Calcolo Parallelo dall'Infrastruttura alla Matematica

Auxiliary Tools

Laurea Triennale e Magistrale in Matematica

Fabio Durastante

May 18, 2023

Dipartimento
di Matematica
Università di Pisa

Table of Contents

Timers and Synchronization

- A timer is specified even though it is not an instruction based on “*message-passing*”: timing parallel programs is important for inquiring on the “*performances*” of your code.

Timers and Synchronization

- A timer is specified even though it is not an instruction based on "*message-passing*": timing parallel programs is important for inquiring on the "*performances*" of your code.
- the timer returns a floating-point number of seconds, representing elapsed wall-clock time since *some time in the past*:

```
double MPI_Wtime(void);
```

the *time in the past* is guaranteed not to change during the life of the process.

Timers and Synchronization

- A timer is specified even though it is not an instruction based on "*message-passing*": timing parallel programs is important for inquiring on the "*performances*" of your code.
- the timer returns a floating-point number of seconds, representing elapsed wall-clock time since *some time in the past*:

```
double MPI_Wtime(void);
```

  the *time in the past* is guaranteed not to change during the life of the process.
- the usual application of a timer is something of the form:

```
double starttime, endtime;
starttime = MPI_Wtime();
< --- foolish things happen here --- >
endtime = MPI_Wtime();
printf("That took %f seconds\n",endtime-starttime);
```

Timers and Synchronization

- A timer is specified even though it is not an instruction based on "*message-passing*": timing parallel programs is important for inquiring on the "*performances*" of your code.
- the timer returns a floating-point number of seconds, representing elapsed wall-clock time since *some time in the past*:

```
double MPI_Wtime(void);
```

  the *time in the past* is guaranteed not to change during the life of the process.
- There exists a tag `MPI_WTIME_IS_GLOBAL` that is `1` if clocks at all processes in `MPI_COMM_WORLD` are synchronized, `0` otherwise.

Timers and Synchronization

- MPI offers a *barrier* function that blocks the caller until all processes in the communicator have called it

  ```
  int MPI_Barrier(MPI_Comm comm)
  ```

  that is, the call returns at any process only after all members of the communicator have entered the call.

Timers and Synchronization

- MPI offers a *barrier* function that blocks the caller until all processes in the communicator have called it

  ```
  int MPI_Barrier(MPI_Comm comm)
  ```

  that is, the call returns at any process only after all members of the communicator have entered the call.

- It can be used together with the `MPI_Wait` function to force a synchronization point in the program.

Timers and Synchronization

- MPI offers a *barrier* function that blocks the caller until all processes in the communicator have called it

  ```
  int MPI_Barrier(MPI_Comm comm)
  ```

  that is, the call returns at any process only after all members of the communicator have entered the call.

- It can be used together with the `MPI_Wait` function to force a synchronization point in the program.

- It can be used to regulate the access to an external resource (e.g., a file) in such a way that every processor accesses it in an order way: if you are interested in writing file in parallel you can look at Chapter 13 of the MPI guide[1]

[1] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 3.1. `https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`, High Performance Computing Center Stuttgart (HLRS).

Evaluating performances

You can use the `MPI_Wtime()` to give a simple evalaution of the performances of your program.

Consider, e.g., the two programs for the computation of the π constant. You can evaluate the weak scalability of your code by looking at the time spent in doing the whole computation for growing size of processor numbers and samples.

We can compute the efficiency of the code by measuring:

$$E = t(1)/t(N) \in [0, 1]$$

where

- $t(1)$ is the amount of time to complete a work unit with 1 processing element,
- $t(N)$ is the amount of time to complete N of the same work units with N processing elements.

Further modifications

For the derivative program:

- In every case the function `void firstderiv1Dp_vec` wants to exchange information between two adjacent processes, i.e., every process wants to “swap” is halo with its adjacent process. We can rewrite the whole function by using the `MPI_Sendrecv_replace` point-to-point communication routine.
- We can rewrite the entire program in an “embarrassing parallel” way, if every process has access to f, and are assuming that all the interval are partitioned the same way, by using the knowledge of our `rank` we can compute what are the boundary elements at the previous and following process. Thus, no communication at all!

For the π programs,

- Make a graph of the timings to evaluate the weak scaling efficiency.

Table of Contents

A Queue Manager

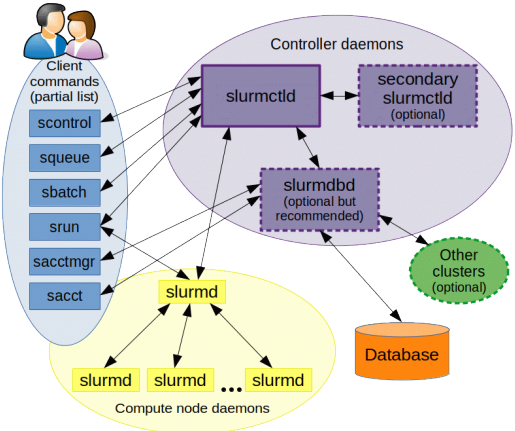
*"Slurm is an **open source**, **fault-tolerant**, and highly scalable cluster management and **job scheduling system** for large and small Linux clusters*[1]*."*

- It **allocates** exclusive and/or non-exclusive **access to resources** (compute nodes) to users for some duration of time so they can perform work;
- It provides a **framework** for **starting**, **executing**, and **monitoring work** (normally a parallel job) on the set of allocated nodes;
- It **arbitrates contention** for resources by managing a queue of pending work.

[1] https://slurm.schedmd.com/quickstart.html

The *Slurm* architecture

2 A Queue Manager

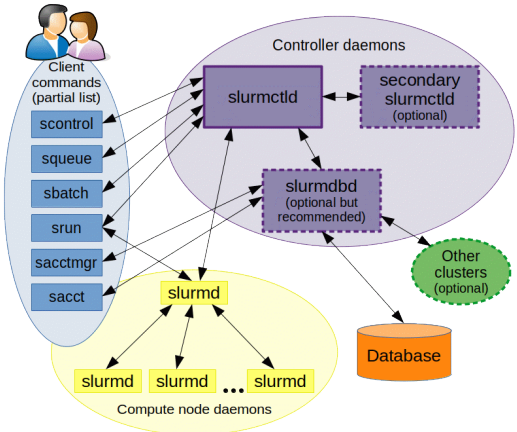


Slurm consists of a

- `slurmd` daemon running on each compute node and
- a central `slurmctld` daemon running on a management node (there may be fail-over twins, but not in our case. . .).

The *Slurm* architecture

2 A Queue Manager



Slurm consists of a

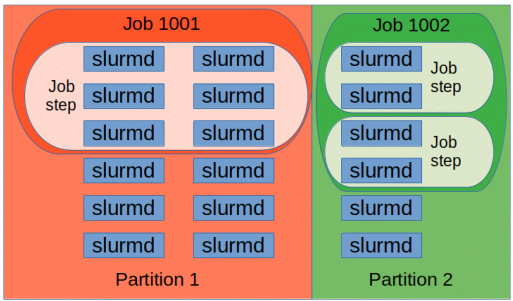
- `slurmd` daemon running on each compute node and
- a central `slurmctld` daemon running on a management node (there may be fail-over twins, but not in our case. . .).

Daemon

A **daemon** is a service process that runs in the background and supervises the system or provides functionality to other processes.

What does Slurm control?

2 A Queue Manager



Entities managed by Slurm daemons include:

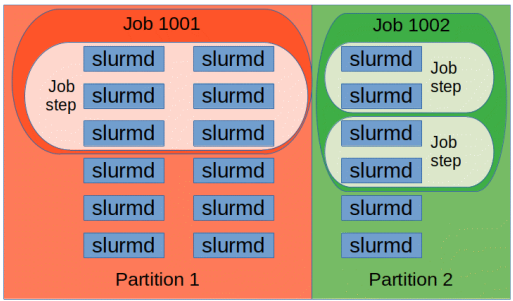
nodes the compute resource in Slurm,

partitions which group nodes into logical (possibly overlapping) sets,

jobs allocations of resources assigned to a user for a specified amount of time.

What does Slurm control?

2 A Queue Manager



Entities managed by Slurm daemons include:

nodes the compute resource in Slurm,

partitions which group nodes into logical (possibly overlapping) sets,

jobs allocations of resources assigned to a user for a specified amount of time.

More generally, **job steps**, which are sets of (possibly parallel) tasks within a job.

What configuration do we have?

In our case the system is configured as having

`steffe0` running the `slurmctld` daemon,

`steffe[1-19]` running the `slurmd` daemon.

For the moment we also have a **single partition** called “`production`”, the **partition** can be considered a **job queue**, usually coming with an assortment of constraints such as job size limit, job time limit, users permitted to use it, *etc.*

What configuration do we have?

In our case the system is configured as having

`steffe0` running the `slurmctld` daemon,

`steffe[1-19]` running the `slurmd` daemon.

For the moment we also have a **single partition** called "`production`", the **partition** can be considered a **job queue**, usually coming with an assortment of constraints such as job size limit, job time limit, users permitted to use it, *etc.*

To discover information on the system we can run the command `sinfo`:

```
fdurastante@steffe0:~$ sinfo
PARTITION   AVAIL  TIMELIMIT  NODES  STATE NODELIST
production*    up   infinite     19   idle steffe[1-19]
```

What configuration do we have?

Further information on the available partition could be obtained by doing:

```
fdurastante@steffe0:~$ scontrol show partition
PartitionName=production
AllowGroups=ALL AllowAccounts=ALL AllowQos=ALL
AllocNodes=ALL Default=YES QoS=N/A
DefaultTime=NONE DisableRootJobs=NO ExclusiveUser=NO GraceTime=0 Hidden=NO
MaxNodes=UNLIMITED MaxTime=UNLIMITED MinNodes=0 LLN=NO MaxCPUsPerNode=UNLIMITED
Nodes=steffe[1-19]
PriorityJobFactor=1 PriorityTier=1 RootOnly=NO ReqResv=NO OverSubscribe=NO
OverTimeLimit=NONE PreemptMode=OFF
State=UP TotalCPUs=114 TotalNodes=19 SelectTypeParameters=NONE
JobDefaults=(null)
DefMemPerNode=UNLIMITED MaxMemPerNode=UNLIMITED
```

Running a *job*: the interactive case

- **Interactive jobs** allow a user to interact with applications on the compute nodes.
- With an **interactive job**, you **request time** and **resources** to work on a compute node.

They are mostly used to *spawn* interactive shells on a compute node via the `srun` command, e.g., the following command

```
srun --partition=production --time=00:30:00 --nodes=1 --pty bash -i
```

gives us

- 1 node (`--nodes=1`),
- from the partition `production` (`--nodes=1`),
- for half an our (`--time=00:30:00`),
- running an interactive `bash` shell `--pty bash -i`

Running a *job*: the interactive case

Let us try it:

```
fdurastante@steffe0:~$ srun --partition=production --time=00:30:00
↪  --nodes=1 --pty bash -i
fdurastante@steffe2:~$
```

the code returns us a *shell* on the node `steffe2` and we have it available for half an hour. In our case all nodes are equal, but we may ask for specific properties:

- `--mem=<size>[units]` Specify the real memory required per node. Default units are megabytes. Different units can be specified using the suffix [K—M—G—T]. Default value is **DefMemPerNode** and the maximum value is **MaxMemPerNode**.
- `-c, --cpus-per-task=<ncpus>` Request that `ncpus` be allocated per process. This may be useful if the **job is multithreaded** and requires more than one CPU per task for optimal performance,

Running a *job*: the interactive case

- `--ntasks=<number>` Specify the **number of tasks to run**. Request that `srun` allocate resources for `ntasks` tasks.
  The **default** is **one task per node**, but note that the `--cpus-per-task` option will change this default.
- `--ntasks-per-node=<ntasks>` Request that `ntasks` be **invoked on each node**. If used with the `--ntasks` option, the –ntasks option will take precedence and the `--ntasks-per-node` will be treated as a maximum count of tasks per node. Meant to be used with the `--nodes` option. This is related to `--cpus-per-task=ncpus`, but *does not require knowledge of the actual number of cpus on each node*.

Running a *job*: the interactive case

- `--exclusive[={user|mcs}]` This option applies to job and job step allocations, and has two slightly different meanings for each one.
  When **used to initiate a job**, the job allocation cannot share nodes with other running jobs (or just other users with the `=user` option or `=mcs` option).
  If user/mcs are not specified (i.e. the job allocation can not share nodes with other running jobs), the job is allocated all CPUs and GRES on all nodes in the allocation, but is only allocated as much memory as it requested.

Use cases for interactive shells are, *e.g.*,

- working with interactive software that requires many resources,
- getting a node just to *compile* a project in parallel,
- *testing* and *debugging*.

Running a *job*: the *batched* case

In a general setting, a **batch script** is an *unformatted script file* which contains *multiple commands* to achieve a certain task.

Commands are executed by command line interpreter that for Slurm is called `sbatch`.

- `sbatch` exits immediately after the script is successfully transferred to the Slurm controller and assigned a Slurm job ID.
- The batch script is **not necessarily granted resources immediately**, it may sit in the **queue of pending jobs** for some time before its required resources become available.

Running a *job*: the *batched* case

Let us start from **an example**, we want to run our `midpointintegral.c` code on our machine, thus we create a bash script called `run.sh` (e.g., `touch run.sh` and then our editor, or whatever editor we like `vim run.sh`.).

The **simplest script** we can write is:

```
#!/bin/bash
#SBATCH -n 10
#SBATCH --time=12:00:00
mpirun ./midpointintegral 100
```

Then we can **put it into the queue** by doing

```
sbatch run.sh
```

that will answer something like: `Submitted batch job 29`.

Running a *job*: the *batched* case

If we look into the folder where we have launched the script we will find a file called `slurm-29.out` containing:

```
fdurastante@steffe0:~/simpletests$ more slurm-29.out
pi is approximately 3.1415927369231262, Error is 8.333333e-08
```

Running a *job*: the *batched* case

If we look into the folder where we have launched the script we will find a file called `slurm-29.out` containing:

```
fdurastante@steffe0:~/simpletests$ more slurm-29.out
pi is approximately 3.1415927369231262, Error is 8.333333e-08
```

What did we do?

- We have run the program `midpointintegral` on `-n 10` tasks,
- this means that we have **used 2 nodes** of our cluster (each node has only 6 CPUs),
- we have also told the machine that the upper-bound time for our execution was 12 minutes (we finished way earlier. . .).

Running a *job*: the *batched* case

What about the queue?

If we run again the `sbatch` command followed by the `squeue` we can visualize it:

```
fdurastante@steffe0:~/simpletests$ sbatch run.sh; squeue
Submitted batch job 30
JOBID  PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
   30  productio   run.sh fdurasta  R       0:00      2 steffe[2-3]
```

The `squeue` command gives us all the **information** relevant to the **queued jobs**.

Running a *job*: the *batched* case

What about the queue?
If we run again the `sbatch` command followed by the `squeue` we can visualize it:

```
fdurastante@steffe0:~/simpletests$ sbatch run.sh; squeue
Submitted batch job 30
JOBID  PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
   30  productio   run.sh fdurasta  R       0:00      2 steffe[2-3]
```

The `squeue` command gives us all the **information** relevant to the **queued jobs**.

Before investigating it further, let us **go back** to the options we can add to our **batch script**.

Running a *job*: the *batched* case

The options overlap with the ones we have seen for the `srun` command.

- *#SBATCH --job-name=<jobname>*
  Specify a name for the job allocation. The specified name will appear along with the job id number when querying running jobs on the system.
- *#SBATCH --mem=<size>[units]*
- *#SBATCH -t, --time=<time>*
  Set a limit on the total run time of the job allocation. If the requested time limit exceeds the partition's time limit, the job will be left in a PENDING state (possibly indefinitely). The default time limit is the partition's default time limit. When the time limit is reached, each task in each job step is sent SIGTERM followed by SIGKILL. Acceptable time formats include "minutes", "minutes:seconds",

"hours:minutes:seconds", "days-hours", "days-hours:minutes" and "days-hours:minutes:seconds".

- *#SBATCH --nodelist=<node_name_list>*
  Request a specific list of hosts. The job will contain all of these hosts and possibly additional hosts as needed to satisfy resource requirements. The list may be specified as a comma-separated list of hosts, a range of hosts (`steffe[1-5,7,...]` for example), or a filename. The host list will be assumed to be a filename if it contains a / character.
- *#SBATCH-p, --partition=<partition_names>*
  Request a specific partition for the resource allocation. If not specified, the default behavior is to allow the slurm controller to select the default partition as designated by the system administrator.

Running a *job*: the *batched* case

- *#SBATCH -N, --nodes=<minnodes>[-maxnodes]|<size_string>*
  Request that a minimum of `minnodes` nodes be allocated to this job. A maximum node count may also be specified with `maxnodes`. If only one number is specified, this is used as both the minimum and maximum node count.
- *#SBATCH -n, --ntasks=<number>*
  `sbatch` does not launch tasks, it requests an allocation of resources and submits a batch script. The default is one task per node, but note that the `--cpus-per-task` option will change this default.
- *#SBATCH --cpus-per-task=<ncpus>*
  Advise the Slurm controller that ensuing job steps will require `ncpus` number of processors per task. Without this option, the controller will just try to allocate one processor per task.

Running a *job*: the *batched* case

- *#SBATCH --ntasks-per-node=<ntasks>*
  Request that ntasks be invoked on each node. If used with the -ntasks option, the `--ntasks` option will take precedence and the `--ntasks-per-node` will be treated as a maximum count of tasks per node. Meant to be used with the `--nodes` option.
- *#SBATCH --exclusive*
- *#SBATCH --input=<filename_pattern>*
  Instruct Slurm to connect the batch script's standard input directly to the file name specified in the `filename pattern`.
- *#SBATCH -o, --output=<filename_pattern>*
  Instruct Slurm to connect the batch script's standard output directly to the file name specified in the `filename pattern`.

Running a *job*: the *batched* case

`sbatch` allows commands that have the **filename argument** to contain one or more substitution symbols usually preceded by "%" and followed by a letter:

Substitution	Symbol
`\\`	Do not process any of the substitution symbols.
`%%`	The character "%"
`%j`	jobid of the running job.
`%N`	short hostname. This will create a separate IO file per node.
`%n`	Node identifier for the current job (eg "0" is the first node of the running job) This will create a separate IO file per node.
`%s`	stepid of the running job.
`%u`	Username
`%x`	Job name

Running a *job*: the *batched*

- *#SBATCH --mail-type=<type>*
  Notify user by email when certain event types occur. Valid type values are `NONE`, `BEGIN`, `END`, `FAIL`, `REQUEUE`, `ALL` (equivalent to `BEGIN`, `END`, `FAIL`, `INVALID_DEPEND`, `REQUEUE`, and `STAGE_OUT`), `INVALID_DEPEND` (dependency never satisfied), `STAGE_OUT` (burst buffer stage out and teardown completed), `TIME_LIMIT`, `TIME_LIMIT_90` (reached 90 percent of time limit), `TIME_LIMIT_80` (reached 80 percent of time limit), `TIME_LIMIT_50` (reached 50 percent of time limit). Multiple type values may be specified in a comma separated list. The user to be notified is indicated with `--mail-user`.
- *#SBATCH --mail-user=<user>*
  User to receive email notification of state changes as defined by –mail-type. The default value is the submitting user.

Checking the status and canceling a *job*

To check the *queue* a useful option is passing to the command the `username`:

```
squeue -u username
```

To have more information we can pass the flag:

```
squeue --long
```

Another useful command is the `scontrol` command, it can be used to check that the controller is UP:

```
scontrol ping
Slurmctld(primary) at steffe0 is UP
```

and to get further information on a *job*

```
scontrol show jobid
```

To *cancel a job* you can use: `scancel jobid`.

Available online resources

The Slurm configuration may have kinks that depends on the machine that is running it. . . so always check the relevant documentation, see, e.g., the documentation for

Marconi-100

Other two general resources are:

- https://slurm.schedmd.com/tutorials.html,
- https://slurm.schedmd.com/pdfs/summary.pdf (this is a `.pdf` file).

Calcolo Parallelo dall'Infrastruttura alla Matematica

Thank you for listening!

Any questions?