

Fine-tuning Job Specifications

As HPC system administrators, we often observe that the HPC resources are not optimally (or wisely) used. For example, we regularly notice that several cores on a computing node are not utilised, due to the fact that one sequential program uses only one core on the node. Or users run I/O intensive applications on nodes with "slow" network connections.

Users often tend to run their jobs without specifying specific PBS Job parameters. As such, their job will automatically use the default parameters, which are not necessarily (or rarely) the optimal ones. This can slow down the run time of your application, but also block HPC resources for other users.

Specifying the "optimal" Job Parameters requires some knowledge of your application (e.g., how many parallel threads does my application use, is there a lot of inter-process communication, how much memory does my application need) and also some knowledge about the HPC infrastructure (e.g., what kind of multi-core processors are available, which nodes have InfiniBand).

There are plenty of monitoring tools on Linux available to the user, which are useful to analyse your individual application. The HPC environment as a whole often requires different techniques, metrics and time goals, which are not discussed here. We will focus on tools that can help to optimise your Job Specifications.

Determining the optimal computer resource specifications can be broken down into different parts. The first is actually determining which metrics are needed and then collecting that data from the hosts. Some of the most commonly tracked metrics are CPU usage, memory consumption, network bandwidth, and disk I/O stats. These provide different indications of how well a system is performing, and may indicate where there are potential problems or performance bottlenecks. Once the data have actually been acquired, the second task is analysing the data and adapting your PBS Job Specifications.

Another different task is to monitor the behaviour of an application at run time and detect anomalies or unexpected behaviour. Linux provides a large number of utilities to monitor the performance of its components.

This chapter shows you how to measure:

1. Walltime
2. Memory usage
3. CPU usage
4. Disk (storage) needs

5. Network bottlenecks

First, we allocate a compute node and move to our relevant directory:

```
qsub -I  
cd ~/examples/Fine-tuning-Job-Specifications
```

Specifying Walltime

One of the most important and also easiest parameters to measure is the duration of your program. This information is needed to specify the *walltime*.

The *time* utility **executes** and **times** your application. You can just add the time command in front of your normal command line, including your command line options. After your executable has finished, **time** writes the total time elapsed, the time consumed by system overhead, and the time used to execute your executable to the standard error stream. The calculated times are reported in seconds.

Test the time command:

```
$ time sleep 75  
real 1m15.005s  
user 0m0.001s  
sys 0m0.002s
```

It is a good practice to correctly estimate and specify the run time (duration) of an application. Of course, a margin of 10% to 20% can be taken to be on the safe side.

It is also wise to check the walltime on different compute nodes or to select the "slowest" compute node for your walltime tests. Your estimate should be appropriate in case your application will run on the "slowest" (oldest) compute nodes.

The walltime can be specified in a job scripts as:

```
#PBS -l walltime=3:00:00:00
```

or on the command line

```
qsub -l walltime=3:00:00:00
```

It is recommended to always specify the walltime for a job.

Specifying memory requirements

In many situations, it is useful to monitor the amount of memory an application is using. You

need this information to determine the characteristics of the required compute node, where that application should run on. Estimating the amount of memory an application will use during execution is often non-trivial, especially when one uses third-party software.

Available Memory on the machine

The first point is to be aware of the available free memory in your computer. The *"free"* command displays the total amount of free and used physical and swap memory in the system, as well as the buffers used by the kernel. We also use the options *"-m"* to see the results expressed in Mega-Bytes and the *"-t"* option to get totals.

```
$ free -m -t
```

	total	used	free	shared	buffers	cached
Mem:	16049	4772	11277	0	107	161
-/+ buffers/cache:		4503	11546			
Swap:	16002	4185	11816			
Total:	32052	8957	23094			

Important is to note the total amount of memory available in the machine (i.e., 16 GB in this example) and the amount of used and free memory (i.e., 4.7 GB is used and another 11.2 GB is free here).

It is not a good practice to use swap-space for your computational applications. A lot of "swapping" can increase the execution time of your application tremendously.

On the UGent clusters, there is no swap space available for jobs, you can only use physical memory, even though *"free"* will show swap.

Checking the memory consumption

To monitor the memory consumption of a running application, you can use the *"top"* or the *"htop"* command.

top

provides an ongoing look at processor activity in real time. It displays a listing of the most CPU-intensive tasks on the system, and can provide an interactive interface for manipulating processes. It can sort the tasks by memory usage, CPU usage and run time.

htop

is similar to *top*, but shows the CPU-utilisation for all the CPUs in the machine and allows to scroll the list vertically and horizontally to see all processes and their full command lines.

Setting the memory parameter

Once you gathered a good idea of the overall memory consumption of your application, you can define it in your job script. It is wise to foresee a margin of about 10%.

The maximum amount of physical memory used by the job per node can be specified in a job script as:

```
#PBS -l mem=4gb
```

or on the command line

```
qsub -l mem=4gb
```

If you do not request memory (neither in the job script nor on the command line), the default memory that your job will get access is the proportional share of the total available memory on the node: If you request a full node, all usable memory will be available. If you request N cores on a cluster where nodes have M cores, you will get N/M of the total usable memory on the node. For the number of cores and available memory per cluster, please see our [infrastructure](#), or you can use the [web portal](#), open the desktop app and there you can browse it per cluster and core using the submission form (there is no need to start an actual desktop).

Specifying processors requirements

Users are encouraged to fully utilise all the available cores on a certain compute node. Once the required numbers of cores and nodes are decently specified, it is also good practice to monitor the CPU utilisation on these cores and to make sure that all the assigned nodes are working at full load.

Number of processors

The number of core and nodes that a user shall request fully depends on the architecture of the application. Developers design their applications with a strategy for parallelization in mind. The application can be designed for a certain fixed number or for a configurable number of nodes and cores. It is wise to target a specific set of compute nodes (e.g., Westmere, Harpertown) for your computing work and then to configure your software to nicely fill up all processors on these compute nodes.

The `/proc/cpuinfo` stores info about your CPU architecture like number of CPUs, threads, cores, information about CPU caches, CPU family, model and much more. So, if you want to detect how many cores are available on a specific machine:

```
$ less /proc/cpuinfo
processor          : 0
```

```
vendor_id      : GenuineIntel
cpu family     : 6
model          : 23
model name     : Intel(R) Xeon(R) CPU E5420 @ 2.50GHz
stepping       : 10
cpu MHz        : 2500.088
cache size     : 6144 KB
...
```

Or if you want to see it in a more readable format, execute:

```
$ grep processor /proc/cpuinfo
processor : 0
processor : 1
processor : 2
processor : 3
processor : 4
processor : 5
processor : 6
processor : 7
```



Note

Unless you want information of the login nodes, you'll have to issue these commands on one of the workernodes. This is most easily achieved in an interactive job, see [the chapter on Running interactive jobs](#).

In order to specify the number of nodes and the number of processors per node in your job script, use:

```
#PBS -l nodes=N:ppn=M
```

or with equivalent parameters on the command line

```
qsub -l nodes=N:ppn=M
```

This specifies the number of nodes (nodes=N) and the number of processors per node (ppn=M) that the job should use. PBS treats a processor core as a processor, so a system with eight cores per compute node can have ppn=8 as its maximum ppn request.

You can also use this statement in your job script:

```
#PBS -l nodes=N:ppn=all
```

to request all cores of a node, or

```
#PBS -l nodes=N:ppn=half
```

to request half of them.

Note that unless a job has some inherent parallelism of its own through something like MPI or OpenMP, requesting more than a single processor on a single node is usually wasteful and can impact the job start time.

Monitoring the CPU-utilisation

This could also be monitored with the **htop** command:

```
htop
```

Example output:

```
 1 [|||| 11.0%]  5 [||  3.0%]  9 [||  3.0%] 13 [  0.0%]
 2 [|||||100.0%] 6 [  0.0%] 10 [  0.0%] 14 [  0.0%]
 3 [||  4.9%]  7 [||  9.1%] 11 [  0.0%] 15 [  0.0%]
 4 [||  1.8%]  8 [  0.0%] 12 [  0.0%] 16 [  0.0%]
Mem[|||||||||||||59211/64512MB] Tasks: 323, 932 thr; 2 running
Swp[||||||||||||| 7943/20479MB] Load average: 1.48 1.46 1.27
                                Uptime: 211 days(!), 22:12:58
```

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
22350	vsc00000	20	0	1729M	1071M	704	R	98.0	1.7	27:15.59	bwa index
7703	root	0	-20	10.1G	1289M	70156	S	11.0	2.0	36h10:11	/usr/lpp/mmfs/bin
27905	vsc00000	20	0	123M	2800	1556	R	7.0	0.0	0:17.51	htop

The advantage of htop is that it shows you the cpu utilisation for all processors as well as the details per application. A nice exercise is to start 4 instances of the "cpu_eat" program in 4 different terminals, and inspect the cpu utilisation per processor with monitor and htop.

If **htop** reports that your program is taking 75% CPU on a certain processor, it means that 75% of the samples taken by top found your process active on the CPU. The rest of the time your application was in a wait. (It is important to remember that a CPU is a discrete state machine. It really can be at only 100%, executing an instruction, or at 0%, waiting for something to do. There is no such thing as using 45% of a CPU. The CPU percentage is a function of time.) However, it is likely that your application's rest periods include waiting to be dispatched on a CPU and not on external devices. That part of the wait percentage is then very relevant to understanding your overall CPU usage pattern.

Fine-tuning your executable and/or job script

It is good practice to perform a number of run time stress tests, and to check the CPU

utilisation of your nodes. We (and all other users of the HPC) would appreciate that you use the maximum of the CPU resources that are assigned to you and make sure that there are no CPUs in your node who are not utilised without reasons.

But how can you maximise?

1. Configure your software. (e.g., to exactly use the available amount of processors in a node)
2. Develop your parallel program in a smart way.
3. Demand a specific type of compute node (e.g., Harpertown, Westmere), which have a specific number of cores.
4. Correct your request for CPUs in your job script.

The system load

On top of the CPU utilisation, it is also important to check the **system load**. The system **load** is a measure of the amount of computational work that a computer system performs.

The system load is the number of applications running or waiting to run on the compute node. In a system with for example four CPUs, a load average of 3.61 would indicate that there were, on average, 3.61 processes ready to run, and each one could be scheduled into a CPU.

The load averages differ from CPU percentage in two significant ways:

1. "*load averages*" measure the trend of processes waiting to be run (and not only an instantaneous snapshot, as does CPU percentage); and
2. "*load averages*" include all demand for all resources, e.g., CPU and also I/O and network (and not only how much was active at the time of measurement).

Optimal load

What is the "*optimal load*" rule of thumb?

The load averages tell us whether our physical CPUs are over- or under-utilised. The **point of perfect utilisation**, meaning that the CPUs are always busy and, yet, no process ever waits for one, is **the average matching the number of CPUs**. Your load should not exceed the number of cores available. E.g., if there are four CPUs on a machine and the reported one-minute load average is 4.00, the machine has been utilising its processors perfectly for the last 60 seconds. The "100% utilisation" mark is 1.0 on a single-core system, 2.0 on a dual-core, 4.0 on a quad-core, etc. The optimal load shall be between 0.7 and 1.0 per processor.

In general, the intuitive idea of load averages is the higher they rise above the number of processors, the more processes are waiting and doing nothing, and the lower they fall below the number of processors, the more untapped CPU capacity there is.

Load averages do include any processes or threads waiting on I/O, networking, databases or anything else not demanding the CPU. This means that the optimal *number of applications* running on a system at the same time, might be more than one per processor.

The "**optimal number of applications**" running on one machine at the same time depends on the type of the applications that you are running.

1. When you are running **computational intensive applications**, one application per processor will generate the optimal load.
2. For **I/O intensive applications** (e.g., applications which perform a lot of disk-I/O), a higher number of applications can generate the optimal load. While some applications are reading or writing data on disks, the processors can serve other applications.

The optimal number of applications on a machine could be empirically calculated by performing a number of stress tests, whilst checking the highest throughput. There is however no manner in the HPC at the moment to specify the maximum number of applications that shall run per core dynamically. The HPC scheduler will not launch more than one process per core.

The manner how the cores are spread out over CPUs does not matter for what regards the load. Two quad-cores perform similar to four dual-cores, and again perform similar to eight single-cores. It's all eight cores for these purposes.

Monitoring the load

The **load average** represents the average system load over a period of time. It conventionally appears in the form of three numbers, which represent the system load during the last **one**-, **five**-, and **fifteen**-minute periods.

The **uptime** command will show us the average load

```
$ uptime
10:14:05 up 86 days, 12:01, 11 users, load average: 0.60, 0.41, 0.41
```

Now, compile and start a few instances of the "*eat_cpu*" program in the background, and check the effect on the load again:

```
$ gcc -O2 eat_cpu.c -o eat_cpu
$ ./eat_cpu&
$ ./eat_cpu&
$ ./eat_cpu&
$ uptime
10:14:42 up 86 days, 12:02, 11 users, load average: 2.60, 0.93, 0.58
```

You can also read it in the **htop** command.

Fine-tuning your executable and/or job script

It is good practice to perform a number of run time stress tests, and to check the system load of your nodes. We (and all other users of the HPC) would appreciate that you use the maximum of the CPU resources that are assigned to you and make sure that there are no CPUs in your node who are not utilised without reasons.

But how can you maximise?

1. Profile your software to improve its performance.
2. Configure your software (e.g., to exactly use the available amount of processors in a node).
3. Develop your parallel program in a smart way, so that it fully utilises the available processors.
4. Demand a specific type of compute node (e.g., Harpertown, Westmere), which have a specific number of cores.
5. Correct your request for CPUs in your job script.

And then check again.

Checking File sizes & Disk I/O

Monitoring File sizes during execution

Some programs generate intermediate or output files, the size of which may also be a useful metric.

Remember that your available disk space on the HPC online storage is limited, and that you have environment variables which point to these directories available (i.e., `$VSC_DATA`, `$VSC_SCRATCH` and `$VSC_DATA`). On top of those, you can also access some temporary storage (i.e., the `/tmp` directory) on the compute node, which is defined by the `$VSC_SCRATCH_NODE` environment variable.

It is important to be aware of the sizes of the file that will be generated, as the available disk space for each user is limited. We refer to section [How much disk space do I get?](#) on [Quotas](#) to check your quota and tools to find which files consumed the "quota".

Several actions can be taken, to avoid storage problems:

1. Be aware of all the files that are generated by your program. Also check out the hidden files.
2. Check your quota consumption regularly.
3. Clean up your files regularly.
4. First work (i.e., read and write) with your big files in the local `/tmp` directory. Once finished,

you can move your files once to the VSC_DATA directories.

5. Make sure your programs clean up their temporary files after execution.
6. Move your output results to your own computer regularly.
7. Anyone can request more disk space to the HPC staff, but you will have to duly justify your request.

Specifying network requirements

Users can examine their network activities with the `htop` command. When your processors are 100% busy, but you see a lot of red bars and only limited green bars in the `htop` screen, it is mostly an indication that they lose a lot of time with inter-process communication.

Whenever your application utilises a lot of inter-process communication (as is the case in most parallel programs), we strongly recommend to request nodes with an "InfiniBand" network. The InfiniBand is a specialised high bandwidth, low latency network that enables large parallel jobs to run as efficiently as possible.

The parameter to add in your job script would be:

```
#PBS -l ib
```

If for some other reasons, a user is fine with the gigabit Ethernet network, he can specify:

```
#PBS -l gbe
```