LA-UR-17-24453
Approved for public release; distribution is unlimited.

Title: Compiling on Linux Clusters

Author(s): Garrett, Charles Kristopher

Intended for: Presentation for LANL's Parallel Computing Summer Research Internship

Issued: 2017-06-02

# Compiling on Linux Clusters

**Kris Garrett**

June 2017

# Compiling Stages

- **Three stages of compiling**
  - Preprocessing
  - Compiling to object files
  - Linking

} Usually done in one step

# Stage 1: Preprocessing

- **Acts only on preprocessing directives in code**
  - Lines that begin with #
- **Examples**
  - `#include “file.h”`
    - Replace line with contents in `file.h`
  - `#ifdef NAME`
    - Only insert code if NAME is defined
  - `#define N 100`
    - Replace N with 100 everywhere below this line (string replacement)
- **This is a rudimentary language to change your source code**
- **To only do preprocessing:** `gcc –E`

# Stage 1: Preprocessing

- **Uses**
  - Platform check: `#ifdef __GNUC__`
  - Debug version: `#ifdef DEBUG`
  - Macros: `#define MAX(a,b) ((a) > (b) ? (a) : (b))`
  - Include guards:

```
#ifndef __FILE_H__
#define __FILE_H__
... Code in header file ...
#endif
```

  - Language additions with pragma:

```
#pragma omp parallel
#pragma ivdep
```

# Stage 2: Compiling

- **Compiles each source file into an object file ending in `.o`**
  - `gcc -c file.c`
  - Output: `file.o`
  - Also calls the preprocessor

- **Object files are machine code (possibly optimized)**
  - Missing addresses to global functions and variables

- **Allows recompiling only a few files when making small changes**
  - Saves a lot of time for builds that take hours

# Stage 3: Linking

- **Link together `.o` files to create executable**
  - `gcc file1.o file2.o -o program.x`
- **This invokes the linker `ld`**
  - You can use `ld` directly if you want
- **You can compile and link with one command**
  - `gcc file1.c file2.c -o program.x`
  - Convenient for small programs

# Linking with a library

- **Two library types: static and dynamic**
  - Static: all machine code is copied into the executable
    - Name: `liblibrary.a`
  - Dynamic: only hooks into the library are put into the executable
    - When executable is run: need to know where the library is
    - Name: `liblibrary.so`

- **If dynamic link library is not in standard location: Two Options**
  1. Set `LD_LIBRARY_PATH=<path_to_library>:$LD_LIBRARY_PATH`
  2. Set `rpath` in link command
     `gcc ... -Wl,-rpath=<path_to_library>`

# Linking with a library

- **Link command**
  - `gcc file1.o file2.o \`
    `-L/path/to/library -llibrary -o program.x`
  - `-L` flag: Add library path if library is not in standard location
  - `-l` flag: link either `liblibrary.a` or `liblibrary.so`

- **Link order matters**
  - Library name needs to be after all source/object/library files that use the library
  - Best practice: put library after all source/object files
  - `gcc file1.o file2.o -llib1 -llib2`
    - Files 1 and 2 depend on libraries and lib1 depends on lib2

# Useful Compile Flags

- **Debug**
  - `-g` (Adds source code information into executable)
- **Optimize**
  - `-O2` or `-O3` (capital letter Oh)
  - `-march=...` (Specify CPU architecture)
- **To profile executable: enable both debug and optimize flags**

# Useful Compile Flags

- **Include directory**
  - `-I/path/to/include` (Directory to check for `#include`)
- **Warning flags**
  - `-Wall -Wextra`
- **OpenMP**
  - GCC: `-fopenmp`
  - Intel: `-qopenmp`

# Common Build Processes

- **Autotools**
  - `./configure` `make` `make install`
  - Can change several options by adding parameters to configure
  - May need `sudo make install` if installing to a system directory

- **cmake**
  - `cmake .` `<change options>` `make` `make install`
  - Change options by
    - editing `CMakeCache.txt`
    - `ccmake .`

# Common Build Processes

- **What does configure do?**
  - Setup compile and link flags
    - Debug vs Optimized
  - Set library paths
  - Set install directory
  - Make sure compiler exists and can compile up to your standard
  - Create makefiles

# Compiling with MPI

- **Will need at least two modules**
  - Load compiler and then MPI (in this order)
  - Example: `module load gcc openmpi`

- **Use compiler wrapper to build**
  - Same wrapper used for many compiler vendors: GCC, Intel, PGI, ...
  - Links to MPI libraries
  - OpenMPI wrappers
    - `mpicc` for C
    - `mpicxx` for C++
    - `mpifort` for Fortran

# Compiler Wrappers are not Magic

```
$ mpicc --showme
gcc
-I/usr/projects/hpcsoft/toss2/moonlight/openmpi/1.10.5-gcc-5.3.0/include/openmpi/opal/mca/hwloc/hwloc191/hwloc/include
-I/usr/projects/hpcsoft/toss2/moonlight/openmpi/1.10.5-gcc-5.3.0/include/openmpi/opal/mca/event/libevent2021/libevent
-I/usr/projects/hpcsoft/toss2/moonlight/openmpi/1.10.5-gcc-5.3.0/include/openmpi/opal/mca/event/libevent2021/libevent/include
-I/usr/projects/hpcsoft/toss2/moonlight/openmpi/1.10.5-gcc-5.3.0/include
-I/usr/projects/hpcsoft/toss2/moonlight/openmpi/1.10.5-gcc-5.3.0/include/openmpi
-pthread
-Wl,-rpath -Wl,/usr/projects/hpcsoft/toss2/moonlight/openmpi/1.10.5-gcc-5.3.0/lib
-L/usr/projects/hpcsoft/toss2/moonlight/openmpi/1.10.5-gcc-5.3.0/lib -lmpi
```

# Pinning Ranks

- **Depending on setup: MPI processes can float between cores**
- **Some applications get better performance if ranks are ‘pinned’ to a core (or socket or hyperthread)**
  - Pinning means the process (rank) doesn’t change cores
  - Use: on core resources don’t need to move such as cached data
- **The following examples are run on:**
  - 2 sockets
  - 18 cores per socket

# Pinning Ranks

```
$ srun --cpu_bind=verbose -n 4 ./hello.x
cpu_bind=MASK - sn089, task  1  1 [45589]: mask 0xffffffffff set
cpu_bind=MASK - sn089, task  2  2 [45590]: mask 0xffffffffff set
cpu_bind=MASK - sn089, task  3  3 [45591]: mask 0xffffffffff set
cpu_bind=MASK - sn089, task  0  0 [45588]: mask 0xffffffffff set
Rank 2 of 4
Rank 3 of 4
Rank 0 of 4
Rank 1 of 4
```

Bit mask of possible process placement

Ranks can bind to any cores

# Pinning Ranks

```
$ srun --cpu_bind=verbose,cores -n 4 ./hello.x
cpu_bind=MASK - sn089, task  0  0 [45666]: mask 0x00001 set   (Core 0)
cpu_bind=MASK - sn089, task  1  1 [45667]: mask 0x40000 set   (Core 18)
cpu_bind=MASK - sn089, task  2  2 [45668]: mask 0x00002 set   (Core 1)
cpu_bind=MASK - sn089, task  3  3 [45669]: mask 0x80000 set   (Core 19)
Rank 0 of 4
Rank 1 of 4
Rank 2 of 4
Rank 3 of 4
```

Bit mask of possible process placement

Ranks bind to one core

# Threading with OpenMP

- **Set number of threads**
  - `export OMP_NUM_THREADS=2`
- **Bind threads to cores (if desired)**
  - `export OMP_PROC_BIND=true`
  - `export OMP_PLACES=cores`
- **Check OpenMP bindings**
  - `export OMP_DISPLAY_ENV=VERBOSE`
- **Each rank should bind to at least `OMP_NUM_THREADS` cores (or hyperthreads)**

# MPI and OpenMP

```
$ srun --cpu_bind=verbose,cores -n 4 -c 2 ./hello_omp.x
cpu_bind=MASK - sn089, task  0  0 [45405]: mask 0x000003 set (Cores 0,1)
cpu_bind=MASK - sn089, task  1  1 [45406]: mask 0x0c0000 set (Cores 18,19)
cpu_bind=MASK - sn089, task  2  2 [45407]: mask 0x00000c set (Cores 2,3)
cpu_bind=MASK - sn089, task  3  3 [45408]: mask 0x300000 set (Cores 20,21)

Rank 1    Thread 0
Rank 1    Thread 1
Rank 2    Thread 0
Rank 2    Thread 1
Rank 3    Thread 0
Rank 3    Thread 1
Rank 0    Thread 0
Rank 0    Thread 1
```

MPI process placement
Ranks bind to two cores
One core per OpenMP thread

# MPI and OpenMP

```
OPENMP DISPLAY ENVIRONMENT BEGIN (for rank 0)
_OPENMP = '201307'
OMP_DYNAMIC = 'FALSE'
OMP_NESTED = 'FALSE'
OMP_NUM_THREADS = '2'
OMP_SCHEDULE = 'DYNAMIC'
OMP_PROC_BIND = 'TRUE'
OMP_PLACES = '{0},{1}'
...
OPENMP DISPLAY ENVIRONMENT END
```

Rank 0 Thread 0 pinned to core 0
Rank 0 Thread 1 pinned to core 1

```
--- OMP_PLACES for all ranks ---
 Rank 0       Rank 1       Rank 2         Rank 3
'{0},{1}'   '{2},{3}'   '{18},{19}'   '{20},{21}'
```

# The End