages and 8792 problem-solving modules (routines)

Optimal use of hardware & software

Search for software according to

- what problem it solves.
- package name.
- module name.
- text in module abstracts.

Go straight to the problem decision tree.

- A Arithmetic, error analysis
- B Number theory
- C Elementary and special functions (search also class L5)
- D Linear Algebra
- E Interpolation
- F Solution of nonlinear equations
- G Optimization (search also classes K, L8)
- H Differentiation, integration
- I Differential and integral equations
- J Integral transforms
- K Approximation (search also class L8)
- L Statistics, probability
- M Simulation, stochastic modeling (search also classes L6 and L10)
- N Data handling (search also class L2)
- O Symbolic computation
- P Computational geometry (search also classes G and Q)
- Q Graphics (search also class L3)
- R Service routines
- S Software development tools
- Z Other

Some of the libraries are unfortunately commercial (line NAG). Check weather they are

installed on the system (at the moment we do not have NAG license). Do *not install libraries* if they are already installed (like blas and lapack) because they are probably much more optimized than your compiled code can be.

8 Mixing Fortran and C++

Let's try to use fortran code inside C++ code.

Suppose we need to calculate erfc function of complex argument. GAMS finds many available routines. Let's pick the one from TOMS package. The header looks like this

Optimal use of hardware & software

```
C       ALGORITHM 680, COLLECTED ALGORITHMS FROM ACM.
C       THIS WORK PUBLISHED IN TRANSACTIONS ON MATHEMATICAL SOFTWARE,
C       VOL. 16, NO. 1, PP. 47.
        SUBROUTINE WOFZ (XI, YI, U, V, FLAG)
C
C  GIVEN A COMPLEX NUMBER Z = (XI,YI), THIS SUBROUTINE COMPUTES
C  THE VALUE OF THE FADDEEVA-FUNCTION W(Z) = EXP(-Z**2)*ERFC(-I*Z),
C  WHERE ERFC IS THE COMPLEX COMPLEMENTARY ERROR-FUNCTION AND I
C  MEANS SQRT(-1).
C  THE ACCURACY OF THE ALGORITHM FOR Z IN THE 1ST AND 2ND QUADRANT
C  IS 14 SIGNIFICANT DIGITS; IN THE 3RD AND 4TH IT IS 13 SIGNIFICANT
C  DIGITS OUTSIDE A CIRCULAR REGION WITH RADIUS 0.126 AROUND A ZERO
C  OF THE FUNCTION.
C  ALL REAL VARIABLES IN THE PROGRAM ARE DOUBLE PRECISION.
C
C
C  THE CODE CONTAINS A FEW COMPILER-DEPENDENT PARAMETERS :
C     RMAXREAL = THE MAXIMUM VALUE OF RMAXREAL EQUALS THE ROOT OF
C                RMAX = THE LARGEST NUMBER WHICH CAN STILL BE
C                IMPLEMENTED ON THE COMPUTER IN DOUBLE PRECISION
C                FLOATING-POINT ARITHMETIC
C     RMAXEXP  = LN(RMAX) - LN(2)
C     RMAXGONI = THE LARGEST POSSIBLE ARGUMENT OF A DOUBLE PRECISION
C                GONIOMETRIC FUNCTION (DCOS, DSIN, ...)
C  THE REASON WHY THESE PARAMETERS ARE NEEDED AS THEY ARE DEFINED WILL
C  BE EXPLAINED IN THE CODE BY MEANS OF COMMENTS
C
C
C  PARAMETER LIST
C     XI     = REAL      PART OF Z
C     YI     = IMAGINARY PART OF Z
C     U      = REAL      PART OF W(Z)
C     V      = IMAGINARY PART OF W(Z)
C     FLAG   = AN ERROR FLAG INDICATING WHETHER OVERFLOW WILL
```

First, we need to compile the Fortran code to produce object file:

```
ifort -c erfc.f
```

If it compiles, we can try to call it from C++. We need to write a prototype for Fortran routine. It is essential, to convert Fortran types to C types correctly. Here are some conversion rules

(Source: `http://www.astro.indiana.edu/~jthorn/c2f.html`)

- use pointers to arguments (in Fortran everything is pointer)
- Name Mangling
  - names of routines should be lowercase
  - in many platforms necessary to add underscore to function names
- Fortran subroutines is equivalent to C function returning void
- Conversion of types:
  - INTEGER $\rightarrow$ int
  - LOGICAL $\rightarrow$ int
  - REAL $\rightarrow$ float
  - DOUBLE PRECISION==REAL*8 $\rightarrow$ double

  - Fortran subroutine or function $\rightarrow$ pointer to a function (Name Mangling applies)
  - CHARACTER $\rightarrow$ usually char* (but be careful)

- Fortran starting index 1 goes to C/C++ starting index 0
- Due to column/row major convention in Fortran/C, the matrixes look transposed

Using the above rules on the fortran code

```
SUBROUTINE WOFZ (XI, YI, U, V, FLAG)
 IMPLICIT DOUBLE PRECISION (A-H, O-Z)
 LOGICAL FLAG
```

we prepare corresponding C++ prototype

```
extern "C"{
  void wofz_(double* xi, double* yi, double* ui, double* vi, int* flag);
}
```

A wrapper C++ function, hides the details of the call

```
std::complex<double> erfc(const std::complex<double>& z)
// C++ wrapper function for call to fortran routine to calculate erfc of a complex number
{
  int info;
  double rz = z.real(), iz = z.imag();
  double ru, iu;
  wofz_(&rz, &iz, &ru, &iu, &info);
  if (info!=0) std::cerr <<"Can't compute error function at "<<z<<" got "<<ru<<","<<iu<<" info="<<info<<std::endl;
  return std::complex<double>(ru,iu); // return value optimization!
}
```

Finally, we can compile C++ code and link together

```
g++ -o erfc cerfc.cc erfc.o
```

which is equivalent to

```
g++ -c cerfc.cc
g++ -o erfc cerfc.o erfc.o
```

See the available ifortran/erfc program for details.

LAPACK & BLAS

If you need to solve a standard linear algebra problem don't even think of writing your own routine and don't take it from your friend's program. It will be orders of magnitude slower. There are very optimized and FREE libraries called blas and lapack.

Blas stands for "Basic Linear Algebra". The basic information is available at `http://www.netlib.org/lapack/lug/node145.html` and also many other locations. Those two libraries are already installed on most of computers (including our cluster). If they are not, consider the following few distributions

- ATLAS: Automatically Tuned Linear Algebra Software

`http://math-atlas.sourceforge.net/`

- GotoBLAS: `http://www.tacc.utexas.edu/resources/software/`

There are ”Level 1”, ”Level 2” and ”Level 3” BLAS routines.

- Level 1 implements vector-vector operations
- Level 2 implements matrix-vector operations
- Level 3 implements matrix-matrix operations

The computation time increases as N for ”Level 1”, as N^2 for ”Level 2” and as N^3 for ”Level 3”. It is crucial to use ”Level 3” routines because they are truly faster than an naive implementation can be. On the other hand, ”Level 1” routines do not give considerable speed improvement and are therefore not mandatory to use. Typical routine from ”Level 3” BLAS is dgemm, which implements matrix multiplication for real matrix (zgemm for complex matrix).

LAPACK is built on top of blas and implements more sophisticated linear algebra operations including solving

- system of linear equations

- linear least square problem
- eigenvalue problem
- singular value problem

For more information, check out `http://www.netlib.org/lapack/lapackqref.ps` and for more detailed explanation take a look at `http://www.cs.colorado.edu/$\sim$jessup/lapack/documentation`

When using LAPACK or BLAS routine for the first time, it is crucial to checks the answer against Mathematica or using a simple example where you know the answer. There are 100 places where something might go wrong when calling BLAS or LAPACK routine from C or C++.

A good idea is to implement wrapper routines (or member routines of your class) which call LAPACK.

Example:

In sections ”Hartree-Fock” and ”Density Functional Theory”, we will need a routine to solve a generalized eigenvalue problem for a real symmetric matrix $\mathbf{A}x = E\mathbf{O}x$

Typing

```
man dsygvd
```

gives the following documentation

```
NAME
       DSYGVD - compute all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite
       eigenproblem, of the form A*x=(lambda)*B*x, A*Bx=(lambda)*x, or B*A*x=(lambda)*x
SYNOPSIS
       SUBROUTINE DSYGVD( ITYPE, JOBZ, UPLO, N, A, LDA, B, LDB, W, WORK, LWORK, IWORK, LIWORK, INFO )
           CHARACTER      JOBZ, UPLO
           INTEGER        INFO, ITYPE, LDA, LDB, LIWORK, LWORK, N
           INTEGER        IWORK( * )
           DOUBLE         PRECISION A( LDA, * ), B( LDB, * ), W( * ), WORK( * )
```

A prototype for subroutine DSYGVD can be expressed by

```
extern "C"
  void dsygvd_(const int* ITYPE, const char* JOBZ, const char* UPLO, const int* N,
               double* A, const int* LDA, double* B, const int* LDB, double* W, double* WORK, const int* LWORK,
               int* IWORK, const int* LIWORK, int* INFO);
```

and a typical call to the subroutine from a C program is

```
dsygvd_(&itype, "V", "U", &N, A, &lda, B, &ldb, eigenval, work, &lwork, iwork, &liwork, &info);
```

Complete implementation can be found in the section on the Hartree-Fock method.

Tools for detecting errors in the code

Debugger

Usually we have many debugers available on a computer. Unfortunately not every debugger is compatible with every compiler.

- For gnu compiler gcc or g77 the best choice is gdb (also part of gnu project and therefore fully compatible).
- For intel compiler ifort the appropriate debugger is idb
- On other non-linux system you will most probabbly find different kind of native compiler and corresponding debuger which is compatible with it.

To debug your program, you need to recompile the code with -g option added (For example: ”g++ -g -c mycode.cc”).

It is convenient to use debuger in a tex editor like emacs. Emacs can displays the source in a separate buffer and displays a pointer to shows you which line of source code will be executed next

- To start gdb within emacs, say M-x gdb <Enter>, edit the gdb execution line and say <Enter> again.
- Set a breakpoint (for example ”b main” meaning break in program main), and type ”r” and <Enter> to run the program. If your program needs input arguments, add them to ”r” command (”r myparam=something”).
- When the program stops, a second emacs buffer window will be created containing the source code with a pointer “=>” indicating the next line of code to be executed. This is your first breakpoint.
- To learn more about available commands, you can type ”help” in debugger buffer.
- Most often used commands are
  - ”n[ext]” (Single-step without descending into functions)
  - ”s[tep]” (Single-step descending into functions)
  - ”p[rint] <variable>” (Prints the content of the variable)

- ”b[reak]” <line number> (Set a breakpoint at line number)
- ”b[reak]” <function name> (Set a breakpoint at function)
- ”b[reak]” <class::name> (Set a breakpoint at class member function)
- ”b[reak]” <class::tab> (Lists members in class)
- ”i[nfo] b[reak]” (Lists breakpoints currently set)
- ”d[elete] 1” (Delete breakpoint number 1)
- ”dis[able] 1” (Disable breakpoint number 1)
- ”en[able] 1” (Enable breakpoint number 1)
- ”c[ontinue]” (Continues to next breakpoint or end)
- ”fin[ish]” (Finish current function, loop, etc.)
- ”x/10f <pointer to memory>” (interprets data following pointer adress as 10 floating point numbers and prints them)
- ”q[uit]” (Leaving debugger)

For more information, google gdb! There are hundreds of nice tutorials. (for example: `http://www-ccrma.stanford.edu/~jos/pasp/Executing_gdb_Emacs.h`
If you want to use Intel debugger ”idb” inside emacs, add the followng lines to your ”.emacs”

file

```
(load-file "/opt/intel_idb_73/bin/idb.el")
```

You need to replace the path to file ”idb.el” with your path. Then say M-x idb <Enter> and continue debugging.

Profiler

Debuggers can help you locate your mistakes. But if your program is working properly but is very slow, you need to speed it up!

Profilers show you how much time is spend in each subprogram or subroutine and (if necessary) even in each line of your code.

If you still remember 80/20 rule you can understand now why profilers are so important.

A good strategy is to write a scientific program which is very readable (highly intuitive and understandable) and not necessary fast or optimized for speed. It usualy takes you much less time to write simple non-optimized code. Then it comes profiling and optimization of those few lines of code that spend most of time. You are going to be surprised many times that the actual time spent in many parts of the code, you believed are crucial for speed, is completely negligable (less than 1% of all time).

”gprof” is very simple profiler (part of gnu package fully compatible with gdb) and most of the time sufficient for our needs.

- Recompile your code with -p option added. If you wish more precise information, add also -g option. The reason is that optimization usually inlines many function calls and then they are not ”seen” by gprof as separate units. The problem, however, is that program compiled with ”-O3” or ”-g” in general does not spend proportional time in each part of the program (some parts can be more optimized than others). Consequently, ”-g” will not always show precise time spent in each subprogram in case of ”-O3” execution. A good strategy is to first profile with -g and then check your results with -O3 option as well.
- execute your program as usual. With profiling turned on, the execution is going to be considerable slower. If your program is already very slow, consider profiling a simplified version.
- type ”gprof <name of your executable>” and you will get profiled information listed on standard output.
- Go into your code and optimize subprogram which spends most of the time. Then compile for profiling again and check weather you won!
- Continue until most of the time is spent in some very optimized library like blas or you have no idea anymore how to speed up the code.

Memory leak detector

You probably need to use memory leak detector if

- your program coredumps and debugging the program does not help
- it corredumps from no obvious reason
- every time you debug corredumps at different place
- when debugging, certain variable is being changed when you really do not expect it to be changed.

The above cases usually occur becuase you are reading from or writting to a memory location outside of the scope of your program or outside the memory location reserved for your variable.

In addition, memory leak detectors will also detect other missusings of dynamic memory allocation and deallocation. For example: if you forget to clean up some memory reserved by new (Each ”new” should be paired by a corresponding ”delete” statement).

One of free available detectors is called ”valgrind” (`http://valgrind.org/`).

You will most probabbly need to install valgrind on your computer on your own. For many common linux distributions, rpm packages are already available. Check them out first.

There are many other Memory Leak Detectors available. Some are for free but most of them are commercial. The website `http://www.linuxjournal.com/article/6556` lists and describes many of them.

More Advanced and very Useful Trick

Example is from minimization of a function but can be equally well used in numerous other applications.

Suppose a function $F(x_0, x_1, y_0, y_1, ...)$ needs to be minimized with respect to variables x_0 and x_1 while variables y_i are kept fixed during minimization.

A typical Fortran program would define global variables y_i which are changed before the minimization routine is called. User-defined function F uses those global variables at each call.

This leads to an extremely unreadable code and always neeeds to be awoided. DO NOT USE GLOBAL VARIABLES! In large scale projects, it is practicaly imposible to keep track of all the statements which could change a global variable and therefore user can not easily figure out when a global variable is changed.

In C++, we have a workaround to this problem. All necessary variables y_i can be hidden in a user-defined class type and remain completely local. The code thus becomes intuitively very clear for an end user. The code that does the trick, is however slightly complicated and needs some thinking.
Let us take a very simple example of a function

$F(x_0, x_1, y_0, y_1) = (x_0 - y_0)^2 + (x_1 - y_1)^2$. The C++ statement that minimizes function F is very clear and simple

```
Parabola parab(y0,y1);
F = Minimize(2, X, Parabola::FCN);
```

Notice that class parab (which is an instance of class type Parabola) is a local variable and will not be used after minimization (will most probabbly be destroyed).
A naive implementation would be

```
class Parabola{
  float u0, u1;
public:
  Parabola(float u0_, float u1_) : u0(u0_), u1(u1_){};
  double Value(double x0, double x1)
    { return sqr(x0-u0)+sqr(x1-u1);}
};
```

and member function Value(x0,x1) would be given to the Fortran subroutine. This will not work because Value is not a static function (not a unique function but different for every instance of a Parabola class).

We need a static function within the class which can be given to Fortran minimization subroutine. However, static function can not use non-static variables and therefore can not access members of a class. The reason is that static function is a common (unique)

function for all instances of a class type while class members are different for different instances. Therefore we need a static copy of the data. The simplest way to do that is to define a static pointer which will point to the instance which was last created.

The class type Parabola needs to be redefined to

```
class Parabola{
  float u0, u1;
  static Parabola* pt;
public:
  Parabola(float u0_, float u1_) : u0(u0_), u1(u1_)
  double Value(double x0, double x1)
    { return sqr(x0-u0)+sqr(x1-u1);}
  static void FCN(const int* N, const float* X, float* F);
};
```

Notice the static pointer and static function FCN. The constructor needs to point the static pointer to itself. The static member FCN can that access all data of a class through the static pointer.

```
Parabola::Parabola(float u0_, float u1_) : u0(u0_), u1(u1_)
{Parabola::pt = this;}

void Parabola::FCN(const int* N, const float* X, float* F)
{
  if (*N!=2) std::cerr<<"Not right number of variables!"<<std::endl;
  *F = pt->Value(X[0],X[1]);
  std::cout<<*F<<" "<<X[0]<<" "<<X[1]<<std::endl;
```

```
}
```

Finally, the function Minimize is a wrapper function to the fortran subroutine and takes the form

```
template <class functor>
float Minimize(int N, float x[], functor& Fun)
{
  float* x0 = new float[N];
  int lw = N*(N+10);
  float* w = new float[lw];
  for (int i=0; i<N; i++) x0[i]=x[i];
  float f; int info;
  uncmin_(&N,x0,Fun,x,&f,&info,w,&lw);
  delete[] w;
  delete[] x0;
  return f;
}
```

where the prototype for Fortran subroutine is

```
extern "C"
void uncmin_(const int* N, float* X0, void (*FCN)(const int* N, const float* X, float*
             float* X, float* F, int* INFO, float* W, int* LW);
```

Fortran subroutine UNCMIN was downloaded from GAMS web page at

`http://gams.nist.gov/serve.cgi/ModuleComponent/5672/Fullsourc`

For details, check the available program static_function/minimize .