Parallel Processing

Parallel processing is a mode of operation in which a process is split into parts, which are executed simultaneously on different processors attached to the same computer. There are several different methods to go around running parallel jobs and different methods have different benefits.

Brief descriptions of various parallel jobs:

- Multi-process program where tasks are split up and run simultaneously on multiple processors with different input.
- Running a multithreaded program with OpenMPI or pthreads.
- Running several instances of a single-threaded program (embarrassingly parallel, or a job array)
- Running one master program controlling several slave programs (master/slave)

Note: For Parallel Processing, Jukka Suomela from Aalto has produced excellent materials and courses.

Why would you do things in paralle? Besides because its fun?

Mostly because when you have more processes to work on a project, it will be completed faster. We'll introduce some analogies here now, so skip this chapter if you don't like them.

You can have one well paid labourer to build a pyramid but it takes quite some time, unless the pyramid is very small. For grande design, you'd probably need a bit more well paid labourers. I think ten thousand ought to do it and you hand down everyone a set of instructions to build a pyramid. All goes well, everyone is silent and shortly you have ten thousand little pyramids.

Well, your well paid labourers are not exactly the smartest bunch on the planet, and nobody told them to build a single pyramid.

You think that probably the workers should be allowed to talk after all.

What is MPI?

The message passing interface (MPI is) is a standardized means of exchanging messages between multiple computers running a parallel program across distributed memory.

From the Slurm and job placement perspective, it is important to know if your application scales well. Understanding of the basic differences of nodes, tasks and cpu's is essential (*for all intents and purposes, cpu is indistinguishable from core*). In short therefore,

- **Task** describes how many instances of your command are executed. In Slurm language, Tasks are referred to as "-–ntasks" or "-n".
- **CPU** describes how many coress your command can use. In Slurm language, CPU's are referred to as "--cpu-per-tasks", or "-c".
- **Node** describes computational unit which may contain many CPU's sharing same memory space. In Slurm language nodes are referred to as "–node" or "-N".

MPI Implementation Considerations

Turso has several different MPI libraries available for use. Generally OpenMPI is universal, while Intel MPI and MVAPICH provide better overall performance. Which one to use depends on your code and personal preferences. Slurm handles different versions somewhat differently.

OpenMPI 2.x does not require specific options for Infiniband, since it is defined as default. However, other MPI implementations do require some specific settings. See below:

Intel MPI requires user to point out explicitly to the PMI library

```
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so
```

Details of Slurm + impi:

https://software.intel.com/en-us/articles/how-to-use-slurm-pmi-with-the-intel-mpi-library-for-linux

OpenMPI 3.x or later requires user to explicitly set pmix to be set for srun:

```
--mpi=pmix
```

MVAPICH requires at the moment for user to explicitly set pmi2 to be set for srun:

```
--mpi=pmi2
```

Message Passing with OpenMPI

You can think of above this way: You decide that you want to have 2 instances (*tasks*) of the command to be executed, and then allocate each instance (*task*) one CPU core to run on. This would make a Message Passing MPI job (*parameters: -n 2, -c 1*). Each task (*instance*) of the command could run in same, or different node. Tasks (*instances*) in separate nodes would communicate with each other over the interconnect network. Example follows:

Note! If you are using OpenMPI 4.x or newer, Infiniband is not assumed as the interconnect. You will need to specify this explicitly with mca -parameter.

```
--mca btl_openib_allow_ib 1
```

You can find example MPI program from Wikipedia: https://en.wikipedia.org/wiki/Message_Passing_Interface#Example_program

First, you will need to compile the program:

```
module load openmpi
mpicc wiki_mpi_example.c -o foobar.mpi
```

Example of job requesting four tasks with 1 core for each task:

```
#!/bin/bash
#
#SBATCH --job-name=foobar_mpi
#SBATCH --output=<mpitest.out>
#SBATCH -c 1                      ## CPU cores per task
#SBATCH -n 4                      ## number of tasks
#SBATCH -N 4                      ## number of nodes
#SBATCH -t 10:00
#SBATCH --mem-per-cpu=100M        ## Memory request

module purge
module load OpenMPI
srun foobar.mpi
```

MPI and Hyperthreaded Cores

If you use nodes with Hyperthreading enabled (*eg. anything but kaXX nodes*) cores you probably would like not to use the HT for performance reasons. To avoid the HT performance impact, and the utilisation of HT cores on the nodes where HT is anabled, you can use following:

```
#SBATCH --hint=nomultithread
```

MPI and Network Filesystem Performance Warning

With certain versions of MPI, you may see an warning message like this concerning performance and network filesystems/Lustre:

```
WARNING: Open MPI will create a shared memory backing file in a
directory that appears to be mounted on a network filesystem.
Creating the shared memory backup file on a network file system, such
as NFS or Lustre is not recommended -- it may cause excessive network
traffic to your file servers and/or cause shared memory traffic in
Open MPI to be much slower than expected.
...
```

In reality, we have found that unless the FS is exceedingly busy, this is somewhat overcautious in case you use $WRKDIR and the performance drop, if any, is negligible.

MPI Performance Options

Following option can be given to srun or sbatch to inform Slurm that multiple nodes are used and job placement is logically contiguous.

```
#BATCH --contiguous
```

```
srun --contiguous <rest of the arguments>
```

Shared memory OpenMP

If however, you have command to run on a single instance (*task*), with 4 cpu's, and use local memory space for communications, it would be Shared Memory OpenMP job (*parameters: -n 1, -c 4*). All requested CPU's have to be from the same node and maximum size for the job equals to the number of CPU's sharing the same memory space. Eg. if a node has 24 cores, that would be the maximum size for the job.

First, you would need to compile the program:

```
gcc -fopenmp wiki_omp_example.c -o fobar.omp
```

OpenMP example requesting one task and 4 cpus for the task:

```
#!/bin/bash
#
#SBATCH --job-name=foobar_omp
#SBATCH -o foobar-omp.out
#
#SBATCH -n 1
#SBATCH -c 4
#SBATCH -t 10:00
#SBATCH --mem-per-cpu=100M
export OMP_PROC_BIND=TRUE
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
./foobar.omp
```

MPI - Consumable resources

Below are some of the most common options to specify when submitting MPI job.

Job Wall Time limit:

```
#SBATCH -t <Wall Time limit>
```

Number of tasks in the MPI job:

```
#SBATCH -n <number of tasks>
```

Job CPU count equals to the cores:

```
#SBATCH -c <Number of CPU cores per task>
```

Job memory limit:

```
#SBATCH --mem-per-cpu=<MB>
```

Job Arrays - Embarrassingly Parallel Jobs

Slurm supports job arrays where multiple jobs can be executed with identical parameters. Each job in the array is considered for execution separately, like any other ordinary batch job and is therefore also subject to the fairness calculations. Ukko2 supports practically unlimited array size. However, please do pay attention to the special nature of the arrays when constructing your own.

Array Job Usage

If using array jobs, and the job requirements change during array run, it is advisable to divide the array to chunks with identical resource requirements. Oversubscribing prohibits other jobs from using reserved resources.

There is no practical limit for number of array jobs that can be submitted, or executed at any one time. However system has default limit of 10001 for a single array.

Example for array job script:

```
#!/bin/bash
#
#SBATCH --job-name=emb_arr
#SBATCH --output=res_emb_arr.txt
#SBACTH -c 1
#SBATCH -n 1
#SBATCH -t 10:00
#SBATCH --mem-per-cpu=100
#
#SBATCH --array=1-8

srun ./example.emb $SLURM_ARRAY_TASK_ID
```

While arrays are extremely efficient way to submit large quantity of jobs, it is not advisable to submit array with entirely different task requirements and base the resource request on the highest array task. (Eg. *2000 1 core tasks using 15Mb of memory and 1000 1 core tasks using 250Gb of memory, and setting the limit to latter for whole 3000 task array. Instead, it is far more efficient to create separate 2000 and 1000 task arrays with appropriate resource requests*).

Performance Impact of Small Array Jobs

If the running time of your program is short, creating a job array will incur a lot of overhead and you should consider *packing* your jobs. For this we can recommend use of somewhat obscurely named option "--exclusive" as in example below. To submit 1000 jobs in packs of 8 (allocating one task for a job, and one core for each task):

```
#!/bin/bash
#
#SBATCH --ntasks=8
for i in {1..1000}
do
   srun -c 1 -n 1 --exclusive ./example.emb $i &
done
wait
```

Array Job e-mail notifications

Additionally if array jobs are used, you may wish to receive single notification once the whole array is done (default behaviour). Or by setting ARRAY_TASKS -option, receive mail notification for every task of the array. Latter may be useful for debugging but may be inconvenient if array has thousands of tasks.

```
#SBATCH --mail-type=ARRAY_TASKS,<other options>
```

Master/Slave

This is typically used in a **producer/consumer** setup where one program creates computing tasks for the other programs to perform (f*or example Hadoop cluster*).

Creating reservation for Master/Slave type jobs, requesting four tasks and one core for each task. Example assumes that task placement is not relevant, and they can end up on same node, or different nodes. See below more details about placement:

```
#!/bin/bash
#
#SBATCH --job-name=test_ms
#SBATCH --output=foobar.out
#SBATCH -n 4
#SBATCH -c 1
#SBATCH -t 10:00
#SBATCH --mem-per-cpu=100

srun --multi-prog master.cfg
```

File "master.cfg" would then state the intended configuration, for example:

```
0      echo Master
1-3    echo Slave %t
```

This instructs Slurm to create four tasks. One of them running "Master" and other three "Slave".

Task placement to multiple nodes

At times, there may be need to place each task to a separate node (*performance, scalability testing, I/O, or other reasons*). Example to submit a four task job, one core for each task and each task placement to a separate node:

```
#!/bin/bash
#SBATCH --tasks-per-node=1
#SBATCH - n 4
#SBATCH - c 1
#SBATCH -o out.put
srun foobar
```

Advanced Parallel Processing

There are several special MPI hybrid cases that may be very useful in specific circumstances. Below a few examples to show how they can be done in Slurm.

MPI and multithreading

First, it is possible to mix MPI and OpenMP multithreading in a simple job (job called foobar assumes reservation of 8 tasks, 4 cores each):

```
#!/bin/bash
#
#SBATCH -n8
#SBATCH -c 4
module load OpenMPI
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
srun ./foobar
```

MPI and Array Jobs

Besides of mixing MPI and multithreading, you can also mix array jobs, below a brief example:

```
#!/bin/bash
#
#SBATCH --array=1-10
#SBATCH -n 8
#SBATCH -c 4
module load OpenMPI
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
srun ./fobar $SLURM_ARRAY_TASK_ID
```

GPU Jobs

Example GPU batch script could look like following:

```
#!/bin/bash
#SBATCH --job-name=GPUbar
#SBATCH -n 1
#SBATCH -c 1
#SBATCH --ntasks-per-node=1
#SBATCH --time=1:00:00
#SBATCH --mem-per-cpu=1000
#SBATCH -p gpu
#SBATCH --gres=gpu:1

module load <application/version>

executable <input.dat>
```


GPU Usage

If you need GPU the queue needs to be specified with -p parameter as well as the requested GPU resource:

```
#SBATCH -p gpu

#SBATCH --gres=gpu:1
```

Note that it is always good idea to request at least as many CPU cores as GPUs.

Advanced GPU optimization

Resource affinity may have significant impact on the GPU performance. For this purpose "--accel-bind" was added to Slurm. Following options are supported:

- g Bind each task to GPUs which are closest to the allocated CPUs.
- m Bind each task to MICs which are closest to the allocated CPUs.
- n Bind each task to NICs which are closest to the allocated CPUs.
- v Verbose mode. Log how tasks are bound to GPU and NIC devices.

Here is an example srun command:

```
srun --gres=gpu:2 --ntasks-per-socket=2 --accel-bind=g -n 4 ./foobar-MPI
```