Using Glenn, the IBM Opteron 1350 Compilers

October 19-20, 2010

Table of Contents

Compilers available on Glenn/BALE
Introduction to compiling using GNU
MPI compiler wrappers
Libraries
Debuggers
Material in this class can also be found at
http://www.osc.edu/supercomputing/software/apps/pgi.shtml
http://www.osc.edu/supercomputing/software/apps/intel_compile.shtml

Retrieve Workshop Problems

Retrieve examples/exercises:

```
>> svn checkout  http://svn.osc.edu/repos/softdevtools/trunk/Compiling
```

More on SVN tomorrow. You may follow along with the Examples during the lecture. Time will be provided to complete the exercises later.

Compiling

Compilers at OSC

Myriad of compilers in use

On OSC HPC machines

- gcc, g++ - the Gnu C compiler
- ifort, icc - Intel compilers
- mpicc, mpif90 - compiling MPI programs

Use gcc to illustrate compiling concepts

Compilers do the same thing

Some arguments differ in syntax

- -O3 (gcc)
- -fast (ifort)

Good compiling practices enhance software development

Main interests

Programs that run

And run fast

Forego the usual software engineering arguments

Good and bad practices

GNU Compilers

Compiler suite from the Free Software Foundation
Freely available open source compiler system
C (**`gcc`**)
C++ (**`g++`**)
Fortran 77/90 (**`g77/gfortran`**)
Installed in **`/usr/bin`** and are on all users' **`$PATH`**
Quite good compilers in terms of standards conformance
Primarily used in Makefiles for open source software builds
Do not generate as fast code as other compilers – not recommended for high performance scientific code

Portland Group Compilers

Available on Glenn and BALE clusters

Better performance for AMD Opteron processors

Complete development environments

C (**`pgcc`**)
C++ (**`pgCC`**)
Fortran 77 (**`pgf77`**)
Fortran 90 (**`pgf90`**)
High Performance Fortran (**`pghpf`**)

Portland Group Compilers

Includes a debugger (**pgdbg**) and a profiler (**pgprof**)

Complete manuals can be found at
http://www.pgroup.com/resources/docs.htm

Portland Group Compilers

Load the PGI compilers into your environment using the `module` command:

```
module load pgi
```

The **pgi** module is loaded into each user's environment by default

Current default versions: 9.0-4 (glenn), 7.1-5 (BALE)
Other versions available: through 10.5 (glenn), 9.0-1 (BALE)

```
module switch pgi pgi-10.5
```

Portland Group Compilers

Compiler binaries (executables)

pgf77 (Fortran 77)
pgf90 (Fortran 90)
pgcc (C)
pgCC (C++)

Portland Group Compilers

Always use **man** pages when uncertain!

General options

- **-c** (compile only, do not link; produces object file with **.o** suffix)
- -**DMACRO[=*value*]** (defines preprocessor macro ***MACRO*** with optional ***value***; default value is 1)
- **-g** (generate symbols for debugging; disables optimization)
- **-I/dir/name** (add **/dir/name** to the list of directories to be searched for **#include**d files)
- **-l*name*** (add library **libname.{a|so}** to the list of libraries to be linked)
- **-L/dir/name** (add **/dir/name** to the list of directories to be searched for library files)
- **-o outfile** (name executable file **outfile**; default is **a.out**)
- **-U*MACRO*** (removes definition of ***MACRO*** from preprocessor)

Portland Group Compilers

Optimization options

`-fast` ("best" optimization) – on Glenn, equivalent to:

```
-O2 -Munroll=c:1 -Mnoframe -Mlre -Mvect=sse
    -Mscalarsse -Mcache_align -Mflushz
```

`-O0` (no optimization)

`-O1` (light optimization; default)

`-O2` (heavy optimization, same as **`-O`**)

`-O3` (aggressive optimization)

`-Munroll` (enables loop unrolling)

`-Minline` (controls function inlining)

`-Mvect=cachesize:`*`numbytes`* (sets assumed L2 cache size to ***`numbytes`*** bytes)

`-Mconcur` (enables automatic parallelization of loops)

`-mp` (enables support for OpenMP and SGI-style PCF pragmas for parallelization)

Portland Group Compilers

C Options

`-Xa` (enforces strict ANSI C compliance)
`-Xc` (enforces loose ANSI C compliance)
`-Xs` (enforces strict K&Rv1 C compliance)
`-Xt` (enforces loose K&Rv1 C compliance)

Recommended flags:

```
-Xa -fast
```

Portland Group Compilers

C++ Options

`-A` (enforces strict ANSI C++ compliance)

`-exceptions` (enables ANSI C++ exceptions)

`--prelink_objects` (enables support for template libraries within template libraries)

Recommended flags

```
-A -fast --prelink-objects
```

Portland Group Compilers

F77/F90 Options

`-byteswapio` (uses byte-swapping with unformatted I/O; compatible with Sun and SGI systems)

`-i4` (assumes 4-byte **`INTEGER`**s; default)

`-i8` (assumes 8-byte **`INTEGER`**s)

`-r4` (interpret **`DOUBLE PRECISION`** variables as **`REAL`**)

`-r8` (Interpret **`REAL`** variables as **`DOUBLE PRECISION`**)

Recommended flags:

`-fast`

Intel Compilers

Better performance than GNU (excellent performance)
Includes support for:
C (**`icc`**)
C++ (**`icpc`**)
Fortran 77 and 90 (**`ifort`**)
Includes a debugger and a profiler
Debug with **`idb`**
Profile with **`gprof`**
Vendor documentation can be found here:

http://www.osc.edu/supercomputing/manuals/

Intel Compilers

Latest version:

glenn: 11.1.056

BALE: 10.0.023

```
module load intel-compilers-11.1
```

intel-compilers-11.1 points to the 11.1.056 software

Compiler binaries (executables)

`ifort` (Fortran 77)

`ifort` (Fortran 90)

`icc` (C)

`icpc` (C++)

Intel Compilers

General Options

-c (compile only; do not link)

-D*MACRO*[=*value*] (defines preprocessor macro ***MACRO*** with optional ***value***; default value is 1)

-g (generate symbols for debugging; disables optimization)

-I/dir/name (add **/dir/name** to the list of directories to be searched for **#include**d files)

-l*name* (add library **libname.{a|so}** to the list of libraries to be linked)

-L/dir/name (add **/dir/name** to the list of directories to be searched for library files)

-o outfile (name resulting output file **outfile**; default is **a.out**)

-U*MACRO* (removes definition of ***MACRO*** from preprocessor)

Intel Compilers

Optimization Options

`-fast` ("good" optimization, can cause numerical problems) – on Glenn, equivalent to:

`-O3 –ipo -static`

`-O0` (no optimization)

`-O1` (light optimization; default)

`-O2` (heavy optimization)

`-O3` (aggressive optimization, may change numerical results)

`-ipo` (enable interprocedural optimizations between files)

`-funroll-loops` (enables loop unrolling)

`-parallel` (enables auto-parallelizer to generate multi-threaded code for safe loops)

`-openmp` (enables parallelization using OpenMP directives)

Intel Compilers

C/C++ Options

`-strict_ansi` (enforces strict ANSI C/C++ compliance)

`-ansi` (enforces loose ANSI C/C++ compliance)

`-Wall` (enable all warnings)

Recommended flags: `-O2 -ansi`

Intel Compilers

Fortran 77, Fortran 90 Options

`-convert bigendian` (uses unformatted I/O compatible with Sun and SGI systems)

`-convert cray` (uses unformatted I/O compatible with Cray systems)

`-i8` (makes 8-byte **`INTEGER`**s the default)

`-module /dir/name` (adds **`/dir/name`** to the list of directories searched for F90 modules)

`-r8` (makes 8-byte **`REAL`**s the default)

`-warn` (enables all warning messages)

`-warn nousage` (suppresses warnings about questionable programming practices)

Recommended flags: **`-O2`**

Transitioning From GNU to Intel/PGI

Makefiles that use GNU compilers can be modified to use Intel or PGI compilers:
Generally the PGI compilers are more likely to work with an existing GNU Makefile
Be careful with optimization flags

- Start with little or no optimization
- **Verify numerical correctness (**some optimization techniques can affect numerical stability)
- Increase optimization

Compiling – Examples using GNU

Simple Compile

Small C program

hello.c:

```
#include <stdio.h>
int main (void) {
     printf ("Hello, world!\n");
     return 0;
}
```

Minimal single file compile

`gcc hello.c`

Produces executable `a.out`

Execute program with command `./a.out`

If $PATH includes `./`, may type `a.out`

Executable file should have ‘x’ set in permissions

```
“./a.out: Permission denied.”
8 -rwxr-xr-x    1 pete     G-0541        5144 Feb 24 05:18 a.out
chmod a+x a.out
```

Compiling

Simple Compile

Direct executable name to alternate file name

`gcc hello.c –o hello`

Execute with command `./hello`

Most compilers recognize the `'-o'` option

Second simple program

duh.c:

```
#include <stdio.h>
int main (void) {
     printf ("Two plus two is %f\n", 4);
     return 0;
}
```

Try to compile and execute?

```
>> gcc duh.c –o duh
>> ./duh
Two plus two is -0.139606
>> [Truly, DOH!??]
```

We will revisit this later

The need for feedback during the compile

Compiling

Compiling Multiple Source Files

Having all the source in a single file is limiting

As the file grows

- compilation time tends to grow
- for each little change, the whole program has to be re-compiled

Impossible that several people will work on the same project together in this manner

Managing your code becomes harder

Split the source code into multiple files

each containing a set of closely-related functions

Use a single command line to compile all the files

```
gcc -g -o imgdither make_pal.c convert.c io_image.c img_dither.c
```

All the source files will be recompiled

Even those files not edited

Compiling

Compiling Multiple Source Files

Alternate method

Divide the compile into two phases

Compiling

- Compile object files for each source

```
gcc –c make_pal.c
gcc –c convert.c
gcc –c io_image.c
gcc –c img_dither.c
```

- -c flag indicates just compile the object file
- More flexible
- Unedited files need not be recompiled

Linking

- Link all object files into one executable

```
gcc make_pal.o convert.o io_image.o img_dither.o –o imgdither
```

This will be more efficient when discussing ‘make’ and Makefiles

Compiling

Compiling production line

Objective is to create an executable file

Specific to the architecture of the machine

Compiling is multi-stage process

Refer to gcc

Front ends: *gcc* or *g++*

Assembler: *as*

Linker: *ld*

Don't need to use each individually

The compilation step with gcc uses all

The simple *hello.c* example goes through all the steps

- Uses headers
- Uses external libraries

Use the –v flag during compilation < the output is on the next slide >

The stages are:

Preprocessing (macro expansion)

Compilation (creating assembler language code)

Assembling (create machine code)

Linking (create final binary executable)

Compiling

Compiling production line

```
pete@gromit: ~/ >> gcc -v hello.c

Using built-in specs.
Target: i386-redhat-linux
Configured with: ../configure --prefix=/usr --mandir=/usr/share/man --infodir=/usr/share/info
 --enable-shared --enable-threads=posix --enable-checking=release --with-system-zlib --enable-__cxa_atexit
 --disable-libunwind-exceptions --enable-libgcj-multifile --enable-languages=c,c++,objc,obj-c++,java,fortran,ada
 --enable-java-awt=gtk --disable-dssi --enable-plugin --with-java-home=/usr/lib/jvm/java-1.4.2-gcj-1.4.2.0/jre
 --with-cpu=generic --host=i386-redhat-linux

Thread model: posix
gcc version 4.1.1 20070105 (Red Hat 4.1.1-51)
 /usr/libexec/gcc/i386-redhat-linux/4.1.1/cc1 -quiet -v hello.c -quiet -dumpbase hello.c -mtune=generic
 -auxbase hello -version -o /tmp/ccbDwhXh.s

ignoring nonexistent directory "/usr/lib/gcc/i386-redhat-linux/4.1.1/../../../../i386-redhat-linux/include"
#include "..." search starts here:
#include <...> search starts here:
 /usr/local/include
 /usr/lib/gcc/i386-redhat-linux/4.1.1/include
 /usr/include
End of search list.
GNU C version 4.1.1 20070105 (Red Hat 4.1.1-51) (i386-redhat-linux)
        compiled by GNU C version 4.1.1 20070105 (Red Hat 4.1.1-51).
GGC heuristics: --param ggc-min-expand=98 --param ggc-min-heapsize=129117
Compiler executable checksum: 98782966c6e2b1983484cce1a314172a
  as -V -Qy -o /tmp/ccr9lLjm.o /tmp/ccbDwhXh.s
GNU assembler version 2.17.50.0.6-2.fc6 (i386-redhat-linux) using BFD version 2.17.50.0.6-2.fc6 20061020
 /usr/libexec/gcc/i386-redhat-linux/4.1.1/collect2 --eh-frame-hdr -m elf_i386 --hash-style=gnu -dynamic-linker
 /lib/ld-linux.so.2 /usr/lib/gcc/i386-redhat-linux/4.1.1/../../../crt1.o /usr/lib/gcc/i386-redhat-linux/4.1.1/../../../crti.o
 /usr/lib/gcc/i386-redhat-linux/4.1.1/crtbegin.o -L/usr/lib/gcc/i386-redhat-linux/4.1.1 -L/usr/lib/gcc/i386-redhat-linux/4.1.1
 -L/usr/lib/gcc/i386-redhat-linux/4.1.1/../../.. /tmp/ccr9lLjm.o -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed
 -lgcc_s --no-as-needed /usr/lib/gcc/i386-redhat-linux/4.1.1/crtend.o /usr/lib/gcc/i386-redhat-linux/4.1.1/../../../crtn.o
pete@gromit: ~/ >>
```

Compiling

Compiling production line

Preprocessing

Source code

- Expand macros
- Included headers

Usually files not saved unless the `-save-temps` option used

- ***.i*** for C code
- ***.ii*** for C++ code

Compiling

Compilation of preprocessed source code to assembly language

For a specific processor

```
>> gcc -Wall -S hello.i
```

Use the command-line option `-S`

- Convert the preprocessed C source code to assembly language :
- Assembly language is stored in the file `hello.s`:

Compiling

Compiling production line

Compiling

Assembler file for an Intel 386 processor on Fedora Core 6

hello.s:

```
            .file       "hello.c"
            .section    .rodata
.LC0:
            .string     "Hello, World!."
            .text
.globl main
            .type       main, @function
main:
            leal        4(%esp), %ecx
            andl        $-16, %esp
            pushl       -4(%ecx)
            pushl       %ebp
            movl        %esp, %ebp
            pushl       %ecx
            subl        $4, %esp
            movl        $.LC0, (%esp)
            call        puts
            movl        $0, %eax
            addl        $4, %esp
            popl        %ecx
            popl        %ebp
            leal        -4(%ecx), %esp
            ret
            .size       main, .-main
            .ident      "GCC: (GNU) 4.1.1 20070105 (Red Hat
4.1.1-51)"
            .section    .note.GNU-stack,"",@progbits
```

Compiling

Compiling production line

Assembler

Assembler converts assembly language into machine code

Generates object file

Calls to external functions

- addresses of the external functions are left undefined
- Filled in later by the linker.
- Assembler invoked with the `–o` flag in the command line

```
>> as hello.s –o  hello.o
```

- `hello.o` has the machine instruction for the C code in `hello.c`
  - Undefined reference to ***puts***

Linking

Final stage of compilation

Linking of object files to create an executable

Executable requires many external functions

- From system and C run-time (crt) libraries
- Internal link command, ***ln***, is complicated

```
ld --eh-frame-hdr -m elf_i386 --hash-style=gnu -dynamic-linker
 /lib/ld-linux.so.2 /usr/lib/gcc/i386-redhat-linux/4.1.1/../../../crt1.o /usr/lib/gcc/i386-redhat-linux/4.1.1/../../../crti.o
 /usr/lib/gcc/i386-redhat-linux/4.1.1/crtbegin.o -L/usr/lib/gcc/i386-redhat-linux/4.1.1 -L/usr/lib/gcc/i386-redhat-linux/4.1.1
 -L/usr/lib/gcc/i386-redhat-linux/4.1.1/../../.. /tmp/ccr9lLjm.o -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed
 -lgcc_s --no-as-needed /usr/lib/gcc/i386-redhat-linux/4.1.1/crtend.o /usr/lib/gcc/i386-redhat-linux/4.1.1/../../../crtn.o
```

Compiling production line

Linker

Transparent to the process, fortunately

Links the object file

- To the C standard library
- To any other libraries as directed
  - Using `-L` in command line
  - Environment variable : `$LD_LIBRARY_PATH`

Produces the executable file `a.out`

Compiling

Using Options For Compiling

Control compiling features

Search paths used for locating libraries and include files

Use of additional warnings and diagnostics

Preprocessor macros

C language dialects.

Some desirable options are not default

For specific options use the ***man***

For our purposes '`man gcc`'

Information overload

But very specific to the architecture

Eventually programmers must refer to the options

Commonly-used GCC compiler options

Debugging with `-g`

- Builds symbol table of the code
  - Names of variables
  - Line numbers of commands and variables
- File size differ appreciably
  - **`gcc –g main.c`**
    - `8 -rwxr-xr-x 1 pete G-0541 6356 Feb 24 05:44 a.out`
  - **`gcc main.c`**
    - `8 -rwxr-xr-x 1 pete G-0541 5144 Feb 24 05:45 a.out`
- Look at this in greater detail in the ***debugging*** section

`-c` and `-o`

We have seen these in action

Compiling

Using Options For Compiling – Warnings

`-W(exp)` : Commonly-used GCC compiler options

Warnings with `-Wall`

- Essential potpourri of individual warning flags
- Should always be included in the compile
- Can catch many errors during compilation
- Full list of options included in `man gcc`
- Revisit our previous `duh.c` problem

```
>> gcc duh.c
>> ./a.out
Two plus two is -0.703465.
>>
```

- Problem printing correct value
- `-Wall` includes the warning `–Wformat`
  - Checks for incorrect use of format strings
  - In functions such as `printf` and `scanf`
  - format *specifier* does not agree with the variable type

```
>> gcc -Wall duh.c
duh.c: In function 'main':
duh.c:4: warning: format '%f' expects type 'double', but argument 2 has type 'int'
>> ./a.out
Two plus two is 4.
>>
```

  - All listed in ***`man gcc`***
    - Will say something like **`(Included in –Wall)`**

Compiling

Using Options For Compiling - Warnings

Other warning options not included in `-Wall`

Source code may be technically valid

But may cause problems

Not included in `–Wall`

- Flag only possible problems

`-W`

- General option similar to `-Wall`
- Warns about common programming errors
- Functions which can return without a value
  - "falling off the end of the function"
- Comparisons between signed and unsigned values
  - testing if an unsigned integer is negative

```
int testInt (unsigned int x) {
    if (x < 0) return 0;
    else return 1;
}


>> gcc -W -c testInt.c
testInt.c: In function 'testInt':
testInt.c:2: warning: comparison of unsigned expression < 0 is always false
>>
```

Compiling

Using Options For Compiling - Optimizing

Optimizing the code

Want the program to run faster or take less space

Replace the '-g' flag with the '-0' argument

Compilation takes longer

- Compiler applies various optimization algorithms

Optimization is designed to be conservative

- Ensures code will function the same as without optimization

Can ramp up the optimization

- Add number arguments to '-0x': '-02', '-03', '-04'
- The higher the number the greater the optimization and slower the compiler

Optimization can alter code

- Chances are higher that an improper optimization will actually alter our code
- Some of them tend to be non-conservative, complex, and contain bugs

Compiling

Using Options For Compiling - Optimizing

Common in-source optimization

References programming practices

Eliminate subexpressions

- Reduce the number of operators

```
X = (1 – t 2)  +  3*(1 – t 2) 3  –  2*(1 – t 2) 5;
```

- Pre-calculate a common expression

```
m = 1 – t 2;

X = (m) +  3 * (m) 3 – 2 * (m) 5;
```

- Reduces the size of the code and increases the speed

Function inlining

- Overhead calling functions

```
double square (double x) {
     return x * x;
}

for (i=0; i<infinity; i++) {
     sum += square(i + 3.14);
}
```



```
for (i=0; i<infinity; i++)
{
     double t = (i+ 3.14);
     sum += t * t;
}
```

Compiling

Using Options For Compiling - Optimizing

Control compilation-time and compiler memory usage

Tradeoffs between speed and space for the resulting executable

GCC provides a range of general optimization levels

- Numbered from 0—3
- Individual options for specific types

An optimization level is given in the command line options

- `-O`*`LEVEL`*`, `*`LEVEL`* is `0-3`

`-O0` or no `-O` option (default)

- Does not perform any optimization
- Compiles the source code in the most straightforward way possible
  - Each command converted directly to the corresponding instructions
- Best option to use when debugging

`-O1` or `-O`

- Most common form of optimization
- Executables should be smaller and faster than with `-O0`
- Expensive optimizations are not used at this level.
  - Instruction scheduling
- Takes less time than compiling with `-O0`
  - Reduced amounts of data not processed after simple optimizations.

Compiling

Using Options For Compiling - Optimizing

`-02`

Turns on further optimizations
- Additional to those used by `-01`
- Includes instruction scheduling

No optimizations that require any speed-space tradeoffs are used
- executable should not increase in size

Takes longer to compile and requires more memory than with `-01`

Best choice
- Provides maximum optimization without increasing the executable size

Default optimization level for GNU

`-03`

More expensive optimizations
- Function inlining
- As well as all the optimizations of the levels `-02` and `-01`

May increase the speed executable

Could also increase its size

May make a program slower and create spurious results

Not recommended

Compiling

Using Options For Compiling - Optimizing

`-funroll-loops`

Turns on loop-unrolling
Independent of other optimization options
Will increase the size of an executable
Results unclear, must be considered case-by-case

`-Os`

Reduces the size of an executable
Produces the smallest possible executable
- For systems constrained by memory or disk space

Possible that smaller executable runs faster
- Better cache usage.

Cost of optimization

Greater complexity in debugging
Increased time and memory requirements during compilation
Best *rule of thumb*
- Use `-O0` for debugging
- Use `-O2` for production

Compiling

Using Options For Compiling - Optimizing

Optimization and compiler warnings

Some compiler warnings do not appear without optimization

Data flow analysis

- Compiler examines the use of all variables and initial values
- Basis of optimization strategies

Compiler can detect the use of un-initialized variables

Create a file with the following code:

`-Wuninitialized` (included in `-Wall`)

- Warns about un-initialized variables
- Only works with optimization turned on

Try the two compile commands

```
gcc –Wall –c checkex.c
```

```
gcc –Wall –O2 –c checkex.c
```

checkex.c:

```
float check (float x) {
     float s;
     if (x == 0.0)
          s = 0.0;
     else if (x != 0.0)
          s = 1/x;
     return s;
}
```

Compiling

Using Options For Compiling – Libraries

Precompiled object files which can be linked into programs

Most commonly used in the C library

sqrt, fabs functions in the C *math* library

Two flavors

static libraries

- Have the extension ***.a***
- May be referenced directly in the compile command

```
gcc -Wall calc.c /usr/lib/libm.a -o calc
```

shared libraries

- Have the extension ***.so***
- More compatible with code reuse
- Linked in at ***runtime***

Library locations

Specify specific libraries in compile command

- `-l(name) – [lower case el](name)`
- Example: `libm.a` would be `–lm`
  - Take off the ***'lib'*** and the ***'.a'***
- `gcc –g –Wall –o myfunc myfunc.c –L/home/user/libs -lmylibs`
  - **Compiler will look for a library** `libmylibs.a` **or** `libmylibs.so`
  - **In the directory** `/home/user/libs`

Compiling

Using Options For Compiling – Libraries

Library locations

As with header files, libraries may reside outside the system install

- Compilers look in `/usr/lib` and `/usr/local/lib` by default
- `-Ldir` in the compile command
  - May be repeated

```
gcc –L. –L/home/me/libs …
```

- `$LD_LIBRARY_PATH`
  - The command ***module*** user interface to Modules package
  - Provides for dynamic modification of user environment

```
[opt-login01] ~ :: module list
Currently Loaded Modulefiles:
  1) pgi                              4) torque
            7) modules
  2) intel-compilers-10.0             5) mpi
  3) moab                             6) mpirun-compat
[opt-login01] ~ :: module show  intel-compilers-10.0
-------------------------------------------------------------------
/usr/local/share/modulefiles/intel-compilers-10.0:


module-whatis    loads the Intel C/C++/F95 compilers
prepend-path     LM_LICENSE_FILE 28519@license2.osc.edu
prepend-path     MANPATH /usr/local/intel-10.0.023/man
prepend-path     PATH /usr/local/intel-10.0.023/bin
append-path      LD_LIBRARY_PATH /usr/local/intel-10.0.023/lib
set-alias        efc ifort
set-alias        ecc icc
-------------------------------------------------------------------
[opt-login01] ~ ::
```

Compiling

Using Options For Compiling – Include Files

Include files contain information for a program

Variable declaration and initialization

Function declaration and prototyping

`<stdio.h>`

- Directed to look in standard system default locations
- `/usr/include`, `/usr/local/include`

`"/home/me/include/myown.h"`

- Look in a specified directory

Include locations for header files

Header files in your source code

- Most system header files in `/usr/include` and `/usr/local/include`
- `-Idir` – that's `[cap i]dir`
  - Also repeatable
- Need to access other directories

```
gcc –g –Wall –I/home/me/includes
```

- Direct the compiler to look into /home/me/includes for referenced headers
- Use the <> in the source file

Compiling

Using Options For Compiling – Include Files

Include locations for header files

Environment variables

- `C_INCLUDE_PATH`
- `CPLUS_INCLUDE_PATH`

Using Options For Compiling – Search Paths

Order of search

1. Command-line options `-I` and `-L`, left to right
2. Environment variables
3. Default system directories

Compiling

Using Options For Compiling – Preprocessing

Alter execution of program

Using `#ifdef`

```
#include <stdio.h>
int main(void) {
#ifdef DEBUG
   printf (“This might be interesting.\n");
#endif
   printf (“This is running mode.\n");
   return 0;
}
```

- Preprocessor includes code
- `#endif` terminates block

`-D`*name* defines a macro

- Part of the command line

```
gcc –DDEBUG …
```

Compiling

Making Static Libraries

Additional commands

`ar` GNU program which creates, extracts and modifies from archives

- An archive is single file holding a collection of other files

`ranlib` generates an index to the contents of an archive and stores it in the archive.

In our image processing example we wish to create a static library for the files `make_pal.o convert.o io_image.o`

```
gcc –c convert.c
gcc –c make_pal.c
gcc –c io_image.c
ar rc libimgdith.a make_pal.o convert.o io_image.o
ranlib libimgdith.a
          (nm libimgdith.a)
gcc –c img_dither.c
gcc img_dither.o –L. –limgdith –o imgdither
```

Compiling

Making Static Libraries : Creating

Often code may be collected and used over and over again
This involves compiling the reusable code into a static library
First
Create source files
Containing functions that will be used
Library can contain multiple object files
Compile files into object files
Creating a library

```
ar rc speciallib.a objfile1.o objfile2.o objfile3.o
```

Create a static library
Rename the "`speciallib`" portion of the name
Create an index inside the library

```
ranlib libmylib.a
```

copying the library

- use the `-p` option with `cp` to preserve permissions

Compiling

Making Static Library : Usage

Prototype library function calls

Do not want implicit declaration errors

When linking to the libraries

```
gcc -o main -L. -lmylib main.o
```

Specify where the library can be found

`-L.` tells `gcc` to look in the current directory for `libmylib.a`.

Compiling

Making Shared Libraries : Creating

Creating shared or dynamic libraries is simple also.
Using the previous example, to create a shared library:

```
>> gcc -fPIC -c objfile1.c
>> gcc -fPIC -c objfile2.c
>> gcc -fPIC -c objfile3.c
>> gcc -shared -o libmylib.so objfile1.o objfile2.o objfile3.o
```

`-fPIC` option

tells compiler to create Position Independent Code

- **create libraries using relative addresses**
- **no absolute addresses because these libraries can be loaded multiple times**

`-shared` option

specifies architecture-dependent shared library is being created
not all platforms support this flag.

Compile the actual program

using the libraries:

```
>> gcc -o foo -L. -lmylib foo.o
```

- **Same as creating a static library**
- **none of the actual library code is inserted into the executable**

Compiling

Making Shared Libraries : Usage

Programs using static libraries

Can run on its own

Shared libraries dynamically access libraries **at run-time**

program needs to know where the shared library is stored

The advantage of using Dynamic Libraries

- The executable is much smaller
- No need to compile it into the executable at compile time

Programs working with dynamic libraries use `LD_LIBRARY_PATH` environment variable

Make sure to **append** the desired path to the variable

```
>> echo $LD_LIBRARY_PATH
/usr/lib:/local/lib:/local/peteFiles/vtk5.0/lib:/local/peteFiles/AVS7.0/express/lib/linux
>> setenv LD_LIBRARY_PATH /home/pete/libs:${LD_LIBRARY_PATH}
>> echo $LD_LIBRARY_PATH
/home/pete/libs:/usr/lib:/local/lib:/local/peteFiles/vtk5.0/lib:/local/peteFiles/AVS7.0/express/lib/linux
```

Not overwrite it – erases all the system settings

Mistake:

```
setenv LD_LIBRARY_PATH /home/pete/libs
```

Parallel Program Development

MPI Compiler Wrappers

The MVAPICH implementation of MPI for Infiniband

- Uses a set of compiler scripts
- Need not remember **`path`** names for libraries and include files
- MPI compilation scripts support the following languages
  - C (**`mpicc`** – wrapper for **`pgcc`**)
  - C++ (**`mpiCC`** – wrapper for **`pgCC`**)
  - Fortran 77 (**`mpif77`** – wrapper for **`pgf77`**)
  - Fortran 90 (**`mpif90`** – wrapper for **`pgf90`**)
- Accept the same arguments as the compiler they wrap
  - **`mpicc`** accepts the same arguments as **`pgcc`**
  - **`mpif77`** accepts the same arguments as **`pgf77`**
  - See manual pages for details on argument options

Parallel Program Development

- MPI Compiler Wrappers
  - Default compiler suite is PGI
  - To use Intel compilers:

```
module switch mpi mvapich-1.1-intel
```

Parallel Program Development

MPI Compiler Wrappers Break

Occasionally, quoting of compiler arguments:

- will not work with the MPI compiler wrappers (which are, after all, only shell scripts)
- in these cases, you can use the Portland Group compilers directly:
  - Compile with
    - **$MPI_CFLAGS** (C)
    - **$MPI_CXXFLAGS** (C++)
    - **$MPI_FFLAGS** (F77)
    - **$MPI_F90FLAGS** (F90)
  - Link with **$MPI_LIBS**

Parallel Program Development

- MPI Compiler Wrappers Break
  - Example

```
[opt-login1] pgcc -O2 \
$MPI_CFLAGS -DVERSION=' " v0.2 build 1/21/00" ' -c vbcast.c
```

  - Example

```
[opt-login1] pgf90 -O2 \
$MPI_FFLAGS -DSIZE='128' cp3.F -o cp3-128 $MPI_LIBS
```

Program Development Tools and Libraries

Libraries

Several Fortran numerical libraries installed

- Can be used in conjunction with the compiler being used

AMD Core Math Library

- Includes BLAS, LAPACK, FFT
- Link with $ACML
- Requires loading the module specific to the compiler

```
[opt-login1] module avail acml
…
acml-gfortran acml-gnu acml-intel acml-pgi
…
[opt-login1] module load acml-pgi
```

- Type the command '**module show acml-*xxx***' for specific settings

Debuggers

gdb command line symbolic debugger

Intel compilers include a debugger, **idb**

Portland Group compilers include a debugger, **pgdbg**

totalview parallel debugger

```
module load totalview
```

Debuggers

`totalview` within a batch job

Designed for parallel programs

MPI, OpenMP, or pthreads

MPI jobs run ***only*** through the PBS batch system

PBS allows for interactive batch jobs

- Used to run interactive programs
- Within the framework of a batch job
- **`qsub -I`**

Debuggers

`totalview` example

Specify on the command line

```
[opt-login1.osc.edu] > qsub -I -l nodes=2:ppn=4  \
  -l walltime=1:00:00 -j oe -N totalview -S /bin/ksh \
  –v DISPLAY
```

Obtain an interactive shell

- On one of the compute nodes
- Has all the limits of your batch request
- **Run `mpiexec` with the `-tv` option**

```
[opt0838] > cd $PBS_O_WORKDIR
[opt0838] > module load totalview
[opt0838] > mpiexec –tv a.out
```

Debuggers

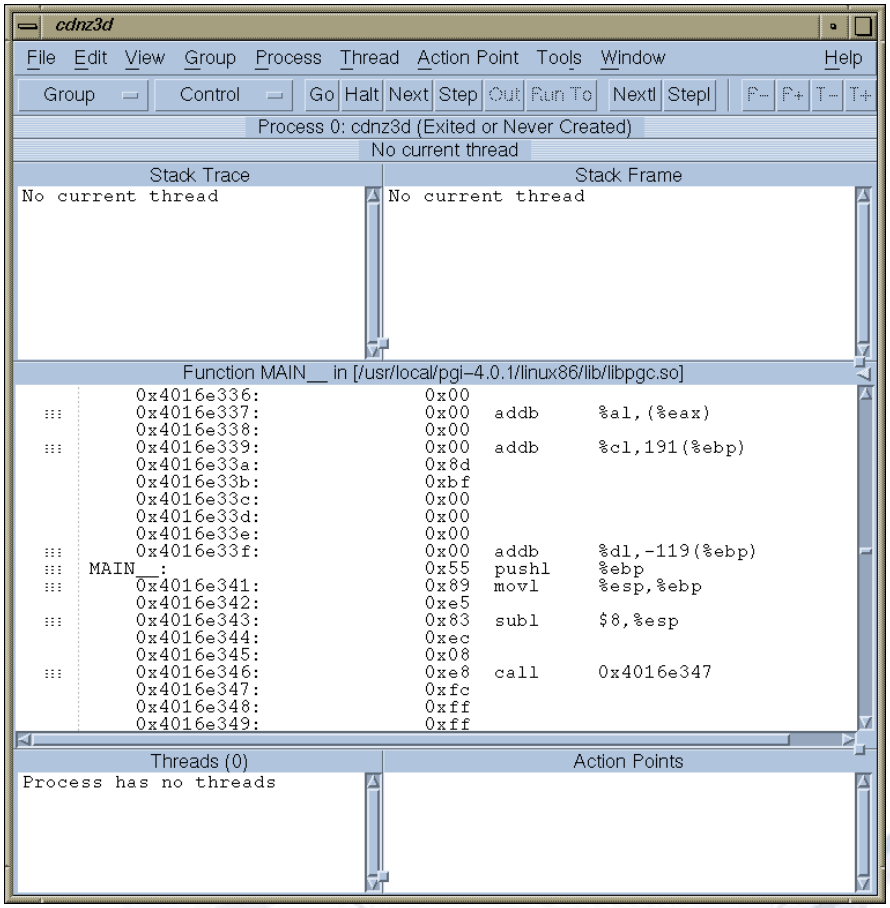
Totalview basic look

Process window

- Pull-down menus
- Control buttons
- 4 panes

Interact with totalview via mouse

- Set breakpoints
- Examine variables
- “step” button



- A good tutorial for totalview can be found at
  `https://computing.llnl.gov/tutorials/totalview/`

Reference

1. http://www.network-theory.co.uk/docs/gccintro/index.html, good online reference

Compiling Exercises

Retrieve the copies of the examples and exercises from Subversion repository

1. svn checkout http://svn.osc.edu/repos/softdevtools/trunk/Compiling
2. Each directory has a ‘README’ and/or ‘HTML’ file that will direct you through the exercise.