Submitting Jobs and Running Programs

Blue Waters Petascale Institute 2016

Week 1, Tuesday (May 31), 10:55-11:40am and 3:00-4:45pm

Author: Aaron Weeden, Shodor

Materials

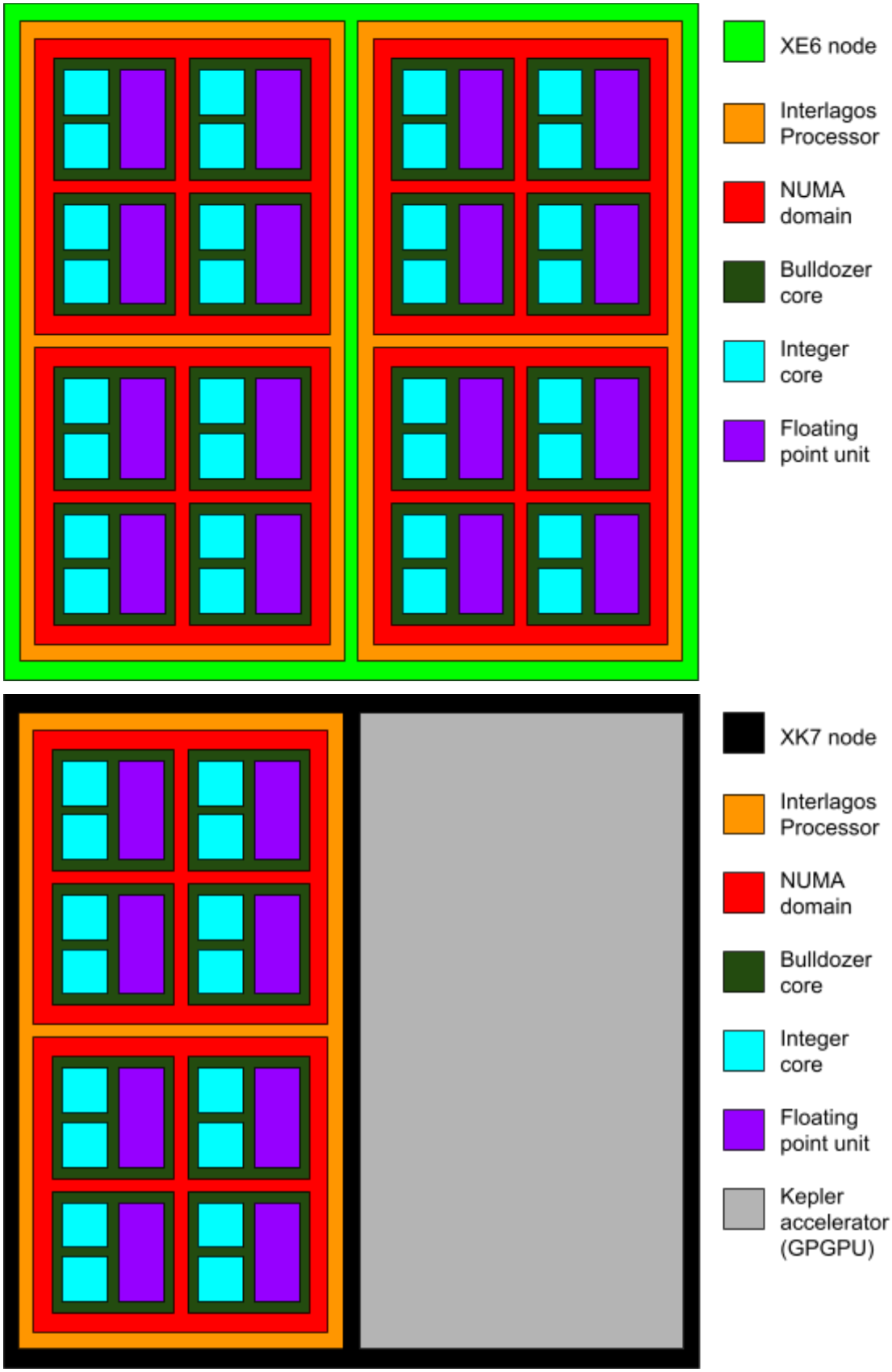
- Example code

https://shodor.org/media/content//petascale/materials/BW2016/day02/day02-submitting-and-running.zip

Introduction

The goal of this section is to introduce how to submit jobs and run programs on Blue Waters. Whenever we want to run a program on Blue Waters, we first have to submit a **job** to the **scheduler**. A job is a specification of the time and space requirements and the commands that need to be executed in order to run a program. The scheduler is an automated system that keeps track of which jobs need to run and allocates time and space on the machine for those jobs to run. Time is measured in hours, minutes, and seconds of real time, whereas space is measured in **nodes** and **processors per node**.

Blue Waters has two types of nodes: XE6 and XK7. These are diagramed below. The primary difference between them is that XE6 nodes have 2 Interlagos processors, while XK7 nodes have a single Interlagos processor in addition to an NVIDIA Kepler accelerator (or **GPGPU**, General-Purpose Graphics Processing Unit, a graphics card that is generally-programmable to do calculations traditionally performed by a CPU in a massively parallel fashion).


XE6 node
Interlagos Processor
NUMA domain
Bulldozer core
Integer core
Floating point unit
XK7 node
Interlagos Processor
NUMA domain
Bulldozer core
Integer core
Floating point unit
Kepler accelerator (GPGPU)

When we submit a job on Blue Waters, we need to tell it how many and what kind of nodes we need, in addition to how many *integer cores* we will be using.

Submitting Jobs

Submitting a job is accomplished using a command named **qsub**. This command allows us to submit two different kinds of jobs: **interactive** jobs and **batch** jobs. Interactive jobs allow us to type commands interactively in order to get our program to run. Batch jobs specify the commands ahead of time, and the system runs them for us automatically.

A typical interactive job submission looks like the following command:

```
qsub
 -I
 -l nodes=<# of nodes>:ppn=<# of integer cores>:<xe or xk>
 -l walltime=<hours>:<minutes>:<seconds><ENTER>
```

In this command, spaces are highlighted in this color, and things to be replaced are in this color. This indicates the enter key should be pressed: **<ENTER>**

Let's break this command down into pieces:

- **qsub** is the command we are running, which allows us to submit a job to the scheduler.
- **-I** is used to indicate that this is an *interactive*, NOT a *batch* job.
- **-l** is used to indicate we are about to specify properties of the job. For example:
    - **nodes=<# of nodes>:ppn=<# of integer cores>:<xe or xk>** lets us specify how many *nodes* we want to use, how many *integer cores per node* we want to use, and whether the type of the nodes should be *xe* (XE6) or *xk* (XK7).
    - **walltime=<hours>:<minutes>:<seconds>** lets us specify the maximum amount of time our job needs to run; after this, the scheduler will terminate our job.

Here is an example for a job that should run on 1 XE node, use all 32 of its integer cores, running for at most 10 minutes:

```
qsub -I -l nodes=1:ppn=32:xe -l walltime=00:10:00<ENTER>
```

Here is an example for a job that should run on 2 XK nodes, using 8 of each node's integer cores, running for at most 5 and a half hours:

```
qsub -I -l nodes=2:ppn=8:xk -l walltime=05:30:00<ENTER>
```

Let's try it now using the command below. Log into Blue Waters, then run this command:

```
qsub -I -l nodes=2:ppn=32:xe -l walltime=00:30:00<ENTER>
```

Once we submit a job, it is placed in a **queue**, where it waits for the scheduler to allow it to start running using the time and space requested. When we run the command to submit a job, the output looks like the following:

```
qsub: waiting for job 4757284.bw to start
```

This shows us the ID of our job, `4757284.bw`. If we want to monitor the status of this job, we can do so by logging into Blue Waters in another window and running the following command:

```
qstat <ID><ENTER>
```

For the example shown above, we would replace `<ID>` with `4757284.bw`. The output of the command will look something like the following:

```
Job ID                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
4757284.bw                 STDIN            aweeden                0 Q normal
```

This shows us a few things: the ID, which we already know is `4757284.bw`, the name of the job, which is `STDIN` because we are using an interactive job that will be running using the *standard input* of the command line, the username of the person who submitted the job (in this case `aweeden`), the amount of time the job has used so far (in this case 0 because the job has not started yet), the *status* of the job (in this case Q because it is *waiting in the queue*. Other job statuses we could see are `R` (*running*) or `C` (*completed*)), and the queue into which the job was submitted (the default is `normal`).

If we forget the ID of our job but still want to see the status of all of our recent jobs, we can instead run this command:

```
qstat -u <username><ENTER>
```

If we want to cancel a job that is in the queue, we can use the following command:

```
qdel <ID><ENTER>
```

Let's practice the **`qdel`** now on the interactive job we just submitted. In the second Blue Waters window (the one where you ran **`qstat`**), run the command below and replace `<ID>` with the ID of the job you submitted:

```
qdel <ID><ENTER>
```

Run qstat to confirm the job now has a status of C, complete:

```
qstat <ID><ENTER>
```

The other window, where you typed the **qsub** command, may still be waiting. You can terminate the job by entering **Ctrl-C**. This will prompt you with the following:

```
Do you wish to terminate the job and exit (y|[n])?
```

Type the following to terminate the job:

```
y<ENTER>
```

Using Screen

It may sometimes be the case that you will lose your network connection to Blue Waters while you are running an interactive job. In order to get back to the job, you would have to log back in to Blue Waters and request a new interactive job. To avoid this, there is a command you can run called **screen**, which will save your current command line session such that if the session crashes, you can resume it later. In order to use **screen** on the Blue Waters login nodes, we first need to identify which login node we are connected to. This will show up in your prompt; the login node's hostname will start with **h2ologin**. Write down the number that appears after it; this is the ID of the login node to which you are currently connected; you will need to know this when you go to resume the screen session. Blue Waters has 4 login nodes, whose IDs range from **0** through **3**.

Once you have written down the ID of the login node, start up a screen by typing the following:

```
screen<ENTER>
```

Then, press the spacebar or the enter key.

So that you recognize the screen when you resume it later, type a message to yourself:

```
echo hello<ENTER>
```

Now, close the window, open a new window, and connect to Blue Waters.

Check your command prompt; if you are now on login node whose ID is different than the one you wrote down, type the following command to connect to that login node, replacing **<ID>** with the number you wrote down (Note: this command will only work if you have token access to BW. Otherwise, you will have to rely on being randomly assigned the correct login node when you SSH to **bwbay**):

```
ssh h2ologin<ID><ENTER>
```

Your command prompt should show that you are now on the correct login node. Type the following command to resume the screen you started earlier:

```
screen -r<ENTER>
```

Confirm the message you sent to yourself earlier appears there; this shows us that the screen has successfully resumed. If we want to detach from the screen but keep it running, we can type the following command (try it now):

```
screen -d<ENTER>
```

If this worked, we will get the message `[remote detached]`. We can resume the screen using the same command we used earlier:

```
screen -r<ENTER>
```

There should now be the message `[15537.pts-127.h2ologin1 detached.]`, though the highlighted part is likely different.

With a screen running, we can now safely request an interactive job without worrying about having to-submit the request if we lose connection to Blue Waters. Enter the following command to request a job running on 2 XE nodes with 32 integer cores per node for a maximum time of 30 minutes:

```
qsub -I -l nodes=2:ppn=32:xe -l walltime=01:45:00<ENTER>
```

While we wait for our interactive job to become active, let's talk about the other way to submit jobs on Blue Waters: using **batch** mode. In batch mode, a job is submitted by specifying the parameters of the job in a file known as a **batch script**. Below is the syntax of a batch script used to request a batch job using the same parameters as the interactive job we just submitted:

```
#!/bin/bash
#PBS -l nodes=2:ppn=32:xe
#PBS -l walltime=01:45:00
```

The first line indicates the location of the shell command used to read this script. Note that the shell we are using is **BASH** which is NOT the same thing as *batch*. **BASH** stands for Bourne-Again Shell and is a *shell*, which is used to execute Unix/Linux commands. **Batch** refers to a specific kind of job that we submit to a scheduler, namely one that is not interactive.

Because the batch job is NOT interactive, once we submit the job, we are not able to execute commands ourselves within the job. Instead, we use the script to run the commands for us. We will return to batch jobs later.

Once our interactive job starts running, we are given an interactive shell (command line interface) in the same window we requested the job, and we will be allowed to type commands to run our program. Note that whereas we requested the job on a **login** node (its hostname starts with **`h2ologin`**), the interactive shell is on a **MOM** node (its hostname starts with **`nid`**). This MOM node is used to control programs on other nodes.

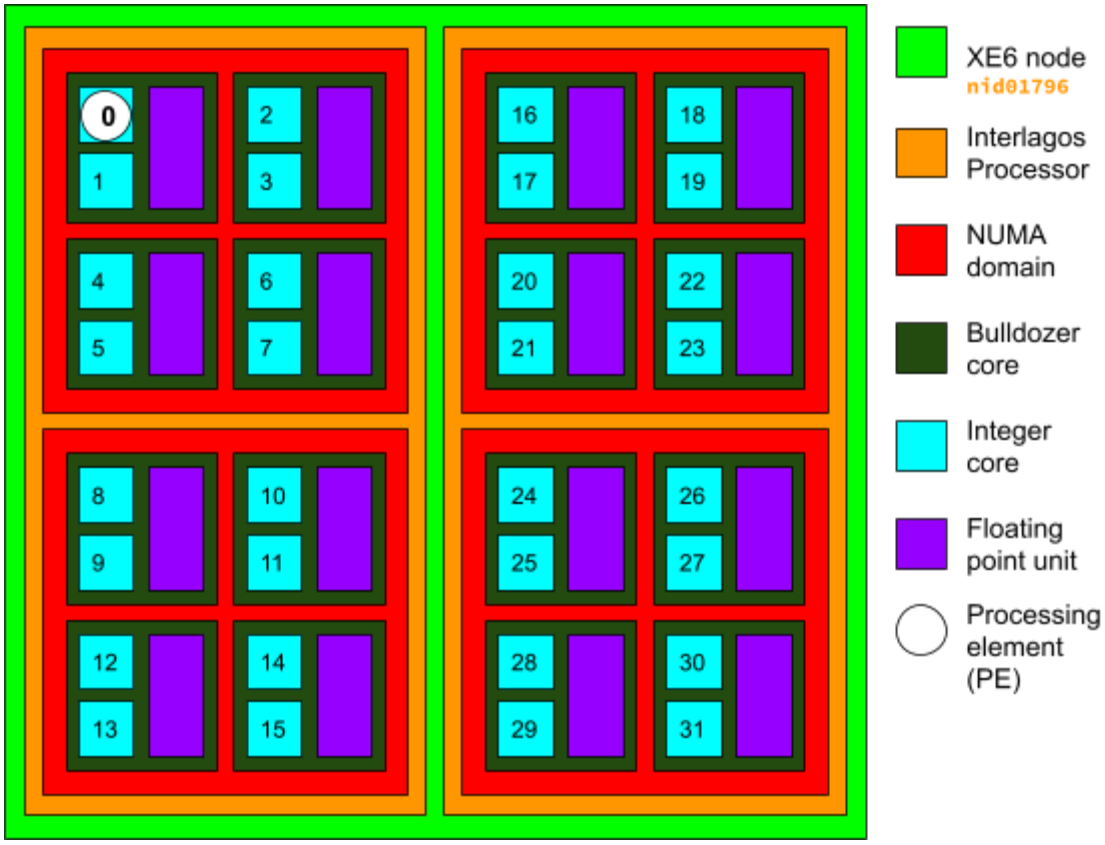
Running Programs

The way we run a program is using the **`aprun`** command. Let's try this on a working program. Copy the **Example code** (https://shodor.org/media/content//petascale/materials/BW2016/day02/day02-submitting-and-running.zip) to Blue Waters. There is an example executable file called **`test.exe`**, which simply reports information about the program you run. Try running the program once to see what it does. **Note:** this must occur from a MOM node; if you try to run it on a login node, it will fail; BW policy is that programs should run on compute nodes (i.e. one starting with **`nid`**), not login nodes (i.e. ones starting with **`h2ologin`**):

```
aprun ./test.exe<ENTER>
PE  0 is on core 0 on node nid01796
```

`aprun` works by assigning pieces of a running program to the integer cores on a node. These pieces of the program are known as **processing elements, or PEs**. The **`test.exe`** program tells each PE (numbered starting at 0) to print out the node and integer core (also numbered starting at 0) on which it is run. You should notice that the ID of the node it prints out (**`nid01796`** in the example shown above) is different than the ID of the node that appears on your prompt. This because the ID on your prompt refers to the MOM node, which controls *other* nodes known as *compute nodes*. The compute nodes contain the processors that actually execute your running program using **`aprun`**.

A diagram of the PE placement we just saw is shown below. PE 0 is running on core 0 of XE node **`nid01796`**.



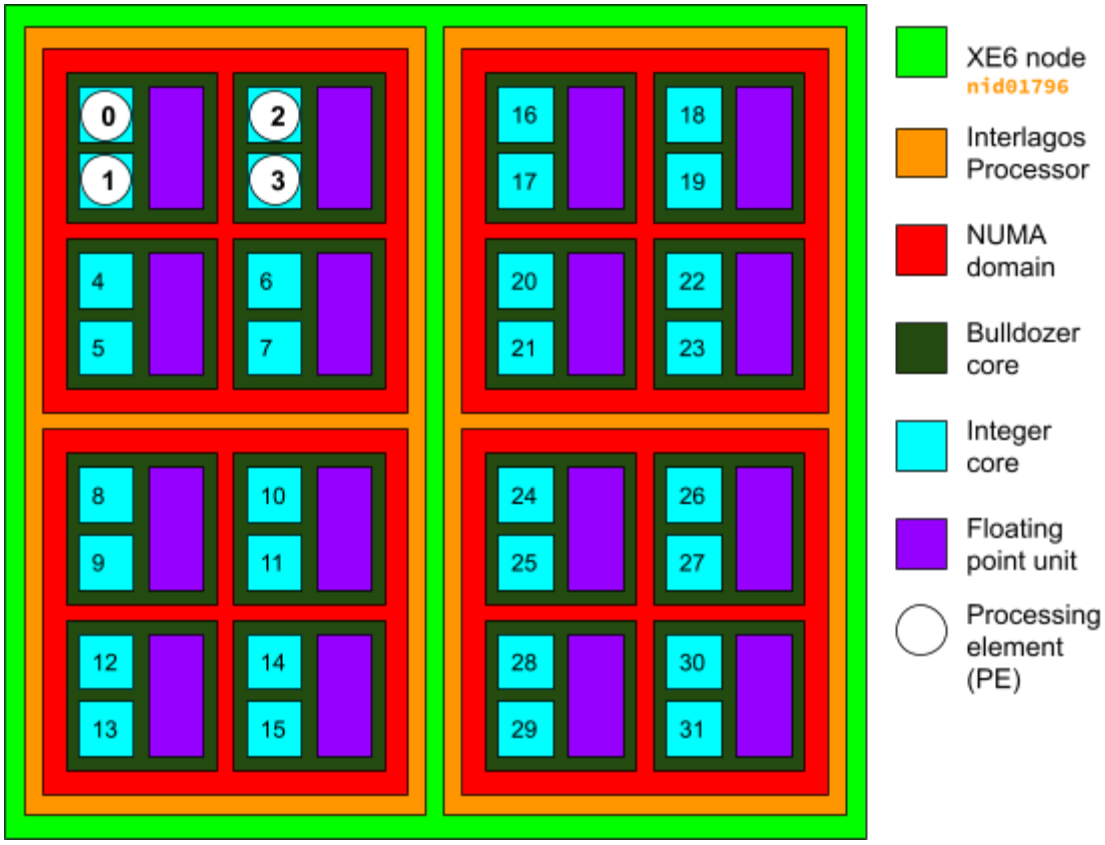
We can change the number of PEs used to run the program using the **-n** option to **aprun**. Let's try it now with 4 PEs:

```
aprun -n 4 ./test.exe<ENTER>
PE  0 is on core 0 on node nid01796
PE  1 is on core 1 on node nid01796
PE  3 is on core 3 on node nid01796
PE  2 is on core 2 on node nid01796
```

Note that the order in which the PEs report their status is random; all PEs are executing the print instruction in parallel, so whichever one happens to execute it first will be the one whose status is shown first. We can sort the output to make it easier to read:

```
aprun -n 4 ./test.exe|sort<ENTER>
PE  0 is on core 0 on node nid01796
PE  1 is on core 1 on node nid01796
PE  2 is on core 2 on node nid01796
PE  3 is on core 3 on node nid01796
```

A diagram of this placement is shown below.

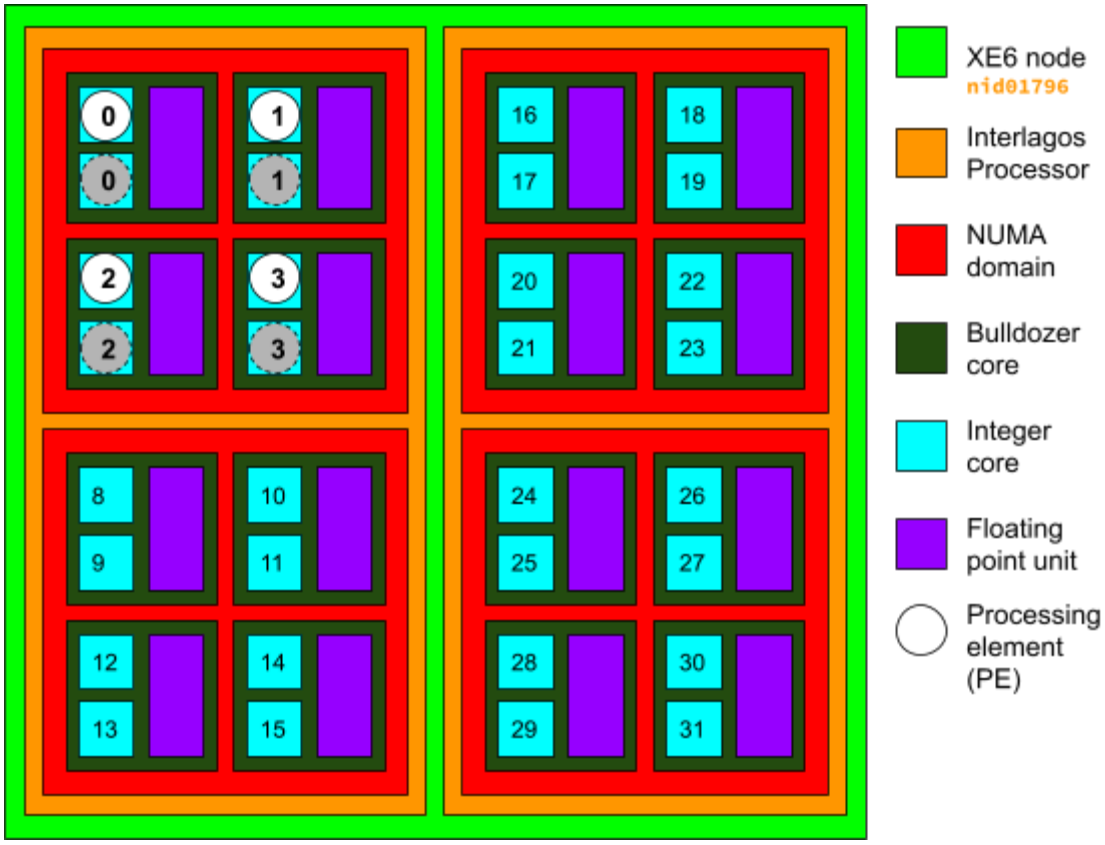


If we use **aprun -n #**, we would expect the first # cores on the node to each be assigned a PE.

We can also use a "depth" parameter, **-d**, to specify the number of cores to assign for each PE. For example, let's run with 4 PEs again but assign each one 2 cores:

```
aprun -n 4 -d 2 ./test.exe|sort<ENTER>
PE  0 is on core 0 on node nid01796
PE  1 is on core 2 on node nid01796
PE  2 is on core 4 on node nid01796
PE  3 is on core 6 on node nid01796
```

A diagram of this placement is shown below.



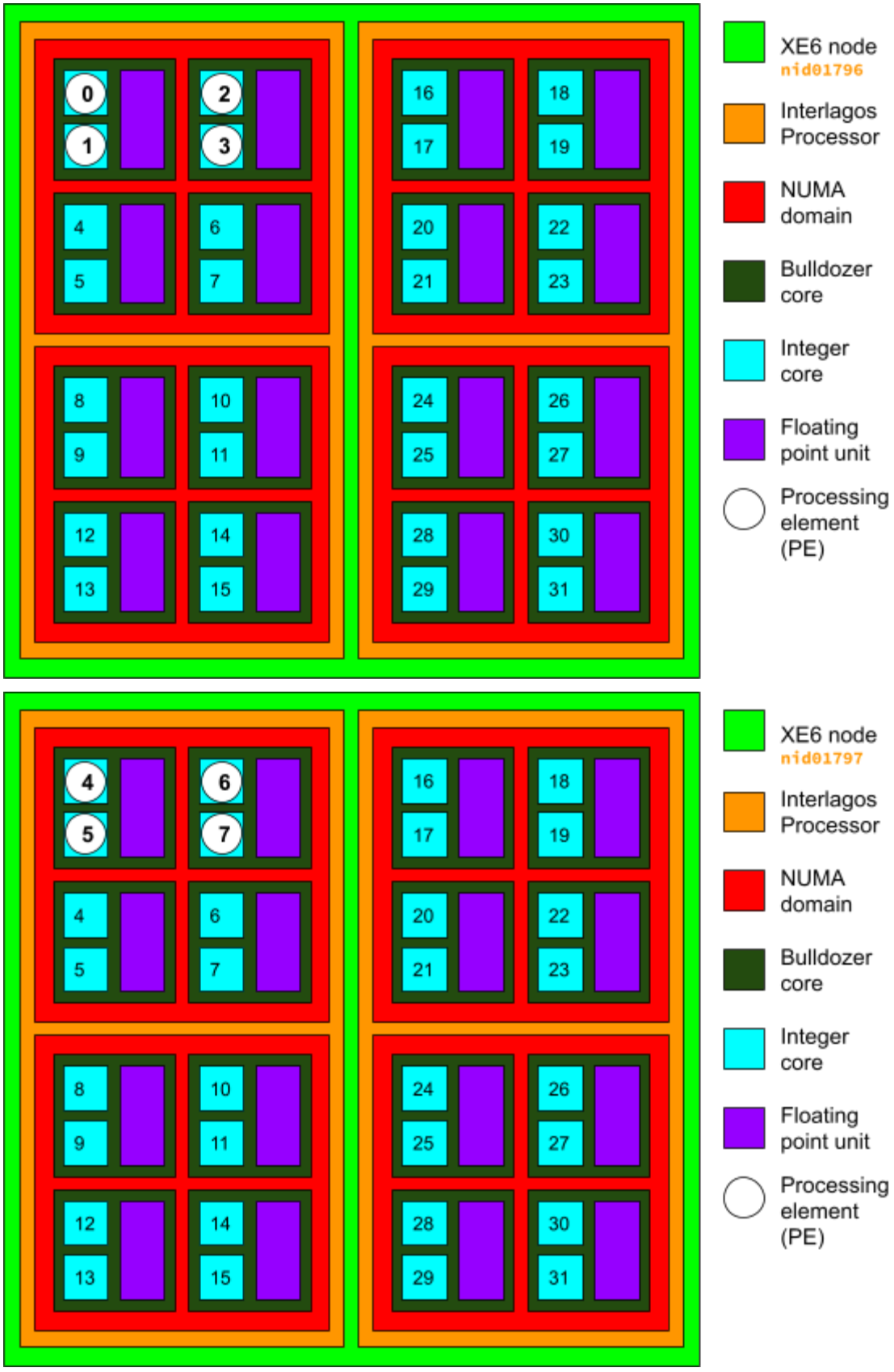
Note that although each PE has access to 2 cores, *it is only running on 1 of those cores*. We can only have each PE actually use its multiple cores if we take advantage of **multi-threading**, which is a topic we will not cover in this session. Thus, for **single-threaded applications**, the **-d** option is useful primarily to spread out the PEs, if this would be useful to a particular application.

Thus far we have only had PEs assigned to a single node. If we want to assign PEs to multiple nodes, we can use the **-N** option, which allows us to specify the number of PEs per node. Let's try running with 8 total PEs but with 4 PEs per node:

```
aprun -n 8 -N 4 ./test.exe|sort<ENTER>
PE  0 is on core 0 on node nid01796
PE  1 is on core 1 on node nid01796
PE  2 is on core 2 on node nid01796
PE  3 is on core 3 on node nid01796
PE  4 is on core 0 on node nid01797
PE  5 is on core 1 on node nid01797
PE  6 is on core 2 on node nid01797
PE  7 is on core 3 on node nid01797
```

Note that now we also have run on the second node, **`nid01797`**.

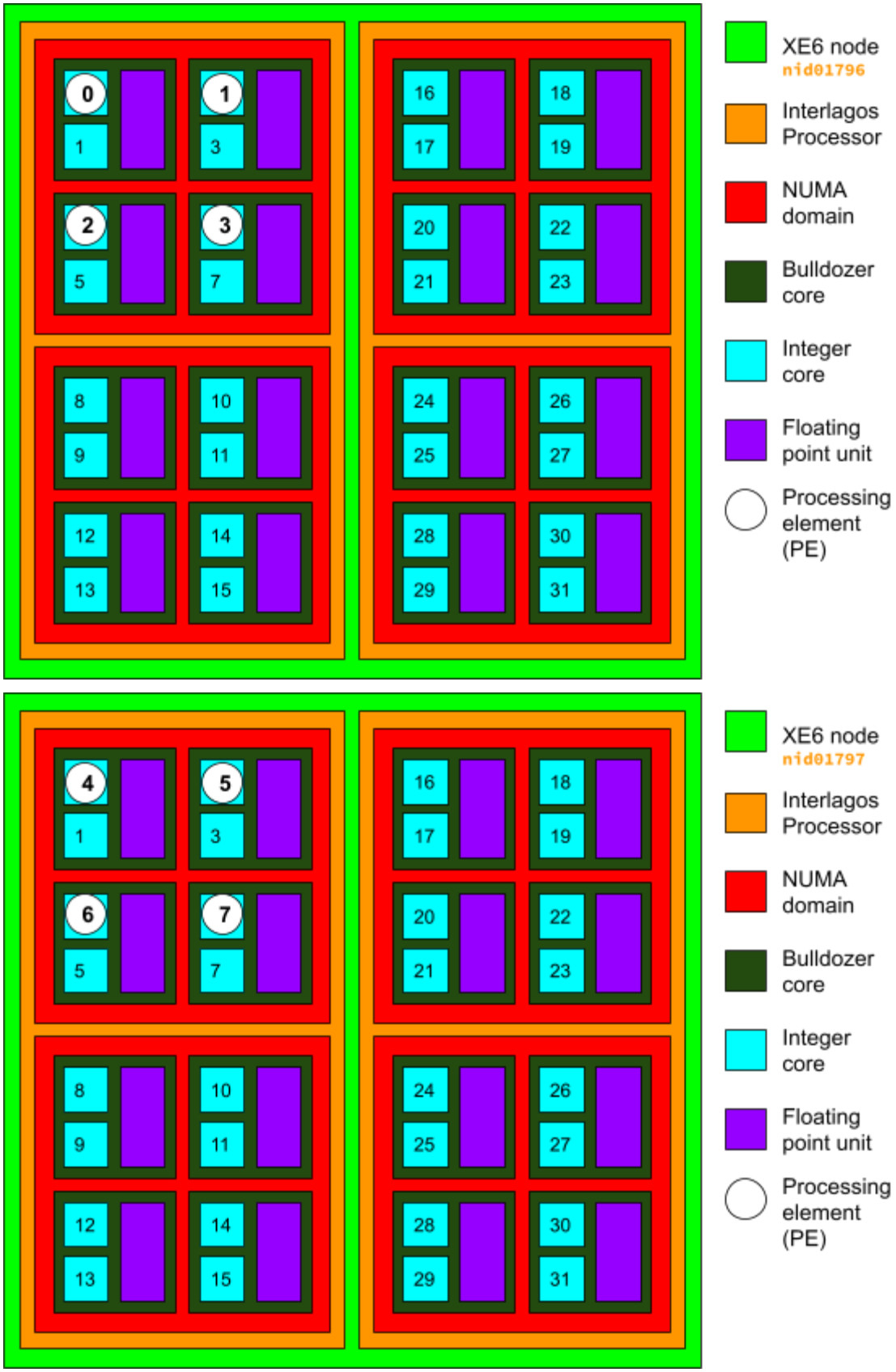
A diagram of this placement is shown below.


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
XE6 node
nid01796
Interlagos Processor
NUMA domain
Bulldozer core
Integer core
Floating point unit
Processing element (PE)
4
5
6
7
XE6 node
nid01797
Interlagos Processor
NUMA domain
Bulldozer core
Integer core
Floating point unit
Processing element (PE)

We can, of course, combine the **-N** and **-d** options, for example running with 8 PEs, 4 PEs per node, and 2 cores per PE:

```
aprun -n 8 -N 4 -d 2 ./test.exe|sort<ENTER>
PE  0 is on core 0 on node nid01796
PE  1 is on core 2 on node nid01796
PE  2 is on core 4 on node nid01796
PE  3 is on core 6 on node nid01796
PE  4 is on core 0 on node nid01797
PE  5 is on core 2 on node nid01797
PE  6 is on core 4 on node nid01797
PE  7 is on core 6 on node nid01797
```

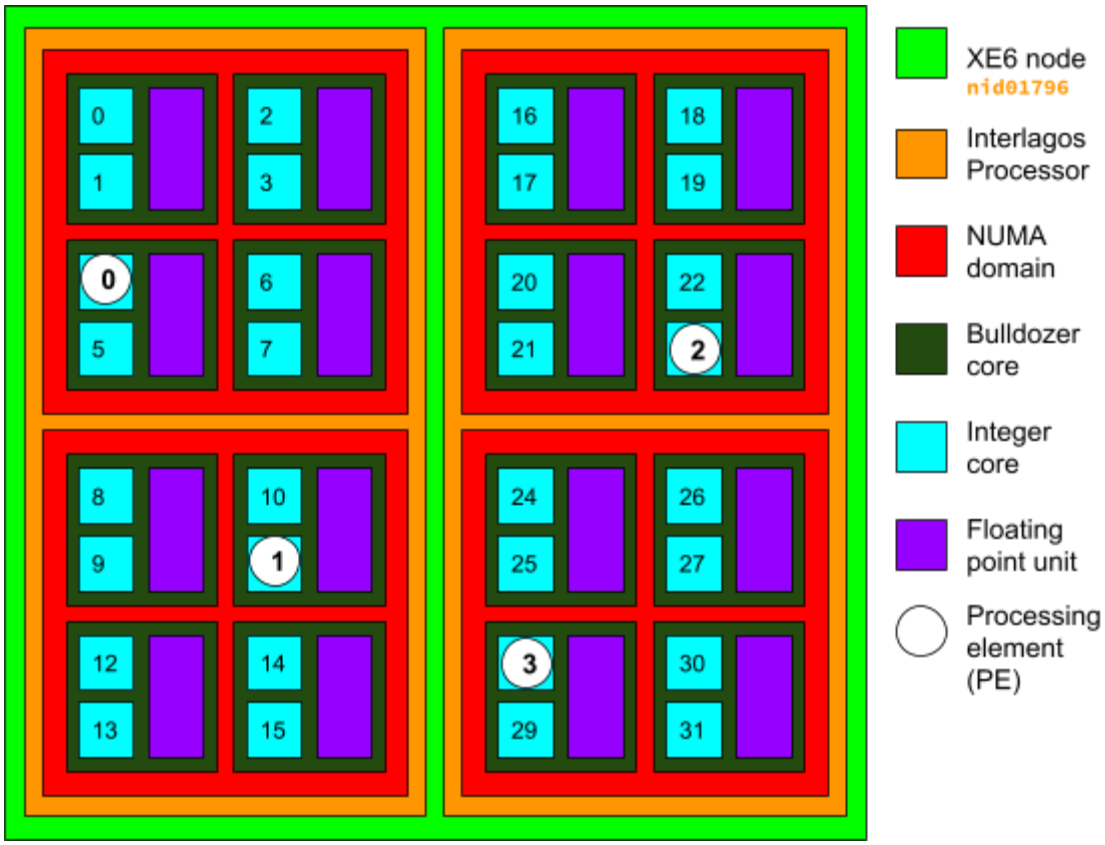
The placement looks like this:


0
1
1
3
2
5
3
7
16
17
18
19
20
21
22
23
8
9
10
11
12
13
14
15
24
25
26
27
28
29
30
31
XE6 node
nid01796
Interlagos Processor
NUMA domain
Bulldozer core
Integer core
Floating point unit
Processing element (PE)
4
1
5
3
6
5
7
7
16
17
18
19
20
21
22
23
8
9
10
11
12
13
14
15
24
25
26
27
28
29
30
31
XE6 node
nid01797
Interlagos Processor
NUMA domain
Bulldozer core
Integer core
Floating point unit
Processing element (PE)

If we want to assign specific PEs to specific cores, we can do this using the **-cc** option. We can list out the IDs of each core, separated by commas. The PEs will assign themselves to these cores, in order. For example:

```
aprun -n 4 -cc 4,11,23,28 ./test.exe|sort<ENTER>
PE  0 is on core 4 on node nid01796
PE  1 is on core 11 on node nid01796
PE  2 is on core 23 on node nid01796
PE  3 is on core 28 on node nid01796
```

The placement looks like this:

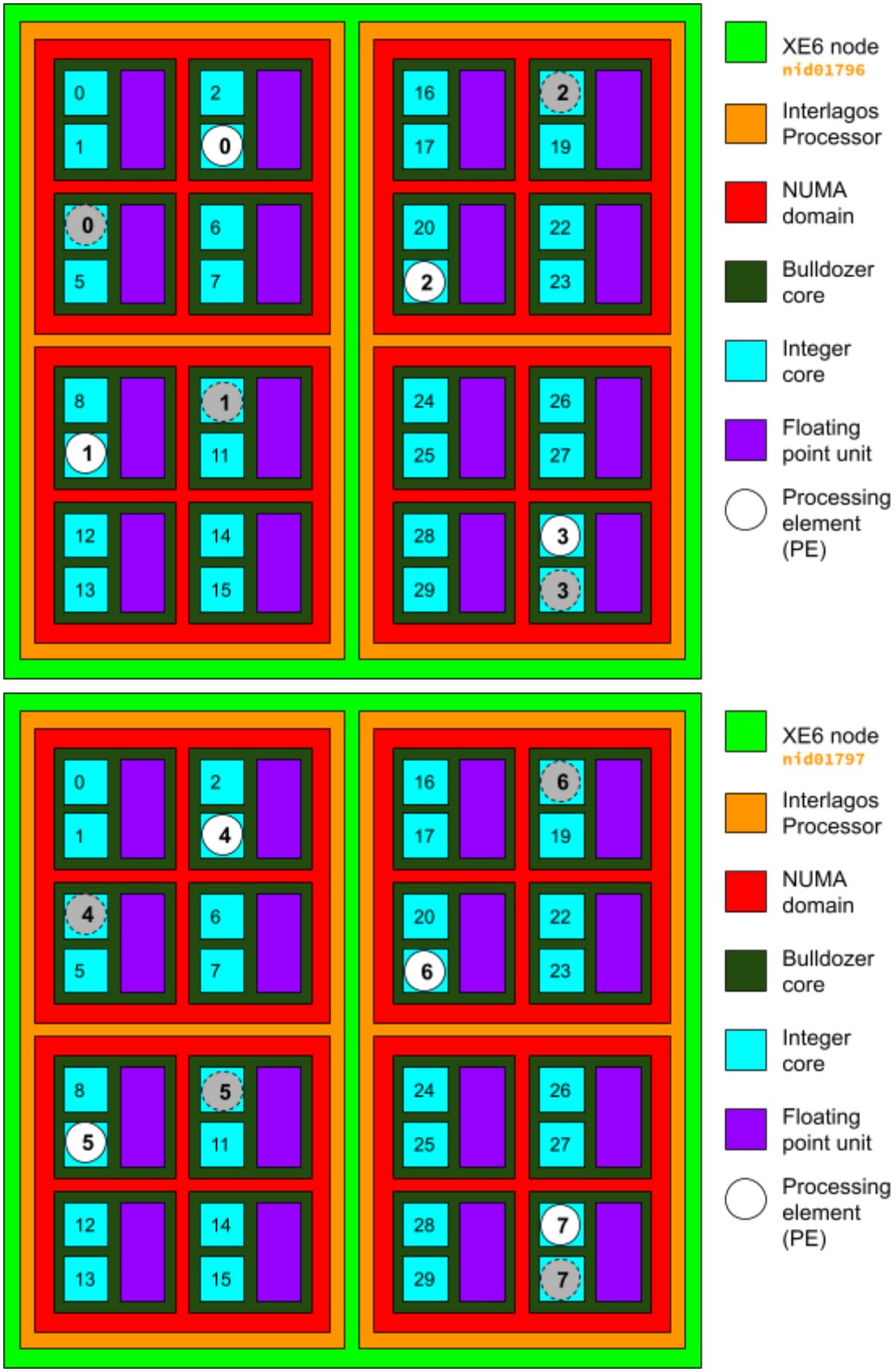


We can also combine **-cc** option with the **-d** and/or the **-N** options; for example:

```
aprun -n 8 -N 4 -d 2 -cc 3,9,21,30 ./test.exe|sort<ENTER>
PE  0 is on core 3 on node nid01796
PE  1 is on core 9 on node nid01796
PE  2 is on core 21 on node nid01796
PE  3 is on core 30 on node nid01796
PE  4 is on core 3 on node nid01797
```

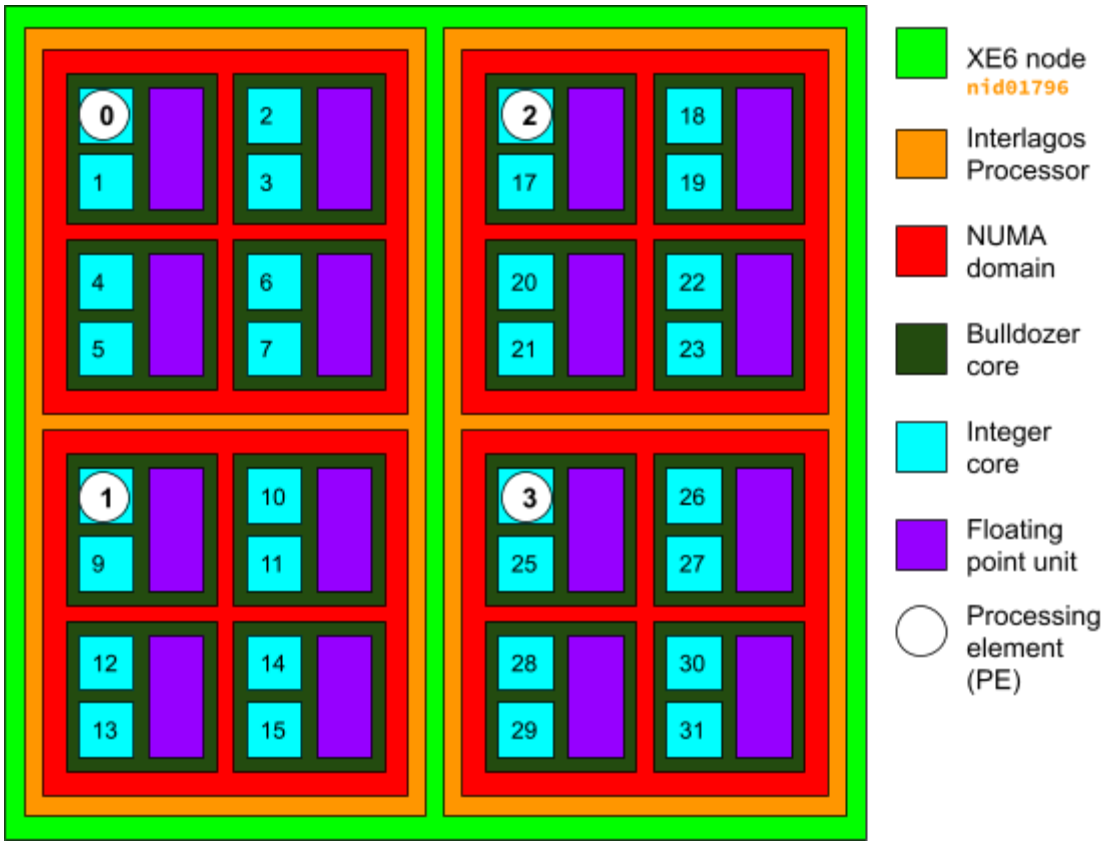
```
PE  5 is on core 9 on node nid01797
PE  6 is on core 21 on node nid01797
PE  7 is on core 30 on node nid01797
```

The placement looks like this:


0
1
2
0
0
5
6
7
16
17
2
19
20
2
22
23
8
1
1
11
24
25
26
27
12
13
14
15
28
29
3
3
XE6 node
nid01796
Interlagos Processor
NUMA domain
Bulldozer core
Integer core
Floating point unit
Processing element (PE)
0
1
2
4
4
5
6
7
16
17
6
19
20
6
22
23
8
5
5
11
24
25
26
27
12
13
14
15
28
29
7
7
XE6 node
nid01797
Interlagos Processor
NUMA domain
Bulldozer core
Integer core
Floating point unit
Processing element (PE)

We can use the **-S** option to specify how many PEs to assign per NUMA domain. For example, if we want to run with 4 PEs, each on its own NUMA domain, we can do the following:

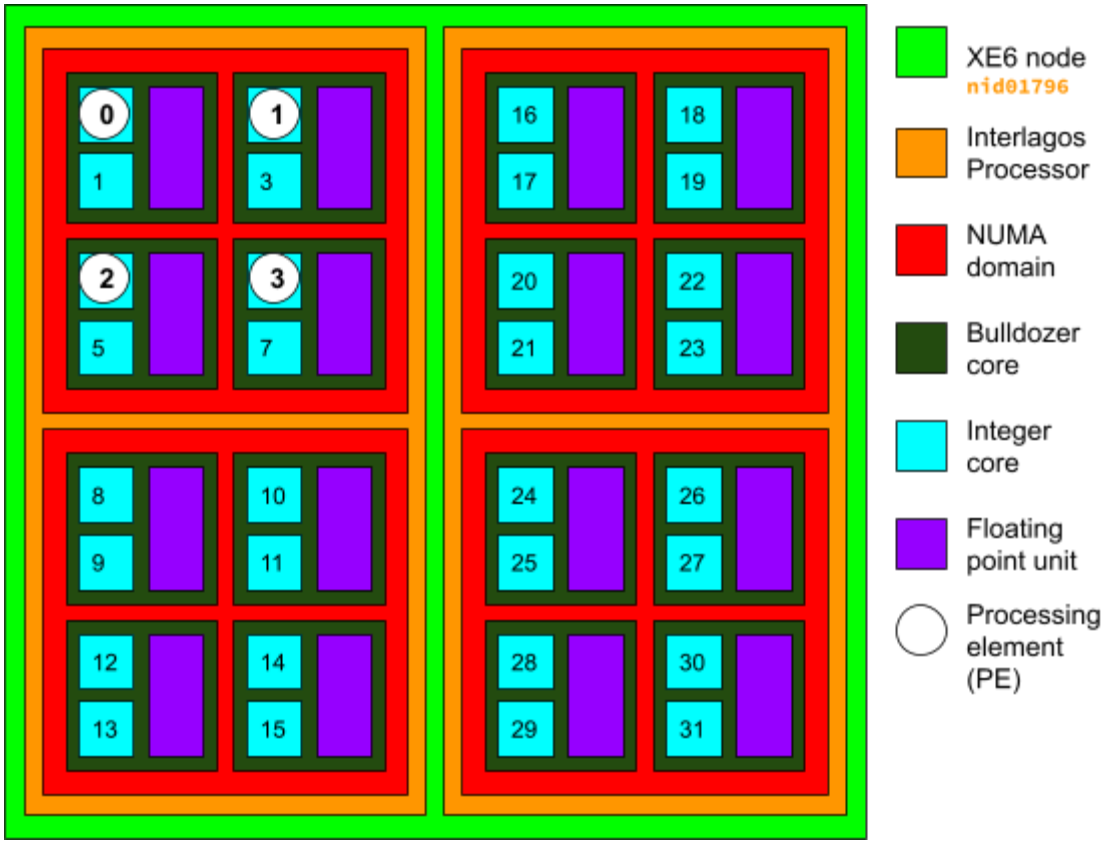
```
aprun -n 4 -S 1 ./test.exe|sort<ENTER>
PE  0 is on core 0 on node nid01796
PE  1 is on core 8 on node nid01796
PE  2 is on core 16 on node nid01796
PE  3 is on core 24 on node nid01796
```



As can be seen, this placement has one PE per NUMA domain ( ).

We can use the **-j** option to specify how many PEs to assign per Bulldozer core. For example, if we want to run with 4 PEs, each on its own Bulldozer core, we can do the following:

```
aprun -n 4 -j 1 ./test.exe|sort<ENTER>
PE  0 is on core 0 on node nid01796
PE  1 is on core 2 on node nid01796
PE  2 is on core 4 on node nid01796
PE  3 is on core 6 on node nid01796
```



As can be seen, this placement has one PE per Bulldozer core ( ).

In parallel computing, we usually want to measure the performance of the programs we run, i.e how long it takes, measured in real time. Linux has a built-in command called **`time`** that can be used in conjunction with any other command to report the number of minutes and seconds of real time it takes to execute the command. For example, the command below will time how long it takes to run the **`test.exe`** program with 4 PEs:

```
time aprun -n 4 ./test.exe|sort<ENTER>
PE  0 is on core 0 on node nid01796
PE  1 is on core 1 on node nid01796
PE  2 is on core 2 on node nid01796
PE  3 is on core 3 on node nid01796

real    0m2.936s
user    0m0.404s
sys     0m0.028s
```

Of these, the **`real`** time measures the amount of elapsed time from when the program started running to when it finished (0 minutes and 2.936 seconds in the example shown above). **`user`**

and **`sys`** time only measure the amount of the time the program was actually using the CPU and not idly waiting for other processes or data; this is broken into user-space time (**`user`**) and kernel-space time (**`sys`**). When measuring performance of a program, we usually care about the **`real`** time and not the **`user`** or **`sys`** time, since the **time to solution** or **time to science** is a measure of how long it takes for us to get a result after we press the enter key to tell the program to run, not how long the CPU thinks it spent actually doing work.

Batch Jobs

Let's move into a discussion of batch jobs. You can exit your interactive session by entering **`quit`**, **`exit`**, or **Ctrl-D**.

Interactive jobs are useful when we need to *test* or *debug* our programs, because this often involves a lot of running the program, seeing what it does, changing something, running the program again, seeing what it does, changing something, etc. We will also be using interactive jobs throughout this 2-week institute, because we will be employing the **expectation**, **observation**, and **reflection** method - identifying how we expect the model to behave, observing how it actually behaves, and reflecting on why it behaves that way. This is a highly interactive process. However, once it comes time to run programs automatically without needing user intervention (saving us time to work on something else while the computer does our work for us), we will want to submit **batch jobs**.

A batch job behaves very similarly to an interactive job, except it happens automatically, without the need for user intervention. Just like with an interactive job, we need to specify what resources are needed when we submit the job. We need to specify the number of nodes, cores, and time needed to run the job. The main behavioral difference is that instead of us typing the commands needed to run our programs, we specify what the commands will be ahead of time, and the scheduler runs these commands automatically for us.

We send a batch job to the Blue Waters scheduler using a batch script. We saw an example earlier of the syntax for specifying the resources needed for the job:

```
#!/bin/bash
#PBS -l nodes=2:ppn=32:xe
#PBS -l walltime=00:00:05
```

Note that the **`walltime`** is measuring how long the entire job will take to run. In the interactive job we used earlier, we wanted to request enough time for us to be able to enter commands, observe the output, reflect on what the output meant, and decide what to type next. In a batch job, we only need to allocate enough time for the program to run by itself. Since we saw earlier that the **`aprun`** command only took about 2 seconds to run, we can use a **`walltime`** of **`00:00:05`**, i.e. 5 seconds, which is more than enough time for the entire job to execute.

Let's try submitting a sample batch script. In the example code you copied to Blue Waters earlier, there is a **`simple.pbs`** file. This contains the 3 lines of the batch job we were just looking at. We can submit this file as a **batch script** by using qsub as follows (try it now):

```
qsub simple.pbs<ENTER>
Job submitted to account: jtp
4800977.bw
```

Note that the output lists the account to which this job was submitted, **`jtp`**. This is a unique code that Blue Waters uses to identify your project. You will see a different code than **`jtp`**, because you have an account through a different project.

Just as before, we can monitor the status of the job using **`qstat`**:

```
qstat <Job ID><ENTER>
```

or

```
qstat -u <your username><ENTER>
```

When the batch job runs, it generates an **output** file and an **error** file. The output file contains any text that was sent to the **standard output (stdout)** of the Unix shell, while the error file contains any text that was sent to the **standard error (stderr)** of the Unix shell.

If you enter **`ls`** in the directory where you submitted the job, you should see two files:

```
simple.pbs.o<Job ID>
simple.pbs.e<Job ID>
```

These are the default names of the files generated by the job: the name of the job (or the name of the batch script file if no name is specified) followed by **`.o`** (for output) or **`.e`** (for error), followed by the ID of the job. Thus, each job will have a unique pair of files that it generates.

You can view the contents of these (or any) files in multiple ways:

- Through a text editor like **`vi`**, **`emacs`**, or **`jpico`**.
- Through the **`less`** command, which lets you scroll with arrow keys, move forward one page at a time using **Ctrl-F** and backward one page at a time using **Ctrl-B**, and quit using **`q`**.
- Through the **`cat`** command, which just dumps all the contents of the file into the terminal.

Try this now with the output file. The output should look something like this:

```
----------------------------------------
Begin Torque Prologue on nid25356
at Thu May 19 16:25:45 CDT 2016
Job Id:             4800977.bw
Username:           aweeden
Group:              EOT_jtp
Job name:           simple.pbs
Requested resources:
neednodes=2:ppn=32:xe,nodes=2:ppn=32:xe,walltime=00:00:05
Queue:              normal
Account:      jtp
End Torque Prologue:  0.041 elapsed
----------------------------------------
```

This shows information about the job that ran. **Torque** is the name of the type of scheduler Blue Waters uses; Torque manages the **qsub** command.

The error file should be empty.

The **simple.pbs** batch script requests resources, but it does not actually run any commands to use those resources. Let's look at the contents of the **run.pbs** file:

```
#!/bin/bash
#PBS -l nodes=2:ppn=32:xe
#PBS -l walltime=00:00:05

cd $PBS_O_WORKDIR
time aprun -n 8 -N 4 ./test.exe|sort
```

The first few lines of this batch script are the same as the **simple.pbs** batch script. Then we have the following line, which tells the job to change directories into the same directory from which we submitted the job.

```
cd $PBS_O_WORKDIR
```

Finally, we have the line that executes an example **aprun** command on our **test.exe** program, sorting the output, and timing how long it takes.

Let's submit a job using this batch script:

```
qsub run.pbs<ENTER>
Job submitted to account: jtp
4801046.bw
```

When the job finishes, the error file should be empty, and this should be the contents of the output file (after the Torque prologue and the bit about the Application resources), noting that the node IDs will probably be different for you:

```
PE  0 is on core 0 on node nid02908
PE  1 is on core 1 on node nid02908
PE  2 is on core 2 on node nid02908
PE  3 is on core 3 on node nid02908
PE  4 is on core 0 on node nid02909
PE  5 is on core 1 on node nid02909
PE  6 is on core 2 on node nid02909
PE  7 is on core 3 on node nid02909
```

Charging

Each research and education allocation on Blue Waters is allocated a certain number of **node-hours**. This is a figure that can be calculated by multiplying the aggregate number of nodes used for all jobs by the aggregate number of hours used for all jobs. Each time a job is run, Blue Waters **charges** you for the time and space used; this gets deducted from your node-hours count. In the case of batch jobs, you are only charged for the amount of time the job is actually running (not while it is waiting in the queue) and for the nodes requested and obtained by the job (even if you only use **`aprun`** on one of the nodes!). For interactive jobs, you are also charged for the duration of the job, *which also includes time you are not entering commands. As long as the interactive shell is running, you will be charged time*. Thus, it can be more expensive to use an interactive job.

Blue Waters offers certain charging discounts if your jobs meet certain criteria. This is described at the page linked below:

https://bluewaters.ncsa.illinois.edu/manage-news/-/blogs/charge-factor-discounts-for-jobs-on-blue-waters

Other options to `qsub`

If you want to use other parameters or options to qsub, you can do so either from the command line (e.g. for the **`-q low`** option to change the queue used):

```
qsub -q low -l nodes=....
```

Or in the top part of a batch script using the **`#PBS`** syntax:

```
#PBS -q low
```

Additional Resources

- https://bluewaters.ncsa.illinois.edu/user-guide
- https://bluewaters.ncsa.illinois.edu/running-your-jobs and the menu on the left.
- http://hpcuniversity.org/media/TrainingMaterials/32/ABlueWatersUsageGuide.pdf
- https://bluewaters.ncsa.illinois.edu/manage-news/-/blogs/charge-factor-discounts-for-jobs-on-blue-waters