Choosing the Number of Nodes, CPU-cores and GPUs

researchcomputing.princeton.edu/support/knowledge-base/scaling-analysis

OUTLINE

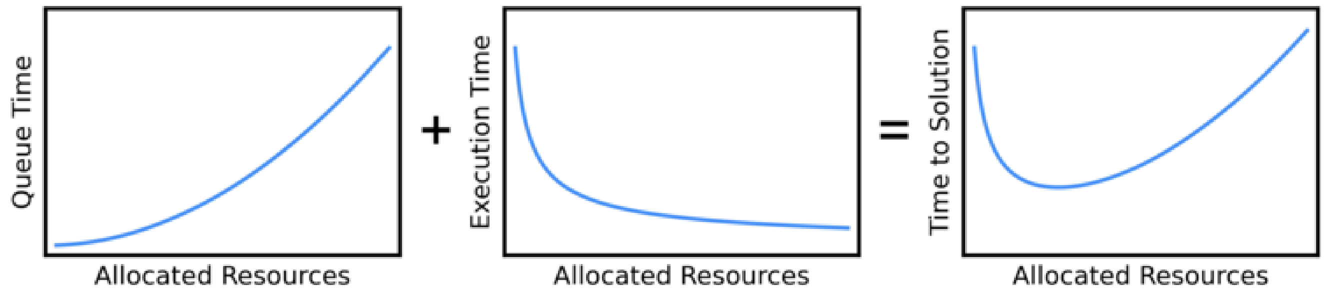
Introduction

Before you start doing production runs with a parallelized code on the HPC clusters, you first need to find the optimal number of nodes, tasks, CPU-cores per task and in some cases the number of GPUs. This page demonstrates how to conduct a **scaling analysis** to find the optimal values of these parameters for different types of parallel codes.

Time to Solution

When a job is submitted to the Slurm scheduler, the job first waits in the queue before being executed on the compute nodes. The amount of time spent in the queue is called the queue time. The amount of time it takes for the job to run on the compute nodes is called the execution time.

The figure below shows that the queue time increases with increasing resources (e.g., CPU-cores) while the execution time decreases with increasing resources. One should try to find the optimal set of resources that minimizes the "time to solution" which is the sum of the queue and execution times. **A simple rule is to choose the smallest set of resources that gives a reasonable speed-up over the baseline case.**



Note that the information on this page only applies to parallel codes. If your code is not parallelized then using more resources will not improve its performance. Instead, doing so will waste resources and it will lower the priority of your next job.

You should not try to explicitly compute the time to solution. This is because the queue time for a given job varies widely depending on your academic department, your fairshare value, the QOS of the job, the time of year and so on. Instead of trying to estimate the queue time, simply keep in mind that, in general, the more resources you request, the more time your job will spend in the queue before running. Execution times are easy to measure and they are reported as "Job Wall-clock time" in the Slurm email report of a completed job.

Note that when performing a scaling analysis you do not need to run your code for hours to get meaningful data. However, you do need to run it for long enough such that one-time start-up operations can be ignored. If necessary add timing statements in your code so that only the relevant sections are measured. Note that you should make sure you run for at least tens of seconds so that unavoidable system operations can be neglected.

Below we demonstrate how to carry out a scaling analysis for the different types of parallel codes. The scaling analysis allows us to estimate the optimal values of the Slurm directives. As just explained, the queue time is not taken into account when a scaling analysis is performed.

Serial Codes

For a serial code there is only once choice for the Slurm directives:

```
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
```

Using more than one CPU-core for a serial code will not decrease the execution time but it will waste resources and leave you with a lower priority for your next job. See a sample Slurm script for a serial job.

Multithreaded Codes

Some software like the linear algebra routines in NumPy and MATLAB are able to use multiple CPU-cores via libraries that have been written using shared-memory parallel programming models like OpenMP, Intel Threading Building Blocks (TBB) or pthreads. For pure multithreaded codes, only a single node and single task can be used (i.e., nodes=1 and ntasks=1) and the optimal value of cpus-per-task is sought:

```
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=<T>
```

See a sample Slurm script for a multithreaded job. To find the optimal value of <T> one must conduct a scaling analysis where cpus-per-task is varied and the execution time of the code is recorded for each choice:

ntasks	cpus-per-task	execution time	speed-up ratio	parallel efficiency

ntasks	cpus-per-task	execution time	speed-up ratio	parallel efficiency
1	1	42.0	1.0	100%
1	2	22.0	1.9	95%
1	4	16.0	2.6	66%
1	8	7.8	5.4	67%
1	16	6.5	6.5	40%
1	32	7.1	5.9	18%

In the table above, the execution time is how long it took the job to run (i.e., wall clock) and the **speed-up ratio** is the serial execution time (cpus-per-task=1) divided by the execution time. The **parallel efficiency** is measured relative to the serial case. That is, for cpus-per-task=2, we have 42.0 / (22.0 × 2) = 0.95. The parallel efficiency is approximately equal to "CPU Efficiency" in Slurm email reports.

The data in the table above reveal two key points:

- The execution time decreases with increasing number of CPU-cores until cpus-per-task=32 is reached when the code actually runs slower than when 16 cores were used. This shows that the goal is not use as many CPU-cores as possible but instead to find the optimal value.
- The optimal value of cpus-per-task is either 2, 4 or 8. The parallel efficiency is too low to consider 16 or 32 CPU-cores.

In this case, your Slurm script might use these directives:

```
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=2
```

Multinode or Parallel MPI Codes

For a multinode code that uses MPI, for example, you will want to vary the number of nodes and ntasks-per-node. Only use more than 1 node if the parallel efficiency is very high when a single node is used. To minimize the time to finish, choose the smallest set of the Slurm directives that gives a reasonable speed-up. For a pure MPI code that does not use threading (e.g., OpenMP), cpus-per-task=1 and the goal is to find the optimal values of nodes and ntasks-per-node:

```
#SBATCH --nodes=<M>
#SBATCH --ntasks-per-node=<N>
#SBATCH --cpus-per-task=1
```

See a full example of a Slurm script for an MPI job. Below is a sample scaling analysis for a parallel MPI code:

nodes	ntasks-per-node	CPU-cores	execution time	speed-up ratio	parallel efficiency
1	1	1	1200	1.0	100%
1	2	2	605	2.0	99%
1	4	4	306	3.9	98%
1	8	8	157	7.6	96%
1	16	16	78	15	96%
1	32	32	40	30	94%
2	32	64	21	57	90%
3	32	96	15	80	83%
4	32	128	14	86	67%

We see that the code performs very well until four nodes or 128 CPU-cores were used. A good choice is probably to use two nodes where the parallel efficiency is still 90%. See a sample Slurm script for a pure MPI code.

Hybrid Multithreaded, Multinode Codes

Some codes take advantage of both shared- and distributed-memory parallelism (e.g., OpenMP and MPI). In these cases you will need to vary the number of nodes, ntasks-per-node and cpus-per-task. Construct a table as above except include a new column for cpus-per-task. Note that when taking full nodes, the product of ntasks-per-node and cpus-per-task should be equal to the total number of CPU-cores per node. Use the "snodes" command to find the total number of CPU-cores per node for a given cluster.

Find the optimal values for these Slurm directives:

```
#SBATCH --nodes=<M>
#SBATCH --ntasks-per-node=<N>
#SBATCH --cpus-per-task=<T>
```

See a sample Slurm script for a multithreaded, multinode job.

GPU Codes

Before considering multiple GPUs, one should first demonstrate high GPU utilization when a single GPU is used. See the GPU Computing page to learn about measuring and improving the utilization. If the GPU utilization is sufficiently high for the single GPU case then you should explore using multiple GPUs by performing a scaling analysis such as in the table below:

nodes	GPUs	execution time	speed-up ratio	parallel efficiency
1	1	212	1.0	100%
1	2	140	1.5	75%
1	3	110	1.9	64%
1	4	105	2.0	50%
2	8	145	1.5	18%

The scaling analysis above reveals that the code does not perform well when multiple GPUs are used. That is, linear scaling is not observed. For instance, the performance with two GPUs is not twice as fast as the case with one. Keeping in mind that the queue time increases with increasing resources, it may not make sense to use two GPUs for this specific code.

Notice that we did not mention the number of CPU-cores being used in the analysis above. This is because of the far greater computational power of a GPU in comparison to a multicore CPU. **However, it is often critical to the performance of a GPU-enabled code that one or more CPU-cores be fully leveraged.** For instance, for the deep learning codes TensorFlow and PyTorch, optimal performance can only be achieved when multiple CPU-cores are used to keep the GPU busy by feeding it data.

Many scientific codes use OpenMP, MPI and GPUs. In this case one seeks the optimal values for nodes, ntasks-per-node, cpus-per-task and gres.

See a sample Slurm script for a simple GPU job.

Getting Help

If you encounter any difficulties while conducting a scaling analysis on the HPC clusters then please send an email to cses@princeton.edu or attend a help session.