Stephen Nicholas Swatman
Version `e25f274*`

DAS Tutorial

The *Distributed ASCI Supercomputer* (DAS) is a series of medium-scale compute clusters designed for research in computer science. Unline many supercomputers which focus on how computing can be used to support domain science, DAS is designed to study the behaviour of computer systems in their own right. The DAS systems are federated compute clusters, with nodes hosted at *Vrije Univeriteit Amsterdam*, *Leiden University*, *University of Amsterdam*, *Delft University of Technology*, and *ASTRON*. At the time of writing, two DAS systems are active: DAS-5 and DAS-6. This tutorial serves to introduce you to these systems (which operate in a very similar fashion); most of the examples will be run on DAS-6, but the commands should also work on DAS-5.

Thoughout this tutorial we will assume basic knowledge of tools like bash and SSH. If you need a refresher on these topics, the internet has many great resources on them. Please note that this document was written by a happy DAS user, but not by one of the DAS admins. You should always consider the information on the official DAS website to be definitive. This document is also not definitive documentation of SLURM; that can be found here.

Notes on Acknowledgements

It is important that researchers who use the DAS systems acknowledge this in their papers and talks. In order to do so, please reference the paper *"A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term"* with DOI `10.1109/MC.2016.127`. You can also use following BibTeX entry:

```
@article{DAS,
  author = {H. Bal and D. Epema and C. de Laat and R. van Nieuwpoort and J.
Romein and F. Seinstra and C. Snoek and H. Wijshoff},
  title = {A Medium-Scale Distributed System for Computer Science Research:
Infrastructure for the Long Term},
  journal = {Computer},
  year = {2016},
  volume = {49},
  number = {05},
  issn = {1558-0814},
  pages = {54-63},
```

```
  doi = {10.1109/MC.2016.127},
  publisher = {IEEE Computer Society},
  address = {Los Alamitos, CA, USA},
  month = {may}
}
```

Accessing DAS

In order to access DAS, you will need an account. Accounts can be requested by emailing the DAS administrators, or an account may be provided for you if you need access to DAS in the context of, for example, a course at an affiliated university. DAS accounts are specific to iterations of DAS, but are usable across the partitions. In other words, a DAS-5 account will work on both the VU and UvA partitions of DAS-5, but it will not work on DAS-6. The DAS clusters are accessible only from institute networks of through proxies. For VU users, you can use the `ssh.data.vu.nl` machine as a proxy jump, LIACS users can use `ssh.liacs.nl`, and UvA users can use UvA-VPN.

Let's now connect to a DAS machine for the first time. We will connect to the VU node for the DAS-6 cluster first, which is accessible through `fs0.das6.cs.vu.nl`. We use the `ssh` command to do so. Please insert your own username in the following command:

```
$ ssh sswatman@fs0.das6.cs.vu.nl
```

The other partitions are available as `fs1.das6.liacs.nl`, `fs2.das6.science.uva.nl`, `fs3.das6.tudelft.nl`, `fs4.das6.science.uva.nl`, and `fs5.das6.astron.nl`. The DAS-5 hostnames are similar, replacing `das6` by `das5`. Unless you have a particularly good memory, it may be useful to add the DAS hosts to your SSH configuration, usually located in `~/.ssh/config`. If you are looking to use DAS-5, you can add the following hosts to this file (replacing my username with yours, of course):

```
$ cat ~/.ssh/config
# DAS-5 hosts.
Host das5-vu
    Hostname fs0.das5.cs.vu.nl
    User sswatman

Host das5-leiden
    Hostname fs1.das5.liacs.nl
    User sswatman

Host das5-uva
    Hostname fs2.das5.science.uva.nl
    User sswatman

Host das5-delft
    Hostname fs3.das5.tudelft.nl
    User sswatman
```

```
Host das5-mn
    Hostname fs4.das5.science.uva.nl
    User sswatman

Host das5-astron
    Hostname fs5.das5.astron.nl
    User sswatman
```

If you do this, you will now be able to access the DAS-5 nodes using the short-form syntax, which will also automatically plug in the right username:

```
$ ssh das5-vu
```

If you intend to use the DAS-6, you can use the following nodes in your configuration:

```
$ cat ~/.ssh/config
# DAS-6 hosts.
Host das6-vu
    Hostname fs0.das6.cs.vu.nl
    User sswatman

Host das6-leiden
    Hostname fs1.das6.liacs.nl
    User sswatman

Host das6-uva
    Hostname fs2.das6.science.uva.nl
    User sswatman

Host das6-delft
    Hostname fs3.das6.tudelft.nl
    User sswatman

Host das6-ai
    Hostname fs4.das6.science.uva.nl
    User sswatman

Host das6-astron
    Hostname fs5.das6.astron.nl
    User sswatman
```

Tip: If you are using a proxy machine (e.g. `ssh.data.vu.nl`) to access the DAS, you can add this to your SSH configuration using the `ProxyJump` attribute. For example:

```
Host vu
  Hostname ssh.data.vu.nl
  User sswatman
```

```
Host das6-vu
  Hostname fs0.das6.cs.vu.nl
  User sswatman
  ProxyJump vu
```

Now, the `ssh das6-vu` command will automatically jump through `ssh.data.vu.nl`, saving you time and effort!

If, like me, you are especially lazy and want to remember neither the host names nor your own password, the DAS nodes support public key authentication. If you have a public-private key pair (usually `~/.ssh/id_rsa.pub`, which you can create using `ssh-keygen`), you can copy this to the DAS partitions (sadly, you need to do this separately) using the following command:

```
$ ssh-copy-id das6-vu
```

After you fill out your password one more time, you should now be able to log in to the DAS machines without needing to enter your password.

From this point forward, any commands in this tutorial will assume we are logged into the DAS systems, unless otherwise noted. Connecting to DAS will connect us to the so-called *head nodes*, which serve as the platform from which we submit jobs to worker nodes. It is worth noting that each partition has its own head node, and the VU head node is in charge of managing passwords; if you want to change your password, you should always do some from the VU head node. You should never need to SSH into any of the worker nodes. Many people rely on the cluster management software to ensure they have exclusive ownership of a given machine so they can gather publication-quality results without noise from other users, so please do not manually connect to the worker nodes.

Warning: The DAS head nodes are **not** meant for compute jobs; they should only be used for filesystem management, submission of jobs to worker nodes, compilation, and other short tasks. Using the headnodes for intensive computation degrades the performance of the cluster for all DAS users and risks getting your account blocked.

Cluster Information

The modern DAS systems are managed using the *SLURM* cluster management software. This allows users to submit jobs to the cluster (we will get to this later), and SLURM will manage those jobs for them. SLURM handles the scheduling, and ensures that jobs get access to the resources they need. In this section, we will observe the operation of a SLURM cluster. Let's start by asking the SLURM manager about the cluster status using the `sinfo` command:

```
$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
defq*        up   infinite      1 drain* node021
defq*        up   infinite      1  down* node010
defq*        up   infinite      1  drain node024
```

```
defq*        up   infinite     21  alloc node[001-005...]
defq*        up   infinite      7   idle node[006-007...]
defq*        up   infinite      1   down node027
fatq         up   infinite      1  alloc node028
fatq         up   infinite      1   idle node029
```

This gives us some information on the thirty-four nodes of the VU DAS-6 partition. The first column, `PARTITION` indicates the queue through which the nodes can be accessed. Across DAS, you will usually find two queues: `defq` is the default queue which should be used for most work, and `fatq` contains special nodes with very powerful hardware for special applications. You are free to use the so-called fat nodes, but jobs will not be submitted to them by default to ensure that they remain available to people who really need that extra power. We will discuss queues in more detail later. The `AVAIL` column can be ignored. The `TIMELIMIT` column specifies the maximum amount of time jobs can run on each node. The `NODES` column specifies how many nodes are grouped together in each row, and the `STATE` column gives us information about the state of the nodes. Nodes in the `alloc` state are busy processing jobs, nodes in the `idle` state are available for work, and nodes in the `down` or `drain` state are not available for work. Nodes can also be in a `mix` state, indicating they they are partially occupied and could fit additional non-exclusive jobs. The final column shows the node(s) described by that particular row.

By default, SLURM will only give us a small amount of information about each node. We can request additional information by specifying an output format string, such as in the following command:

```
$ sinfo -o "%.40N %.6D %.5P %.11t %.4c %.8z %.30f %.20E %.20G"
NODELIST NODES PARTI  STATE CPUS  S:C:T AVAIL_FEATURES   REASON        GRES
 node021     1 defq* drain*   48 1:24:2        cpunode GPU test       gpu:1
 node010     1 defq*  down*   48 1:24:2  gpunode,A4000 GPU test gpu:A4000:1
...
```

The `CPUS` column will now tell us the total number of CPU threads available on each of the nodes. The `S:C:T` column will tell us how many sockets the machine has, how many cores each socket has, and how many threads each core has. The `AVAIL_FEATURES` column tells us the features available on the machine, and we will make use of this later. The `REASON` column will tell us any additional information about the node state, and the `GRES` column will tell us the "generic consumable resources" available on the node. This, too, will become relevant later.

Tip: If you have a particular favourite `sinfo` output format, you can set it to be the default whenever you run `sinfo` by saving it into the `SINFO_FORMAT` environment variable:

```
$ export SINFO_FORMAT="%.20N %.6D %.5P %.11t %.4c %.8z %.6m %.30f %.10E %.20G"
```

You can also put this in your `~/.bashrc` (or `~/.zshrc`, etc.) to make it more permanent.

Let's now take a look at what other users are using the cluster for. The relevant command for this is `squeue`, which lets us view the job queue. Let's try that now:

```
$ squeue
       JOBID     NAME     USER ST       TIME  NODES NODELIST(REASON)
57624_[5-16]     BiNE   rpe201 PD       0:00      1 (Resources)
57628_[6-16]     BiNE   rpe201 PD       0:00      1 (Priority)
       55765 test-dmg   tyo201  R 6-03:59:07      1 node015
       55766  test-md   tyo201  R 6-03:57:43      1 node001
       56921     bash   ddz500  R 2-22:46:17      1 node030
       57623 prun-job  xouyang  R    3:05:25      1 node005
```

The `JOBID` column tells us the unique identifier of each job in the list. In some cases this is just a number (`55765`), and in other cases it consists of two numbers separated by an underscore (`57624_5`). The former is a simple job, and the latter is a *job array*. We will discuss what this means later. The `NAME` column specifies the name of the job. This can be set by the user, whose username is usefully specified in the `USER` column. The `ST` column gives us the state of the job, which is either running (`R`) or pending (`PD`). The `TIME` column tells us how long a job has been running, and the `NODES` column tells us how many nodes the job is using.

The `NODELIST(REASON)` column will tell us the nodes a given job is using or, if the job is not running, the reason why it is not. Common reasons why a job might be pending are `Resources`, which implies that the job is waiting for certain resources to become available and `Priority`, which means that a job is ready to run, but that starting it would take away resources from a higher priority job which is waiting for some of the resources this job would take. In this particular case, the job array `57628` is ready to run, but starting it would further delay job array `57624` which has a higher priority. SLURM manages job according to their priority, a number which will slowly increase as a job waits to be executed. This ensures that all jobs have a fair chance of running. If you want to get more detailed information about jobs, including their priority, you can use a custom format specifier:

```
$ squeue -o "%.18i %.9P %.30j %.8u %.10T %.10M %.13l %.6D %Q %R"
```

Tip: As with `sinfo`, you can store your favourite format specifier for `squeue` in an environment variable:

```
$ export SQUEUE_FORMAT="%.18i %.9P %.30j %.8u %.10T %.10M %.13l %.6D %Q %R"
```

If you want to truly interrogate SLURM about the state of a job, you can use the following command. Hopefully, you will not need to use this:

```
$ scontrol show jobid -d 57624
```

File Storage

If you want to run real research jobs on the DAS clusters you will no doubt need some files, whether they are software or data. Data on the DAS should be permanently stored on one

of two places. The first is your home directory, in `/home/[username]`. Your home directory is small but aggressively backed up, so it is ideal for small, important data which is not backed up elsewhere. You can store larger amounts of data on your scratch space, located at `/var/scratch/[username]`. Keep in mind that the scratch space is not backed up, so use it only for results that are easily replicated and software which is available from other sources.

Both your home directory and your scratch space are available from the worker nodes as well as the head nodes, but each worker has additional space in `/local/[username]` (you may need to create this directory) which you can use. Please be considerate of your fellow DAS users and clean this memory when you no longer need it, since it is not automatically deleted by default. In general, you should not rely on the persistency of this data between jobs, so use it strictly for temporary data needed by a single job.

Remember that the file systems of the different DAS partitions are fully separate, i.e. you cannot access your data on the VU partition from a node in the Delft partition, and so forth. In most cases, you should be well served using just one of the partitions for your work but in some cases – particularly if you want to study a program across a wide variety of hardware – you may need to copy data between head node file systems.

Submitting Jobs

We now know everything we need to know to get started running jobs on the DAS worker nodes. There are two ways of doing this, and we will discuss the `srun` command first. To run our very first cluster job, let's request a worker node and have it report its own hostname. To do so, we execute the following command:

```
$ srun -- hostname
srun: job 57739 queued and waiting for resources
srun: job 57739 has been allocated resources
node022
```

Great! What has happened here? We passed the `hostname` command on to `srun`, which created a new SLURM job for us with the identifier `57739`. This job briefly waited in a queue to be executed and was then mapped onto a node. The node then executed the `hostname` command, printed its own name, and SLURM passed this back to us on the head node. We can use the same command to run a short compute job which uses the file system:

```
$ cat test.txt
5
1
2
3
5
$ srun -- awk '{sum+=$1;} END{print sum;}' test.txt
srun: job 57740 queued and waiting for resources
srun: job 57740 has been allocated resources
16
```

It is worth convincing yourself that the first command runs on the head node: the head node accesses the file system and prints the contents of a file. We then use `srun` to submit a job to one of the worker nodes which counts up the numbers in that same file and prints them. Adding five numbers is hardly worth using a supercomputer for, of course, but consider this a demonstration of how we can read files. The same works for writing files:

```
$ srun -- touch my_file.txt
srun: job 57741 queued and waiting for resources
srun: job 57741 has been allocated resources
$ ls -la
-rw-rw-r--    1 sswatman sswatman    0 May  7 04:28 my_file.txt
```

In general, SLURM jobs inherit their environment from the shell that launches them, including the working directory. This is convenient because it ensures that – in most cases – if you can access a file or a command from the head note, you will be able to access it from your script as well. Let's try to run some jobs in parallel. In the next example, we will asks SLURM to create four *tasks*, separate instances of the command, which together form a single job:

```
$ srun --ntasks=4 -- hostname
srun: job 57749 queued and waiting for resources
srun: job 57749 has been allocated resources
node022
node022
node022
node022
```

In this case, SLURM decided to schedule all tasks in our job on the same node. This makes sense, because the node in question has 48 threads available and SLURM assumes – by default – that each task requires a single thread. If our tasks can use multiple threads, we might want to tell SLURM to reserve a certain number of threads for each task:

```
$ srun --cpus-per-task=24 --ntasks=4 -- hostname
srun: job 57753 queued and waiting for resources
srun: job 57753 has been allocated resources
node022
node023
node023
node022
```

Notice how SLURM has decided to schedule this job across two nodes, reserving 24 threads (CPUs) for each of the four tasks. We can achieve the same result by requesting SLURM to reserve two nodes, and to balance the jobs across them:

```
$ srun --nodes=2 --ntasks=4 -- hostname
srun: job 57754 queued and waiting for resources
srun: job 57754 has been allocated resources
node022
```

```
node023
node022
node023
```

Alternatively, we could leave out the number of tasks and instead request a number of nodes and a number of tasks per node:

```
$ srun --nodes=2 --ntasks-per-node=2 -- hostname
srun: job 57755 queued and waiting for resources
srun: job 57755 has been allocated resources
node022
node022
node023
node023
```

Long story short, there are a lot of way of asking SLURM for job configurations. What is best for you will depend on your application. It is worth keeping in mind, however, that a SLURM job must always run in one go: all of its tasks must execute simultaneously. This is excellent news if our application needs communication to other nodes (via, say, MPI), but it is bad news if we need a lot of tasks. If you need a large number of tasks which do not need to run at the same time, stick around until we discuss job arrays:

```
$ srun --nodes=1000 -- hostname
srun: Requested partition configuration not available now
```

Let's now see if we can get ourselves a GPU to play with. Since all the GPUs in DAS-6 are made by NVIDIA, we should be able to find them easily in the list of PCI devices. We can cross our fingers and hope we get a GPU by random chance:

```
$ srun -- lspci | grep NVIDIA || echo "No NVIDIA devices found"
srun: job 57759 queued and waiting for resources
srun: job 57759 has been allocated resources
No NVIDIA devices found
```

Sadly, that did not work. Indeed, simply hoping that something will happen is not usually a good strategy in computing. Instead, we can request specific features from the SLURM cluster manager. Let's ask it for a GPU in two different ways:

```
$ srun -C gpunode -- lspci | grep NVIDIA || echo "No NVIDIA devices found"
srun: job 57768 queued and waiting for resources
srun: job 57768 has been allocated resources
41:00.0 VGA compatible controller: NVIDIA Corporation GA104GL [RTX A4000]

$ srun --gres=gpu -- lspci | grep NVIDIA || echo "No NVIDIA devices found"
srun: job 57767 queued and waiting for resources
srun: job 57767 has been allocated resources
41:00.0 VGA compatible controller: NVIDIA Corporation GA104GL [RTX A4000]
```

The good news is that we managed to get access to a GPU (an NVIDIA RTX A4000) in both cases, but why do we need different ways of acquiring these cards? The reason is because these approaches are slightly different, and they are both useful in different scenarios. The `-C` flag request nodes with a specific *feature*. If you remember, we saw these in the `AVAIL_FEATURES` column in the output of `sinfo`. These are flags that are added manually by the DAS admins, and GPUs are a useful kind of feature. When requesting features, SLURM will ensure that that feature is available on every single node that the job runs on. Specific GPU types are also features, so we can use the `-C` flag to request a specific GPU, such as an RTX A6000:

```
$ srun -C A6000 -- lspci | grep NVIDIA || echo "No NVIDIA devices found"
srun: job 57773 queued and waiting for resources
srun: job 57773 has been allocated resources
41:00.0 VGA compatible controller: NVIDIA Corporation GA102GL [RTX A6000]
```

The `--gres` flag specifies general resources, which includes GPUs. Like the `-C` flag, specifying GRES requirements allows us to request specific GPU types, and it also requires the resources to be available on every single node in the job.

```
$ srun --gres=gpu:2 -- lspci | grep NVIDIA || echo "No NVIDIA devices found"
srun: job 57774 queued and waiting for resources
srun: job 57774 has been allocated resources
41:00.0 VGA compatible controller: NVIDIA Corporation GA104GL [RTX A4000]
42:00.0 VGA compatible controller: NVIDIA Corporation GA104GL [RTX A4000]
```

GRES also allows us to request both a specific type, as well as a specific number of GPUs:

```
$ srun --gres=gpu:A2:4 -- lspci | grep NVIDIA || echo "No NVIDIA devices
found"
srun: job 57775 queued and waiting for resources
srun: job 57775 has been allocated resources
41:00.0 3D controller: NVIDIA Corporation Device 25b6
42:00.0 3D controller: NVIDIA Corporation Device 25b6
c5:00.0 3D controller: NVIDIA Corporation Device 25b6
c6:00.0 3D controller: NVIDIA Corporation Device 25b6
```

Warning: On some partitions, using the `-C` flag will make tasks run on GPU nodes, but SLURM will unhelpfully make the GPUs invisible to the command you are running, in case anyone else wants to non-exclusively use the GPUs on that node. In order to prevent this, I recommend you use `--gres` over `-C`.

Finally, let's run a very big compute job. For this one, we will want to use a node with 96 cores. We know that such nodes (`node028` and `node029`) are available:

```
$ sinfo -e -o "%.22N %.5P %.4c %.8z"
              NODELIST PARTI CPUS    S:C:T
```

```
 node[001-027,030-034] defq*   48   1:24:2
         node[028-029]  fatq   96   2:24:2
```

However, when we try to submit our job, we get the following error:

```
$ srun --cpus-per-task=96 --nodes=1 -- hostname
srun: error: CPU count per node can not be satisfied
srun: error: Unable to allocate resources: Requested node configuration is
not available
```

Note that this is *not* SLURM telling us that the nodes are busy; in that case, it would have put our jobs in a queue, and they would eventually execute. No, this is SLURM telling us that the job we requested is simply impossible to execute. But why is this? It's because – by default – DAS jobs are submitted to the `defq` queue, and the nodes which have enough cores to satify our requirements are only allocatable via the `fatq` queue. We must, therefore, specify the queue explicitly:

```
$ srun -p fatq --cpus-per-task=96 --nodes=1 -- hostname
srun: job 57809 queued and waiting for resources
srun: job 57809 has been allocated resources
node029
```

Exclusivity

Depending on which of the DAS partitions you are using, nodes may be – by default – allocated exclusively or not. When you reserve a node exclusively, no other jobs can run on that node at the same time as your job. Non-exclusive jobs can share resources on the same node between then. For example, a node with 10 CPU cores can comfortably have a non-exclusive 6-core job and a non-exclusive 4-core job from a different user running on it at the same time.

Whether or not you want exclusivity or not depends on the kind of research you are doing. If the the desired output of your job is the result of some computation, e.g. you are running a fluid simulation or training a neural network, then you don't need to worry about exclusivity. If the output of your job is some metrics about the computation *itself*, e.g. if you are benchmarking software on a specific GPU, or if you are measuring power consumption, then you should request exclusive access to a node using the `--exclusive` flag. If you are happy to share nodes between your own jobs but not with other users' jobs, you can use `--exclusive=user`.

Batch Files

We now know how to submit jobs to the grid, but passing the commands directly to `srun` is only convenient for very simple commands. If we want to make our jobs more complex (possibly including multiple steps) and more reproducible (meaning we don't need to remember all the flags we pass to `srun`), we will need to learn about SLURM *batch* scripts. Batch scripts are very similar to shell scripts, except that they can carry additional metadata about how we want SLURM to treat our jobs. Let's write our very first batch script, `example.sh`, now:

```
#!/bin/bash

echo "Hello world"
hostname
```

The structure of this file is simple: the first line contains a so-called *shebang*, which indicates the shell to use when intepreting the script. In most cases, `/bin/bash` will do just fine. The remainder of the script contains the commands we want to execute. Batch scripts are submitted to SLURM using the `sbatch` command, which simply takes the name of the script we want to execute. By default, `sbatch` will output to a file rather than to the command line, and batch jobs run asynchronously: the `sbatch` command completes immediately, but this does not mean that the job is actually finished. We may need to wait for it to be allocated and for it to finish running:

```
$ sbatch example.sh
Submitted batch job 58144

# The sbatch command completes immediately
# This does not mean the job is complete!

$ cat slurm-58144.out
Hello world
node004
```

The main advantage of using batch files is that we can encode properties about *how* we want to run jobs into the job itself. This means that we don't need to remember the flags to `srun`, we can simply put them in the script itself. We do this using `SBATCH` comments at the top of the file. Consider the following batch script which runs on a node with an NVIDIA A4000 and has a maximum running time of one minute. Notice also that this allows us to run multiple commands in sequence!

```
#!/bin/bash
#SBATCH -C A4000
#SBATCH -t 0:01:00

hostname
lspci | grep NVIDIA
```

Running this job gives us the expected result:

```
$ sbatch example2.sh
Submitted batch job 65765

$ cat slurm-65765.out
node003
43:00.0 VGA compatible controller: NVIDIA Corporation GA104GL [RTX A4000]
(rev a1)
```

```
43:00.1 Audio device: NVIDIA Corporation GA104 High Definition Audio
Controller (rev a1)
```

To reiterate, the advantage of using batch scripts – from what we have seen so far – is that they allow us to encode SLURM flags into the script itself, and that they allow us to run multiple commands in sequence. Let's extend our script to run on multiple nodes, just like we did with `srun`. We'll consider the following batch file:

```
#!/bin/bash
#SBATCH --nodes=4

hostname
```

Running this script gives the following output:

```
$ sbatch example3.sh
Submitted batch job 65766

$ cat slurm-65766.out
node031
```

Interestingly, we only get the output from a single node. This behaviour differs from what we saw with `srun`: when submitting a job to multiple nodes using `srun`, they *all* run the command. When we submit a job to multiple nodes using `sbatch`, only one of the nodes runs the job. This may seem unintuitive and silly, but it is actually a powerful idea: SLURM is allowing us to carve up the reservation however we please. If we reserve five nodes, we can have three of them run one part of the job and two run another part.

This partitioning is done, confusingly, using `srun`. Thus, `srun` does two different things depending on when and where you run it: if you use `srun` *outside* of an existing SLURM allocation, it will make a new allocation for you and distribute work over it. If you use `srun` *inside* an existing allocation, it allows you to carve up that partition into smaller *steps*. As an example, let's construct a job that reserves four nodes. First, we'll run a single step on each of those nodes; then, in the same allocation, we'll run *multiple* steps on each of the reserved nodes:

```
#!/bin/bash
#SBATCH --nodes=4

echo "The script is executed by $(hostname)"
echo "STEP 1:"
srun --tasks-per-node=1 hostname
echo "STEP 2:"
srun --tasks-per-node=2 hostname
```

Running this code results in the following output:

```
$ sbatch example4.sh
Submitted batch job 65863

$ cat slurm-65863.out
The script is executed by node031
STEP 1:
node031
node033
node032
node034
STEP 2:
node031
node033
node032
node034
node031
node033
node032
node034
```

It is valuable to consider what happened here. The `--nodes=4` flag at the top of the batch file requests a *reservation* of four nodes. In this particular case, we were allocated nodes 31, 32, 33, and 34. As the node is executed, the script is executed a *single* time, by node 31. This node requests the allocation to be partitioned into steps, with each node in the allocation handling one task. Thus, we see four printouts during the first step. Nodes 32, 33, and 34 then return to being idle and node 31 issues a second command, requesting *two* commands per node this time, resulting in eight more print-outs.

Sometimes we want nodes to perform heterogeneous work. SLURM allows us to run `srun` commands in the background and thereby run multiple different partitions at the same time. We use the ampersand shell operator for this: placing `&` at the end of a command returns immediately, even if the command has not finished running. We can issue multiple commands into the background and wait for them using the `wait` command. In this case, we also need to – confusingly – pass the `--exclusive` flag to the `srun` commands; this flag has a different meaning in step allocations (running `srun` inside jobs) than it has in job allocations. Observe the following batch script:

```
#!/bin/bash
#SBATCH --tasks=8

srun --exclusive --ntasks=4 echo "STEP A" &
srun --exclusive --ntasks=2 echo "STEP B" &
srun --exclusive --ntasks=2 echo "STEP C" &
wait
```

And here is what happens when we run it:

```
$ sbatch example5.sh
Submitted batch job 65879

$ cat slurm-65879.out
STEP C
STEP A
STEP A
STEP B
STEP B
STEP C
STEP A
STEP A
```

Notice how the print-outs are out of order: they are being executed in parallel. We can also use steps to partition special resources across tasks. We will now construct a rather odd batch script to demonstrate the control SLURM gives you over job allocation:

```
#!/bin/bash
#SBATCH -C "[A2*3&A4000*1]"
#SBATCH --ntasks=6
#SBATCH -w node020
#SBATCH --nodes=6

srun --exclusive --nodes=1 --ntasks=1 --gres=gpu:A4000:1 echo "This job has
an A4000 GPU" &
srun --exclusive --nodes=2 --ntasks=2 --gres=gpu:A2:1 echo "This job has an
A2 GPU" &
srun --exclusive --nodes=1 --ntasks=1 --gres=gpu:A2:2 echo "This job has two
A2 GPUs" &
srun --exclusive --nodes=2 --ntasks=2 --gres=none echo "This job is a CPU
job" &
wait
```

It is unlikely that you will need to write such complex SLURM batch scripts, but let's go over what this program does. The `[A2*3&A4000*1]` constraint specifies that we want three nodes with *at least* one NVIDIA A2 GPU, as well as one node with an NVIDIA A4000 GPU. Then, we specify that we want six nodes and six tasks. Finally, we specify that we want node 20 specifically. The reason for this is that it the only node with multiple NVIDIA A2 GPUs. While the `-C` syntax allows us to specify that we want nodes with *at least one* A2, we cannot specify that we want nodes with *at least two* A2 GPUs. Including node 20 in the nodelist explicitly ensures that we have access to the right node. Finally, the job issues four steps, requesting the necessary resources for each job.

By default, steps inherit options from the `sbatch` script that invokes them. For example, if your batch script sets `--gres=gpu:1`, this will be automatically inherited by all steps in it. If you want some of your steps not to use a GPU, you can use `--gres=none` on the `srun` commands to override the inherited behaviour.

Job Arrays

The combination of batch scripting an `srun` allows us to run jobs in which we precisely control what happens, in which order, where, and when. This is useful for application in which synchronization is important, e.g. when using MPI to compute things which would not fit in the memory of a single machine. In other cases, our goal is not to do one enormous computation at once but rather to performance many independent computations in any order. It is in these cases that *job arrays* can be more appropriate than single, large allocations.

Job arrays do not require that the entire computation happens simultaneously which eases resource requirements significantly. A compute job that takes twenty nodes might take a while to be allocated on DAS, but twenty jobs that take one node each will be completed much more quickly. Job arrays can be specified using the `-a` flag in `sbatch`. For example, the following batch script creates a job array of four jobs, numbered 0 through 3:

```
#!/bin/bash
#SBATCH -a 0-3
#SBATCH -t 0:01:00

echo "I am job ${SLURM_ARRAY_TASK_ID} on node $(hostname)!"
```

Executing this job gives the following result. Note how the outputs are split between different files!

```
$ sbatch example_array.sh
Submitted batch job 65836

$ cat slurm-65836_0.out
I am job 0 on node node015!

cat slurm-65836_*.out
I am job 0 on node node015!
I am job 1 on node node001!
I am job 2 on node node002!
I am job 3 on node node031!
```

To drive home the utility of job arrays, let's do something that would be impossible to do without job arrays: launch multiple jobs that occupy *the same* node entirely:

```
#!/bin/bash
#SBATCH -a 0-15
#SBATCH -w node001
#SBATCH --tasks-per-node
#SBATCH -t 0:01:00

sleep 10
echo "I am job ${SLURM_ARRAY_TASK_ID} on node $(hostname)!"
```

Because job arrays do not need to run at the same time, this overallocation is not a problem. We can see this in the job queue:

```
$ sbatch example_array2.sh
Submitted batch job 65844

$ squeue
       JOBID PARTITION     USER ST       TIME  NODES NODELIST(REASON)
65844_[8-15]      defq sswatman PD       0:00      1 (Resources)
     65844_7      defq sswatman  R       0:08      1 node001
```

As you can see, one of the jobs in the array (number 7) is current running on the specified node. Jobs 0 through 6 have already completed, and jobs 8 through 15 are waiting to execute.

Cancelling Jobs

Sometimes, we start jobs and realize that we mistyped some important configuration parameter, or we notice a bug in the software. In those cases, please cancel the job to ensure that resources become available – to yourself or to your fellow DAS users – as soon as possible. In order to cancel a job, we simply use the `scancel` command:

```
$ sbatch --wrap="sleep 100000"
Submitted batch job 60421

# Oh no! Let's not waste 100,000 seconds of compute time!

$ scancel 60421
```

It is also possible to retroactively modify properties of jobs using the `scontrol` command. You will rarely need this and you can use this command in many different ways. It's outside of the scope of this tutorial, but you can consult the official SLURM documentation for more information.

Environment Modules

By default, the DAS machines give you access to only a few compilers and libraries. If you're looking to compile something for CUDA devices, you'll soon notice that the `nvcc` compiler is not available:

```
$ nvcc
zsh: command not found: nvcc
```

Don't worry! The CUDA tookit *is* installed, it is simply not enabled by default. You can enable the CUDA toolkit using *environment modules*, which describe exactly and reversibly how to modify your environment in order to make software available to you. As an example, the following will make CUDA available to you on the VU partition on DAS-6:

```
$ module load cuda11.7/toolkit/11.7
```

```
$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Wed_Jun__8_16:49:14_PDT_2022
Cuda compilation tools, release 11.7, V11.7.99
Build cuda_11.7.r11.7/compiler.31442593_0
```

If you are on the VU partition of DAS-5, the invocation is very similar:

```
$ module load cuda11.0/toolkit/11.0.3

$ $ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2020 NVIDIA Corporation
Built on Wed_Jul_22_19:09:09_PDT_2020
Cuda compilation tools, release 11.0, V11.0.221
Build cuda_11.0_bu.TC445_37.28845127_0
```

Installed software differs slightly between DAS-5 and DAS-6. You can always find out which modules are available by using the following command:

```
$ module avail

------------------ /cm/local/modulefiles ------------------
cluster-tools/8.0 etcd/3.1.5        module-git
cmd               flannel/0.7.0     module-info
cmsh              freeipmi/1.5.5    null
cm-upgrade/7.2    gcc/6.3.0         openldap
cm-upgrade/8.0    ipmitool/1.8.18   shared
dot               lua/5.3.4

----------------- /cm/shared/modulefiles ------------------
acml/gcc/64/5.3.1
acml/gcc/fma4/5.3.1
acml/gcc/mp/64/5.3.1
acml/gcc/mp/fma4/5.3.1
...
```

All of these packages can be loaded as desired, and you can even install your own packages in a personal module environment on `/var/scratch/`. If you need any inspiration on how to setup your own personal modules, you can take a look at the `/var/scratch/sswatman/.modules/` directory on VU DAS-6.

You can find out which modules you have loaded using `module list`, and you can unload module at any time using the `module unload` command. For example:

```
$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
```

```
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Wed_Jun__8_16:49:14_PDT_2022
Cuda compilation tools, release 11.7, V11.7.99
Build cuda_11.7.r11.7/compiler.31442593_0

$ module unload cuda11.7/toolkit/11.7

$ nvcc --version
zsh: command not found: nvcc
```

Finally, it is worth noting that virtually all SLURM commands inherit their environment from the invocation site. Thus, loading software on the head node makes it available on the worker nodes as well:

```
$ nvcc --version # On the head node...
zsh: command not found: nvcc

$ srun nvcc --version # On a worker node...
slurmstepd: error: execve(): nvcc: No such file or directory
srun: error: node015: task 0: Exited with exit code 2

$ module load cuda11.7/toolkit/11.7 # On the head node...

$ nvcc --version # On the head node...
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Wed_Jun__8_16:49:14_PDT_2022
Cuda compilation tools, release 11.7, V11.7.99
Build cuda_11.7.r11.7/compiler.31442593_0

$ srun nvcc --version # On a worker node...
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Wed_Jun__8_16:49:14_PDT_2022
Cuda compilation tools, release 11.7, V11.7.99
Build cuda_11.7.r11.7/compiler.31442593_0
```