Effective Use of Clusters

2025–2026

The ARC Team

Introduction

Before we start...

- Glossary
  - **core** = unit that does the work (sometimes use CPU as a synonym)
  - **processor** = collection of cores in a single package all sharing the same memory
  - **node** = a collection of processors all sharing the same memory
  - **interconnect** = the network in a machine that joins together the separate nodes

Note: each node has its own memory and cannot directly 'see' another node's memory.

- Distinction between processor, process and thread
  - **processor** = a physical piece of **hardware**
  - **process** = an instance of a running program **(software)**
    - essentially it has two components: instructions to execute and associated data
    - in parallel programming we often have multiple instances (processes) of the same program...
  - a process always consists of one or more **threads** of execution

Models of Parallelism
Distributed Memory

Distributed Memory Programming Model:

- multi-core system, each core has its own private memory
- local core memory is invisible to all other processors
- agent of parallelism: the **process** (program = collection of processes)
- exchanging information between processes requires **explicit** message passing
- the dominant programming standard: **MPI**

Distributed Memory Hardware:

- conceptually, many PCs connected together (traditional Beowulf cluster)
- current approach:
  - multi-core computer nodes (high-density blades) with own memory
  - high-bandwidth, low-latency network connection
  - off-the shelf modular technology (high-end CPUs, standard hard disk)
  - accounts for the largest HPC systems

Distributed Memory ARC systems: the **ARC** cluster (but any machine can be programmed using this model)

Models of Parallelism
Distributed Memory

Shared Memory Programming Model:

- multi-core system
- each core has access to a shared memory space
- agent of parallelism: the **thread** (program = collection of threads)
- threads exchange information **implicitly** by reading/writing shared variables
- the dominant programming standard: **OpenMP**

Shared Memory Hardware:

- conceptually, a single PC, with a large memory and many cores
- accounts for both small and inexpensive systems (desktops) and very large and expensive system (with very expensive high bandwidth memory access)

Shared Memory ARC Systems: **HTC cluster** and any single node of the **ARC cluster**.

Distributed Memory v. Shared Memory

Distributed Memory:

- **Can scale to any number of cores**
- Requires special tools to compile and run the code
  - Typically `mpicc` or `mpif90` to compile, `mpirun` to run it
- Can be harder to program that shared memory
- But will generally perform better if done well
- And it teaches good parallel programming ‘habits’

Shared memory:

- **Is usually limited to the number of cores in a node**
  - Can overpopulate, good for debug, bad idea for performance
- Generally just requires an extra flag on the compiler
- Can be easier to program than distributed memory
- It is often hard to get good parallel performance
  - Sharing things is not good for parallelism…
- Can easily let people be a bit sloppy when programming…

Distributed Memory v. Shared Memory

Distributed Memory:

- **Can scale to any number of cores**
- Requires special tools to compile and run the code
  - Typically `mpicc` or `mpif90` to compile, `mpirun` to run it
- Can be harder to program that shared memory
- But will generally perform better if done well
- And it teaches good parallel programming 'habits'

Shared memory:

- **Is usually limited to the number of cores in a node**
  - Can overpopulate, good for debug, bad idea for performance
- Generally just requires an extra flag on the compiler
- Can be easier to program than distributed memory
- It is often hard to get good parallel performance
  - Sharing things is not good for parallelism…
- Can easily let people be a bit sloppy when programming…

Batch Scripts

Batch Scripts

- Now we know about the types of parallelism we can structure our batch script in such a way that we can efficiently use the resources.
- For the clusters provided by ARC we need to know that…
  - Each node has typically 48 cores
- Also **remember** the first part of the script reserves resources for you, while the second says what you want to do with it…

Some quick solutions — ARC/MPI

Example for HPC-type job script:

- parallel (MPI) application
- single large problem, too large for single node
- one single input file
- job uses many compute nodes
- For best resource usage use multiple of 48 cores

```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=48
#SBATCH --mem-per-cpu=2G
#SBATCH --time=00:10:00
#SBATCH --partition=devel
#SBATCH --job-name=myjob

module load mpitest/1.0
mpirun mpihello
```

Some quick solutions — HTC

Example for HTC-type job script:

- serial (or multi-threaded) application
- parametric study, many input files
- processing in batches of 48
- each job uses 1 compute node
- ideally, processing should be balanced

```
#!/bin/bash
...
## Reserve 1 node for 10 hours
#SBATCH --walltime=10:00:00
#SBATCH --nodes=1
...
## Run 48 jobs in the background
for ID in {1..48}; do
    serialApp test_$ID.dat &
done
...
## Wait for all the jobs to complete
wait
```

Array Jobs

- Job arrays allow you to submit the same batch script many times over.

```
[user@arc-login ~]$ sbatch -array=1-10 myscript.sh
```

- By default you can distinguish between members of the array with the `$SLURM_ARRAY_TASK_ID` environment variable. For example we could modify the previous example to use multiple directories based upon this variable...

```
job_name.1/test_1.dat .. test_48.dat
..
job_name.10/test_1.dat .. test_48.dat
```

```
1  #!/bin/bash
2  ...
3  #SBATCH --walltime=10:00:00
4  #SBATCH --nodes=1
5  ...
6  cd job_name.$SLURM_ARRAY_TASK_ID
7  for ID in {1..48}; do
8    serialApp test_$ID.dat &
9  done
10 ...
11 wait
```

Array Jobs

- The main use is to allow the HTC user to use more than one node
  - However there is no reason why an MPI user can't use them
- And also note there is no performance difference from submitting each of the job members individually
- The main reason is convenience
  - Can submit all with one command
  - Can use `scancel` to cancel all the jobs with one command
- Strong recommendation:
  Don't use very large job arrays, if things go wrong things can go VERY wrong!
  - Do you want emails from each of 10 000 failing jobs all at the same time?

Load Balancing and Array Jobs

- When we pack jobs up into groups of 48 the time taken is determined by the one that takes the longest
- This can cause efficiency problems if one or two of the jobs take very much longer than the others as you will have to wait for the longest to complete irrespective of how quick the others are.
- In other words you want the group of 48 to be load balanced
- Not much you can do if you just have 48 jobs
- But if you are using a job array try to make each member of the array as balanced as possible
  - You will generally have some kind of feeling which runs are quick to complete and which are slow, so group the quick with the quick and the slow with the slow
- So a bit of thought can help your efficiency quite a lot!

Hybrid OpenMP/MPI Jobs

Some applications can use MPI for communication between nodes and OpenMP for parallelism within the node. This Hybrid type of jobs can be handled as follows:

As an example if we want to use 2 nodes, with 1 MPI task per node and 12 OpenMP threads as the resources should be:

```
1  #!/bin/bash
2  #SBATCH --nodes=2
3  #SBATCH --ntasks-per-node=1
4  #SBATCH --cpus-per-task=12
5
6  module load mpitest
7
8  export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
9
10 mpirun --map-by numa:pe=${SLURM_CPUS_PER_TASK} mpisize
```

```
Hello from host "arc-c303". This is MPI task 1, the total MPI Size is 2, and there are 12 CPU core(s)
allocated to *this* MPI task, these being { 0 1 2 3 4 5 6 7 8 9 10 11 }
Hello from host "arc-c302". This is MPI task 0, the total MPI Size is 2, and there are 12 CPU core(s)
allocated to *this* MPI task, these being { 0 1 2 3 4 5 6 7 8 9 10 11 }
```

Multi-threaded R or Python Jobs

Both R and Python have relatively easy to use multi-threading libraries available.

When using an script which uses these it is important to use the correct SLURM resource directives.

As an example if we want to use 48 cores on 1 node, the resources should be:

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=48
```

GPUs

GPUs on HTC

- HTC has a wide range of GPU accelerator resources available
- An up-to-date list of GPUs is available here:
  `https://arc-user-guide.readthedocs.io/en/latest/arc-systems.html#gpu-resources`
- GPUs are highly contended resources
  - Some are co-investment resources which may be reserved at certain times.
  - The NVIDIA drivers on the compute nodes are updated twice per year (May/November).
- The process requires nodes to be taken offline.

Application Containers

What is a container?

- A Unix operating system is broken into two primary components, the kernel space, and the user space.
  - The kernel talks to the hardware, and provides core system features.
  - The user space is the environment that most people are most familiar with. It is where applications, libraries and system services run.
- If you have access to a machine running CentOS (like the ARC clusters) then you cannot install software that was packaged for Ubuntu on it, because the user space of these distributions is not compatible.
- Containers change the user space into a swappable component. This means that the entire user space portion of a Linux operating system, including programs, custom configurations, and environment can be independent of whether your system is running CentOS, Fedora etc., underneath.
- Software developers can now build their stack onto whatever operating system base fits their needs best, and create distributable runtime environments so that users never have to worry about dependencies and requirements, that they might not be able to satisfy on their systems.

From: `https://apptainer.org/docs/user/main/introduction.html#why-use-containers`

Application Containers

- The ARC clusters have **Apptainer** (formerly Signularity) installed — no need to load a module.
- Why not **Docker**?
  - **Docker** users a client-server model which cannot integrate with the SLURM batch system.
  - It requires superuser privileges to run
  - **Docker** container data is isolated from the host — so no access to data or host drivers.
- **Apptainer** is designed specifically for HPC environments
  - The container runs as a 'child' of the current shell
  - Allows access to all host resources including storage, Infiniband, GPUs etc..

Application Containers — cont...

Simple to run published containers:

```
[user@arc-interactive ~]$ apptainer remote add --no-login -n sylabsed cloud.sylabs.io
[user@arc-interactive ~]$ apptainer run library://sylabsed/examples/lolcow
INFO:    Downloading library image
79.9MiB / 79.9MiB [============================================================================] 100 % 20.1 MiB/s 0s

 ----------------------------------------
/ "You have been in Afghanistan, I       \
| perceive."                             |
|                                        |
| -- Sir Arthur Conan Doyle, "A Study in |
\ Scarlet"                               /
 ----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Application Containers — cont...

Simple to run published Docker containers:

```
[user@arc-interactive ~]$ apptainer run docker://sylabsio/lolcow:latest
INFO:    Converting OCI blobs to SIF format
INFO:    Starting build...
Getting image source signatures
Copying blob 16ec32c2132b done
Copying blob 5ca731fc36c2 done
Copying config fd0daa4d89 done
Writing manifest to image destination
Storing signatures
2025/05/02 13:16:01  info unpack layer: sha256:16ec32c2132b43494832a05f2b02f7a822479f8250c173d0ab27b3de78b2f058
2025/05/02 13:16:02  info unpack layer: sha256:5ca731fc36c28789c5ddc3216563e8bfca2ab3ea10347e07554ebba1c953242e
INFO:    Creating SIF file...
 _____________________________
< Fri May 2 13:16:08 BST 2025 >
 -----------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Application Containers — cont...

Apptainer containers are compatible with MPI and NVIDIA GPUs

- You need to ensure that the NVIDIA driver version inside the container matches the version on our compute nodes.
- See the following link for the Apptainer documentation: `https://apptainer.org/docs/user/main/`
- Note: Apptainer was formerly known as Singularity
- Containers are unlikely to be as efficient as code built natively on the system. This is the cost of convenience.

`https://arc-software-guide.readthedocs.io/en/latest/apps/arc_apptainer.html`

Memory Efficiency

Identifying & Fixing Memory Issues

- **The most common** issue: An application runs out of memory.
- There are things you can do...
  - Use `squeue` to identify machines job is running on.
  - `ssh` to one of those machines, and use the Linux `top` command to examine process resources.
- **If** memory is the problem. Firstly ensure you are requesting enough memory using the `#SBATCH --mem` directive.
- You can also waste cores to gain more memory per process.
  - Most machines have 48 cores and usable 360 GB so this is approx. 7.5 GB per core. If you use a whole node and 24 cores you now have 15 GB per process.

Identifying & Fixing Memory Issues

```
top - 08:07:09 up 6 days, 21:12,  2 users,  load average: 10.01, 2.79, 1.00
Tasks: 662 total,   5 running, 657 sleeping,   0 stopped,   0 zombie
%Cpu(s):  2.1 us,  0.1 sy,  0.0 ni, 95.8 id,  0.0 wa,  1.9 hi,  0.1 si,  0.0 st
MiB Mem : 386398.6 total, 368922.7 free,  15453.6 used,   2022.2 buff/cache
MiB Swap:   1908.0 total,    938.4 free,    969.6 used. 365768.2 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
2056621 ouit0554  20   0 1641888   1.1g  23248 R  98.0   0.3   0:28.68 mpiprimes
2056648 ouit0554  20   0 2880696 180980  23320 R   4.0   0.0   0:13.84 mpiprimes
2056631 ouit0554  20   0 2993992 165092  23292 S   3.6   0.0   0:13.11 mpiprimes
2056634 ouit0554  20   0 2889096 167968  23056 S   3.6   0.0   0:13.25 mpiprimes
2056661 ouit0554  20   0 2998192 163488  23188 S   3.6   0.0   0:14.93 mpiprimes
2056662 ouit0554  20   0 2855516 185188  23164 S   3.6   0.0   0:14.96 mpiprimes
2056665 ouit0554  20   0 2905872 183776  23184 S   3.6   0.0   0:15.14 mpiprimes
```

Identifying & Fixing Memory Issues

MPI example

- 7.5 GB per core

```
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=48
#SBATCH --mem=0
```

- 15 GB per core

```
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=24
#SBATCH --mem=0
```

- Other combinations are available…

Identifying & Fixing Memory Issues

Threaded example

- 7.5 GB per core

```
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=48
#SBATCH --mem=0
```

- 15 GB per core

```
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=24
#SBATCH --mem=0
```

Identifying & Fixing Memory Issues

Requesting exclusive access to node resources...

- MPI

```
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=2
#SBATCH --exclusive
```

- Threaded

```
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=24
#SBATCH --exclusive
```

Improving Performance

What else can we improve

- There are a number of other ways to generate your results faster on a cluster:
  - **Make the program run faster by either better use of the compiler or libraries**
  - **Better use of the disks**
  - **Using an appropriate number of cores for your MPI or OpenMP program**
  - *Use of area specific or application specific knowledge*
- The first three we'll discuss in the following sections.
  - We will also cover how to measure parallel performance.
- The last is a huge area and mostly beyond what we cover today
  - Note especially for MPI programs there are often application specific 'tricks' that can help you obtain your answer more efficiently on a cluster

Compilers

Programming Languages

Programming languages for scientific computing:

- **Fortran** and **C** account for most computation intensive codes
  - computation engines of many applications
  - Fortran is more 'natural' than C for scientific computing
    - use Fortran 95 or later, Fortran 77 is dead!
  - performance libraries are written in C (FFTW) or Fortran (LAPACK)
- **C++**
  - OOP allows (it is claimed) better software design and re-use of code
- **JAVA**, **C#**, etc.
  - normally only used for front-ends and GUIs, etc.
- **Matlab**, **Python**
  - interpreters, interactive use (data inspection, plotting capabilities)
  - numerically intensive parts written in C/Fortran (mex functions, modules)

Compiled Languages

Fortran, **C**, and **C++** are compiled

- the computer cannot understand the program (human readable) directly
- the compiler is a tool used to translate the whole program into the instructions that a computer can understand
- there are many ways to do this translation; how fast the resulting program runs will depend upon how 'good' a job the compiler does

Compare with **Matlab**, **Python**, and **R**

- interpreted languages
- again, a tool is required to turn the program into something the computer can understand, but it is done one 'line' at a time
  - easy on the tool and convenient in some ways (e.g. what if your program is 1000s of lines long and you change one line only?)
  - but typically *much* lower performance than compiled programs

Making the most of compilers

So, we want our program to run as fast as possible

There a two major ways we can affect its performance via the complier:

- the **choice** of compiler, and
- the **use** of compiler (i.e. the choice of compiler flags)

Compiliers

On the ARC systems, several compilers are available.

- The GNU compiler collection (standard Linux compilers — available everywhere and free):
  `gcc/g++/gfortran`
- The Intel compiler suite:
  `icc/icpc/ifort`
- The Portland Group compilers:
  `pgcc/pgCC/pgf90`
  The Portland Group compilers are now part of the NVidia HPC toolkit.

Choosing the compiler (base)

How to choose which compiler you are using varies from system to system, but on the ARC cluster we use **environment modules**, which are a general method to manage software installations.

For instance on ARC systems the following will pick versions of the appropriate compiler:

```
module load intel-compilers
module load GCC
module load PGI
```

Note:

Fortran programmers — note these are entirely separate from and have nothing to do with Fortran modules

Choosing the compiler (toolchain)

The ARC environment is built using the EasyBuild framework and this standardises the naming of certain compiler types and also gathers them together with popular libraries such as BLAS/LAPACK and MPI to form what is known as a toolchain.

For example you can load the GCC 14.3.0 compiler alone (not recommended) using:

```
module load GCCcore/14.3.0
```

Or GCC with compatible binutils and zlib (recommended) with:

```
module load GCC/14.3.0
```

Or, if you load the whole toolchain:

```
module load foss/2025b
```

Choosing the Compiler (Toolchain)

`foss/2025b` loads the following modules:

1) GCCcore/14.3.0
2) zlib/1.3.1-GCCcore-14.3.0
3) binutils/2.44-GCCcore-14.3.0
4) GCC/14.3.0
5) numactl/2.0.19-GCCcore-14.3.0
6) XZ/5.8.1-GCCcore-14.3.0
7) libxml2/2.14.3-GCCcore-14.3.0
8) libpciaccess/0.18.1-GCCcore-14.3.0
9) hwloc/2.12.1-GCCcore-14.3.0
10) OpenSSL/3
11) libevent/2.1.12-GCCcore-14.3.0
12) UCX/1.19.0-GCCcore-14.3.0
13) libfabric/2.1.0-GCCcore-14.3.0
14) PMIx/5.0.8-GCCcore-14.3.0
15) PRRTE/3.0.11-GCCcore-14.3.0
16) UCC/1.4.4-GCCcore-14.3.0
17) OpenMPI/5.0.8-GCC-14.3.0
18) OpenBLAS/0.3.30-GCC-14.3.0
19) FlexiBLAS/3.4.5-GCC-14.3.0
20) FFTW/3.3.10-GCC-14.3.0
21) gompi/2025b
22) FFTW.MPI/3.3.10-gompi-2025b
23) ScaLAPACK/2.2.2-gompi-2025b-fb
24) foss/2025b

Key:
Black — General modules
Blue — Compiler modules
Red — MPI related modules
Green — Maths library modules

Choosing the compiler (toolchain)

For more information on available toolchains see:
https://docs.easybuild.io/en/latest/Common-toolchains.html#common-toolchains

N.B. Not all toolchains will be available on ARC. Use `module spider` to check.

Compiler Performance

As an example of how the compiler can affect the run time here are two examples of `DL_POLY_4` run on 16 cores of ARC (one single node) with the three different compilers.

Versions of the compilers are indicated. Times are in seconds.

Compiler	Runtime (s)	
	Sodium Chloride	Gramicidin
gcc 4.8.2	241.686	189.514
Intel 14.0.2	342.201	246.520
Portland Group 13.10-0	**205.602**	**129.827**

Invoking the compiler

- However it's not just which compiler you use, it's also how you invoke it — this usually makes more difference than the choice of compiler
- So let's have a look at how a compiler works in practice…
- There are actually a number of stages, but only two are of interest to us
  - Compilation
  - Linking

Compilation

- Remember one program can be contained in many source files
- Compilation is the stage that takes an individual file and translates it into instructions the computer can understand.
- These instructions are placed in an object file with a `.o` suffix
- You can compile only (no linking) by use of the `-c` flag:

```
[user@arc-interactive ~]$ ls
file.f90
[user@arc-interactive ~]$ gfortran -c file.f90
[user@arc-interactive ~]$ ls
file.f90 file.o
```

Linking

- Remember that a program can be in many different source files
- Linking takes all compiled object files and links them together into a single *executable*
- Linking is normally managed through the compiler tool itself (which uses the default linux linker `ld` to do the work)

```
[user@arc-interactive ~]$ ls
file1.f90 file2.f90 file3.f90
[user@arc-interactive ~]$ ifort -c file1.f90
[user@arc-interactive ~]$ ifort -c file2.f90
[user@arc-interactive ~]$ ifort -c file3.f90
[user@arc-interactive ~]$ ls
file1.f90 file1.o file2.f90 file2.o file3.f90 file3.o
[user@arc-interactive ~]$ ifort file1.o file2.o file3.o -o exe
[user@arc-interactive ~]$ ls
exe file1.f90 file1.o file2.f90 file2.o file3.f90 file3.o
```

Compile and link flags

- We have already used flags to change the (default) behaviour of the compiler and linker
  - `-c` specifies 'compile only' and `-o` specifies the executable file
- All compilers have many, many flags
- Similarly, linkers have many flags
  - the most important are those telling the linker where to find files (esp. libraries)
- The usual flags at the compile stage include
  - Help with debugging the program
  - Making sure the programmer sticks to international standards
  - Telling the compiler where to find files — e.g. `-I` for include files
  - Optimisation flags (these are the most important flags for us)

Optimisation flags

- All compilers have a flag of the form `-ON` where `N` is an integer, typically in the range 0 to 3
- Use of this will make the compiler analyse each file it is working on in an attempt to produce a faster executable code
- The larger the number, the harder it will try to do so:
  - `-O0` — don't optimise the code
  - `-O1` — do quick and easy optimisations
  - `-O2` — try hard to get the best performance
  - `-O3` — try really hard to get the best performance!
- Not quite a free lunch
  - Longer compiler times
  - More likely to show up compiler bugs
  - Also more likely to show up software bugs in strange ways…
- But you should really use the highest of these for productions runs!

What difference does it make?

Table: Compiled with `-O0` flag

Compiler	Runtime (s)	
	Sodium Chloride	Gramicidin
gcc 4.8.2	241.686	189.514
Intel 14.0.2	342.201	246.520
Portland Group 13.10-0	**205.602**	**129.827**

Table: Compiled with `-O3` flag

Compiler	Runtime (s)	
	Sodium Chloride	Gramicidin
gcc 4.8.2	115.956	84.919
Intel 14.0.2	**99.942**	**76.556**
Portland Group 13.10-0	108.292	78.945

Other optimisation flags

- All compilers have many optimisation flags
- Unfortunately apart from `-O` they are almost always specific to the compiler
  - You will have to look at the **man** page or the user guide
- Some suggestions (apply to all languages we have considered):

```
gcc -O3 -funroll-loops -march=native ...
icc -O3 -xHost -ipa ...
```

- ipa = **i**nter-**p**rocedural **a**nalysis, analyses **all** source code (not just one file at a time), looking for optimisation opportunities **across** files
- `-ipa` can MASSIVELY increase compile time

Optimisation Flags

Table:

gcc flags	Runtime NaCl (s)
`-O0`	241.686
`-O1`	132.373
`-O2`	120.537
`-O3`	115.596
`-O3 -funroll-loops`	119.578
`-O3 -funroll-loops -march=native`	106.882

Table:

icc flags	Compilation time (s)	Runtime NaCl (s)
`-O0`	65	342.201
`-O1`	142	114.795
`-O2`	237	99.47
`-O3`	267	99.942
`-O3 -xHost`	285	96.769
`-O3 -xHost -ipa`	4202	97.516

Short Break…

Makefiles, MPI, & The Linker

In Practice — the 'Makefile'

You don't always compile the whole program from the command line

- Often something called a **Makefile** is supplied which will automate the build process. This can also be generated by **Automake** or **CMake**.
- How to set the compiler and compiler flags in the case will vary from case to case

However…

- Commonly you set a variable called `CC` to the name of the C compiler
- And one called `FC` or `F90` for the Fortran compiler
- The C compile flags are usually called `CFLAGS`
- And the Fortran compile flags `FFLAGS`, `FCFLAGS` or `F90FLAGS`

Example Makefile

```
# serial compiler
CC      = icc
# compiler flags
CFLAGS  = -O2 -xHost -Wall
# include files
INC     = -I$(MKLROOT)/include
# libraries
LDFLAGS = -L$(MKLROOT)/lib/intel64 -openmp -mkl=parallel -lpthread -lm

# rules
.SUFFIXES:
.SUFFIXES: .c .h .o

.c.o:
    $(CC) $(INC) $(CFLAGS) $(COPTS) -c $<

.DEFAULT: blas
blas: blas_demo.o blas_demo_aux.o
    $(CC) $(CFLAGS) $(COPTS) -o blas_demo blas_demo.o blas_demo_aux.o $(LDFLAGS)
```

Compiling MPI programs

- Some parallel programs use MPI
  - As discussed already
- These should be compiled using the MPI wrapper for the compiler
  - Usually called `mpicc`/`mpif90`
- This takes exactly the same flags as the normal invocation of the compiler
  - In fact all it really is is the normal invocation with a few extra flags added for you!

Linker flags

- Similar to the compile stage the linker can also use many flags
- By far the most important of these for us are
  - `-o` which names the executable
  - flags to tell the linker where to find ‘extra’ object files
- This last point takes us towards libraries, our next point
- Libraries allow us to use very efficient code that somebody has already written
  - And so more efficiently use the cluster

Libraries

Use the Centrally Installed Libraries!

- When we install software on ARC systems we always try to install software using the best performing libraries — the toolchains help here.
- This is why you should always use the modules we provide
  - Unless you have a very specific need
- The performance of python, MATLAB, R,... really depends on these — use the centrally installed software if at all possible and don't install your own.
- Your first option should always be to use the toolchain libraries!

Useful Libraries — BLAS and LAPACK

- There are many useful libraries for Scientific computing and I'll mention a few over the next few slides
- Possibly the most important are
  - BLAS — Basic Linear Algebra Subprograms
  - LAPACK — Linear Algebra Package
- Reference versions are available from `https://www.netlib.org`
- However you **should not** use these
- Rather you should use one of the optimised implementations
  - MKL on Intel
  - ACML on AMD
  - OpenBLAS — Portable, optimised BLAS, continuation of GotoBLAS
- ARC machines mainly use Intel CPUs MKL is provided as part of the `intel` toolchain. OpenBLAS is also available.

Other useful Scientific Libraries

- FFTW — the de facto method for ffts (`https://www.fftw.org`)
- Boost — libraries for C++ programmers
- GSL — GNU Scientific Library (`https://www.gnu.org/software/gsl`)
- ScaLAPACK — distributed memory version of LAPACK
- NetCDF and HDF5 — libraries to make input/output easier and data more portable

Parallel libraries

- Some libraries are parallel
  - They can use multiple cores to accelerate the computation
- ScaLAPACK is distributed memory
  - To use it requires code changes
- But many implementations of BLAS and LAPACK can use shared memory parallelism
- So we can use this to make our calculations faster without changing the code
- As we saw previously — maths libraries are made available with the `foss` or `intel` toolchains.

Storage

Use of Storage

- Getting the best out of many applications depends on getting the best out of using the filesystem where the files your application uses are read from or written to
- How to best use the filesystem is cluster specific, but what is best for ARC often can be adapted with only small changes for other clusters
- There are 2 main issues
  1. Amount of I/O — i.e. using lots of disk
  2. Efficiency of I/O — i.e. accessing the filesystem as quickly as possible

Large Disk Usage

- On ARC systems you have two areas on the disk
- A small ‘home’ area — this is where you log into
- A much larger ‘data’ area
- Thus for your batch jobs we strongly suggest you use the data area
- The data area can be accessed from home via `cd $DATA`

```
*****************************************************************
* Welcome to ARC                                                *
*                                                               *
* Available clusters are:                                       *
* arc (large parallel jobs)                                     *
* htc (single-core to single-node jobs                          *
*                                                               *
*****************************************************************

Last login: Thu Oct 06 10:56:05 2022 from somewhere.arc.ox.ac.uk
[user@arc-login ~]$ pwd
/home/ouit0554
[user@arc-login ~]$ cd $DATA
[user@arc-login user]$ pwd
/data/myproject/user
```

Faster Disk Usage

- ARC also provides faster `$SCRATCH` space which is allocated on a per job basis:
  - This requires input data to be copied to the `$SCRATCH` area before running the application and the results copied back to `$DATA` upon completion.
    This must be performed in your submission script.
- for detailed information and examples on how to use `$SCRATCH` see the following page:
  `https://arc-user-guide.readthedocs.io/en/latest/arc-storage.html`

Efficient Disk Usage

- The filesystem on ARC systems is something called NetApp.
- This is a ‘high performance filesystem’
- However the way it works means that you will get best performance if you use a small number of large files accessing them in large chunks
  - Actually this is true for most filesystems
- Storing and accessing a very large number of small files will cripple your performance on ARC
- We have had cases of users having 100 000+ small (a few kbyte) files all in one directory. This will lead to **very** slow performance
- And what is more as the disks are shared it’s not just slow for you, it’s slow for everybody!

A Final Word on storage

- Please note the disks on ARC systems are not there for permanent storage
- They are **not** backed up
- After you have generated your data you should transfer it back to your ‘home machines’ via SFTP or similar mechanisms, and then delete it from ARC.

Measures of Performance

The most important measure

- In this section we will introduce a few measures of performance for computer codes, both serial and parallel
- But never forget that what you ultimately want to maximise is the amount of ***science per second*** that you generate
- This may be as simple as minimising the run time of your program
- But it may involve other factors
  - Using a more **familiar** application
  - Getting the best out of your computer budget
  - Turnaround on the cluster
    - **Higher core counts tend to turn around more slowly**

Parallel measures of performance

- Measuring the performance of parallel codes generally asks questions related to how much better are things running on multiple cores when compared to running on a single core or node
- We'll look at
  - Speed Up
  - Cost

Speed Up

- Speed up answers the question ‘How much faster does my program run if I use P cores’

$$S(P) = \frac{T_1}{T(P)}$$

- So if I use 100 processors it will run 100 times faster, right?
- And it can’t run more than 100 times faster, right?
- Also, I have 100 times as much memory so I can run 100 times bigger a problem, right?

Absolute Speed Up

- What we should really measure is Absolute Speed Up

$$S(P) = \frac{T_s}{T(P)}$$

- Where T_s is the time to run the best implementation of the serial program, and $T(P)$ is the time to run the parallel code on P cores
  - You may use a different algorithm in the parallel code from the serial code

Relative Speed Up

- However what is almost always measured is Relative Speed Up

$$S(P) = \frac{T(1)}{T(P)}$$

- i.e. we compare the speed against the time taken on 1 core by the parallel program
- Saves writing both the serial and parallel code

Linear Speed Up

- Linear speed up is simply

$$S(P) = P$$

- So if you run on P processors, it runs P times quicker
- This is the ideal situation
- Also called perfect scaling

What does it look like?

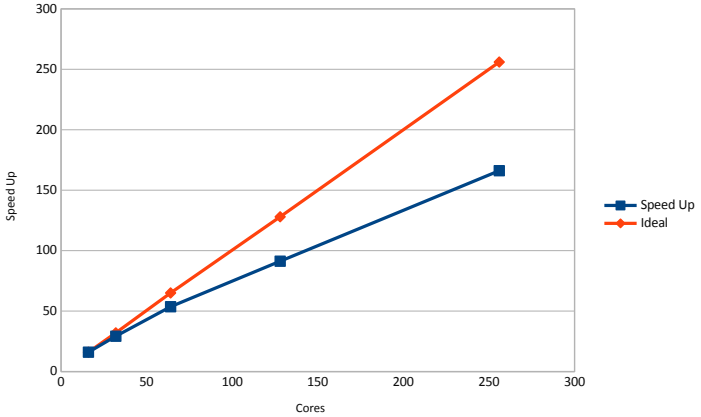

Figure: DL_POLY, 512 000 particles of NaCl on ARC

So Why is it Not Perfect!?

- Many Possible Reasons!
- We've touched on load balance
- To go beyond this again is beyond what we are trying to cover here
- The main thing is NOT to expect perfect speed up
- And to be aware that using too many cores can actually **DEGRADE** your performance

So How Many Cores Should I Use

- Well, firstly **don't put more processes or threads** on a node than there are cores!!
- It depends…
- It depends on the code you are running
  - It may have special parallel options which you should learn about
- It depends on the case you are running
- It depends on the computer you are running on
- More cores will cost you more
- More cores may even slow you calculation down
- More cores will probably mean slower turn around
- It depends upon you and how important cost and turnaround are
- BUT DON'T JUST GUESS!
- One thing you can do is to run a little experiment

An Experiment

- Many Scientific codes do the same kind of thing many times
  - e.g. timesteps
  - e.g. iterations in a solver
- So plot the speed up curve for a few iterations and from that decide on a good number of cores for you
- For instance a full `DL_POLY` run will require at the very least many thousands of times steps
- So first run it for 100 timesteps on 1, 2, 4, 8, 16,... cores and see what the speed up curve looks like
- And then use that number of cores for the full run

Summary

- It's very difficult to predict the performance of a real application *a priori*
- So you will have to do experiments
- Many applications are iterative
- So measure the performance on a number of different cores for a small number of iterations and use that to work out what to use for the full run

Thank You!

Any Questions?