Running multiple tasks on one node

- What and why?
- Job types
  - Serial (single threaded) jobs
  - MPI jobs
  - Threaded (OpenMP) jobs
- SLURM multi-prog support
- Cluster Considerations
  - Deneb
  - Fidis
- Other considerations
  - Memory bandwidth limited jobs
  - NUMA and CPU Affinity

What and why?

The majority of the SCITAS managed clusters are intended for parallel computations where many compute nodes are used as part of a distributed memory (MPI) programme. As such the minimum unit of resource allocation is a whole compute node.

It may be that users wish to use the SCITAS clusters for non-parallel tasks, or for shared memory tasks that cannot make use of all the cores on a node. In this case there are several options available:

1. Use a cluster/partition configured for high throughput computing (HTC) such as the serial partition on Deneb where nodes are shared and individual cores can be allocated - this is explained in the Cluster Considerations section
2. Use Job Arrays to create multiple tasks that can share nodes
3. Write a job script that runs multiple instances of the code so as to fill a compute node.

This guide covers the third case.

For Job Arrays please see:

https://slurm.schedmd.com/job_array.html
https://c4science.ch/source/scitas-examples/browse/master/Advanced/

Job types

There are three main types of task that one may wish to run. We discuss each type individually.

Also, please see the examples in https://c4science.ch/source/scitas-examples/browse/master/Advanced/OccupyOneNode/

Serial (single threaded) jobs

In this case we have a single threaded code (one that can only make use of one core) and we wish to run multiple instances at the same time so as to maximise the utilisation of the compute node.

The simplest way of achieving this is to have a job file that launches a certain number of independent tasks:
This is tedious to write but easy to understand. 16 tasks are launched in the background (the "&") and then the shell waits (the "wait") until they have all completed before the job script exits.

This can be simplified with the use of a BASH for loop

```bash
#!/bin/bash

#SBATCH --nodes=1
#SBATCH --ntasks=16
#SBATCH --mem=32G

mytask -in input01 -out output01 &
mytask -in input02 -out output02 &
mytask -in input03 -out output03 &
mytask -in input04 -out output04 &
mytask -in input05 -out output05 &
mytask -in input06 -out output06 &
mytask -in input07 -out output07 &
mytask -in input08 -out output08 &
mytask -in input09 -out output09 &
mytask -in input10 -out output10 &
mytask -in input11 -out output11 &
mytask -in input12 -out output12 &
mytask -in input13 -out output13 &
mytask -in input14 -out output14 &
mytask -in input15 -out output15 &
mytask -in input16 -out output16 &

wait
```

More complicated logic and flow control is also possible.

```bash
#!/bin/bash

#SBATCH --nodes=1
#SBATCH --ntasks=16
#SBATCH --mem=32G

for A in $(seq -w 1 16); do
  mytask -in input$A -out output$A &
done

wait
```

The following example allows us to run an arbitrary number of serial tasks (300 in this case), without having to tailor the script to a particular number of cores (i.e. machine type). This example is even suitable to fill multiple nodes with serial tasks (by simply changing the `--nodes` option).
#!/bin/bash

#SBATCH --nodes=1
#SBATCH --cpus-per-task 1
#SBATCH --mem-per-cpu 4G

echo started at `date`

module purge
module load intel intel-mpi
module load chem2020

# calculate how many tasks can we run simultaneously (i.e. how many cores are available)
NR_TASKS=$(echo $SLURM_TASKS_PER_NODE | sed 's/\([0-9]\+\)(x\([0-9]\+\))/\1*\2/' | bc)

# execute the 300 tasks, NR_TASKS tasks at a time

cat <(seq 1 300) | xargs -I{} --max-procs=${NR_TASKS} bash -c "srun -N 1 -n 1 --exclusive chem2020.x -in myinput.{} -out myout.{}"

echo finished at `date`

The key here to running multiple jobs is using the "cpus-per-task" directive and then setting the amount of memory per CPU (i.e. per task) with #SBATCH --mem-per-cpu 4G rather than asking for an overall amount of memory per node. Failure to do this may result in strange and sub-optimal behaviour!

**MPI jobs**

The typical problem is an MPI code that scales well to a certain point and that point is significantly less than the number of cores in a compute node. Here we use the example executable chem2020.x which scales well up to 8 ranks.

We will run this code on a cluster that has 24 cores per compute node.

*Warning: Running multiple MPI jobs per node is currently only possible with IntelMPI and MVAPICH2 (i.e. OpenMPI does not work)*

As 24 / 8 = 3 we can run three instances at once. This means that no cores are wasted

#!/bin/bash

#SBATCH --nodes=1
#SBATCH --ntasks=24
#SBATCH --mem=48G

module purge
module load intel intel-mpi
module load chem2020

srun -n 8 --exclusive --mem=16G chem2020.x -in myinput.1 -out myout.1 &
srun -n 8 --exclusive --mem=16G chem2020.x -in myinput.2 -out myout.2 &
srun -n 8 --exclusive --mem=16G chem2020.x -in myinput.3 -out myout.3 &

wait

echo finished at `date`

Here the important points to node are that we background each srun with the "&", allocated one third of the memory to each task and then tell the shell for wait for all child processes to finish before exiting.
If the tasks are not run in the background then they will run one after the other and if the memory is not divided then the first srun will take the entire allocation thus preventing the others from starting which also causes the sequential execution of the calls to srun.

**Threaded (OpenMP) jobs**

The typical problem is an OpenMP code that scales well to a certain point and that point is significantly less than the number of cores in a compute node. Here we use the example executable bio2020.x which scales well up to 6 threads.

We will run this code on a cluster that has 24 cores per compute node.

As 24 / 6 = 4 we can run four instances at once.

By default OpenMP tasks will run as many threads as there are cores on the compute node. We need to modify this behaviour using the OMP_NUM_THREADS variable.

```bash
#!/bin/bash -l

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

module purge
module load intel
module load bio2020

export OMP_NUM_THREADS=6

srun -n 1 --exclusive bio2020.x -in myinput.1 -out myout.1 &
srun -n 1 --exclusive bio2020.x -in myinput.2 -out myout.2 &
srun -n 1 --exclusive bio2020.x -in myinput.3 -out myout.3 &
srun -n 1 --exclusive bio2020.x -in myinput.4 -out myout.4 &
wait

echo finished at `date`
```

**SLURM multi-prog support**

The SLURM batch system provides an alternative mechanism which can be used instead of or in addition to the aforementioned methods.

```bash
srun -n 24 --multi-prog tasks.conf
```

Where the tasks.conf is a description of what tasks to run on which processors.

```bash
# My multi-prog conf file saved as tasks.conf
#
# Here we run three MPI programs, the first uses 12 ranks and the second and third have 6 ranks.
#
0-11: code.x -in in.1 -out out.1
12-17: code.x -in in.2 -out out.2
18-23: code.x -in in.3 -out out.3
```
Cluster Considerations

Deneb
Deneb has a dedicated serial partition which allows users to request CPUs and memory. The jobs running on this partition may well share the node with other jobs as explained at [https://scitas-data.epfl.ch/kb/Deneb%3A+using+the+sequential+partition](https://scitas-data.epfl.ch/kb/Deneb%3A+using+the+sequential+partition)

Fidis
On Fidis we set the SLURM option "ExclusiveUser=YES" which allows jobs from the same user to share a node. This means that if one submits 28 jobs each requesting 1 core and 4GB of memory then they will be able to all run on one node.

Although this helps help improve utilisation when running multiple jobs which do not fully occupy a node it is not optimal use of the Fidis parallel partition. There will still be many situations in which most of a node will be blocked by a job which doesn't fully utilise it.

We cannot use this option on Deneb due to some subtleties of the QLogic/Intel Infiniband Interconnect behaviour.

If, on Fidis, you really want to run a less than 28 CPU job and not have any other of your jobs running alongside then you need to pass the "--exclusive" option to sbatch.

Other considerations

Memory bandwidth limited jobs
It is not always correct to assume that using all the cores on a compute node will give the best performance/throughput as many codes are limited by the bandwidth to the main memory.

In this case you may be better off running fewer instances per CPU socket so as to maximise the available bandwidth per thread/rank.

NUMA and CPU Affinity
In the case of the MPI tasks that use 8 cores we see that we may end up in the following situation where one of the tasks runs across both CPUs. Here the second tasks runs with 4 tasks on CPU0 and 4 on CPU1

```
CPU 0           CPU 1
cccccccccccccccccccc|cccccccccccccccc
000000000111111122222
123456789012345678901234
|srun -n 8      |srun -n 8      |srun -n 8      |
```

This can be a problem due to the way memory is physically distributed across the system. In the above case, as each MPI rank has its own memory space then, with correct pinning/affinity there should be no performance degradation.

Similar problems can occur whenever the number of cores per MPI/OpenMP task is not exactly 1/2 of the total. In this case it is up to you to use CPU binding to ensure the optimal placement of the tasks on the underlying hardware. This can be via a CPU mask given to srun or using numactl for threaded tasks along with an appropriate affinity variable.

Please see the page on [CPU affinity](https://scitas-data.epfl.ch/kb/) for more details.
The course on using MPI may also be of interest - the slides are available HERE.