Parallelism

If you are working on complex and large-scale problems that require high performance computing (HPC), you may wonder how to choose the best architecture for your needs.

HPC systems can be classified into two main types: ***shared memory*** and ***distributed memory_ systems***.

Shared memory parallelism (threading)

In *shared-memory parallelism (SM)*, applications achieve parallelism by executing more than one thread at a time across cores within one node. Each of the threads see and manipulate the same memory that the initial process allocated. Threads are light-weight and fast, independently processing portions of work. With SM parallelism, **jobs are limited to the total amount of memory and cores on one node**.

To run a threaded application properly through Slurm, you will need to specify a number of Slurm constraints. For example, to launch a Slurm job for your threaded application that will use 16 threads do:

```
#!/bin/bash
#SBATCH -J my_job
# Numero de Nodos y cores por nodo
#SBATCH -N 1
#SBATCH --cpu-per-task=16
#SBATCH --mem=57000
#SBATCH --partition=batch

module load GCC/12.2.0
srun ./myapp
```

If you opt to not use *srun* to launch your application, you must add this to your Slurm job script: `export OMP_NUM_THREADS=16` This number must match *--cpus-per-task*.

Distributed memory parallelism (MPI)

In *distributed-memory parallelism (DM)*, an application achieves parallelism by running multiple instances of itself across multiple nodes to solve the problem. Each instance is allocated its own chunk of virtual memory and communicates to other instances via a message passing interface such as MPI.

For example, to use 4 nodes and 16 processors per node,

```
#!/bin/bash
#SBATCH -J my_job
#SBATCH -N 4
#SBATCH --tasks=16
#SBATCH --mem=28000
#SBATCH --partition=batch

module load   intel/2021b   intel-compilers/2021.4.0   impi/2021.4.0
srun ./myapp
```

DM + SM parallelism (hybrid)

In hybrid parallelism, applications achieve parallelism with the use of both threads and MPI tasks. This type of parallelization combines the features of the previous two models, allowing a program to run threading on multiple nodes across the cluster.

Hybrid jobs can potentially run more efficiently (consuming less memory and scaling further) by reducing the MPI communication overhead of a job. This can be done by substituting light-weight threads for MPI ranks.

To set up jobs to use DM+SM, first make sure your app supports it.

```
#!/bin/bash
#SBATCH –cpus-per-task=12  #same as -c 12
#SBATCH –ntasks=4          #same as -n 4
#SBATCH –ntasks-per-node=2

module load gompi/2022b
srun ./myapp
```

DM jobs can be a performance hit due to the MPI communication steps when running the application over too many nodes.

For example, if you have a DM job that runs best with 64 cores, you can then request that your job run on only 4 nodes with *nodes=4*. Of course, by doing this your job will likely take longer to start due to the extra job constraint, but it could be worth it for your analyses. The next best thing to do is to ask Slurm to preferably use 4 nodes, but if not possible, use 5, or 6. You would specify this with *--nodes=4-6*. Without specifying *--nodes*, your MPI job could run on any number of nodes across the cluster.

OpenMP/Multithreading vs. MPI

While there are several ways to request a certain amount of CPU cores for your program notice the following distinction:

- **Requests *ntask* (no of) CPU cores for MPI ranks (distinct processes)**

These can be distributed over several compute nodes

```
#SBATCH --ntask= <ntask>
```

- **Requests *nodes* and *ntask-per-node* (no of) CPU cores for MPI ranks (distinct processes) in the same node**

These distribute over *no of* compute nodes *X* task.

```
#SBATCH --nodes= <nodes>
#SBATCH --ntask-per-node= <ntas-per-node>
```

- **Request *cpu-per-task* (no of) CPU cores for multithreaded applications (eg. OpenMP)**

In contrast the option "--cpus-per-task" specify how many CPUs each task can use. These will always be allocated inside one single compute node, never to several nodes!

```
#SBATCH --cpus-per-task= <cpus-per-task>
```

Some advice about this.

For a plain MPI application, use

```
--ntasks_,
usin Distributed Memory (across nodes)
requires MPI.
```

For a plain OpenMP/multithreaded application, use

```
--ntasks=1
--cpus-per-task=X,
using Shared Memory (inside a single node).
```

For a hybrid application, use

```
--ntasks=<no of nodes>
--cpus-per-task=<no of cores per node>
using both SM and DM,
requires MPI.
```

The SBATCH option *--ntasks-per-core=#* is only suitable for compute nodes having HyperThreading enabled in hardware/BIOS

In TeideHPC, **by default HyperThreading is disabled**, that is to say `#SBATCH --ntasks-per-core=1`

Do your own scalability tests.

Start with 2 nodes and study the results and increase the number of nodes or tasks little by little.

Basic example: *use srun or not*

```
#!/bin/bash

#SBATCH --ntasks=8
## more options
echo hello
```

This should always output only a single line, because the script is only executed on the submitting node not the worker.

```
#!/bin/bash
#SBATCH --ntasks=8
## more options
srun echo hello
```

srun causes the script to run your command on the worker nodes and as a result you should get 8 lines of hello.

Basic examples: *use ntask and 1 thread applications.*

- Using the default *ntasks=1*

```
#!/bin/bash

#SBATCH --ntasks=1

srun sleep 10 &
srun sleep 12 &
wait
```

The number of tasks by default was specified to one, and therefore the second task cannot start until the first task has finished. This job will finish in around 22 seconds. To break this down:

`sacct -j1425 --format=JobID,Start,End,Elapsed,NCPUS`

```
       JobID             Start               End    Elapsed      NCPUS
------------ ------------------- ------------------- ---------- ----------
1425         2023-07-13T20:51:44 2023-07-13T20:52:06   00:00:22          1
1425.batch   2023-07-13T20:51:44 2023-07-13T20:52:06   00:00:22          1
1425.0       2023-07-13T20:51:44 2023-07-13T20:51:56   00:00:12          1
1425.1       2023-07-13T20:51:56 2023-07-13T20:52:06   00:00:10          1
```

Here task 0 started and finished (in 12 seconds) followed by task 1 (in 10 seconds). To make a total user time of 22 seconds.

- Using *ntasks=2*

```
#!/bin/bash

#SBATCH --ntasks=2

srun --ntasks=1 sleep 10 &
srun --ntasks=1 sleep 12 &
wait
```

Running the same sacct command as specified above:

`sacct -j 515064 --format=JobID,Start,End,Elapsed,NCPUS`

```
JobID             Start               End    Elapsed      NCPUS
------------ ------------------- ------------------- ---------- ----------
515064       2023-07-13T21:34:08 2023-07-13T21:34:20   00:00:12          2
515064.batch 2023-07-13T21:34:08 2023-07-13T21:34:20   00:00:12          2
515064.0     2023-07-13T21:34:08 2023-07-13T21:34:20   00:00:12          1
515064.1     2023-07-13T21:34:08 2023-07-13T21:34:18   00:00:10          1
```

More examples

Visit our repository at github https://github.com/hpciter/user_codes.git