Introduction to Compiling and Profiling at MSI

Minnesota Supercomputing Institute

University of Minnesota

Welcome to "Compiling and Profiling" tutorial

Tutorial

- Training level
  - Beginner
- Presenters
  - Ham Lam (lamx0031@umn.edu)
  - Angel Mancebo Jr. (mance012@umn.edu)
- Recommended background
  - Familarity with Linux
  - Basic programming experience

Tutorial outline

- Compilers at MSI
- Compiling and Linking Libraries
- Automated building system (GNU Make)
- Profiling applications
- Hands-On demo

HPC clusters at MSI

Agate

- 412 nodes, AMD processors with 64-128 cores per node
- 344 CPU compute nodes (244 w/ 512g memory, 100 w/ 2TB memory)
- 264 Nvidia A100 GPUs are available
  - 50 nodes have 4 A100 GPUs connected via NVLink and 512 GB of memory
  - 8 nodes have 8 A100s and 1 TB of memory
- 10 GPU interactive nodes, 8 A40 GPUs 512g memory each

Mesabi

- 741 Compute nodes (Intel CPU) with 17,784 total cores
- 40 nodes with 2x Nvidia Tesla GPUs
- 32 nodes with 480g SSDs

HPC clusters at MSI

Mangi

- 164 AMD ROME nodes
- 20,992 cores
- AMD 2TB RAM nodes
- V100 GPU nodes, V100 4-way and 8-way nodes

Compilers

- A compiler is a program to turn human readable source code into a machine code "program" for execution.
- Not all programming languages use compilers, but the fastest executing ones do.
- In the process of compiling most compilers will partially "optimize" code for faster more efficient execution.

Interpreters vs Compilers

An interpreter (e.g. Python, Perl)

- Reads each character then figure out what has to be done.
- When you use an interpreter you are actually using a sophisticated machine language program that someone wrote.
- Your program stays in text file format and is run as a text file.
- An interpreted program must be re-interpreted each time it is run.

Interpreters vs Compilers

A compiler(C,C++,fortran)

- More difficult to use than interpreters
- Your program is translated into a machine language which is then run
- Once a program has been compiled it may be run as many times as you want without recompiling
- Compilers are used by people who need programs that run faster than an interpreter would allow
- Many compilers also attempt to optimize the code for performance

What a compiler does?

1. Preprocessing: Parses and alters code
2. Compiling: Translate to Assembly language
3. Assembly: Translate into machine code
4. Linking: Links machine code pieces and libraries into final machine code program

It is possible to stop at intermediate stages

Compilers available at MSI

Intel compilers

- C compiler: icc
- C++ compiler: icpc
- fortran compiler: ifort
- Good optimization and integration with Intel libraries, e.g. MKL

module avail intel *list all available versions
module load intel *load default version of the intel compiler

Compilers available at MSI

GNU compilers

- C compiler: gcc
- C++ compiler: g++
- fortran compiler: gfort
- Free and open source

module avail gcc *list all available versions
module load gcc *load default version of the gcc compiler

Compilers available at MSI

PGI compilers

- C compiler: pgcc
- C++ compiler: pgcpp
- fortran compiler: pgfortran

```
module avail pgi
module load pgi
```

Compilers available at MSI

Clang compilers

- C compiler: clang
- C++ compiler: clang++
- fortran compiler: No fortran compiler

```
module avail clang
module load clang
```

Compilers available at MSI

Nvidia's nvcc compilers

- C compiler: nvc
- C++ compiler: nvc++
- fortran compiler: nvfortran

module avail nvidia-hpc-toolkit
module load nvidia-hpc-toolkit/20.11

Where to compile code?

Users should compile on the node type where they intend to run the code. Most compiler are smart enough to query the local machine type and will tailor the compilation toward that specific architecture (note, one can usually also force machine-specific compilations using compiler options).

- Use a dedicated interactive session
- Submit a batch job (if building large software)

Compiling commands

Let's clarify some terms:

- code, just a file containing a computer program.
- source code (source file), a program written in a higher level language such as Python, C, or even assembler.
- object code, the output of a compiler or an assembler
- executable code, a file that is ready to run on the machine

Compiling commands

- Usual format: **compiler-name flags sourcefiles**
- **compiler-name** should be the name of the compiler being used.
- **flags** are optional arguments (called flags) that change compiling options. Most flags begin with the minus sign (-).
- **sourcefiles** are the names of the source code files that are being compiled.
- By default, most compilers name the compiled program **a.out**

Example

```
gcc -o my.exe mysourcecode.c
```

Useful compiler options

- To specify a name for the compiled program use the **-o** flag followed by the desired program name.
  - Intel compiler examples:
  - icpc -o myprogram mysourcecode.cpp
  - icc -o myprogram mysourcecode.c
  - ifort -o myprogram mysourcecode.f
- For multiple source files the command looks like this:
  - g++ -o myprogram mysourcecode1.cpp mysourcecode2.cpp

Optimization flags

- The compiler can attempt to perform some automatic optimizations of the source code, and this can often tremendously improve program speed.
- The flags controlling optimization begin with **-O** (oh, not zero), and most compilers support three options: **-O1, -O2, -O3**.
- The automatic optimizations can remove unnecessary portions of code, and reorganize code so that it performs more efficiently.
- Using optimization flags can sometimes slightly alter the output of a program. Usually any difference will occur in the smallest digits of numbers being calculated, and will only be significant if the program is very sensitive to such values.

Compiling using object files

- It is sometimes useful to compile in steps, first compiling to **object (.o) files**, and then linking the object files.
- Use the **-c** flag: **icpc -c mysource1.cpp mysource2.cpp**
- This will produce files named **mysource1.o mysource2.o**
- To perform the linking use the command:
- **icpc -o myprogram mysource1.o mysource2.o**
- This will link the object files and create an executable named myprogram.

Compiling using object files

Advantages

- It reduces the time to recompile after making small changes to a large program. In such a case only a few of many object files will need to be recompiled. It also reduces compile time in many files that have shared dependencies.
- It makes it easier to work collaboratively on code, with each coder compiling their own object files which are part of a larger program.
- It allows the linking of object files that were created via different programming languages (care must be taken with this).

Disadvantages

- It can prevent some compiler optimizations because it makes it harder for the compiler to determine how all the pieces fit together. This effect is usually minor.

Preprocessor Commands

Preprocessor statements within program code can be used to include other code and insert constants and code snippets. Most compilers use # to indicate a preprocessor directive.

Include statements insert code

#include <stdio.h> This will cause the preprocessor to insert the contents of the stdio.h file at this place in the code.

Define statements insert constants or code snippets

#define pi 3.141592653 This will cause the preprocessor to replace every instance of pi with the numerical representation.

Linking with Libraries

Linking with Libraries

Linking wth Libraries

Library files come in two types: dynamic and static

- Dynamic (shared) library files have names ending in **.so**
- Static library files have names ending in **.a**

Dynamic (shared) libraries remain distinct from programs, and are loaded by a program during execution when they are needed. Dynamic libraries can be upgraded without recompiling the programs using them (within limits).

Static libraries become part of the program using them. When a static library is used while compiling the program generated will always load that version of the library.

Where the Compiler Searches for Files

The compiler needs to locate:

- Source code files
- Files included via preprocessor statements
- Library files

The compiler will search for source code files in the place the compiling command is executed.

Include files and library files are looked for in a number of locations governed by compiling options.

Where the Compiler Searches for Files

- **Include files** are files containing code (often "header files") that will be inserted into the source code by the preprocessor.
- Places specified in the compiling command using the -I (uppercase i) flag:
- For example: **gcc -I /soft/fftw/include -o myexe source.c**
- Places referenced by environmental variables: **CPATH, FPATH, C_INCLUDE_PATH, CPLUS_INCLUDE_PATH, INCLUDE**
- The variables are mostly C, C++, Fortran specific
- "Default" locations like: /usr/include etc

Where the Compiler Searches for Files

- **Library files** are files containing machine code that the program will link to during the linking step of compiling.
- Places specified in the compiling command using the **-L** flag,
- For example: **gcc -L/soft/fftw/lib -o myexe source.c**
- Places referenced by environmental variables: **LD_LIBRARY_PATH, LIBRARY_PATH**
- Default locations like: /usr/lib, /usr/lib64, etc

Where the Compiler Searches for Files

- It is possible to alter where the compiler searches for files by altering environment variables:
- **export LD_LIBRARY_PATH=/soft/fftw/lib:$LD_LIBRARY_PATH**
- This would make the compiler search within /soft/fftw/lib directory
- Equivalent at compile time:
- **icc -L/soft/fftw/lib mysource.c**

Additional Tips

- When a compiler module is loaded it makes changes to the environment, mostly by altering environmental variables. To see what a module does use the command: **module show module-name**
- By examining what changes the module makes you can determine where the library and header files are located. For compiling some complex programs you may need to specify these locations using the **-L** and **-I** flags.
- Sometimes compiling is sensitive to the order in which the library files are linked. If linking with libraries seems to fail you may need re-order the linking flags in the compile command.
- It is generally unsafe to have multiple module versions of the **same** compiler loaded simultaneously.

Utilities for Examining the Compiled Files

- **file**: determine the File type
- **nm**: List symbol table of object files
- **ldd**: List dynamic-linked libraries

Bulding systems

- Most codes are complex with interdependent source files.
- Individual files need to be compiled into object files (.o) then linked into a binary (-c option)
    - gcc -c foo.c -o foo.o
    - gcc -c bar.c -o bar.o
    - gcc foo.o bar.o -o foobar.exe
- NOTE: Linking is required to satisfy **undefined external references**
- At scale (10+ files), this becomes cumbersome
- We need automation!!

GNU Make utility

GNU Make

What is GNU Make?

- GNU make is a tool which controls the generation of executables and other non-source files of a program from the program's source files.
- It automatically determines which parts to (re)build
- It uses a **"Makefile"** to keep track of which files need to be recompiled
- Uses "Date Modified" timestamp to know when changes are made to any files
- If a single dependency changes it is recompiled plus all dependent sources
- Run one command (make) and let it worry about each call to compile and link.

What is a Makefile?

- Input file specifically for the GNU Make utility
- Similar to a shell script
- Specifies **targets, rules, and dependencies** for each part of build
- **Targets:** callable scriptlets that have dependencies and rules
- **Rules:** build tasks/actions (command lines)
- **Dependencies:** control target call order

Using a Makefile

- Executing Make: >**make**
- Make looks for a file named **makefile** or **Makefile** in the current directory
- Specify a different file: >**make -f other.mk**
- Specify a **target:** >**make test.exe**
- Use all available cores: >**make -j**
  (best on a compute node, e.g. >**srun --ntasks=4**)

Profiling

- The goal of profiling is to gain insight into program execution that helps to identify any potential performance problems. These can be the algorithmic code makeup, memory management, communication, or I/O.
- Profiling allows us to pinpoint the most resource-intensive spots (hotspot) in an application. A hotspot is the parts of code the program spends most of its time executing.
- Profiling can also identify bottlenecks that affect application's performance.

Python example: standard libraries

- `time`
    - For simple runtime measurements.
- `cProfile`
    - Fast profiler
- `profile`
    - Compatible with `cProfile` but slower, useful for extending the capabilities
- `pstats`
    - For viewing profiler results generated from `cProfile` or `profile`

Task: Use a profiler to improve the following code

```
"""Generates random integers and returns how many are positive."""

import numpy as np

def is_positive(x):
    """Return True if x is a positive number, otherwise return zero."""
    return x > 0

def count_positive(arr):
    """Loop through numbers in arr and return how many are positive."""
    output = np.array([], dtype='uint32')
    for n in arr:
        output = np.concatenate([output, [is_positive(n)]])
    result = sum(output)
    return result

np.random.seed(0)
myarray = np.random.randint(low=-100, high=100, size=300000)
result = count_positive(myarray)
print('Result:', result)
```

```
Result: 148828
```

Timing code

```
import time

t0 = time.time()
result = count_positive(myarray)
print('Result:', result)
print(f'{time.time() - t0:.3g} s')
```

```
Result: 148828
4.16 s
```

Viewing profiler results

```
import cProfile

profiler = cProfile.Profile()
profiler.enable()
result = count_positive(myarray)
profiler.disable()

# Write to a file for viewing later
profiler.dump_stats('code_profile')

print('Result:', result)
profiler.print_stats(sort='cumtime')
```

Viewing profiler results

```
Result: 148828
         1200004 function calls in 4.265 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.114    0.114    4.265    4.265 python-RS8HnE:9(count_positive)
   300000    0.060    0.000    4.091    0.000 <__array_function__ internals>:2(conc
   300000    4.020    0.000    4.020    0.000 {built-in method numpy.core._multiarr
   300000    0.039    0.000    0.039    0.000 python-RS8HnE:5(is_positive)
        1    0.020    0.020    0.020    0.020 {built-in method builtins.sum}
   300000    0.011    0.000    0.011    0.000 multiarray.py:148(concatenate)
        1    0.000    0.000    0.000    0.000 {built-in method numpy.array}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profile
```

Addressing the hotspots

```
def count_positive(arr):
    result = sum(is_positive(arr))
    return result

profiler2 = cProfile.Profile()

profiler2.enable()
result2 = count_positive(myarray)
profiler2.disable()

# Make sure the new result is identical!
assert result == result2
profiler2.dump_stats('code_profile2')
print('Result2:', result2)

profiler2.print_stats(sort='cumtime')
```

Addressing the hotspots

```
Result2: 148828
         4 function calls in 0.288 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.288    0.288 python-xmbMGk:1(count_positive)
        1    0.288    0.288    0.288    0.288 {built-in method builtins.sum}
        1    0.000    0.000    0.000    0.000 python-RS8HnE:5(is_positive)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profile
```

Profiling Python code from the command line

```
python -m cProfile -o profiler_results myscript.py
python -m pstats profiler_results  # Opens interactive viewer
```

Gotchas

- Use tests to verify that your optimized code yields the same results as your old code
- Avoid premature optimization
  - It can be better to start out writing code that is easy to read and to validate *before* sacrificing these things for perceived performance gains
  - Don't spend time guessing: profile your code
- There's no free lunch: profilers incur an overhead, so keep that in mind and calibrate the profiler when it matters.

Thank you!

- If you have questions:
- help@msi.umn.edu
- lamx0031@umn.edu
- mance012@umn.edu