Handout for "Compiling code and using MPI"

- Introduction
- The ugly realities of hardware
  - This is hardware
  - Infiniband
  - NUMA
  - CPU affinity
  - SIMD instructions and marketing numbers
- Parallel programming
  - SIMD
  - SPMD
    - Shared Memory
    - Distributed Memory
    - MPI
    - OpenMP
- Compiling Code
  - Modules
    - No modules loaded
    - Compiler loaded
    - Compiler and MPI loaded
    - Compiler, MPI and Linear Algebra Library loaded
  - Compilation
  - Linking
    - Compiling and linking a simple code
    - Making a shared library
    - Using a shared library
    - What's linked?
    - Libraries (modules) that you should know about
- Build Tools
  - GNU autotools
  - CMake
- Optimisation
  - Compilers are lazy
  - Compilers are stupid
- mpicc and friends
- Compiling OpenMP codes
- Options for Intel and GCC
- Running parallel codes
  - Job launchers
  - Specifying what you want
    - OpenMP Code
    - MPI Code
    - Hybrid Code with open rank per node
    - Hybrid code with multiple ranks per node
  - Specifying where you want code to run
    - Bind by rank
    - Bind to sockets
    - Bind with whatever mask you feel like
    - Mismatches
    - OpenMP binding
- Runtime errors
- Going Further
  - Courses
  - References and further reading

Target Audience

The intended audience is a new PhD student who has done a bit of programming and is told by their supervisor "Here's the code written by X who has left - compile and run it on the SCITAS cluster".

The aim of this course is to make you aware of a number of subtle problems that can arise when you compile and run parallel codes on compute clusters.

We are not going to discuss how to write code - there are other SCITAS courses for that.

You are not expected to understand everything in the course but at least you will know that it exists.
The course covers the following topics

- The realities of hardware
- Very basic concepts of parallel programming
- Compiling and linking codes
- Running a parallel code on the clusters

The first two can be considered the "why" and the second two the "how".

There are no practical exercises as part of the course.

The ugly realities of hardware

This is hardware

A compute node looks a bit like:

The important bits are the two CPUs each with RAM and the Infiniband card in the upper right-hand corner.

Infiniband

Latency is often more important than bandwidth for parallel codes which is why HPC clusters have a special interconnect.

<table>
<thead>
<tr>
<th>Interconnect</th>
<th>Bandwidth</th>
<th>Latency 8 bytes</th>
<th>Message Rate 8 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 GE Ethernet</td>
<td>10 Gb/s</td>
<td>~8 us</td>
<td>~1 million/s</td>
</tr>
<tr>
<td>FDR Infiniband</td>
<td>56 Gb/s</td>
<td>0.7 us</td>
<td>137 million/s</td>
</tr>
<tr>
<td>OmniPath</td>
<td>100 Gb/s</td>
<td>0.6 us</td>
<td>200 million/s</td>
</tr>
</tbody>
</table>
EDR Infiniband  |  100 Gb/s  |  0.6 us  |  150 million/s

This isn't something that we will discuss further during the course but it is something to be aware of if/when you want to run parallel codes at large scale and is why a significant part of the cost of real HPC systems is the fast interconnect.

**NUMA**

Or as it should be, ccNUMA, *cache coherent non uniform memory architecture*. All modern multi socket computers look something like the diagram below with multiple levels of memory and some of this being distributed across the system.

Cache coherence is the name for the process that ensures that if one core updates information that is also in the cache of another core the change is propagated.

The bandwidth decreases and the latency increases as we move further away from the processor itself with access to the main memory taking roughly 50 times as long as from the L1 cache. The bandwidth between the two processors is significantly lower than between a processor and its "local" memory.

Memory is allocated by the operating system when asked to do so by your code (e.g. via malloc) but the physical location is not defined until the moment at which the memory page is accessed (i.e. written to for the first time). The default is to place the page on the closest physical memory (the memory directly attached to the socket) as this provides the highest performance. If the thread accessing the memory moves to the other socket the memory will not follow!

**CPU affinity**

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

The aim is to improve performance by

- keeping a thread near the memory that it is accessing
- preventing the invalidating and reloading of L1/L2/L3 cache if a thread gets moved to another core

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 mask means that the process is bound to CPUs 7 & 8.

```
11000000
```

This number can be conveniently written in hexadecimal as c0 (192 in decimal) and 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: 000111000000
pid 8097's current affinity mask: 000111000000000000000000
```

This shows that the OS scheduler has the choice of three cores on which it can run these single threads.
These masks can be seen (and changed) using the taskset tool:

```
$ taskset -p <PID of the process>
```

Hybrid codes, that is to say mixing MPI with threads or OpenMP also present a challenge. By default Linux threads inherit the mask of the spawning process so if you want your threads to have free use of all the available cores please take care! This is discussed more in the Running Parallel Codes section.

**SIMD instructions and marketing numbers**

Or why processing "power" has increased while clock speeds have decreased.

SIMD = Single Instruction Multiple Data

We generally talk about "vectorisation" and operating on a vector of a certain size composed of multiple values.

### SIMD Operation of Vector Length 4

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_x$</td>
<td>+</td>
<td>$B_x$</td>
</tr>
<tr>
<td>$A_y$</td>
<td>+</td>
<td>$B_y$</td>
</tr>
<tr>
<td>$A_z$</td>
<td>+</td>
<td>$B_z$</td>
</tr>
<tr>
<td>$A_w$</td>
<td>+</td>
<td>$B_w$</td>
</tr>
</tbody>
</table>

*Intel® Architecture currently has SIMD operations of vector length 4, 8, 16*

---

A Floating point double/real uses 64 bits

- 256 bit vectors (AVX) are composed of 4 double precision numbers so 4 operations at a time
- 512 bit vectors (AVX512) are composed of 8 double precision numbers so 8 operations at a time

Add Fused-Multiply-Add (FMA) and we can double that to get ridiculous FLOPS values as we perform two operations (multiple and add) in one cycle.

One Node on Gacrux with two Intel Sylake processors has a theoretical performance of 2.3 TFLOPS:

$$2.6x10^{9} \text{ (Hz) x 14 (cores per socket) x 2 (number of sockets) x 8 (number of doubles per operation) X 2 (number of FP math units per core) x 2 (FMA so two operations per cycle) = marketing numbers}$$
If we multiply this by the number of nodes we get the "peak" theoretical floating point performance (Rpeak) of a cluster.

A few examples of the leading HPC systems and their performance and efficiency are given below:

<table>
<thead>
<tr>
<th>Machine</th>
<th>HPL Performance</th>
<th>% of Rpeak</th>
<th>HPCG Performance</th>
<th>% of Rpeak</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summit (ORNL)</td>
<td>122,300 PF</td>
<td>65%</td>
<td>2,926 PF</td>
<td>1.5%</td>
</tr>
<tr>
<td>K computer (Riken)</td>
<td>10,510 PF</td>
<td>93%</td>
<td>0.603 PF</td>
<td>5.3%</td>
</tr>
<tr>
<td>Piz Daint (CSCS)</td>
<td>19,590 PF</td>
<td>77%</td>
<td>0.486 PF</td>
<td>1.9%</td>
</tr>
<tr>
<td>Sunway TaihuLight</td>
<td>93,015 PF</td>
<td>74%</td>
<td>0.481 PF</td>
<td>0.4%</td>
</tr>
</tbody>
</table>

HPCG (High Performance Conjugate Gradient) provides an alternative to the well known "HPL" benchmark for ranking supercomputers and represents certainly widely used types of codes that are more limited by memory and interconnect bandwidth than by raw floating point capacity.

Parallel programming

**SIMD**

As we've already seen Single Instruction Multiple Data vectorisation is a form of parallel processing and is probably the easiest way to make your application run faster - in general this is something the compiler does for you and it's rare to hand code SIMD instructions.

At the SIMD level we are looking at making each individual thread as fast as possible.

In order to reach the 2.3 TFLOPS for a node we saw that a factor of 16 comes from SIMD so if you do not exploit this then you are potentially wasting 95% of the compute power!

**SPMD**

Single Program Multiple Data is generally what we refer to as parallel programming. Multiple copies/instances of the same program run on separate cores or nodes and synchronise occasionally. This is different to SIMD where all operations are synchronous. Unlike SIMD which is performed by the compiler SPMD involves you writing to code to implement the programming model.

There are a few different ways to use the SPMD model and these are discussed below.

**Shared Memory**

In a shared memory model all data (memory) are visible to all threads (workers) in your application. This means that they can get required data easily but also means that one has to be careful that when updating memory only one thread is writing to a particular address.

The main limitation is one of system size and anything more than a 2 socket system and 1TB of memory gets very expensive. The largest systems available have 32 processors with 48TB of memory.

**Distributed Memory**

Here the workers only see a small part of the overall memory and if they need data that are visible to another worker they have to explicitly ask for it. The advantage is that, if the algorithm scales, we can use more nodes to increase the performance and/or problem size. All the largest HPC systems use such a distributed memory model.

**MPI**

MPI is the de-facto distributed memory model. There is a specification agreed by the MPI Forum and anybody can implement their own code that adheres (or tries to adhere) to the standard. MPI 2.0 was agreed in 1996 and MPI version 4.0 is in the process of being defined with version 3.1 being the widely implemented and used standard.

There are other approaches to distributed memory programming such as PVM but 99% of codes use MPI.

Some widely used MPI distributions (implementations) are:

- MPICH2
- MVAPICH2
Of these MPICH2, MVAPICH2 and Intel MPI share a common ancestry and behave in a similar manner.

Caveat Emptor - Just because something is in the standard doesn't necessarily mean that your preferred MPI distribution has actually implemented it.

OpenMP

OpenMP is the most widely used shared memory model in scientific computing. As with MPI there have been many versions with 4.5 being the most recent and 4.0 being widely used.

OpenMP uses multi-threading but hides the complexity of using threads directly and also supports accelerators such as GPUs.

Compiling Code

Modules

Before we look at how to compile a code it's useful to know what has already been compiled for you on the clusters.

Modules are a way of allowing multiple incompatible belief systems (compilers and MPI flavours) to co-exist in a harmonious manner.

We use LMod rather than the traditional modules and packages are organised in a three level hierarchy:

$ module load Compiler
$ module load MPI flavour
$ module load BLAS/LAPACK implementation

On the SCITAS clusters we support two main stacks - one open source and the other proprietary.

<table>
<thead>
<tr>
<th></th>
<th>Open Source &quot;A&quot;</th>
<th>Open Source &quot;B&quot;</th>
<th>Proprietary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>GCC</td>
<td>GCC</td>
<td>Intel Composer</td>
</tr>
<tr>
<td>MPI</td>
<td>MVAPICH2</td>
<td>OpenMPI</td>
<td>Intel MPI</td>
</tr>
<tr>
<td>BLAS</td>
<td>OpenBLAS</td>
<td>OpenBLAS</td>
<td>Intel MKL</td>
</tr>
</tbody>
</table>

Technically there is no reason not to mix open source and proprietary but we restrict the choice for support reasons. Once you have chosen your compiler the MPI and BLAS implementations are fixed.

Going through the steps of loading these modules we see:

No modules loaded
<table>
<thead>
<tr>
<th>package</th>
<th>version</th>
<th>version</th>
</tr>
</thead>
<tbody>
<tr>
<td>adf/2017.11</td>
<td>gcc/7.3.0</td>
<td>mercurial/4.4.1</td>
</tr>
<tr>
<td>ansys/17.1</td>
<td>git/2.17.0</td>
<td>molder/5.7</td>
</tr>
<tr>
<td>autoconf/2.69</td>
<td>hadoop/3.1.0</td>
<td>sbt/1.1.4</td>
</tr>
<tr>
<td>automake/1.16.1</td>
<td>intel/18.0.2</td>
<td>scala/2.11.11</td>
</tr>
<tr>
<td>cfdplusplus/16.1</td>
<td>jdk/8u141-b16</td>
<td>smr/2017.06</td>
</tr>
<tr>
<td>cmake/3.11.1</td>
<td>libtool/2.4.6</td>
<td>spark/2.3.0</td>
</tr>
<tr>
<td>comsol/5.3</td>
<td>likwid/4.3.0</td>
<td>tar/1.30</td>
</tr>
<tr>
<td>curl/7.59.0</td>
<td>m4/1.4.18</td>
<td>tmux/2.7</td>
</tr>
<tr>
<td>fdtd/8.19.1416-1</td>
<td>maple/2017</td>
<td>totalview/2017.2.11</td>
</tr>
<tr>
<td>gaussian/g16-A.03</td>
<td>mathematica/11.1.1</td>
<td></td>
</tr>
<tr>
<td>gcc/6.4.0</td>
<td>(D)</td>
<td>matlab/R2018a</td>
</tr>
</tbody>
</table>

Where:

D: Default Module

This is the list of modules that depend only on the operating system.

**Compiler loaded**
Here we see the previous list of modules as well as a new list that contains packages that are compiled with the (Intel) compiler.

**Compiler and MPI loaded**
[user@fidis ~]$ module load intel-mpi

[user@fidis ~]$ module avail

/ssoft/spack/paien/v2/share/spack/lmod/linux-rhel7-x86_E5v4_Mellanox/intel-mpi/2018.2.199-u525w4z/intel/18.0.2
boost/1.67.0-mpi
gromacs/2018.1-mpi
cgal/4.12
fftw/3.3.7-mpi
fftw/3.3.7-mpi-openmp
foam-extend/4.0

/ssoft/spack/paien/v2/share/spack/lmod/linux-rhel7-x86_E5v4_Mellanox/intel/18.0.2
argtable/2-13
bedtools2/2.27.1
binutils/2.29.1
bison/3.0.4
bzip2/1.0.6
eigen/3.3.4
fftw/3.3.7-mpi
fftw/3.3.7-mpi-openmp
font-util/1.3.1
fontconfig/2.12.3
gdb/8.1
gdbm/1.14.1
gmp/6.1.2
gettext-introspection/1.49.2
gperf/3.0.4
gsl/2.4
hdf5/1.10.1
htslib/1.8
icu4c/60.1
intel-mkl/2018.2.199
intel-mpi/2018.2.199
intel-tbb/2018.2
jdk/8u141-b16
libffi/3.2.1
libjpeg-turbo/1.5.3
libpng/1.6.34
libsigsegv/2.11

----- /ssoft/spack/stable/share/spack/lmod/linux-rhel7-x86_E5v4_Mellanox/Core -----
adf/2017.111
ansys/17.1
autoconf/2.69
automake/1.16.1
cmake/3.11.1
comsol/5.3
curl/7.59.0
fdtd/8.19.1416-1
gaussian/g16-A.03
gcc/6.4.0

Compiler, MPI and Linear Algebra Library loaded

[user@fidis ~]$ module load intel-mkl
module avail

/ssoft/spack/paien/v2/share/spack/lmod/linux-rhel7-x86_E5v4_Mellanox/intel/18.0.2
arpack-ng/3.5.0-mpi

cp2k/5.1-mpi

cp2k/5.1-mpiplumed

cpmd/v4.1

elpa/2016.05.004

gromacs/2016.4-mpi-plumed

hp1/2.2

lammps/20180316-mpi

/ssoft/spack/paien/v2/share/spack/lmod/linux-rhel7-x86_E5v4_Mellanox/intel-mkl/2018.2.199-3uufzzq/intel/18.0.2
arpack-ng/3.5.0 (D)    r/3.5.0    suite-sparse/5.2.0

boost/1.67.0-mpi

cgal/1.12

fftw/3.3.7-mpi

fftw/3.3.7-openmp

fftw/3.3.7-openmp

font-util/1.3.1

gdb/8.1

gmp/6.1.2

gobject-introspection/1.49.2

gperf/3.0.4

gsl/2.4

hdf5/1.10.1 (D)

hisat2/2.1.0

htslib/1.8

icu4c/60.1

intel/mpi/2018.2.199

intel-tbb/2018.2

jdk/8u141-b16

libffi/3.2.1

libjpeg-turbo/1.5.3

libpng/1.6.34

libsigsegv/2.11

------ /ssoft/spack/stable/share/spack/lmod/linux-rhel7-x86_E5v4_Mellanox/Core ------

adl/2017.111

ansys/1.71

autoconf/2.69

automake/1.16.1

cfdplusplus/16.1

cmake/3.11.1

comsol/5.3

curl/7.59.0

fdtd/8.19.1416-1

gaussian/gl6-A.03

gcc/6.4.0 (D)

Where:
L: Module is loaded
D: Default Module
Compilation

Compilation is the process by which human readable code (Fortran, C, C++, etc) is transformed into instructions that the CPU understands. At compilation time, one can apply optimisations and thus make the executable code run faster.

The "problem" is that optimisation is a hard job and so, by default, the compiler will do as little as possible. This means that anything compiled without asking explicitly for optimisation will be running a lot slower than it could be.

Compilation is a multi-stage process: from human readable code to assembler (which is still readable), then from assembler to a binary object.

We provide examples using the syntax of the Intel Composer. A table showing the equivalent options for the GCC compilers can be found at the bottom of the section.

To compile something we need the following information:

- The name of the source code file(s)
- The libraries to link against
- Where to find these libraries
- Where to find the header files
- A nice name for the executable

Putting this together, we get something like

```
compiler -l libraries -L <path to libraries> -I <path to header files> -o <name of executable> mycode.c
```

where the compiler could be gcc, icc, ifort or something else.

For icc we might do:

```
$ pwd
/home/user/qmcode

$ ls
lib
include
src

$ ls src/
qmsolve.c

$ ls lib/
libfastqm.so

$ ls include/
fastqm.h

$ icc -lfastqm -L/home/user/qmcode/lib -I/home/user/qmcode/include -o qmsolve src/qmsolve.c
```

To this we may add options such as:

```
-O3 -xCORE-AVX2
```
This will perform aggressive optimisation and use the features for the CORE-AVX2 architecture (Haswell) as explained later.

A code compiled with the above options will run optimally on Haswell CPUs, but will not run at all on older systems.

We could also compile a code as follows:

```
-O0 -Wall -g -check bounds
```

This performs no optimisation, gives lots of warnings and adds full debugging information in the binary as well as bounds checking for Fortran arrays.

This code will run slowly but will point out syntax problems, tell you if you make errors when accessing arrays and provides clear information when run through a debugger such as GDB or TotalView.

### Linking

In 99% of cases a better programmer than you has already had the same problem and has written code that you can reuse. In the other 1% of cases you want to easily re-use the code that you have written.

For both cases the technique to make life simpler is to use shared libraries.

It would be technically possible to write your "re-usable" code and then copy and paste it into your new code each time but this is less than optimal.

#### Compiling and linking a simple code

```
$ ls
hello.c output.c output.h

$ gcc -c output.c
$ gcc -c hello.c
$ ls
hello.c hello.o output.c output.o .. ..

$ gcc -o hello output.o hello.o

$ ./hello
Hello World!
```

#### Making a shared library

We can do the same thing but instead of directly linking output.o we create a shared library called liboutput.so that can be linked against any future code that needs to print "Hello World!".
Here we note that in the first step we pass the "-c" flag to tell the compiler to only compile as the default behaviour is to compile and link. In the second step the "-shared" flag tells the compiler we want to create a shared library. 

There is a convention that the name of a library is `libmylibraryname.so` where the mylibraryname should be something meaningful and not the same as a system library!

**Using a shared library**

The recipe for using a shared library is the same as we have just seen:

- `-l<name of the library>`
- `-L<location of the library>`
- `-I<location of the header files containing the library function definitions>`

When using the SCITAS provided modules it's best to use the environmental variables we provide which are of the form NAME_ROOT:
To link against FFTW we would therefore write

```bash
$ module load gcc fftw
$ gcc -lfftw3 -L$(FFTW_ROOT)/lib -I$(FFTW_ROOT)/include -o mycode.x mycode.c
```

What's linked?

```bash
$ ldd hi
linux-vdso.so.1 => (0x00007ffdff1f3000)
liboutput.so => /home/user/mycode/liboutput.so (0x00002b4fa13a5000)
libc.so.6 => /lib64/libc.so.6 (0x00002b4fa15ba000)
/lib64/ld-linux-x86-64.so.2 (0x00005605a1b01000)
```

If we unset the LD_LIBRARY_PATH variable we see a common problem:
$ unset LD_LIBRARY_PATH

$ ./hi
./hi: error while loading shared libraries: liboutput.so: cannot open shared object file: No such file or directory

$ ldd hi
    linux-vdso.so.1 =>  (0x00007ffe10bc1000)
    liboutput.so => not found
    libc.so.6 => /lib64/libc.so.6 (0x00002ade4a69a000)
    /lib64/ld-linux-x86-64.so.2 (0x000055903b50d000)

Note that setting LD_LIBRARY_PATH isn't the only way for executables to find libraries but it is the most common. For the software built by SCITAS we use the RPATH mechanism which embeds the path to the library in the executable itself.

Libraries (modules) that you should know about

Some of the more important libraries provided as modules on the SCITAS clusters are:

Intel Math Library

The Intel Math library is part of the Intel Compiler Suite and provides optimised and vectorised implementations of the standard mathematical functions. There is a math.h that is compatible with the GCC implementation as well as mathimf.h which contains additional functions. To make use of the math library it is vital to not specify "-lm" but rather "-limf".

Intel MKL

The Intel Math Kernel Library provides optimised mathematical routines for science, engineering, and financial applications. Core functions include BLAS, LAPACK, ScaLAPACK, sparse solvers, fast Fourier transforms, and vector maths. These routines are often hand optimised to take full advantage of the Intel processors and are far faster than anything you will ever be able to write.

OpenBLAS

OpenBLAS is, as the name suggests, an open source BLAS implementation. This is the linear algebra package that is provided with the GCC based stacks.

FFTW

The Fastest Fourier Transform in the West is a C subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions, of arbitrary input size, and of both real and complex data. The Intel MKL might be faster for certain transforms but FFTW is more general.

Eigen

Eigen is a C++ template library for linear algebra, matrices, vectors, numerical solvers, and related algorithms. There are no *.so shared libraries as it's a header only library.

Build Tools

Once you have more than one source file compilation becomes tedious (and error prone) so it's worth automating the process as much as possible.

There are two main approaches to managing the build process:
**GNU autotools**

This is the traditional way to build codes and is widely used. There is a configure script that takes parameters and generates a Makefile. This Makefile is then used to compile and install (and perhaps test) the code.

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

A real example for the FFTW library as installed on the compute nodes is:

```bash
./configure '--prefix=/ssoft/spack/paien/v2/opt/spack/linux-rhel7-x86_E5v4_Mellanox/gcc-6.4.0/fftw-3.3.7-aqeffyxtgjw5r5hnx74jrkzhstdk2q0' '--enable-shared'
'--enable-threads' '--disable-sse' '--disable-sse2' '--enable-avx' '--enable-avx2' '--enable-avx512' '--disable-avx-128-fma' '--disable-kcvi' '--disable-altivec'
'--disable-vsx' '--disable-neon' '--disable-generic-simd128' '--disable-generic-simd256' '--enable-fma
```

**CMake**

CMake does the same thing as autotools but in a more modern and cross platform compatible manner.

```bash
$ cmake -DCMAKE_INSTALL_PREFIX:PATH=X -DOption=Y <sources>
$ make
$ make install
```

CMake has a graphical interface called ccmake and is our recommendation if you are starting a project without an existing build system.

A real example for netlib-scalapack installed on the compute nodes is:

```bash
cmake '/ssoft/spack/paien/spack.v2/var/spack/stage/netlib-scalapack-2.0.2-bxvv2dvaro2ty2t2114u3xrqtu46qah /scalapack-2.0.2' '-G' 'Unix Makefiles'
'--DCMAKE_INSTALL_PREFIX:PATH=/ssoft/spack/paien/v2/opt/spack/linux-rhel7-x86_E5v4_Mellanox/gcc-6.4.0/netlib-scalapack-2.0.2-bxvv2dvaro2ty2t2114u3xrqtu46qah'
'--DCMAKE_BUILD_TYPE:STRING=RelWithDebInfo' '--DCMAKE_VERBOSE_MAKEFILE:BOOL=ON' '--DCMAKE_INSTALL_RPATH_USE_LINK_PATH:BOOL=FALSE' '--DBUILD_SHARED_LIBS:BOOL=ON' '--DBUILD_STATIC_LIBS:BOOL=OFF' '--DLAPACK_FOUND=true'
'--DLAPACK_INCLUDE_DIRS=/ssoft/spack/paien/v2/opt/spack/linux-rhel7-x86_E5v4_Mellanox/gcc-6.4.0/openblas-0.2.20-nc3s6m7ojtks32jz2rlo2w4xuilfmexx/include'
'--DLAPACK_LIBRARIES=/ssoft/spack/paien/v2/opt/spack/linux-rhel7-x86_E5v4_Mellanox/gcc-6.4.0/openblas-0.2.20-nc3s6m7ojtks32jz2rlo2w4xuilfmexx/lib/libopenblas.so'
'--DBLAS_LIBRARIES=/ssoft/spack/paien/v2/opt/spack/linux-rhel7-x86_E5v4_Mellanox/gcc-6.4.0/openblas-0.2.20-nc3s6m7ojtks32jz2rlo2w4xuilfmexx/lib/libopenblas.so'
```

**Optimisation**

Compiling is not an easy task and it's also impossible for the compiler to guess what you want to achieve. For example you might want to compile the code such that:

- The size of the resulting executable is as small as possible
- The code runs as fast as possible with the risk of some numerical inaccuracy
- The code has to have "perfect" numerical accuracy
The first case might be true for embedded applications and the third for control systems for a plane. For scientific applications we often target the second goal.

**Compilers are lazy**

When you compile code you are asking for a binary executable that does the same thing (gives you the same result) as your human readable code but there is no guarantee that the compiler will use the same logic/procedure to achieve this!

Let’s take a simple C function and compile it with GCC 4.8.3 (gcc -s matest.c )

```c
float matest(float a, float b, float c) {
    a = a*b + c;
    return a;
}
```

Note that a float in C uses 32 bits and is single precision.

This then gives us the assembler instructions that will be run on the CPU (after having been converted to binary)

```
matest(float, float, float):
push   rbp
mov    rbp,rsp
movss  DWORD PTR [rbp-0x4],xmm0
movss  DWORD PTR [rbp-0x8],xmm1
movss  DWORD PTR [rbp-0xc],xmm2
movss  xmm0,DWORD PTR [rbp-0x4]
mulss  xmm0,DWORD PTR [rbp-0x8]
addss  xmm0,DWORD PTR [rbp-0xc]
movss  DWORD PTR [rbp-0x4],xmm0
mov    eax,DWORD PTR [rbp-0x4]
mov    DWORD PTR [rbp-0x10],eax
movss  xmm0,DWORD PTR [rbp-0x10]
pop    rbp
ret
```

The assembler instructions used are:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
<td>Push Word, Doubleword or Quadword Onto the Stack</td>
</tr>
<tr>
<td>mov</td>
<td>Move</td>
</tr>
<tr>
<td>movss</td>
<td>Move Scalar Single-FP Values</td>
</tr>
<tr>
<td>mulss</td>
<td>Multiply Scalar Single-FP Value</td>
</tr>
<tr>
<td>addss</td>
<td>Add Scalar Single-FP Values</td>
</tr>
<tr>
<td>pop</td>
<td>Pop a Value from the Stack</td>
</tr>
</tbody>
</table>

What we note is that there are a lot of them!

**Performance tuning**

You can ask the compiler to try and make your code run faster with the -O flag. The variants -O1, -O2 and -O3 are described below:

| O1 | Enables optimisations for speed and disables some optimisations that increase code size and affect speed. To limit code size, this option enables global optimisation; this includes data-flow analysis, code motion, strength reduction and test replacement, split-lifetime analysis, and instruction scheduling. |
O2 Enables optimisations for speed. This is the generally recommended optimisation level. Vectorization is enabled at O2 and higher levels.

O3 Performs O2 optimisations and enables more aggressive loop transformations such as Fusion, Block-Unroll-and-Jam, and collapsing IF statements.

The recommended level is O2 and it's entirely possible that O3 can make your code run slower. Please don't assume that O3 is always better!

Returning to the example shown in the introduction if we compile with -O2 the assembler becomes

```assembly
matest(float, float, float):
mulss xmm0,xmm1
addss xmm0,xmm2
ret
```

It's not difficult to see that this requires far fewer steps than the unoptimised version.

In addition to the "general" optimisation, different CPUs have different instructions that can be used to make operations faster, with the Intel AVX (Advanced Vector eXtensions) being a good example. The Haswell processors introduced AVX2 which adds a Fused Multiply-Add (FMA) instruction so that $a=b\times a+c$ is performed in one step rather than requiring two instructions.

<table>
<thead>
<tr>
<th>Processor Family</th>
<th>Vector Instruction Set</th>
<th>What is does</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium</td>
<td>SSE</td>
<td>128 bit vector instructions</td>
</tr>
<tr>
<td>Sandy Bridge</td>
<td>AVX</td>
<td>256 bit vector instructions</td>
</tr>
<tr>
<td>Haswell</td>
<td>AVX2</td>
<td>256 bit vector instructions with FMA and other enhancements</td>
</tr>
<tr>
<td>Skylake</td>
<td>AVX512</td>
<td>512 bit vector length operations</td>
</tr>
</tbody>
</table>

Because the compiler doesn't know where you want to run your code it will create a binary for the lowest common denominator which is usually SSE. This means that a code compiled with just -O3 will not be able to take advantage of 95% of the performance of the latest processors via SIMD instructions and other improvements.

Taking the simple example and optimising for Haswell processors (read on for how to do this) the assembler becomes

```assembly
matest(float, float, float):
vfmadd132ss xmm0,xmm2,xmm1
ret
```

So rather than two steps to calculate the product, it takes now only one.

As the instructions operate on vectors we see that it would be possible to carry out multiple operations at once (SIMD).

**Compilers are stupid**

It is entirely possible that despite having specified "-O3 -xCORE-AVX512" the compiler does not manage to fully optimise or vectorise your code. In this case manual intervention (changing the code) is often needed to achieve the desired result.

A classic example of this is loop unrolling which can be performed automatically but not always.
Here the problems are twofold:

1. In C an array can point to another array so the compiler has to assume that array1 and array2 are not independent.
2. The compiler has to be able to group operations together and the unit of "work" is often seen as the part inside the innermost loop.

We can "unroll" this by a factor of 8 and add a few hints to tell the compiler that it can vectorise.

```c
void myfunc( double *restrict array1, double *restrict array2, double *prod, int count)
{
    for(int x=0; x < count; x=x+8)
    {
        prod[x] = array1[x] * array2[x];
        prod[x+1] = array1[x+1] * array2[x+1];
        prod[x+2] = array1[x+2] * array2[x+2];
        prod[x+3] = array1[x+3] * array2[x+3];
        prod[x+4] = array1[x+4] * array2[x+4];
        prod[x+5] = array1[x+5] * array2[x+5];
        prod[x+6] = array1[x+6] * array2[x+6];
        prod[x+7] = array1[x+7] * array2[x+7];
    }
}
```

Compiling with

```bash
icc -O2 -xCORE-AVX512 -qopt-zmm-usage=high
```

results in the operations in the loop being converted to

```assembly
vmovups zmm0, ZMMWORD PTR [rdi+r8*8]
vmulpd zmm1, zmm0, ZMMWORD PTR[rsi+r8*8]
vmovupd ZMMWORD PTR[rdx+r8*8], zmm1
```

So the 8 multiplications are carried out as one instruction (the `vmulpd` operation).

The alternative approach would have been to unroll the loop but pass the "-fno-alias" flag to the compiler which tells it to assume that there is no array aliasing. It is up to you as the code author to make sure that this is the case.

```bash
icc -O2 -xCORE-AVX512 -qopt-zmm-usage=high -fno-alias
```

Looking at the compiler documentation:
-fno-alias

Determines whether aliasing is assumed in a program.

Arguments:
None

Default:
-falias

On Linux* and macOS*, aliasing is assumed in the program. On Windows*, aliasing is not assumed in a program.

Description:
This option determines whether aliasing is assumed in a program.

If you specify -fno-alias, aliasing is not assumed in a program.

If you specify -falias, aliasing is assumed in a program. However, this may affect performance.

If you think that you have this kind if problem then contact SCITAS - we have application experts who know what to do.

mpicc and friends

The MPI implementation is just another shared library that we load using modules and then need to link with our code (-lmipi -I path/to/mpi.h etc).

Humans are bad at typing so we have tools to help us compile MPI codes and link them with the correct libraries. Some of these tools are

- mpicc - generic MPI C compiler
- mpiicc - Intel MPI C compiler
- mpicxx - generic MPI C++ compiler
- mpiifort - Intel MPI Fortran compiler

MPICH2 based MPI distributions have the very nice "-show" command which allows us to see that these tools are wrappers for the standard compiler.

$ mpiicc -show mycode.c

icc 'mycode.c'
-I/soft/spack/external/intel/2018.2/mpi/2018.2.199/intel64/include
-Xlinker --enable-new-dtags
-Xlinker -rpath -Xlinker /opt/intel/mpi-rt/2017.0.0/intel64/lib/release_mt
-Xlinker -rpath -Xlinker /opt/intel/mpi-rt/2017.0.0/intel64/lib
-ldmpi -ldl -lmpi -lmpifort

We can also ask the version of mpiicc:

$ mpiicc --version
icc (ICC) 18.0.2 20180210
Copyright (C) 1985-2018 Intel Corporation. All rights reserved.
Which just gets passed through to the compiler behind the wrapper.

All standard compiler options can also be used so an example of compiling a simple MPI code is:

```
$ mpiicc -O2 -g -xCORE-AVX2 -Wall -o mycode.x mycode.c
```

## Compiling OpenMP codes

OpenMP is supported by both the GCC and Intel compilers. One feature of OpenMP enabled code is that it can be compiled with or without OpenMP support (so as to run in parallel or serial) as the OpenMP part is controlled via pragmas:

```c
int main(int argc, char **argv)
{
    double a[100000];

    #pragma omp parallel for
    for (int i = 0; i < 100000; i++) {
        a[i] = i * i;
    }

    return 0;
}
```

In order to activate the OpenMP parallelism we need to tell the compiler to look for the #pragma omp lines

With GCC:

```
gcc -fopenmp my_omp_code.c

gfortran -fopenmp my_omp_code.f95
```

With Intel:

```
icc -qopenmp my_omp_code.c

ifort -qopenmp my_omp_code.f95
```

## Options for Intel and GCC

This table shows the main differences between the options given to the Intel and GCC compilers. The majority of the basic options (-g, -O2, -Wall etc) are the same for both.

<table>
<thead>
<tr>
<th>Intel</th>
<th>GCC</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-xAVX</td>
<td>-march=corei7-avx</td>
<td>SandyBridge optimisations (GCC 4.8.3)</td>
</tr>
</tbody>
</table>
Running parallel codes

We've now compiled our MPI code and we want to run it on the compute cluster.

Job launchers

If you launch an executable directly it runs on the machine on which you typed the command. What we need is a magic way of launching the executable across multiple (maybe thousands) of nodes at once and ensuring that they all have the correct information to communicate with each other.

This is where job launchers come in useful.

**mpirun**

All MPI flavours come with their own job launcher that is, almost invariably, called mpirun.

The typical way of using mpirun is something like:

```plaintext
mpirun -np <number of ranks> -h <file with list of hostnames> mycode.x
```

**srun**

SLURM comes with its own job launcher than aims to be much faster at launching MPI tasks on large numbers of nodes. As it is fully integrated with the batch system there is, generally, no need to pass any options as it already has the required information.

```plaintext
srun mycode.x
```

On the SCITAS clusters you should use srun!

Specifying what you want

The key to getting the resources you need is using SBATCH directives - see the introduction to the clusters course for more details.

It's important to be aware that some directives are per node and some are per job (multiple nodes).

<table>
<thead>
<tr>
<th>SBATCH Directive</th>
<th>Short version</th>
<th>What it means</th>
</tr>
</thead>
<tbody>
<tr>
<td>--nodes</td>
<td>-N</td>
<td>Number of nodes</td>
</tr>
<tr>
<td>--ntasks</td>
<td>-n</td>
<td>Total number of MPI tasks (ranks)</td>
</tr>
<tr>
<td>--ntasks-per-node</td>
<td></td>
<td>Number of tasks per node</td>
</tr>
<tr>
<td>--cpus-per-task</td>
<td>-c</td>
<td>CPU (cores) per task</td>
</tr>
<tr>
<td>--mem</td>
<td></td>
<td>Memory per Node</td>
</tr>
</tbody>
</table>
The constraint options for the SCITAS clusters are:

<table>
<thead>
<tr>
<th>Constraint</th>
<th>What it means</th>
<th>Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>E5v2</td>
<td>Ivy Bridge (AVX)</td>
<td>Deneb</td>
</tr>
<tr>
<td>E5v3</td>
<td>Haswell (AVX2)</td>
<td>Deneb</td>
</tr>
<tr>
<td>E5v4</td>
<td>Broadwell (AVX2)</td>
<td>Fidis</td>
</tr>
<tr>
<td>s6g1</td>
<td>Skylake (AVX 512)</td>
<td>Fidis</td>
</tr>
</tbody>
</table>

The "switches" option should be used with great care as it will probably result in your job taking a long time to schedule. The only justifiable reason would be for a very latency sensitive code using less than 16/24 nodes.

**OpenMP Code**

Here we show an example job script for a pure OpenMP code that uses all the cores on a compute node

```bash
#!/bin/bash

#SBATCH --nodes=1
#SBATCH --cpus-per-task=28
#SBATCH --mem=100G

module purge
module load intel

export OMP_NUM_THREADS=28

./my_omp_code.x --in=myinput.dat
```

Note that we don't actually need to set the OMP_NUM_THREADS variable as the default behaviour is to launch as many threads as are on the node.

**MPI Code**

Here we show an example job script for a pure MPI code than runs with 96 ranks.
Here SLURM will choose sufficient nodes to run the 96 ranks but this may not give a symmetrical load distribution. For example on Fidis/Gacrux the distribution by default will be 28:28:28:12

One alternative is to use the `--ntasks-per-node` directive

```
#!/bin/bash

#SBATCH --ntasks=96
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=4096

module purge
module load gcc
module load mvapich2

srun my_mpi_code.x --in=myinput.dat
```

The second option is to specify that tasks are distributed in a cyclic manner

```
#!/bin/bash

#SBATCH --ntasks=96
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=4096
#SBATCH --distribution=cyclic:block

module purge
module load gcc
module load mvapich2

srun my_mpi_code.x --in=myinput.dat
```

Here the `--distribution` directive controls how remote processes are placed on nodes and CPUs. See the sbatch man page for all the possible options.

**Hybrid Code with open rank per node**

Here we show an example job script for a hybrid MPI/OpenMP code that has one rank per node and runs on 8 nodes.
Hybrid code with multiple ranks per node

Here we show an example job script for a hybrid MPI/OpenMP code that has six ranks and runs on 1 node with four OpenMP tasks per rank (24 threads on total)

```bash
#!/bin/bash

#SBATCH --nodes=1
#SBATCH --ntasks=6
#SBATCH --cpus-per-task=4
#SBATCH --mem=100G

module purge
module load intel
module load intel-mpi

export OMP_NUM_THREADS=4

srun my_hybrid_code.x --in=myinput.dat
```

We could also use the SLURM variable `SLURM_CPUS_PER_TASK` to set the number of OpenMP threads to be the same as the value of `--cpus-per-task`:

```bash
export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}
```

Specifying where you want code to run

CPU Affinity / Binding

In order to set and view CPU affinity with `srun` one needs to pass the "--cpu_bind" flag with some options. We strongly suggest that you always ask for "verbose" which will print out the affinity mask set by SLURM.

Bind by rank
Please be aware that binding by rank is only recommended for pure MPI codes as any OpenMP or threaded part will also be confined to one CPU!

**Bind to sockets**

```bash
:> srun -N 1 -n 4 -c 4 --cpu_bind=verbose,sockets ./hi
```

```plaintext
cpu_bind=MASK - b370, task  1  1 [5376]: mask 0xff00 set
cpu_bind=MASK - b370, task  2  2 [5377]: mask 0xff set
cpu_bind=MASK - b370, task  0  0 [5375]: mask 0xff set
cpu_bind=MASK - b370, task  3  3 [5378]: mask 0xff00 set
Hello world, b370
0: sleep(1)
0: bye-bye
Hello world, b370
2: sleep(1)
2: bye-bye
Hello world, b370
1: sleep(1)
1: bye-bye
Hello world, b370
3: sleep(1)
3: bye-bye
```

**Bind with whatever mask you feel like**

```bash
:> srun -N 1 -n 4 -c 1 --cpu_bind=verbose,rank ./hi
```

```plaintext
cpu_bind=RANK - b370, task  0  0 [5326]: mask 0x1 set
cpu_bind=RANK - b370, task  1  1 [5327]: mask 0x2 set
cpu_bind=RANK - b370, task  3  3 [5329]: mask 0x8 set
cpu_bind=RANK - b370, task  2  2 [5328]: mask 0x4 set
Hello world, b370
0: sleep(1)
0: bye-bye
Hello world, b370
2: sleep(1)
2: bye-bye
Hello world, b370
1: sleep(1)
1: bye-bye
Hello world, b370
3: sleep(1)
3: bye-bye
```
Mismatches

In the case of there being an exact match between the number of tasks and the number of cores srun will bind by rank but by default there is no cpu binding.

In the case of there being an exact match between the number of tasks and the number of cores srun will bind by rank but by default there is no cpu binding.

This may well result in sub optimal performance as one has to rely on the OS scheduler to (not) move things around.

See the --cpu_bind section of the srun man page for all the details!

OpenMP binding

There are two main ways that OpenMP is used on the clusters.

1. A single node OpenMP code
2. A hybrid code with one OpenMP domain per rank

For both Intel and GNU OpenMP there are environmental variables which control how OpenMP threads are bound to cores.

The first step for both is to set the number of OpenMP threads per job (case 1) or MPI rank (case 2). Here we set it to 8

export OMP_NUM_THREADS=8

Intel

The variable here is KMP_AFFINITY
export KMP_AFFINITY=verbose,scatter    # place the threads as far apart as possible
export KMP_AFFINITY=verbose,compact    # pack the threads as close as possible to each other

The official documentation can be found at https://software.intel.com/en-us/node/522691

GNU

with GCC one needs to set either

OMP_PROC_BIND
export OMP_PROC_BIND=SPREAD      # place the threads as far apart as possible
export OMP_PROC_BIND=CLOSE       # pack the threads as close as possible to each other

or GOMP_CPU_AFFINITY which takes a list of CPUs

GOMP_CPU_AFFINITY="0 2 4 6 8 10 12 14"   # place the threads on CPUs 0,2,4,6,8,10,12,14 in this order.
GOMP_CPU_AFFINITY="0 8 2 10 4 12 6 14"   # place the threads on CPUs 0,8,2,10,4,12,6,14 in this order.

The official documentation can be found at https://gcc.gnu.org/onlinedocs/libgomp/Environment-Variables.html#Environment-Variables

Runtime errors

Sometimes, despite ones best efforts, things don't quite go to plan. Some of the more commonly seen ones are:

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

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

Fatal error in MPI_Init:  Other MPI error, error
stack:
  .MPIR_Init_thread(514):
  .MPID_Init(320)........:  channel initialization failed
  .MPID_Init(716)........:  PMI_Get_id returned 14

If you encounter errors when running a batch job then please try interactively - errors are much more visible this way

$ s alloc -N 2 -n 32 -t 01:00:00 -partition debug
$ s run mycode.x < inp.in

Going Further

Courses

If you want to learn more about the topics discussed then SCITAS offers courses in:
Introduction to profiling and software optimisation
MPI: an introduction to parallel programming
MPI: advanced parallel programming
Computing on GPUs

You are also encouraged to look at the courses offered by PRACE (PartneRship for Advanced Computing in Europe) - http://www.training.prace-ri.eu

References and further reading

For system packages the man pages provide the most up to date command reference. Don't forget to load the relevant compiler module first.

man srun
man sbatch
man gcc
man icc

The "Intel® 64 and IA-32 Architectures Optimization Reference Manual" explains in great detail everything you never wanted to know about the internals of Intel Processors


The reference guides for the MKL can be found at


The MPI specifications can be found at

https://www.mpi-forum.org/docs/

If you want to play around with compiler options and versions you can do so interactively with to the Godbolt Compiler Explorer at

https://godbolt.org

The SCITAS documentation is at

https://scitas-data.epfl.ch/kb