Compiling code and using MPI

scitas.epfl.ch

August 25, 2015

Welcome

What you will learn

How to compile and launch MPI codes on the SCITAS clusters along with a bit of the "why"

What you will not learn

How to write parallel code and optimise it - there are other courses for that!

Compilation

From code to binary

Compilation is the process by which code (C, C++, Fortran etc) is transformed into a binary that can be run on a CPU.

CPUs are not all the same

- CPUs have different features and instruction sets
- The same code will need to be recompiled for different architectures

What is MPI?

What is MPI?

- Message Passing Interface
- De facto standard for distributed memory parallelisation
- Open standard with multiple implementations - now at version 3.0
- Scales to very large systems

Shared vs Distributed Memory

- Shared - all tasks see all the memory (e.g. OpenMP)
- Distributed - tasks only see a small part of the overall memory

Clusters are distributed memory systems so MPI is well suited.

MPI Terminology

Words that you are going to hear

- Rank - how MPI tasks are organised
- Rank 0 to N - the "worker" tasks
- Hybrid - a code that combines shared memory parallelisation with MPI

Pure MPI codes generally run one rank per core.

Compilers - Intel vs GCC

GNU Compiler Collection

- The industry standard and available everywhere
- Quick to support new C++ language features
- Fortran support used to be poor

Intel Composer

- Claims to produce faster code on Intel CPUs
- Better Fortran support
- Generally much stricter with bad code!

MPI - Intel vs MVAPICH2 vs OpenMPI

Why are there different flavours?

There are multiple MPI flavours that comply with the specification and each claims to have some advantage over the other. Some are vendor specific and others are open source

The main contenders

- Intel MPI - commercial MPI with support
- MVAPICH2 - developed by Ohio uni for Infiniband
- OpenMPI - Open source and widely used

In SCITAS we support IntelMPI and MVAPICH2

Compiler and MPI choice

First choose your compiler

- GCC or Intel
- This might be a technical or philosophical choice

The associated MPI is then

- GCC with MVAPICH2
- Intel with IntelMPI

This is a SCITAS restriction to prevent chaos - nothing technically stops one from mixing!

Both work well and have good performance.

The dark art of mangling

Mangling?

Mechanism to allow multiple functions with the same name

C/C++

- GCC - `_ZN5NOMAD10Eval_PointD2Ev`
- Intel - `_ZN5NOMAD10Eval_PointD2Ev`

Result: C/C++ libraries are compatible between GCC and Intel

Fortran

- GCC - `__h5f_MOD_h5fget_access_plist_f`
- Intel - `h5f_mp_h5fget_access_plist_f_`

Result: Fortran libraries are not compatible between GCC and Intel!

Linking

How to use libraries

Linking is the mechanism by which you can use libraries with your code.

- static - put everything in your executable
- dymanic - keep the libraries outside and load them as needed

Dynamic by default

There are very few reasons to statically link code.

What is linked?

ldd is your friend

```
ldd run.x
```

```
linux-vdso.so.1 => (0x00007fffbfcf5000)

llibmkl_intel_lp64.so =>

/ssoft/intel/15.0.0/RH6/all/x86_E5v2/composer_xe_2015.2.164/mkl/lib/intel64/libmkl_intel_lp64.so

libmkl_intel_thread.so =>

/ssoft/intel/15.0.0/RH6/all/x86_E5v2/composer_xe_2015.2.164/mkl/lib/intel64/libmkl_intel_thread.so

..

libpthread.so.0 => /lib64/libpthread.so.0 (0x0000003acb200000)

libc.so.6 => /lib64/libc.so.6 (0x0000003acae00000)

libdl.so.2 => /lib64/libdl.so.2 (0x0000003acb600000)

/lib64/ld-linux-x86-64.so.2 (0x0000003aca600000)
```

Example 1 - Build sequential 'Hello World'

Compile the source files

```
gcc -c output.c
gcc -c hello.c
```

Link

```
gcc -o hello output.o hello.o
```

Run

```
./hello
Hello World!
```

Modules

How software is organised on the clusters

Modules is utility that allows multiple, often incompatible, tools and libraries to exist on a cluster.

Naming convention

- name / version / compiler
- hdf5/1.8.14/gcc-4.4.7
- The MPI flavour is implicit!

Note - compilers are backwards compatible so there is no need to have hdf5/1.8.14/gcc-4.8.3!

More Modules

Commands

- module purge
- module load gcc/4.8.3
- module load mvapich2/2.0.1/gcc-4.4.7
- module load hdf5/1.8.14/gcc-4.4.7
- module list
- module show gcc/4.8.3

At present Modules will not prevent you from loading incompatible modules!

MPICC and friends

mpicc / mpiicc / mpicxx / mpif77 / mpif90 / mpiifort

These are wrappers to the underlying compiler that add the correct options to link with the MPI libraries

- mpicc - C wrapper
- mpiicc - Intel C wrapper
- mpiifort - Intel Fortran Compiler

Check the MPI flavour documentation for more details

mpicc mycode.c

To use the wrappers simply type:

- module load mympiflavour/version
- mpicc hello.c -o hi

Example 2 - Build // MPI-based 'Hello World'

Load modules

```
module load intel intelmpi
```

Compile-link

```
mpiicc -g -o hello_mpi hello_mpi.c
```

Run two tasks on two different nodes

```
srun -N2 -n2 --partition=debug ./hello_mpi
Hello world:  I am task rank 1, running on node
'b292'
Hello world:  I am task rank 2, running on node
'b293'
```

Configure and Make

The traditional way to build packages

- `./configure --help`
- `./configure --prefix=X --option=Y`
- `make`
- `make install`

cmake

cmake is a better way to do things!

- `cmake -DCMAKE_INSTALL_PREFIX:PATH=X -DOption=Y .`
- `make`
- `make install`

If you're starting a project from scratch then we recommend using cmake rather than configure. There's also a graphic interface called `ccmake`.

MPI and the Batch System

Telling SLURM what we need

We would like 64 processes over 4 nodes

```
#SBATCH --nodes 4
#SBATCH --ntasks-per-node 16
#SBATCH --cpus-per-task 1
#SBATCH --mem 32000
```

Remember that the memory is per node!

Alternative formulations

We would like 64 processes

```
#SBATCH --ntasks 64
#SBATCH --cpus-per-task 1
#SBATCH --mem 32000
```

SLURM will find the space for 64 tasks on as few nodes as possible

We would like 16 processes each one needing 4 cores

```
#SBATCH --ntasks 16
#SBATCH --cpus-per-task 4
#SBATCH --mem 32000
```

SLURM will allocate 64 cores in total

srun and mpirun

Launching a MPI job

Now that we have a MPI code we need some way of correctly launching it across multiple nodes

- `srun` - SLURM's built in job launcher
- `mpirun` - "traditional" job launcher

To use this we type

```
srun mycode.x
```

With the directives on the previous slide this will launch 64 processes on 4 nodes

Intel mpirun

Using IntelMPI and mpirun

On our clusters IntelMPI is configured to work with srun by default. If you want to use mpirun then do as follows:

- `unset I_MPI_PMI_LIBRARY`
- `mpirun ./mycode.x`

We don't advise doing this and strongly recommend using srun!

CPU affinity

Kesako?

CPU affinity is the name for the mechanism by which a process is bound to a specific CPU (core) or a set of cores.

Pourquoi?

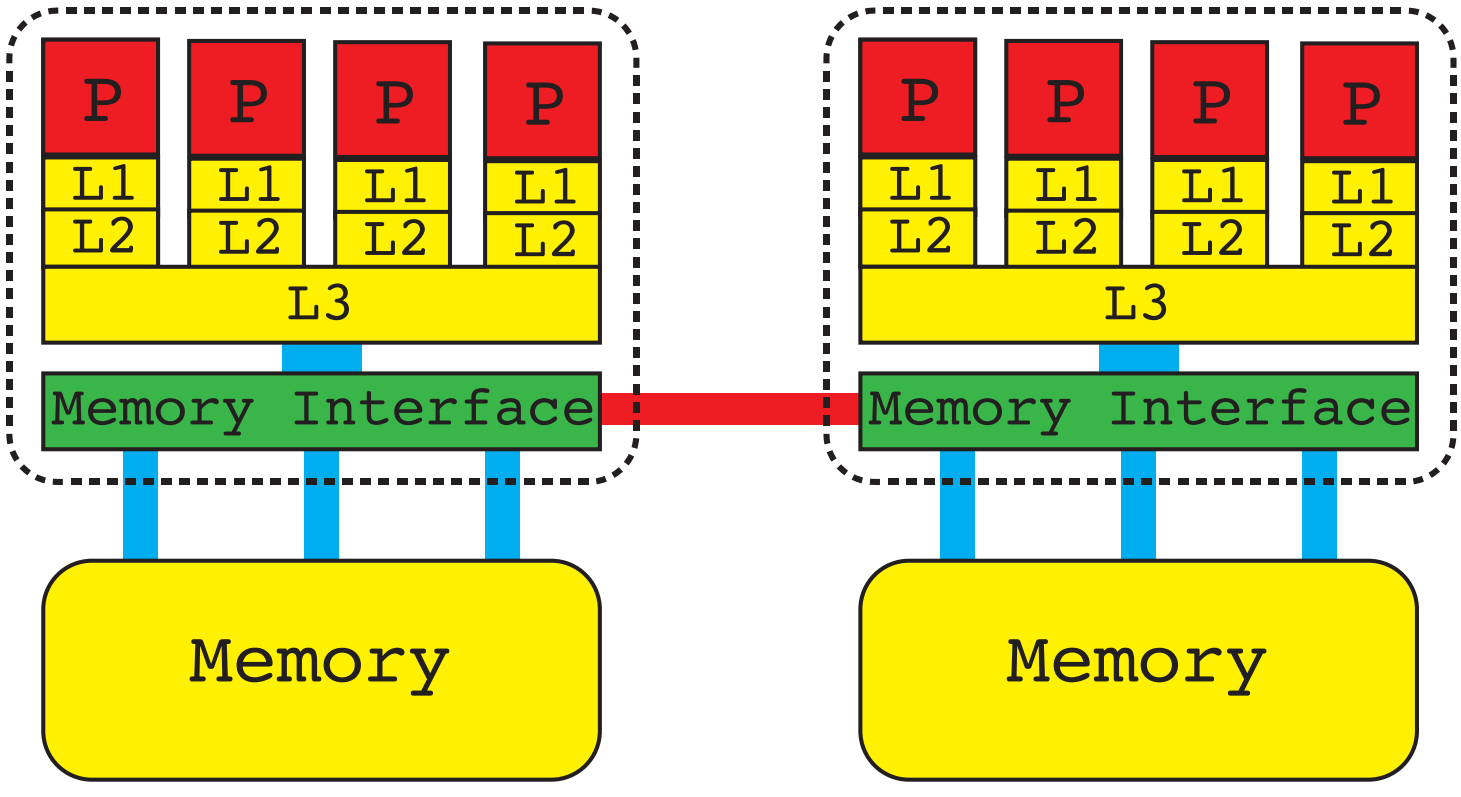
If a mask is not set the OS might place the task on different cores every 100ms or so. For performance this can be a very bad thing to do.

We can also optimise placement of ranks with respect to the underlying hardware.

ccNUMA

Cache Coherent Non Uniform Memory Architecture

This is what compute nodes with more than one processor look like...



CPU bitmasks

11000000

When talking about affinity we use the term "mask" or "bit mask" which is a convenient way of representing which cores are part of a CPU set.

If we have an 8 core system then the following masks mean:

- 10000000 - core 8
- 01000000 - core 7
- 00100000 - core 6
- 11110000 - cores 5 to 8
- 00001111 - cores 1 to 4

CPU bitmasks

11110000 is f0

These numbers can be conveniently written in hexadecimal so if we query the system regarding CPU masks we will see something like:

pid 8092's current affinity mask: 1c0
pid 8097's current affinity mask: 1c0000

In binary this would translate to

pid 8092's current affinity mask: 000000000000000111000000
pid 8097's current affinity mask: 000111000000000000000000

Binding with srun

Examples

```
srun -N 1 -n 4 -c 1 --cpu_bind=verbose rank ./hi 1
cpu_bind=RANK - b370, task 0 :  mask 0x1 set
cpu_bind=RANK - b370, task 1 :  mask 0x2 set
cpu_bind=RANK - b370, task 2 :  mask 0x4 set
cpu_bind=RANK - b370, task 3 :  mask 0x8 set

srun -N 1 -n 4 -c 4 --cpu_bind=verbose,sockets ./hi 1
cpu_bind=MASK - b370, task 1 :  mask 0xff00 set
cpu_bind=MASK - b370, task 2 :  mask 0xff set
cpu_bind=MASK - b370, task 0 :  mask 0xff set
cpu_bind=MASK - b370, task 3 :  mask 0xff00 set
```

Common errors

Compiled on a different machine

```
Please verify that both the operating system and the
processor support Intel MOVBE, FMA, BMI, LZCNT and
AVX2 instructions.
```

Module not loaded - LD_LIBRARY_PATH

```
./run.x
./run.x:  error while loading shared libraries:
libmkl_intel_lp64.so:  cannot open shared object file:
No such file or directory
```

If things don't work

Try interactively

Errors are much more visible this way

- `salloc -N 2 -n 32 -t 01:00:00 --partition debug`
- `srun mycode.x < inp.in`

Check what's going on with htop and ps

- `ssh b123`
- `htop`
- `ps auxf`

If things still don't work

Crashes or won't start

- Reference input files
- GDB
- TotalView Debugger

Crashes after a while

Memory Leak?

- Check with Ganglia
- MemoryScape (TotalView)
- Valgrind

Going further

SCITAS offers courses in

- MPI, an introduction to parallel programming
- MPI, advanced parallel programming
- Introduction to profiling and software optimisation
- Computing on GPUs

Exercise - Build Octopus

Download package

http://www.tddft.org/programs/octopus

Hints

- load modules:

```
intel/15.0.0 intelmpi/5.0.1 fftw/3.3.4/intel-15.0.0 gsl/1.16/intel-15.0.0
```

- build first libxc
- some configure options to use for // octopus:

```
--enable-openmp --enable-mpi

--disable-zdotc-test

--with-blas="-L${MKL_ROOT}/lib/intel64 -lmkl_intel_lp64 -lmkl_core -lmkl_intel_thread -lpthread"

--with-fft-lib="-L${FFTW_LIBRARY} -lfftw3 -lfftw3_threads"
```