

# STAMPEDE2 USER GUIDE

Last update: June 11, 2024

See [revision history](#)

**Stampede2 is now decommissioned.** (06/04/2024)

---

## Notices

**Stampede2 queues are now closed.** (01/31/2024)

Updated Timeline - Stampede2 to Stampede3 Transition.

*All dates subject to change based on hardware availability and condition.*

January 2024 - Reduction in number of SKX/ICX nodes, reduction in queue maximums

January 2024 - Stampede3 file system available for data migration

February 2024 - Early user period for Stampede3

February 2024 - Stampede2 file system decommissioned

March 2024 - Stampede3 in full production

## Introduction

Stampede2, generously funded by the National Science Foundation (NSF) through [award ACI-1540931](#), is one of the Texas Advanced Computing Center (TACC), University of Texas at Austin's flagship supercomputers. Stampede2 entered full production in the Fall 2017 as an 18-petaflop national resource that builds on the successes of the original Stampede system it replaces. The first phase of the Stampede2 rollout featured the second generation of processors based on Intel's Many Integrated Core (MIC) architecture. Stampede2's 4,200 Knights Landing (KNL) nodes represent a radical break with the first-generation Knights Corner (KNC) MIC coprocessor. Unlike the legacy KNC, a Stampede2 KNL is not a coprocessor: each 68-core KNL is a stand-alone, self-booting processor that is the sole processor in its node. Phase 2 added to Stampede2 a total of 1,736 Intel Xeon Skylake (SKX) nodes. The final phase of Stampede2 features the replacement of 448 KNL nodes with 224 Ice Lake nodes.

## System Overview

### KNL Compute Nodes

Stampede2 hosts 4,200 KNL compute nodes, including 504 KNL nodes that were formerly configured as a Stampede1 sub-system.

Each of Stampede2's KNL nodes includes 96GB of traditional DDR4 Random Access Memory (RAM). They also feature an additional 16GB of high bandwidth, on-package memory known as Multi-Channel Dynamic Random Access Memory (**MCDRAM**) that is up to four times faster than DDR4. The KNL's memory is configurable in two important ways: there are BIOS settings that determine at boot time the processor's **memory mode** and **cluster mode**. The processor's **memory mode** determines whether the fast MCDRAM operates as RAM, as direct-mapped L3 cache, or as a mixture of the two. The **cluster mode** determines the mechanisms for achieving cache coherency, which in turn determines latency: roughly speaking, this mode specifies the degree to which some memory addresses are "closer" to some cores than to others. See "[Programming and Performance: KNL](#)" below for a top-level description of these and other available memory and cluster modes.

## Table 1. KNL Compute Node Specifications

Table 1. Stampede2 KNL Compute Node Specifications.

Specification	Value
Model:	Intel Xeon Phi 7250 ("Knights Landing")
Total cores per KNL node:	68 cores on a single socket
Hardware threads per core:	4
Hardware threads per node:	$68 \times 4 = 272$
Clock rate:	1.4GHz
RAM:	96GB DDR4 plus 16GB high-speed MCDRAM. Configurable in two important ways; see " <a href="#">Programming and Performance: KNL</a> " for more info.
Cache:	32KB L1 data cache per core; 1MB L2 per two-core tile. In default config, <a href="#">MCDRAM</a> operates as 16GB direct-mapped L3.
Local storage:	All but 504 KNL nodes have a 107GB <code>/tmp</code> partition on a 200GB Solid State Drive (SSD). The 504 KNLs originally installed as the Stampede1 KNL sub-system each have a 32GB <code>/tmp</code> partition on 112GB SSDs. The latter nodes currently make up the <code>development</code> , <code>long</code> and <code>flat-quadrant</code> <a href="#">queues</a> . Size of <code>/tmp</code> partitions as of 24 Apr 2018.

## SKX Compute Nodes

Stampede2 hosts 1,736 SKX compute nodes.

Table 2. Stampede2 SKX Compute Node Specifications

Specification	Value
Model:	Intel Xeon Platinum 8160 ("Skylake")
Total cores per SKX node:	48 cores on two sockets (24 cores/socket)
Hardware threads per core:	2
Hardware threads per node:	$48 \times 2 = 96$
Clock rate:	2.1GHz nominal (1.4-3.7GHz depending on instruction set and number of active cores)
RAM:	192GB (2.67GHz) DDR4
Cache:	32KB L1 data cache per core; 1MB L2 per core; 33MB L3 per socket. Each socket can cache up to 57MB (sum of L2 and L3 capacity).
Local storage:	144GB <code>/tmp</code> partition on a 200GB SSD. Size of <code>/tmp</code> partition as of 14 Nov 2017.

## ICX Compute Nodes

Stampede2 hosts 224 ICX compute nodes.

Table 2a. Stampede2 ICX Compute Node Specifications

Specification	Value
Model:	Intel Xeon Platinum 8380 ("Ice Lake")
Total cores per ICX node:	80 cores on two sockets (40 cores/socket)
Hardware threads per core:	2
Hardware threads per node:	80 x 2 = 160
Clock rate:	2.3 GHz nominal (3.4GHz max frequency depending on instruction set and number of active cores)
RAM:	256GB (3.2 GHz) DDR4
Cache:	48KB L1 data cache per core; 1.25 MB L2 per core; 60 MB L3 per socket. Each socket can cache up to 110 MB (sum of L2 and L3 capacity)
Local storage:	342 GB <code>/tmp</code> partition

### Login Nodes

The Stampede2 login nodes, upgraded at the start of Phase 2, are Intel Xeon Gold 6132 (SKX) nodes, each with 28 cores on two sockets (14 cores/socket). They replace the decommissioned Broadwell login nodes used during Phase 1.

### Network

The interconnect is a 100Gb/sec Intel Omni-Path (OPA) network with a fat tree topology employing six core switches. There is one leaf switch for each 28-node half rack, each with 20 leaf-to-core uplinks (28/20 oversubscription).

### File Systems Introduction

Stampede2 mounts three shared Lustre file systems on which each user has corresponding account-specific directories `$HOME`, `$WORK`, and `$SCRATCH`. Each file system is available from all Stampede2 nodes; the [Stockyard-hosted work file system](#) is available on most other TACC HPC systems as well. See [Navigating the Shared File Systems](#) for detailed information as well as the [Good Conduct](#) file system guidelines.

Table 3. Stampede2 File Systems

File System	Quota	Key Features
<code>\$HOME</code>	10GB, 200,000 files	<b>Not intended for parallel or high-intensity file operations.</b> Backed up regularly. Overall capacity ~1PB. Two Meta-Data Servers (MDS), four Object Storage Targets (OSTs). Defaults: 1 stripe, 1MB stripe size. Not purged.
<code>\$WORK</code>	1TB, 3,000,000 files across all TACC systems, regardless of where on the file system the files reside.	<b>Not intended for high-intensity file operations or jobs involving very large files.</b> On the Global Shared File System that is mounted on most TACC systems. See <a href="#">Stockyard system description</a> for more information. Defaults: 1 stripe, 1MB stripe size

File System	Quota	Key Features
		Not backed up. Not purged.
\$SCRATCH	no quota	Overall capacity ~30PB. Four MDSs, 66 OSTs. Defaults: 1 stripe, 1MB stripe size. Not backed up. <b>Files are subject to purge if access time* is more than 10 days old.</b>

## Scratch File System Purge Policy



### Warning

The \$SCRATCH file system, as its name indicates, is a **temporary storage space**. Files that have not been accessed\* in ten days are subject to purge. Deliberately modifying file access time (using any method, tool, or program) for the purpose of circumventing purge policies is prohibited.

\*The operating system updates a file's access time when that file is modified on a login or compute node. Reading or executing a file/script on a login node does not update the access time, but reading or executing on a compute node does update the access time. This approach helps us distinguish between routine management tasks (e.g. `tar`, `scp`) and production use. Use the command `ls -ul` to view access times.

## Accessing the System

Access to all TACC systems now requires Multi-Factor Authentication (MFA). You can create an MFA pairing on the TACC User Portal. After login on the portal, go to your account profile (Home->Account Profile), then click the "Manage" button under "Multi-Factor Authentication" on the right side of the page. See [Multi-Factor Authentication at TACC](#) for further information.

## Secure Shell (SSH)

The `ssh` command (SSH protocol) is the standard way to connect to Stampede2. SSH also includes support for the file transfer utilities `scp` and `sftp`. [Wikipedia](#) is a good source of information on SSH. SSH is available within Linux and from the terminal app in the Mac OS. If you are using Windows, you will need an SSH client that supports the SSH-2 protocol: e.g. [Bitvise](#), [OpenSSH](#), [PuTTY](#), or [SecureCRT](#). Initiate a session using the `ssh` command or the equivalent; from the Linux command line the launch command looks like this:

```
localhost$ ssh myusername@stampede2.tacc.utexas.edu
```

The above command will rotate connections across all available login nodes and route your connection to one of them. To connect to a specific login node, use its full domain name:

```
localhost$ ssh myusername@login2.stampede2.tacc.utexas.edu
```

To connect with X11 support on Stampede2 (usually required for applications with graphical user interfaces), use the `-X` or `-Y` switch:

```
localhost$ ssh -X myusername@stampede2.tacc.utexas.edu
```

Use your TACC password for direct logins to TACC resources. You can change your TACC password through the [TACC User Portal](#). Log into the portal, then select "Change Password" under the "HOME" tab. If you've forgotten your password, go to the [TACC User Portal](#) home page and select "Password Reset" under the Home tab.

To report a connection problem, execute the `ssh` command with the `-vvv` option and include the verbose output when submitting a help ticket.

**Do not run the `ssh-keygen` command on Stampede2.** This command will create and configure a key pair that will interfere with the execution of job scripts in the batch system. If you do this by mistake, you can recover by renaming or deleting the `.ssh` directory located in your home directory; the system will automatically generate a new one for you when you next log into Stampede2.

1. execute `mv .ssh dot.ssh.old`
2. log out
3. log into Stampede2 again

After logging in again the system will generate a properly configured key pair.

## Using Stampede2

Stampede2 nodes run Red Hat Enterprise Linux 7. Regardless of your research workflow, **you'll need to master Linux basics** and a Linux-based text editor (e.g. `emacs`, `nano`, `gedit`, or `vi/vim`) to use the system properly. If you encounter a term or concept in this user guide that is new to you, a quick internet search should help you resolve the matter quickly.

## Configuring Your Account

### Linux Shell

The default login shell for your user account is Bash. To determine your current login shell, execute:

```
$ echo $SHELL
```

If you'd like to change your login shell to `csh`, `sh`, `tcsh`, or `zsh`, submit a ticket through the [TACC User Portal](#). The `chsh` ("change shell") command will not work on TACC systems.

When you start a shell on Stampede2, system-level startup files initialize your account-level environment and aliases before the system sources your own user-level startup scripts. You can use these startup scripts to customize your shell by defining your own environment variables, aliases, and functions. These scripts (e.g. `.profile` and `.bashrc`) are generally hidden files: so-called dotfiles that begin with a period, visible when you execute: `ls -a`.

Before editing your startup files, however, it's worth taking the time to understand the basics of how your shell manages startup. Bash startup behavior is very different from the simpler `csh` behavior, for example. The Bash startup sequence varies depending on how you start the shell (e.g. using `ssh` to open a login shell, executing the `bash` command to begin an interactive shell, or launching a script to start a non-interactive shell). Moreover, Bash does not automatically source your `.bashrc` when you start a login shell by using `ssh` to connect to a node. Unless you have specialized needs, however, this is undoubtedly more flexibility than you want: you will probably want your environment to be the same regardless of how you start the shell. The easiest way to achieve this is to execute `source ~/.bashrc` from your `.profile`, then put all your customizations in `.bashrc`. The system-generated default startup scripts demonstrate this approach. We recommend that you use these default files as templates.

For more information see the [Bash Users' Startup Files: Quick Start Guide](#) and other online resources that explain shell startup. To recover the originals that appear in a newly created account, execute `/usr/local/startup_scripts/install_default_scripts`.

### Environment Variables

Your environment includes the environment variables and functions defined in your current shell: those initialized by the system, those you define or modify in your account-level startup scripts, and those defined or modified by the [modules](#) that you load to configure your software environment. Be sure to distinguish between an environment variable's name (e.g. `HISTSIZE`) and its value (`$HISTSIZE`). Understand as well that a sub-shell (e.g. a script) inherits environment variables from its parent, but does not inherit ordinary shell variables or aliases. Use `export` (in Bash) or `setenv` (in `csh`) to define an environment variable.

Execute the `env` command to see the environment variables that define the way your shell and child shells behave.

Pipe the results of `env` into `grep` to focus on specific environment variables. For example, to see all environment variables that contain the string `GIT` (in all caps), execute:

```
$ env | grep GIT
```

The environment variables `PATH` and `LD_LIBRARY_PATH` are especially important. `PATH` is a colon-separated list of directory paths that determines where the system looks for your executables. `LD_LIBRARY_PATH` is a similar list that determines where the system looks for shared libraries.

## Account-Level Diagnostics

TACC's `sanitytool` module loads an account-level diagnostic package that detects common account-level issues and often walks you through the fixes. You should certainly run the package's `sanitycheck` utility when you encounter unexpected behavior. You may also want to run `sanitycheck` periodically as preventive maintenance. To run `sanitytool`'s account-level diagnostics, execute the following commands:

```
login1$ module load sanitytool
login1$ sanitycheck
```

Execute `module help sanitytool` for more information.

## Accessing the Compute Nodes

You connect to Stampede2 through one of four "front-end" login nodes. The login nodes are shared resources: at any given time, there are many users logged into each of these login nodes, each preparing to access the "back-end" compute nodes (Figure 2. Login and Compute Nodes). What you do on the login nodes affects other users directly because you are competing for the same memory and processing power. This is the reason you should not run your applications on the login nodes or otherwise abuse them. Think of the login nodes as a prep area where you can manage files and compile code before accessing the compute nodes to perform research computations. See [Good Conduct](#) for more information.

**You can use your command-line prompt, or the `hostname` command, to tell you whether you are on a login node or a compute node.** The default prompt, or any custom prompt containing `\h`, displays the short form of the hostname (e.g. `c401-064`). The hostname for a Stampede2 login node begins with the string `login` (e.g. `login2.stampede2.tacc.utexas.edu`), while compute node hostnames begin with the character `c` (e.g. `c401-064.stampede2.tacc.utexas.edu`). Note that the default prompts on the compute nodes include the node type (`kn1`, `skx` or `icx`) as well. The environment variable `TACC_NODE_TYPE`, defined only on the compute nodes, also displays the node type. The simplified prompts in the User Guide examples are shorter than Stampede2's actual default prompts.

While some workflows, tools, and applications hide the details, there are three basic ways to access the compute nodes:

1. **Submit a batch job using the `sbatch` command.** This directs the scheduler to run the job unattended when there are resources available. Until your batch job begins it will wait in a [queue](#). You do not need to remain connected while the job is waiting or executing. See [Running Jobs](#) for more information. Note that the scheduler does not start jobs on a first come, first served basis; it juggles many variables to keep the machine busy while balancing the competing needs of all users. The best way to minimize wait time is to request only the resources you really need: the scheduler will have an easier time finding a slot for the two hours you need than for the 48 hours you unnecessarily request.
2. **Begin an [interactive session](#) using `idev` or `srun`.** This will log you into a compute node and give you a command prompt there, where you can issue commands and run code as if you were doing so on your personal machine. An interactive session is a great way to develop, test, and debug code. When you request an interactive session, the scheduler submits a job on your behalf. You will need to remain logged in until the interactive session begins.
3. **Begin an [interactive session](#) using `ssh`** to connect to a compute node on which you are already running a job. This is a good way to open a second window into a node so that you can monitor a job while it runs.



Be sure to request computing resources that are consistent with the type of application(s) you are running:

- A **serial** (non-parallel) application can only make use of a single core on a single node, and will only see that node's memory.
- A threaded program (e.g. one that uses **OpenMP**) employs a shared memory programming model and is also restricted to a single node, but the program's individual threads can run on multiple cores on that node.
- An **MPI** (Message Passing Interface) program can exploit the distributed computing power of multiple nodes: it launches multiple copies of its executable (MPI **tasks**, each assigned unique IDs called **ranks**) that can communicate with each other across the network. The tasks on a given node, however, can only directly access the memory on that node. Depending on the program's memory requirements, it may not be possible to run a task on every core of every node assigned to your job. If it appears that your MPI job is running out of memory, try launching it with fewer tasks per node to increase the amount of memory available to individual tasks.
- A popular type of **parameter sweep** (sometimes called **high throughput computing**) involves submitting a job that simultaneously runs many copies of one serial or threaded application, each with its own input parameters ("Single Program Multiple Data", or SPMD). The `launcher` tool is designed to make it easy to submit this type of job. For more information:

```
$ module load launcher
$ module help launcher
```



Figure 2. Login  
and compute  
nodes

## Using Modules to Manage your Environment

Lmod, a module system developed and maintained at TACC, makes it easy to manage your environment so you have access to the software packages and versions that you need to using your research. This is especially important on a system like Stampede2 that serves thousands of users with an enormous range of needs. Loading a module amounts to choosing a specific package from among available alternatives:

```
$ module load intel          # load the default Intel compiler
$ module load intel/17.0.4   # load a specific version of Intel compiler
```

A module does its job by defining or modifying environment variables (and sometimes aliases and functions). For example, a module may prepend appropriate paths to `$PATH` and `$LD_LIBRARY_PATH` so that the system can find the executables and libraries associated with a given software package. The module creates the illusion that the system is installing software for your personal use. Unloading a module reverses these changes and creates the illusion that the system just uninstalled the software:

```
$ module load ddt # defines DDT-related env vars; modifies others
$ module unload ddt # undoes changes made by load
```

The module system does more, however. When you load a given module, the module system can automatically replace or deactivate modules to ensure the packages you have loaded are compatible with each other. In the example below, the module system automatically unloads one compiler when you load another, and replaces Intel-compatible versions of IMPI and PETSc with versions compatible with gcc:

```
$ module load intel # load default version of Intel compiler
$ module load petsc # load default version of PETSc
$ module load gcc   # change compiler

Lmod is automatically replacing "intel/17.0.4" with "gcc/7.1.0".
```

```
Due to MODULEPATH changes, the following have been reloaded:  
1) impi/17.0.3      2) petsc/3.7
```

On Stampede2, modules generally adhere to a TACC naming convention when defining environment variables that are helpful for building and running software. For example, the `papi` module defines `TACC_PAPI_BIN` (the path to PAPI executables), `TACC_PAPI_LIB` (the path to PAPI libraries), `TACC_PAPI_INC` (the path to PAPI include files), and `TACC_PAPI_DIR` (top-level PAPI directory). After loading a module, here are some easy ways to observe its effects:

```
$ module show papi # see what this module does to your environment  
$ env | grep PAPI  # see env vars that contain the string PAPI  
$ env | grep -i papi # case-insensitive search for 'papi' in environment
```

To see the modules you currently have loaded:

```
$ module list
```

To see all modules that you can load right now because they are compatible with the currently loaded modules:

```
$ module avail
```

To see all installed modules, even if they are not currently available because they are incompatible with your currently loaded modules:

```
$ module spider # list all modules, even those not available to load
```

To filter your search:

```
$ module spider slep # all modules with names containing 'slep'  
$ module spider sundials/2.5.0 # additional details on a specific module
```

Among other things, the latter command will tell you which modules you need to load before the module is available to load. You might also search for modules that are tagged with a keyword related to your needs (though your success here depends on the diligence of the module writers). For example:

```
$ module keyword performance
```

You can save a collection of modules as a personal default collection that will load every time you log into Stampede2. To do so, load the modules you want in your collection, then execute:

```
$ module save # save the currently loaded collection of modules
```

Two commands make it easy to return to a known, reproducible state:

```
$ module reset # load the system default collection of modules  
$ module restore # load your personal default collection of modules
```

On TACC systems, the command `module reset` is equivalent to `module purge; module load TACC`. It's a safer, easier way to get to a known baseline state than issuing the two commands separately.

Help text is available for both individual modules and the module system itself:

```
$ module help swr # show help text for software package swr  
$ module help # show help text for the module system itself
```

See [Lmod's online documentation](#) for more extensive documentation. The online documentation addresses the basics in more detail, but also covers several topics beyond the scope of the help text (e.g. writing and using your own module files).

It's safe to execute module commands in job scripts. In fact, this is a good way to write self-documenting, portable job scripts that produce reproducible results. If you use `module save` to define a personal default module collection, it's rarely



necessary to execute module commands in shell startup scripts, and it can be tricky to do so safely. If you do wish to put module commands in your startup scripts, see Stampede2's default startup scripts for a safe way to do so.

## Managing Your Files


Stampede2 mounts three file Lustre file systems that are shared across all nodes: the home, work, and scratch file systems. Stampede2's startup mechanisms define corresponding account-level environment variables `$HOME`, `$SCRATCH`, and `$WORK` that store the paths to directories that you own on each of these file systems. Consult the [Stampede2 File Systems](#) table for the basic characteristics of these file systems, [File Operations: I/O Performance](#) for advice on performance issues, and [Good Conduct](#) for tips on file system etiquette.

### Navigating the Shared File Systems

Stampede2's `/home` and `/scratch` file systems are mounted only on Stampede2, but the work file system mounted on Stampede2 is the Global Shared File System hosted on [Stockyard](#). Stockyard is the same work file system that is currently available on Frontera, Lonestar6, and several other TACC resources.

The `$STOCKYARD` environment variable points to the highest-level directory that you own on the Global Shared File System. The definition of the `$STOCKYARD` environment variable is of course account-specific, but you will see the same value on all TACC systems that provide access to the Global Shared File System. This directory is an excellent place to store files you want to access regularly from multiple TACC resources.

Your account-specific `$WORK` environment variable varies from system to system and is a sub-directory of `$STOCKYARD` (Figure 3). The sub-directory name corresponds to the associated TACC resource. The `$WORK` environment variable on Stampede2 points to the `$STOCKYARD/stampede2` subdirectory, a convenient location for files you use and jobs you run on Stampede2. Remember, however, that all subdirectories contained in your `$STOCKYARD` directory are available to you from any system that mounts the file system. If you have accounts on both Stampede2 and Frontera, for example, the `$STOCKYARD/stampede2` directory is available from your Frontera account, and `$STOCKYARD/frontera` is available from your Stampede2 account.



**Note**

Your quota and reported usage on the Global Shared File System reflects all files that you own on Stockyard, regardless of their actual location on the file system.

See the example for fictitious user `bjones` in the figure below. All directories are accessible from all systems, however a given sub-directory (e.g. `lonestar6`, `frontera`) will exist **only** if you have an allocation on that system. Figure 3 illustrates account-level directories on the `$WORK` file system (Global Shared File System hosted on Stockyard). Example for fictitious user `bjones`. All directories usable from all systems. Sub-directories (e.g. `lonestar6`, `frontera`) exist only when you have allocations on the associated system.

 Stockyard 2022Figure 3.

Note that resource-specific sub-directories of `$STOCKYARD` are nothing more than convenient ways to manage your resource-specific files. You have access to any such sub-directory from any TACC resources. If you are logged into Stampede2, for example, executing the alias `cdw` (equivalent to `cd $WORK`) will take you to the resource-specific sub-directory `$STOCKYARD/stampede2`. But you can access this directory from other TACC systems as well by executing `cd $STOCKYARD/stampede2`. These commands allow you to share files across TACC systems. In fact, several convenient account-level aliases make it even easier to navigate across the directories you own in the shared file systems:

**Table 4. Built-in Account Level Aliases**

Alias	Command
<code>cd</code> or <code>cdh</code>	<code>cd \$HOME</code>
<code>cdw</code>	<code>cd \$WORK</code>
<code>cds</code>	<code>cd \$SCRATCH</code>
<code>cdy</code> or <code>cdg</code>	<code>cd \$STOCKYARD</code>

## Striping Large Files

Stampede2's Lustre file systems look and act like a single logical hard disk, but are actually sophisticated integrated systems involving many physical drives (dozens of physical drives for `$HOME`, hundreds for `$WORK` and `$SCRATCH`).

Lustre can **stripe** (distribute) large files over several physical disks, making it possible to deliver the high performance needed to service input/output (I/O) requests from hundreds of users across thousands of nodes. Object Storage Targets (OSTs) manage the file system's spinning disks: a file with 16 stripes, for example, is distributed across 16 OSTs. One designated Meta-Data Server (MDS) tracks the OSTs assigned to a file, as well as the file's descriptive data.



### Tip

Before transferring to, or creating large files on Stampede2, be sure to set an appropriate default stripe count on the receiving directory.

To avoid exceeding your fair share of any given OST, a good rule of thumb is to allow at least one stripe for each 100GB in the file. For example, to set the default stripe count on the current directory to 30 (a plausible stripe count for a directory receiving a file approaching 3TB in size), execute:

```
$ lfs setstripe -c 30 $PWD
```

Note that an `lfs setstripe` command always sets both stripe count and stripe size, even if you explicitly specify only one or the other. Since the example above does not explicitly specify stripe size, the command will set the stripe size on the directory to Stampede2's system default (1MB). In general there's no need to customize stripe size when creating or transferring files.

Remember that it's not possible to change the striping on a file that already exists. Moreover, the `mv` command has no effect on a file's striping if the source and destination directories are on the same file system. You can, of course, use the `cp` command to create a second copy with different striping; to do so, copy the file to a directory with the intended stripe parameters.

You can check the stripe count of a file using the `lfs getstripe` command:

```
$ lfs getstripe myfile
```

## Transferring Files

### with `scp`

You can transfer files between Stampede2 and Linux-based systems using either `scp` [↗](#) or `rsync` [↗](#). Both `scp` and `rsync` are available in the Mac Terminal app. Windows [ssh clients](#) typically include `scp`-based file transfer capabilities.

The Linux `scp` (secure copy) utility is a component of the OpenSSH suite. Assuming your Stampede2 username is `bjones`, a simple `scp` transfer that pushes a file named `myfile` from your local Linux system to Stampede2 `$HOME` would look like this:

```
localhost$ scp ./myfile bjones@stampede2.tacc.utexas.edu: # note colon after net address
```

You can use wildcards, but you need to be careful about when and where you want wildcard expansion to occur. For example, to push all files ending in `.txt` from the current directory on your local machine to `/work/01234/bjones/scripts` on Stampede2:

```
localhost$ scp *.txt bjones@stampede2.tacc.utexas.edu:/work/01234/bjones/stampede2
```

To delay wildcard expansion until reaching Stampede2, use a backslash (`\`) as an escape character before the wildcard. For example, to pull all files ending in `.txt` from `/work/01234/bjones/scripts` on Stampede2 to the current directory on your local system:

```
localhost$ scp bjones@stampede2.tacc.utexas.edu:/work/01234/bjones/stampede2/*.txt .
```



#### Note

Using `scp` with wildcard expansion on the remote host is unreliable. Specify absolute paths wherever possible.

Avoid using `scp` for recursive (`-r`) transfers of directories that contain nested directories of many small files:

```
localhost$ scp -r ./mydata bjones@stampede2.tacc.utexas.edu:\$WORK # DON'T DO THIS
```

Instead, use `tar` to create an archive of the directory, then transfer the directory as a single file:

```
localhost$ tar cvf ./mydata.tar mydata # create archive
localhost$ scp ./mydata.tar bjones@stampede2.tacc.utexas.edu:\$WORK # transfer archive
```

## with `rsync`

The `rsync` (remote synchronization) utility is a great way to synchronize files that you maintain on more than one system: when you transfer files using `rsync`, the utility copies only the changed portions of individual files. As a result, `rsync` is especially efficient when you only need to update a small fraction of a large dataset. The basic syntax is similar to `scp`:

```
localhost$ rsync mybigfile bjones@stampede2.tacc.utexas.edu:\$WORK/data
localhost$ rsync -avtr mybigdir bjones@stampede2.tacc.utexas.edu:\$WORK/data
```

The options on the second transfer are typical and appropriate when syncing a directory: this is a recursive update (`-r`) with verbose (`-v`) feedback; the synchronization preserves time stamps (`-t`) as well as symbolic links and other meta-data (`-a`). Because `rsync` only transfers changes, recursive updates with `rsync` may be less demanding than an equivalent recursive transfer with `scp`.

See [Striping Large Files](#) for additional important advice about striping the receiving directory when transferring or creating large files on TACC systems.

As detailed in [Good Conduct](#), it is important to monitor your quotas on the `$HOME` and `$WORK` file systems, and limit the number of simultaneous transfers. Remember also that `$STOCKYARD` (and your `$WORK` directory on each TACC resource) is available from several other TACC systems: there's no need for `scp` when both the source and destination involve sub-directories of `$STOCKYARD`. See [Managing Your Files](#) for more information about transfers on `$STOCKYARD`.

## with Grid Community Toolkit

The Grid Community Toolkit (GCT) is an open-source fork of the [Globus Toolkit](#) and was created in response to the [end-of-support](#) of the Globus Toolkit in January 2018.

Stampede2 has two endpoints, one running Globus gridftp v5.4 software available for [ACCESS](#) (formerly XSEDE) users, and the endpoint running Grid Community Toolkit with CILogon authentication available to all.

## Sharing Files with Collaborators

If you wish to share files and data with collaborators in your project, see [Sharing Project Files on TACC Systems](#) for step-by-step instructions. Project managers or delegates can use Unix group permissions and commands to create read-only or read-write shared workspaces that function as data repositories and provide a common work area to all project members.

## Building Software

The phrase "building software" is a common way to describe the process of producing a machine-readable executable file from source files written in C, Fortran, or some other programming language. In its simplest form, building software involves a simple, one-line call or short shell script that invokes a compiler. More typically, the process leverages the power of [makefiles](#), so you can change a line or two in the source code, then rebuild in a systematic way only the components affected by the change. Increasingly, however, the build process is a sophisticated multi-step automated workflow managed by a special framework like [autotools](#) or [cmake](#), intended to achieve a repeatable, maintainable, portable mechanism for installing software across a wide range of target platforms.

### Basics of Building Software

This section of the user guide does nothing more than introduce the big ideas with simple one-line examples. You will undoubtedly want to explore these concepts more deeply using online resources. You will quickly outgrow the examples here. We recommend that you master the basics of makefiles as quickly as possible: even the simplest computational research project will benefit enormously from the power and flexibility of a makefile-based build process.

### Intel Compilers

Intel is the recommended and default compiler suite on Stampede2. Each Intel module also gives you direct access to `mkl` without loading an `mkl` module; see [Intel MKL](#) for more information. Here are simple examples that use the Intel compiler to build an executable from source code:

```
$ icc mycode.c                # C source file; executable a.out
$ icc main.c calc.c analyze.c # multiple source files
$ icc mycode.c -o myexe        # C source file; executable myexe
$ icpc mycode.cpp -o myexe     # C++ source file
$ ifort mycode.f90 -o myexe    # Fortran90 source file
```

Compiling a code that uses OpenMP would look like this:

```
$ icc -qopenmp mycode.c -o myexe # OpenMP
```

See the published Intel documentation, available both [online](#) and in `${TACC_INTEL_DIR}/documentation`, for information on optimization flags and other Intel compiler options.

### GNU Compilers

The GNU foundation maintains a number of high quality compilers, including a compiler for C (`gcc`), C++ (`g++`), and Fortran (`gfortran`). The `gcc` compiler is the foundation underneath all three, and the term "gcc" often means the suite of these three GNU compilers.

Load a `gcc` module to access a recent version of the GNU compiler suite. Avoid using the GNU compilers that are available without a `gcc` module — those will be older versions based on the "system gcc" that comes as part of the Linux distribution.

Here are simple examples that use the GNU compilers to produce an executable from source code:

```
$ gcc mycode.c                # C source file; executable a.out
$ gcc mycode.c                -o myexe # C source file; executable myexe
$ g++ mycode.cpp              -o myexe # C++ source file
$ gfortran mycode.f90         -o myexe # Fortran90 source file
$ gcc -fopenmp mycode.c -o myexe # OpenMP; GNU flag is different than Intel
```

Note that some compiler options are the same for both Intel and GNU (e.g. `-o`), while others are different (e.g. `-qopenmp` vs `-fopenmp`). Many options are available in one compiler suite but not the other. See the [online GNU documentation](#) for information on optimization flags and other GNU compiler options.

## Compiling and Linking as Separate Steps

Building an executable requires two separate steps: (1) compiling (generating a binary object file associated with each source file); and (2) linking (combining those object files into a single executable file that also specifies the libraries that executable needs). The examples in the previous section accomplish these two steps in a single call to the compiler. When building more sophisticated applications or libraries, however, it is often necessary or helpful to accomplish these two steps separately.

Use the `-c` ("compile") flag to produce object files from source files:

```
$ icc -c main.c calc.c results.c
```

Barring errors, this command will produce object files `main.o`, `calc.o`, and `results.o`. Syntax for other compilers Intel and GNU compilers is similar.

You can now link the object files to produce an executable file:

```
$ icc main.o calc.o results.o -o myexe
```

The compiler calls a linker utility (usually `/bin/ld`) to accomplish this task. Again, syntax for other compilers is similar.

## Include and Library Paths

Software often depends on pre-compiled binaries called libraries. When this is true, compiling usually requires using the `-I` option to specify paths to so-called header or include files that define interfaces to the procedures and data in those libraries. Similarly, linking often requires using the `-L` option to specify paths to the libraries themselves. Typical compile and link lines might look like this:

```
$ icc          -c main.c -I${WORK}/mylib/inc -I${TACC_HDF5_INC}      # compile
$ icc main.o -o myexe -L${WORK}/mylib/lib -L${TACC_HDF5_LIB} -lmylib -lhdf5 # link
```

On Stampede2, both the `hdf5` and `phdf5` modules define the environment variables `$TACC_HDF5_INC` and `$TACC_HDF5_LIB`. Other module files define similar environment variables; see [Using Modules to Manage Your Environment](#) for more information.

The details of the linking process vary, and order sometimes matters. Much depends on the type of library: static (`.a` suffix; library's binary code becomes part of executable image at link time) versus dynamically-linked shared (`.so` suffix; library's binary code is not part of executable; it's located and loaded into memory at run time). The link line can use `rpath` to store in the executable an explicit path to a shared library. In general, however, the `LD_LIBRARY_PATH` environment variable specifies the search path for dynamic libraries. For software installed at the system-level, TACC's modules generally modify `LD_LIBRARY_PATH` automatically. To see whether and how an executable named `myexe` resolves dependencies on dynamically linked libraries, execute `ldd myexe`.

A separate section below addresses the [Intel Math Kernel Library](#) (MKL).

## Compiling and Linking MPI Programs

Intel MPI (module `impi`) and MVAPICH2 (module `mvapich2`) are the two MPI libraries available on Stampede2. After loading an `impi` or `mvapich2` module, compile and/or link using an mpi wrapper (`mpicc`, `mpicxx`, `mpif90`) in place of the compiler:

```
$ mpicc    mycode.c    -o myexe    # C source, full build
$ mpicc -c mycode.c    # C source, compile without linking
$ mpicxx   mycode.cpp  -o myexe    # C++ source, full build
$ mpif90   mycode.f90  -o myexe    # Fortran source, full build
```

These wrappers call the compiler with the options, include paths, and libraries necessary to produce an MPI executable using the MPI module you're using. To see the effect of a given wrapper, call it with the `-show` option:

```
$ mpicc -show # Show compile line generated by call to mpicc; similarly for other wrappers
```

## Building Third-Party Software in Your Own Account

You're welcome to download third-party research software and install it in your own account. In most cases you'll want to download the source code and build the software so it's compatible with the Stampede2 software environment. You can't use yum or any other installation process that requires elevated privileges, but this is almost never necessary. The key is to specify an installation directory for which you have write permissions. Details vary; you should consult the package's documentation and be prepared to experiment. When using the famous [three-step autotools](#) build process, the standard approach is to use the `PREFIX` environment variable to specify a non-default, user-owned installation directory at the time you execute `configure` or `make`:

```
$ export INSTALLDIR=$WORK/apps/t3pio
$ ./configure --prefix=$INSTALLDIR
$ make
$ make install
```

Other languages, frameworks, and build systems generally have equivalent mechanisms for installing software in user space. In most cases a web search like "Python Linux install local" will get you the information you need.

In Python, a local install will resemble one of the following examples:

```
$ pip install netCDF4 --user          # install netCDF4 package to $HOME/.local
$ python setup.py install --user      # install to $HOME/.local
$ pip install netCDF4 --prefix=$INSTALLDIR # custom location; add to PYTHONPATH
```

Similarly in R:

```
$ module load Rstats          # load TACC's default R
$ R                            # launch R
> install.packages('devtools') # R will prompt for install location
```

You may, of course, need to customize the build process in other ways. It's likely, for example, that you'll need to edit a `makefile` or other build artifacts to specify Stampede2-specific [include and library paths](#) or other compiler settings. A good way to proceed is to write a shell script that implements the entire process: definitions of environment variables, module commands, and calls to the build utilities. Include `echo` statements with appropriate diagnostics. Run the script until you encounter an error. Research and fix the current problem. Document your experience in the script itself; including dead-ends, alternatives, and lessons learned. Re-run the script to get to the next error, then repeat until done. When you're finished, you'll have a repeatable process that you can archive until it's time to update the software or move to a new machine.

If you wish to share a software package with collaborators, you may need to modify file permissions. See [Sharing Files with Collaborators](#) for more information.

## Intel Math Kernel Library (MKL)



The [Intel Math Kernel Library](#) (MKL) is a collection of highly optimized functions implementing some of the most important mathematical kernels used in computational science, including standardized interfaces to:

- [BLAS](#) (Basic Linear Algebra Subroutines), a collection of low-level matrix and vector operations like matrix-matrix multiplication
- [LAPACK](#) (Linear Algebra PACKage), which includes higher-level linear algebra algorithms like Gaussian Elimination
- FFT (Fast Fourier Transform), including interfaces based on [FFTW](#) (Fastest Fourier Transform in the West)
- [ScaLAPACK](#) (Scalable LAPACK), [BLACS](#) (Basic Linear Algebra Communication Subprograms), Cluster FFT, and other functionality that provide block-based distributed memory (multi-node) versions of selected [LAPACK](#), [BLAS](#), and [FFT](#) algorithms;
- [Vector Mathematics](#) (VM) functions that implement highly optimized and vectorized versions of special functions like sine and square root.

## MKL with Intel C, C++, and Fortran Compilers

There is no MKL module for the Intel compilers because you don't need one: the Intel compilers have built-in support for MKL. Unless you have specialized needs, there is no need to specify include paths and libraries explicitly. Instead, using MKL with the Intel modules requires nothing more than compiling and linking with the `-mkl` option.; e.g.

```
$ icc -mkl mycode.c
$ ifort -mkl mycode.c
```

The `-mkl` switch is an abbreviated form of `-mkl=parallel`, which links your code to the threaded version of MKL. To link to the unthreaded version, use `-mkl=sequential`. A third option, `-mkl=cluster`, which also links to the unthreaded libraries, is necessary and appropriate only when using ScaLAPACK or other distributed memory packages. For additional information, including advanced linking options, see the [MKL documentation](#) and [Intel MKL Link Line Advisor](#).

## MKL with GNU C, C++, and Fortran Compilers

When using a GNU compiler, load the MKL module before compiling or running your code, then specify explicitly the MKL libraries, library paths, and include paths your application needs. Consult the [Intel MKL Link Line Advisor](#) for details. A typical compile/link process on a TACC system will look like this:

```
$ module load gcc
$ module load mkl                # available/needed only for GNU compilers
$ gcc -fopenmp -I$MKLR00T/include \
    -Wl,-L$MKLR00T/lib/intel64 \
    -lmkl_intel_lp64 -lmkl_core \
    -lmkl_gnu_thread -lpthread \
    -lm -ldl mycode.c
```

For your convenience the `mkl` module file also provides alternative TACC-defined variables like `$TACC_MKL_INCLUDE` (equivalent to `$MKLR00T/include`). Execute `module help mkl` for more information.

## Using MKL as BLAS/LAPACK with Third-Party Software

When your third-party software requires BLAS or LAPACK, you can use MKL to supply this functionality. Replace generic instructions that include link options like `-lblas` or `-llapack` with the simpler MKL approach described above. There is no need to download and install alternatives like OpenBLAS.

## Using MKL as BLAS/LAPACK with TACC's MATLAB, Python, and R Modules

TACC's MATLAB, Python, and R modules all use threaded (parallel) MKL as their underlying BLAS/LAPACK library. This means that even serial codes written in MATLAB, Python, or R may benefit from MKL's thread-based parallelism. This requires no action on your part other than specifying an appropriate max thread count for MKL; see the section below for more information.

## Controlling Threading in MKL

Any code that calls MKL functions can potentially benefit from MKL's thread-based parallelism; this is true even if your code is not otherwise a parallel application. If you are linking to the threaded MKL (using `-mkl`, `-mkl=parallel`, or the equivalent explicit link line), you need only specify an appropriate value for the max number of threads available to MKL. You can do this with either of the two environment variables `MKL_NUM_THREADS` or `OMP_NUM_THREADS`. The environment variable `MKL_NUM_THREADS` specifies the max number of threads available to each instance of MKL, and has no effect on non-MKL code. If `MKL_NUM_THREADS` is undefined, MKL uses `OMP_NUM_THREADS` to determine the max number of threads available to MKL functions. In either case, MKL will attempt to choose an optimal thread count less than or equal to the specified value. Note that `OMP_NUM_THREADS` defaults to 1 on TACC systems; if you use the default value you will get no thread-based parallelism from MKL.

If you are running a single serial, unthreaded application (or an unthreaded MPI code involving a single MPI task per node) it is usually best to give MKL as much flexibility as possible by setting the max thread count to the total number of hardware threads on the node (272 on KNL, 96 on SKX, 160 on ICX). Of course things are more complicated if you are running more than one process on a node: e.g. multiple serial processes, threaded applications, hybrid MPI-threaded applications, or pure MPI codes running more than one MPI rank per node. See <http://software.intel.com/en-us/articles/recommended-settings-for-calling-intel-mkl-routines-from-multi-threaded-applications> and related Intel resources for examples of how to manage threading when calling MKL from multiple processes.

## Using ScaLAPACK, Cluster FFT, and Other MKL Cluster Capabilities

See "Working with the Intel Math Kernel Library Cluster Software" and "Intel MKL Link Line Advisor" for information on linking to the MKL cluster components.

## Building for Performance on Stampede2

### Compiler

When building software on Stampede2, we recommend using the most recent Intel compiler and Intel MPI library available on Stampede2. The most recent versions may be newer than the defaults. Execute `module spider intel` and `module spider impi` to see what's installed. When loading these modules you may need to specify version numbers explicitly (e.g. `module load intel/18.0.0` and `module load impi/18.0.0`).

### Architecture-Specific Flags

To compile for KNL only, include `-xMIC-AVX512` as a build option. The `-x` switch allows you to specify a [target architecture](#), while `MIC-AVX512` is the KNL-specific subset of Intel's Advanced Vector Extensions 512-bit [instruction set](#). Besides all other appropriate compiler options, you should also consider specifying an [optimization level](#) using the `-O` flag:

```
$ gcc -xMIC-AVX512 -O3 mycode.c -o myexe # will run only on KNL
```

Similarly, to build for SKX or ICX, specify the `CORE-AVX512` instruction set, which is native to SKX and ICX:

```
$ ifort -xCORE-AVX512 -O3 mycode.f90 -o myexe # will run on SKX or ICX
```

Because Stampede2 has two kinds of compute nodes, however, we recommend a more flexible approach when building with the Intel compiler: use [CPU dispatch](#) to build a multi-architecture ("fat") binary that contains alternate code paths with optimized vector code for each type of Stampede2 node. To produce a multi-architecture binary for Stampede2, build with the following options:

```
-xCORE-AVX2 -xCORE-AVX512,MIC-AVX512
```

These particular choices allow you to build on any Stampede2 node (KNL, SKX and ICX nodes), and use [CPU dispatch](#) to produce a multi-architecture binary. We recommend that you specify these flags in both the compile and link steps. Specify an optimization level (e.g. `-O3`) along with any other appropriate compiler switches:

```
$ icc -xCORE-AVX2 -axCORE-AVX512,MIC-AVX512 -O3 mycode.c -o myexe
```

The `-x` option is the target base architecture (instruction set). The base instruction set must run on all targeted processors. Here we specify `CORE-AVX2`, which is native for older Broadwell processors and supported on all KNL, SKX and ICX node. This option allows configure scripts and similar build systems to run test executables on any Stampede2 login or compute node. The `-ax` option is a comma-separated list of alternate instruction sets: `CORE-AVX512` for SKX and ICX, and `MIC-AVX512` for KNL.

Now that we have replaced the original Broadwell login nodes with newer Skylake login nodes, `-xCORE-AVX2` remains a reasonable (though conservative) base option. Another plausible, more aggressive base option is `-xCOMMON-AVX512`, which is a subset of `AVX512` that runs on all KNL, SKX and ICX node.

**It's best to avoid building with `-xHost`** (a flag that means "optimize for the architecture on which I'm compiling now"). Using `-xHost` on a SKX login node, for example, will result in a binary that won't run on KNL.

Don't skip the `-x` flag in a multi-architecture build: the default is the very old SSE2 (Pentium 4) instruction set. **Don't create a multi-architecture build with a base option of either `-xMIC-AVX512` (native on KNL) or `-xCORE-AVX512` (native on SKX/ICX);** there are no meaningful, compatible alternate (`-ax`) instruction sets:

```
$ icc -xCORE-AVX512 -axMIC-AVX512 -O3 mycode.c -o myexe      # NO! Base incompatible with alternate
```

On Stampede2, the module files for newer Intel compilers (Intel 18.0.0 and later) define the environment variable `TACC_VEC_FLAGS` that stores the recommended architecture flags described above. This can simplify your builds:

```
$ echo $TACC_VEC_FLAGS                                # env variable available only for intel/18.0.0 and later
-xCORE-AVX2 -axCORE-AVX512,MIC-AVX512
$ icc $TACC_VEC_FLAGS -O3 mycode.c -o myexe
```

Simplicity is a major advantage of this multi-architecture approach: it allows you to build and run anywhere on Stampede2, and performance is generally comparable to single-architecture builds. There are some trade-offs to consider, however. This approach will take a little longer to compile than single-architecture builds, and will produce a larger binary. In some cases, you might also pay a small performance penalty over single-architecture approaches. For more information see the [Intel documentation](#).

For information on the performance implications of your choice of build flags, see the sections on Programming and Performance for [KNL](#) and [SKX and ICX](#) respectively.

If you use GNU compilers, see [GNU x86 Options](#) for information regarding support for KNL, SKX and ICX. Note that GNU compilers do not support multi-architecture binaries.

## Running Jobs

### Job Accounting

Like all TACC systems, Stampede2's accounting system is based on node-hours: one unadjusted Service Unit (SU) represents a single compute node used for one hour (a node-hour). For any given job, the total cost in SUs is the use of one compute node for one hour of wall clock time plus any charges or discounts for the use of specialized queues, e.g. Frontera's `flex` queue, Stampede2's `development` queue, and Longhorn's `v100` queue. The [queue charge rates](#) are determined by the supply and demand for that particular queue or type of node used and are subject to change.

**Stampede2 SUs billed = (# nodes) x (job duration in wall clock hours) x (charge rate per node-hour)**

The Slurm scheduler tracks and charges for usage to a granularity of a few seconds of wall clock time. **The system charges only for the resources you actually use, not those you request.** If your job finishes early and exits properly, Slurm will release the nodes back into the pool of available nodes. Your job will only be charged for as long as you are using the nodes.



TACC does not implement node-sharing on any compute resource. Each Stampede2 node can be assigned to only one user at a time; hence a complete node is dedicated to a user's job and accrues wall-clock time for all the node's cores whether or not all cores are used.

Principal Investigators can monitor allocation usage via the [TACC User Portal](#) under "Allocations->Projects and Allocations". Be aware that the figures shown on the portal may lag behind the most recent usage. Projects and allocation balances are also displayed upon command-line login.



### Tip

To display a summary of your TACC project balances and disk quotas at any time, execute:

```
login1$ /usr/local/etc/taccinfo # Generally more current than balances displayed on the portals.
```

## Slurm Job Scheduler

Stampede2's job scheduler is the [Slurm Workload Manager](#). Slurm commands enable you to submit, manage, monitor, and control your jobs.

## Slurm Partitions (Queues)

Currently available queues include those in [Stampede2 Production Queues](#). See [KNL Compute Nodes](#), [SKX Compute Nodes](#), [Memory Modes](#), and [Cluster Modes](#) for more information on node types.

**Table 5. Production Queues**

Queue Name	Node Type	Max Nodes per Job (assoc'd cores)*	Max Duration	Max Jobs in Queue *	Charge Rate (per node-hour)
development	KNL cache-quadrant	16 nodes (1,088 cores)*	2 hrs	1*	0.8 Service Unit (SU)
normal	KNL cache-quadrant	256 nodes (17,408 cores) *	48 hrs	50 *	0.8 SU
large **	KNL cache-quadrant	2048 nodes (139,264 cores) **	48 hrs	5 **	0.8 SU
long	KNL cache-quadrant	32 nodes (2,176 cores) *	120 hrs	2 *	0.8 SU
flat-quadrant	KNL flat-quadrant	32 nodes (2,176 cores) *	48 hrs	5 *	0.8 SU
skx-dev	SKX	4 nodes (192 cores) *	2 hrs	1 *	1 SU
skx-normal	SKX	128 nodes (6,144 cores) *	48 hrs	20 *	1 SU
skx-large **	SKX	868 nodes (41,664 cores) *	48 hrs	3 *	1 SU
icx-normal	ICX	40 nodes (3,200 cores) *	48 hrs	20 *	1.67 SU

\* Queue status as of March 7, 2022. **Queues and limits are subject to change without notice.** Execute `qlimits` on Stampede2 for real-time information regarding limits on available queues. See [Monitoring Jobs and Queues](#) for additional

information.

\*\* To request more nodes than are available in the normal queue, [submit a consulting \(help desk\) ticket](#). Include in your request reasonable evidence of your readiness to run under the conditions you're requesting. In most cases this should include your own strong or weak scaling results from Stampede2.

\*\*\* For non-hybrid memory-cluster modes or other special requirements, [submit a consulting ticket](#).

## Submitting Batch Jobs with `sbatch`

Use Slurm's `sbatch` command to [submit a batch job](#) to one of the Stampede2 queues:

```
login1$ sbatch myjobscript
```

Here `myjobscript` is the name of a text file containing `#SBATCH` directives and shell commands that describe the particulars of the job you are submitting. The details of your job script's contents depend on the type of job you intend to run.

In your job script you (1) use `#SBATCH` directives to request computing resources (e.g. 10 nodes for 2 hrs); and then (2) use shell commands to specify what work you're going to do once your job begins. There are many possibilities: you might elect to launch a single application, or you might want to accomplish several steps in a workflow. You may even choose to launch more than one application at the same time. The details will vary, and there are many possibilities. But your own job script will probably include at least one launch line that is a variation of one of the examples described here.

Your job will run in the environment it inherits at submission time; this environment includes the modules you have loaded and the current working directory. In most cases you should **run your applications(s) after loading the same modules that you used to build them**. You can of course use your job submission script to modify this environment by defining new environment variables; changing the values of existing environment variables; loading or unloading modules; changing directory; or specifying relative or absolute paths to files. **Do not use the Slurm `--export` option to manage your job's environment**: doing so can interfere with the way the system propagates the inherited environment.

The [Common `sbatch` Options table](#) below describes some of the most common `sbatch` command options. Slurm directives begin with `#SBATCH`; most have a short form (e.g. `-N`) and a long form (e.g. `--nodes`). You can pass options to `sbatch` using either the command line or job script; most users find that the job script is the easier approach. The first line of your job script must specify the interpreter that will parse non-Slurm commands; in most cases `#!/bin/bash` or `#!/bin/csh` is the right choice. Avoid `#!/bin/sh` (its startup behavior can lead to subtle problems on Stampede2), and do not include comments or any other characters on this first line. All `#SBATCH` directives must precede all shell commands. Note also that certain `#SBATCH` options or combinations of options are mandatory, while others are not available on Stampede2.

**Table 6. Common `sbatch` Options**

Option	Argument	Comments
<code>-p</code>	<code>queue_name</code>	Submits to queue (partition) designated by <code>queue_name</code>
<code>-J</code>	<code>job_name</code>	Job Name
<code>-N</code>	<code>total_nodes</code>	Required. Define the resources you need by specifying either: (1) <code>-N</code> and <code>-n</code> ; or (2) <code>-N</code> and <code>--tasks-per-node</code> .
<code>-n</code>	<code>total_tasks</code>	This is total MPI tasks in this job. See <code>-N</code> above for a good way to use this option. When using this option in a non-MPI job, it is usually best to set it to the same value as <code>-N</code> .
<code>--tasks-per-node</code> or <code>--ntasks-per-node</code>	<code>tasks_per_node</code>	This is MPI tasks per node. See <code>-N</code> above for a good way to use this option. When using this option in a non-MPI job, it is usually best to set <code>--tasks-per-node</code> to 1.

Option	Argument	Comments
-t	hh:mm:ss	Required. Wall clock time for job.
--mail-user=	email_address	Specify the email address to use for notifications. Use with the --mail-type= flag below.
--mail-type=	begin, end, fail, or all	Specify when user notifications are to be sent (one option per line).
-o	output_file	Direct job standard output to output_file (without -e option error goes to this file)
-e	error_file	Direct job error output to error_file
-d=	afterok:jobid	Specifies a dependency: this run will start only after the specified job (jobid) successfully finishes
-A	projectnumber	Charge job to the specified project/allocation number. This option is only necessary for logins associated with multiple projects.
-a or --array	N/A	Not available. Use the launcher module for parameter sweeps and other collections of related serial jobs.
--mem	N/A	Not available. If you attempt to use this option, the scheduler will not accept your job.
--export=	N/A	Avoid this option on Stampede2. Using it is rarely necessary and can interfere with the way the system propagates your environment.

By default, Slurm writes all console output to a file named `slurm-%j.out`, where `%j` is the numerical job ID. To specify a different filename use the `-o` option. To save `stdout` (standard out) and `stderr` (standard error) to separate files, specify both `-o` and `-e`.

## Launching Applications

The primary purpose of your job script is to launch your research application. How you do so depends on several factors, especially (1) the type of application (e.g. MPI, OpenMP, serial), and (2) what you're trying to accomplish (e.g. launch a single instance, complete several steps in a workflow, run several applications simultaneously within the same job). While there are many possibilities, your own job script will probably include a launch line that is a variation of one of the examples described in this section:

- [Launching One Serial Application](#)
- [Launching One Multi-Threaded Application](#)
- [Launching One MPI Application](#)
- [Launching One Hybrid \(MPI+Threads\) Application](#)
- [More Than One Serial Application in the Same Job](#)
- [More than One MPI Application Running Concurrently](#)
- [More than One OpenMP Application Running Concurrently](#)

### Launching One Serial Application

To launch a serial application, simply call the executable. Specify the path to the executable in either the PATH environment variable or in the call to the executable itself:

```
myprogram          # executable in a directory listed in $PATH
$WORK/apps/myprov/myprogram # explicit full path to executable
./myprogram        # executable in current directory
./myprogram -m -k 6 input1 # executable with notional input options
```



## Launching One Multi-Threaded Application

Launch a threaded application the same way. Be sure to specify the number of threads. **Note that the default OpenMP thread count is 1.**

```
export OMP_NUM_THREADS=68    # 68 total OpenMP threads (1 per KNL core)
./myprogram
```

## Launching One MPI Application

To launch an MPI application, use the TACC-specific MPI launcher `ibrun`, which is a Stampede2-aware replacement for generic MPI launchers like `mpirun` and `mpiexec`. In most cases the only arguments you need are the name of your executable followed by any arguments your executable needs. When you call `ibrun` without other arguments, your Slurm `#SBATCH` directives will determine the number of ranks (MPI tasks) and number of nodes on which your program runs.

```
#SBATCH -N 5
#SBATCH -n 200
ibrun ./myprogram           # ibrun uses the $SBATCH directives to properly allocate nodes and tasks
```

To use `ibrun` interactively, say within an `idev` session, you can specify:

```
login1$ idev -N 2 -n 80
c123-456$ ibrun ./myprogram    # ibrun uses idev's arguments to properly allocate nodes and tasks
```

## Launching One Hybrid (MPI+Threads) Application

When launching a single application you generally don't need to worry about affinity: both Intel MPI and MVAPICH2 will distribute and pin tasks and threads in a sensible way.

```
export OMP_NUM_THREADS=8    # 8 OpenMP threads per MPI rank
ibrun ./myprogram           # use ibrun instead of mpirun or mpiexec
```

As a practical guideline, the product of `$OMP_NUM_THREADS` and the maximum number of MPI processes per node should not be greater than total number of cores available per node (KNL nodes have 68 cores, SKX nodes have 48 cores, ICX nodes have 80 cores).

## More Than One Serial Application in the Same Job

TACC's `launcher` utility provides an easy way to launch more than one serial application in a single job. This is a great way to engage in a popular form of High Throughput Computing: running parameter sweeps (one serial application against many different input datasets) on several nodes simultaneously. The `launcher` utility will execute your specified list of independent serial commands, distributing the tasks evenly, pinning them to specific cores, and scheduling them to keep cores busy. Execute `module load launcher` followed by `module help launcher` for more information.

## MPI Applications One at a Time

To run one MPI application after another (or any sequence of commands one at a time), simply list them in your job script in the order in which you'd like them to execute. When one application/command completes, the next one will begin.

```
module load git
module list
./preprocess.sh
ibrun ./myprogram input1    # runs after preprocess.sh completes
ibrun ./myprogram input2    # runs after previous MPI app completes
```

## More than One MPI Application Running Concurrently

To run more than one MPI application simultaneously in the same job, you need to do several things:

- use ampersands to launch each instance in the background;
- include a `wait` command to pause the job script until the background tasks complete;
- use the `ibrun -n` and `-o` switches to specify task counts and hostlist offsets respectively; and
- include a call to the `task_affinity` script in your `ibrun` launch line.

If, for example, you use `#SBATCH` directives to request  $N=4$  nodes and  $n=128$  total MPI tasks, Slurm will generate a hostfile with 128 entries (32 entries for each of 4 nodes). The `-n` and `-o` switches, which must be used together, determine which hostfile entries `ibrun` uses to launch a given application; execute `ibrun --help` for more information. **Don't forget the ampersands (`&`)** to launch the jobs in the background, **and the `wait` command** to pause the script until the background tasks complete:

```
ibrun -n 64 -o 0 task_affinity ./myprogram input1 & # 64 tasks; offset by 0 entries in hostfile.
ibrun -n 64 -o 64 task_affinity ./myprogram input2 & # 64 tasks; offset by 64 entries in hostfile.
wait # Required; else script will exit immediately.
```

The `task_affinity` script does two things:

- `task_affinity` manages task placement and pinning when you call `ibrun` with the `-n`, `-o` switches (it's not necessary under any other circumstances); and
- `task_affinity` also manages MCDRAM when you run in flat-quadrant mode on the KNL. It does this in the same way as `mem_affinity`.
- **Don't confuse `task_affinity` with `tacc_affinity`**; the keyword `tacc_affinity` is now a symbolic link to `mem_affinity`. The `mem_affinity` script and the symbolic link `tacc_affinity` manage MCDRAM in flat-quadrant mode on the KNL, but they do not pin MPI tasks.

## More than One OpenMP Application Running Concurrently

You can also run more than one OpenMP application simultaneously on a single node, but you will need to distribute and pin tasks appropriately. In the example below, `numactl -C` specifies virtual CPUs (hardware threads). According to the numbering scheme for KNL hardware threads, CPU (hardware thread) numbers 0-67 are spread across the 68 cores, 1 thread per core. Similarly for SKX: CPU (hardware thread) numbers 0-47 are spread across the 48 cores, 1 thread per core, and for ICX: CPU (hardware thread) numbers 0-79 are spread across the 80 cores, 1 thread per core.

```
export OMP_NUM_THREADS=2
numactl -C 0-1 ./myprogram inputfile1 & # HW threads (hence cores) 0-1. Note ampersand.
numactl -C 2-3 ./myprogram inputfile2 & # HW threads (hence cores) 2-3. Note ampersand.
wait
```

## Interactive Sessions with `idev` and `srn`

TACC's own `idev` utility is the best way to begin an interactive session on one or more compute nodes. To launch a thirty-minute session on a single node in the development queue, simply execute:

```
login1$ idev
```

You'll then see output that includes the following excerpts:

```
...
-----
Welcome to the Stampede2 Supercomputer
-----
...

-> After your idev job begins to run, a command prompt will appear,
-> and you can begin your interactive development session.
```

```
-> We will report the job status every 4 seconds: (PD=pending, R=running).

->job status: PD
->job status: PD
...
c449-001$
```

The `job status` messages indicate that your interactive session is waiting in the queue. When your session begins, you'll see a command prompt on a compute node (in this case, the node with hostname c449-001). If this is the first time you launch `idev`, the prompts may invite you to choose a default project and a default number of tasks per node for future `idev` sessions.

For command line options and other information, execute `idev --help`. It's easy to tailor your submission request (e.g. shorter or longer duration) using Slurm-like syntax:

```
login1$ idev -p normal -N 2 -n 8 -m 150 # normal queue, 2 nodes, 8 total tasks, 150 minutes
```

For more information see the [idev documentation](#).

You can also launch an interactive session with Slurm's `srun` command, though there's no clear reason to prefer `srun` to `idev`. A typical launch line would look like this:

```
login1$ srun --pty -N 2 -n 8 -t 2:30:00 -p normal /bin/bash -l # same conditions as above
```

## Interactive Sessions using `ssh`

If you have a batch job or interactive session running on a compute node, you "own the node": you can connect via `ssh` to open a new interactive session on that node. This is an especially convenient way to monitor your applications' progress. One particularly helpful example: login to a compute node that you own, execute `top`, then press the "1" key to see a display that allows you to monitor thread ("CPU") and memory use.

There are many ways to determine the nodes on which you are running a job, including feedback messages following your `sbatch` submission, the compute node command prompt in an `idev` session, and the `squeue` or `showq` utilities. The sequence of identifying your compute node then connecting to it would look like this:

```
login1$ squeue -u bjones
JOBID      PARTITION   NAME     USER ST       TIME  NODES NODELIST(REASON)
858811     development idv46796 bjones  R         0:39      1 c448-004
login1$ ssh c448-004
...
C448-004$
```

## SLURM Environment Variables

Be sure to distinguish between internal Slurm replacement symbols (e.g. `%j` described above) and Linux environment variables defined by Slurm (e.g. `SLURM_JOBID`). Execute `env | grep SLURM` from within your job script to see the full list of Slurm environment variables and their values. You can use Slurm replacement symbols like `%j` only to construct a Slurm filename pattern; they are not meaningful to your Linux shell. Conversely, you can use Slurm environment variables in the shell portion of your job script but not in an `#SBATCH` directive.



### Danger

For example, the following directive will not work the way you might think:

```
#SBATCH -o myMPI.o${SLURM_JOB_ID} # incorrect
```



### Hint

Instead, use the following directive:

```
#SBATCH -o myMPI.0%j      # "%j" expands to your job's numerical job ID
```

Similarly, you cannot use paths like `$WORK` or `$SCRATCH` in an `#SBATCH` directive.

For more information on this and other matters related to Slurm job submission, see the [Slurm online documentation](#); the man pages for both Slurm itself (`man slurm`) and its individual command (e.g. `man sbatch`); as well as numerous other online resources.

## Job Scripts

This section provides sample Slurm job scripts for each Stampede2 node type: Knight's Landing (KNL), Sky Lake (SKX) and Ice Lake (ICX) nodes. Each section also contains sample scripts for serial, MPI, OpenMP and hybrid (MPI + OpenMP) programming models. Copy and customize each script for your own applications.

### KNL Nodes

Click on a tab for a customizable job-script.

[Serial Job in Normal Queue](#)

[MPI Job in Normal Queue](#)

[OpenMP Job in Normal Queue](#)

[Hybrid Job in Normal Queue](#)

### SKX Nodes

Click on a tab for a customizable job-script.

[Serial Job in Normal Queue](#)

[MPI Job in Normal Queue](#)

[OpenMP Job in Normal Queue](#)

[Hybrid Job in Normal Queue](#)

### ICX Nodes

Click on a tab for a customizable job-script.

[MPI Job in Normal Queue](#)

[OpenMP Job in Normal Queue](#)

[Hybrid Job in Normal Queue](#)

## Monitoring Jobs and Queues

Several commands are available to help you plan and track your job submissions as well as check the status of the Slurm queues.

When interpreting queue and job status, remember that **Stampede2 doesn't operate on a first-come-first-served basis**. Instead, the sophisticated, tunable algorithms built into Slurm attempt to keep the system busy, while scheduling jobs in a way that is as fair as possible to everyone. At times this means leaving nodes idle ("draining the queue") to make room for a large job that would otherwise never run. It also means considering each user's "fair share", scheduling jobs so that those who haven't run jobs recently may have a slightly higher priority than those who have.

## Monitoring Queue Status with `sinfo` and `qlimits`

To display resource limits for the Stampede2 queues, execute "`qlimits`". The result is real-time data; the corresponding information in this document's [table of Stampede2 queues](#) may lag behind the actual configuration that the `qlimits` utility displays.

Slurm's "`sinfo`" command allows you to monitor the status of the queues. If you execute `sinfo` without arguments, you'll see a list of every node in the system together with its status. To skip the node list and produce a tight, alphabetized summary of the available queues and their status, execute:

```
login1$ sinfo -S+P -o "%18P %8a %20F"    # compact summary of queue status
```

An excerpt from this command's output looks like this:

PARTITION	AVAIL	NODES(A/I/O/T)
development*	up	41/70/1/112
normal	up	3685/8/3/3696

The `AVAIL` column displays the overall status of each queue (up or down), while the column labeled `NODES(A/I/O/T)` shows the number of nodes in each of several states ("**A**llocated", "**I**dle", "**O**ffline", and "**T**otal"). Execute `man sinfo` for more information. Use caution when reading the generic documentation, however: some available fields are not meaningful or are misleading on Stampede2 (e.g. `TIMELIMIT`, displayed using the `%l` option).

## Monitoring Job Status with `squeue`

Slurm's `squeue` command allows you to monitor jobs in the queues, whether pending (waiting) or currently running:

```
login1$ squeue          # show all jobs in all queues
login1$ squeue -u bjones # show all jobs owned by bjones
login1$ man squeue      # more info
```

An excerpt from the default output looks like this:

JOBID	PARTITION	NAME	USER	ST	TIME	NODES	NODELIST(REASON)
170361	normal	spec12	bjones	PD	0:00	32	(Resources)
170356	normal	mal2d	slindsey	PD	0:00	30	(Priority)
170204	normal	rr2-a2	tg123456	PD	0:00	1	(Dependency)
170250	development	idv59074	aturing	R	29:30	1	c455-044
169669	normal	04-99a1	aturing	CG	2:47:47	1	c425-003

The column labeled `ST` displays each job's status:

- `PD` means "Pending" (waiting);
- `R` means "Running";
- `CG` means "Completing" (cleaning up after exiting the job script).

Pending jobs appear in order of decreasing priority. The last column includes a nodelist for running/completing jobs, or a reason for pending jobs. If you submit a job before a scheduled system maintenance period, and the job cannot complete before the maintenance begins, your job will run when the maintenance/reservation concludes. The `squeue` command will report `ReqNodeNotAvailable` ("Required Node Not Available"). The job will remain in the `PD` state until Stampede2 returns to production.

The default format for `squeue` now reports total nodes associated with a job rather than cores, tasks, or hardware threads. One reason for this change is clarity: the operating system sees each KNL node's 272 hardware threads (and each SKX node's 96 hardware threads) as "processors", and output based on that information can be ambiguous or otherwise difficult to interpret.

The default format lists all nodes assigned to displayed jobs; this can make the output difficult to read. A handy variation that suppresses the nodelist is:

```
login1$ squeue -o "%.10i %.12P %.12j %.9u %.2t %.9M %.6D" # suppress nodelist
```

The `--start` option displays job start times, including very rough estimates for the expected start times of some pending jobs that are relatively high in the queue:

```
login1$ squeue --start -j 167635 # display estimated start time for job 167635
```

## Monitoring Job Status with `showq`

TACC's `showq` utility mimics a tool that originated in the PBS project, and serves as a popular alternative to the Slurm `squeue` command:

```
login1$ showq          # show all jobs; default format
login1$ showq -u        # show your own jobs
login1$ showq -U bjones # show jobs associated with user bjones
login1$ showq -h        # more info
```

The output groups jobs in four categories: `ACTIVE`, `WAITING`, `BLOCKED`, and `COMPLETING/ERRORED`. A `BLOCKED` job is one that cannot yet run due to temporary circumstances (e.g. a pending maintenance or other large reservation.).

If your waiting job cannot complete before a maintenance/reservation begins, `showq` will display its state as "`WaitNod`" ("Waiting for Nodes"). The job will remain in this state until Stampede2 returns to production.

The default format for `showq` now reports total nodes associated with a job rather than cores, tasks, or hardware threads. One reason for this change is clarity: the operating system sees each KNL node's 272 hardware threads (and each SKX node's 96 hardware threads) as "processors", and output based on that information can be ambiguous or otherwise difficult to interpret.

## Other Job Management Commands (`scancel`, `scontrol`, and `sacct`)

**It's not possible to add resources to a job (e.g. allow more time)** once you've submitted the job to the queue.

To **cancel** a pending or running job, first determine its jobid, then use `scancel`:

```
login1$ squeue -u bjones # one way to determine jobid
JOBID  PARTITION  NAME      USER ST      TIME  NODES NODELIST(REASON)
170361  normal    spec12    bjones PD      0:00    32 (Resources)
login1$ scancel 170361  # cancel job
```

For **detailed information** about the configuration of a specific job, use `scontrol`:

```
login1$ scontrol show job=170361
```

To view some **accounting data** associated with your own jobs, use `sacct`:

```
login1$ sacct --starttime 2017-08-01 # show jobs that started on or after this date
```

## Dependent Jobs using `sbatch`

You can use `sbatch` to help manage workflows that involve multiple steps: the `--dependency` option allows you to launch jobs that depend on the completion (or successful completion) of another job. For example you could use this technique to split into three jobs a workflow that requires you to (1) compile on a single node; then (2) compute on 40 nodes; then finally (3) post-process your results using 4 nodes.



```
login1$ sbatch --dependency=afterok:173210 myjobscript
```

For more information see the [Slurm online documentation](#). Note that you can use `$SLURM_JOBID` from one job to find the jobid you'll need to construct the `sbatch` launch line for a subsequent one. But also remember that you can't use `sbatch` to submit a job from a compute node.

## Visualization and VNC Sessions

Stampede2 uses the SKX and KNL processors for all visualization and rendering operations. We use the Intel OpenSWR library to render raster graphics with OpenGL, and the Intel OSPRay framework for ray traced images inside visualization software. **On Stampede2, `swr` replaces `vglrun` (e.g. `swr glxgears`) and uses similar syntax.** OpenSWR can be loaded by executing `module load swr`. We expect most users will notice little difference in visualization experience on KNL. MCDRAM may improve visualization performance for some users. SKX nodes may provide better interactivity for intensive rendering applications.

There is currently no separate visualization queue on Stampede2. All visualization apps are available on all nodes. VNC and DCV sessions are available on any queue, either through the command line or via the [TACC Analysis Portal](#). We recommend submitting to the `development` queue (for KNL) or the `skx-dev` queue (for SKX) for interactive sessions. If you are interested in an application that is not yet available, please submit a help desk ticket through the TACC User Portal.

## Remote Desktop Access

Remote desktop access to Stampede2 is formed through a VNC connection to one or more visualization nodes. Users must first connect to a Stampede2 login node (see System Access) and submit a special interactive batch job that:

- allocates a set of Stampede2 visualization nodes
- starts a vncserver process on the first allocated node
- sets up a tunnel through the login node to the vncserver access port

Once the vncserver process is running on the visualization node and a tunnel through the login node is created, an output message identifies the access port for connecting a VNC viewer. A VNC viewer application is run on the user's remote system and presents the desktop to the user.

### Tip



If this is your first time connecting to Stampede2, you must run `vncpasswd` to create a password for your VNC servers. This should NOT be your login password! This mechanism only deters unauthorized connections; it is not fully secure, as only the first eight characters of the password are saved. All VNC connections are tunneled through SSH for extra security, as described below.

Follow the steps below to start an interactive session.

### 1. Start a Remote Desktop

TACC has provided a VNC job script (`/share/doc/slurm/job.vnc`) that requests one node in the `development` queue for two hours, creating a [VNC](#) session.

```
login1$ sbatch /share/doc/slurm/job.vnc
```

You may modify or overwrite script defaults with `sbatch` command-line options:

- `-t hours:minutes:seconds` modify the job runtime
- `-A projectnumber` specify the project/allocation to be charged
- `-N nodes` specify number of nodes needed
- `-p partition` specify an alternate queue.

See more `sbatch` options in the [Common `sbatch` Options](#) table above.

All arguments after the job script name are sent to the `vncserver` command. For example, to set the desktop resolution to 1440x900, use:

```
login1$ sbatch /share/doc/slurm/job.vnc -geometry 1440x900
```

The `vnc.job` script starts a `vncserver` process and writes to the output file, `vncserver.out` in the job submission directory, with the connect port for the `vncviewer`. Watch for the "To connect via VNC client" message at the end of the output file, or watch the output stream in a separate window with the commands:

```
login1$ touch vncserver.out ; tail -f vncserver.out
```

The lightweight window manager, `xfce`, is the default VNC desktop and is recommended for remote performance. Gnome is available; to use gnome, open the `~/ .vnc/xstartup` file (created after your first VNC session) and replace `startxfce4` with `gnome-session`. Note that gnome may lag over slow internet connections.

## 2. Create an SSH Tunnel to Stampede2

TACC requires users to create an SSH tunnel from the local system to the Stampede2 login node to assure that the connection is secure. The tunnels created for the VNC job operate only on the `localhost` interface, so you must use `localhost` in the port forward argument, not the Stampede2 hostname. On a Unix or Linux system, execute the following command once the port has been opened on the Stampede2 login node:

```
localhost$ ssh -f -N -L xxxx:localhost:yyyy username@stampede2.tacc.utexas.edu
```

where:

- `yyyy` is the port number given by the `vncserver` batch job
- `xxxx` is a port on the remote system. Generally, the port number specified on the Stampede2 login node, `yyyy`, is a good choice to use on your local system as well
- `-f` instructs SSH to only forward ports, not to execute a remote command
- `-N` puts the `ssh` command into the background after connecting
- `-L` forwards the port

On Windows systems find the menu in the Windows SSH client where tunnels can be specified, and enter the local and remote ports as required, then `ssh` to Stampede2.

## 3. Connecting vncviewer

Once the SSH tunnel has been established, use a [VNC client](#) to connect to the local port you created, which will then be tunneled to your VNC server on Stampede2. Connect to `localhost:xxxx`, where `xxxx` is the local port you used for your tunnel. In the examples above, we would connect the VNC client to `localhost:xxxx`. (Some VNC clients accept `localhost:xxxx`).

We recommend the [TigerVNC](#) VNC Client, a platform independent client/server application.

Once the desktop has been established, two initial xterm windows are presented (which may be overlapping). One, which is white-on-black, manages the lifetime of the VNC server process. Killing this window (typically by typing `exit` or `ctrl-D` at the prompt) will cause the `vncserver` to terminate and the original batch job to end. Because of this, we recommend that this window not be used for other purposes; it is just too easy to accidentally kill it and terminate the session.

The other xterm window is black-on-white, and can be used to start both serial programs running on the node hosting the vncserver process, or parallel jobs running across the set of cores associated with the original batch job. Additional xterm windows can be created using the window-manager left-button menu.

## Running Applications on the VNC Desktop

From an interactive desktop, applications can be run from icons or from xterm command prompts. Two special cases arise: running parallel applications, and running applications that use OpenGL.

## Running Parallel Applications from the Desktop

Parallel applications are run on the desktop using the same `ibrun` wrapper described above (see [Running](#)). The command:

```
c442-001$ ibrun ibrunoptions application applicationoptions
```

will run application on the associated nodes, as modified by the `ibrun` options.

## Running OpenGL/X Applications On The Desktop

Stampede2 uses the OpenSWR OpenGL library to perform efficient rendering. At present, the compute nodes on Stampede2 do not support native X instances. All windowing environments should use a VNC desktop launched via the job script in `/share/doc/slurm/job.vnc` or using the [TACC Analysis Portal](#).

swr: To access the accelerated OpenSWR OpenGL library, it is necessary to use the swr module to point to the swr OpenGL implementation and configure the number of threads to allocate to rendering.

```
c442-001$ module load swr
c442-001$ swr options application application-args
```

## Parallel VisIt on Stampede2

[VisIt](#) was compiled under the Intel compiler and the mvapich2 and MPI stacks.

After connecting to a VNC server on Stampede2, as described above, load the VisIt module at the beginning of your interactive session before launching the VisIt application:

```
c442-001$ module load swr visit
c442-001$ swr visit
```

VisIt first loads a dataset and presents a dialog allowing for selecting either a serial or parallel engine. Select the parallel engine. Note that this dialog will also present options for the number of processes to start and the number of nodes to use; these options are actually ignored in favor of the options specified when the VNC server job was started.

## Preparing Data for Parallel VisIt

VisIt reads [nearly 150 data formats](#). Except in some limited circumstances (particle or rectilinear meshes in ADIOS, basic netCDF, Pixie, OpenPMD and a few other formats), VisIt piggy-backs its parallel processing off of whatever static parallel decomposition is used by the data producer. This means that VisIt expects the data to be explicitly partitioned into independent subsets (typically distributed over multiple files) at the time of input. Additionally, VisIt supports a metadata file (with a `.visit` extension) that lists multiple data files of any supported format that hold subsets of a larger logical dataset. VisIt also supports a "brick of values" (`bov`) format which supports a simple specification for the static decomposition to use to load data defined on rectilinear meshes. For more information on importing data into VisIt, see [Getting Data Into VisIt](#).

## Parallel ParaView on Stampede2

After connecting to a VNC server on Stampede2, as described above, do the following:

1. Set up your environment with the necessary modules. Load the `swr`, `qt5`, `ospray`, and `paraview` modules **in this order**:

```
c442-001$ module load swr qt5 ospray paraview
```

2. Launch ParaView:

```
c442-001$ swr -p 1 paraview [paraview client options]
```

3. Click the "Connect" button, or select File -> Connect
4. Select the "auto" configuration, then press "Connect". In the Paraview Output Messages window, you'll see what appears to be an 'lmod' error, but can be ignored. Then you'll see the parallel servers being spawned and the connection established.

## Programming and Performance

Programming for performance is a broad and rich topic. While there are no shortcuts, there are certainly some basic principles that are worth considering any time you write or modify code.

### Timing and Profiling

**Measure performance and experiment with both compiler and runtime options.** This will help you gain insight into issues and opportunities, as well as recognize the performance impact of code changes and temporary system conditions.

Measuring performance can be as simple as prepending the shell keyword `time` or the command `perf stat` to your launch line. Both are simple to use and require no code changes. Typical calls look like this:

```
perf stat ./a.out      # report basic performance stats for a.out
time ./a.out           # report the time required to execute a.out
time ibrun ./a.out     # time an MPI code
ibrun time ./a.out     # crude timings for each MPI task (no rank info)
```

As your needs evolve you can add timing intrinsics to your source code to time specific loops or other sections of code. There are many such intrinsics available; some popular choices include `gettimeofday`, `MPI_Wtime` and `omp_get_wtime`. The resolution and overhead associated with each of these timers is on the order of a microsecond.

It can be helpful to compare results with different compiler and runtime options: e.g. with and without [vectorization](#), [threading](#), or [Lustre striping](#). You may also want to learn to use profiling tools like [Intel VTune Amplifier](#) or GNU `gprof`.

### Data Locality

**Appreciate the high cost (performance penalty) of moving data from one node to another**, from disk to RAM, and even from RAM to cache. Write your code to keep data as close to the computation as possible: e.g. in RAM when needed, and on the node that needs it. This means keeping in mind the capacity and characteristics of each level of the memory hierarchy when designing your code and planning your simulations. A simple KNL-specific example illustrates the point: all things being equal, there's a good chance you'll see better performance when you keep your data in the KNL's [fast MCDRAM](#) instead of the slower DDR4.

When possible, best practice also calls for so-called "stride 1 access" -- looping through large, contiguous blocks of data, touching items that are adjacent in memory as the loop proceeds. The goal here is to use "nearby" data that is already in cache rather than going back to main memory (a cache miss) in every loop iteration.

To achieve stride 1 access you need to understand how your program stores its data. Here C and C++ are different than (in fact the opposite of) Fortran. C and C++ are row-major: they store 2d arrays a row at a time, so elements `a[3][4]` and

`a[3][5]` are adjacent in memory. Fortran, on the other hand, is column-major: it stores a column at a time, so elements `a(4,3)` and `a(5,3)` are adjacent in memory. Loops that achieve stride 1 access in the two languages look like this:

Fortran example	C example
<pre>real*8 :: a(m,n), b(m,n), c(m,n) ... ! inner loop strides through col i do i=1,n   do j=1,m     a(j,i)=b(j,i)+c(j,i)   end do end do</pre>	<pre>double a[m][n], b[m][n], c[m][n]; ... // inner loop strides through row i for (i=0;i&lt;m;i++){   for (j=0;j&lt;n;j++){     a[i][j]=b[i][j]+c[i][j];   } }</pre>

## Vectorization

**Give the compiler a chance to produce efficient, [vectorized](#) code.** The compiler can do this best when your inner loops are simple (e.g. no complex logic and a straightforward matrix update like the ones in the examples above), long (many iterations), and avoid complex data structures (e.g. objects). See Intel's note on [Programming Guidelines for Vectorization](#) for a nice summary of the factors that affect the compiler's ability to vectorize loops.

It's often worthwhile to generate [optimization and vectorization reports](#) when using the Intel compiler. This will allow you to see exactly what the compiler did and did not do with each loop, together with reasons why.

## Learning More

The literature on optimization is vast. Some places to begin a systematic study of optimization on Intel processors include: Intel's [Modern Code](#) resources; and the [Intel Optimization Reference Manual](#).

## Programming and Performance: KNL

### Architecture

KNL cores are grouped in pairs; each pair of cores occupies a tile. Since there are 68 cores on each Stampede2 KNL node, each node has 34 active tiles. These 34 active tiles are connected by a two-dimensional mesh interconnect. Each KNL has 2 DDR memory controllers on opposite sides of the chip, each with 3 channels. There are 8 controllers for the fast, on-package MCDRAM, two in each quadrant.

Each core has its own local L1 cache (32KB, data, 32KB instruction) and two 512-bit vector units. Both vector units can execute `AVX512` instructions, but only one can execute legacy vector instructions (`SSE`, `AVX`, and `AVX2`). Therefore, to use both vector units, you must compile with `-xMIC-AVX512`.

Each core can run up to 4 hardware threads. The two cores on a tile share a 1MB L2 cache. Different [cluster modes](#) specify the L2 cache coherence mechanism at the node level.

### Memory Modes

The processor's memory mode determines whether the fast MCDRAM operates as RAM, as direct-mapped L3 cache, or as a mixture of the two. The output of commands like `top`, `free`, and `ps -v` reflect the consequences of memory mode. Such commands will show the amount of RAM available to the operating system, not the hardware (DDR + MCDRAM) installed.



Figure

4. KNL

- **Cache Mode.** In this mode, the fast MCDRAM is configured as an L3 cache. The operating system transparently uses the MCDRAM to move data from main memory. In this mode, the user has access to 96GB of RAM, all of it traditional DDR4. **Most Stampede2 KNL nodes are configured in cache mode.**
- **Flat Mode.** In this mode, DDR4 and MCDRAM act as two distinct Non-Uniform Memory Access (NUMA) nodes. It is therefore possible to specify the type of memory (DDR4 or MCDRAM) when allocating memory. In this mode, the user has access to 112GB of RAM: 96GB of traditional DDR and 16GB of fast MCDRAM. By default, memory allocations occur only in DDR4. To use MCDRAM in flat mode, use the `numactl` utility or the `memkind` library; see [Managing Memory](#) for more information. If you do not modify the default behavior you will have access only to the slower DDR4.
- **Hybrid Mode (not available on Stampede2).** In this mode, the MCDRAM is configured so that a portion acts as L3 cache and the rest as RAM (a second NUMA node supplementing DDR4).

## Cluster Modes

The KNL's core-level L1 and tile-level L2 caches can reduce the time it takes for a core to access the data it needs. To share memory safely, however, there must be mechanisms in place to ensure cache coherency. Cache coherency means that all cores have a consistent view of the data: if data value *x* changes on a given core, there must be no risk of other cores using outdated values of *x*. This, of course, is essential on any multi-core chip, but it is especially difficult to achieve on manycore processors.

The details for KNL are proprietary, but the key idea is this: each tile tracks an assigned range of memory addresses. It does so on behalf of all cores on the chip, maintaining a data structure (tag directory) that tells it which cores are using data from its assigned addresses. Coherence requires both tile-to-tile and tile-to-memory communication. Cores that read or modify data must communicate with the tiles that manage the memory associated with that data. Similarly, when cores need data from main memory, the tile(s) that manage the associated addresses will communicate with the memory controllers on behalf of those cores.

The KNL can do this in several ways, each of which is called a cluster mode. Each cluster mode, specified in the BIOS as a boot-time option, represents a tradeoff between simplicity and control. There are three major cluster modes with a few minor variations:

- **All-to-All.** This is the most flexible and most general mode, intended to work on all possible hardware and memory configurations of the KNL. But this mode also may have higher latencies than other cluster modes because the processor does not attempt to optimize coherency-related communication paths. Stampede2 does not have nodes in this cluster mode.
- **Quadrant (variation: hemisphere).** This is Intel's recommended default, and the cluster mode of most Stampede2 queues. This mode attempts to localize communication without requiring explicit memory management by the programmer/user. It does this by grouping tiles into four logical/virtual (not physical) quadrants, then requiring each tile to manage MCDRAM addresses only in its own quadrant (and DDR addresses in its own half of the chip). This reduces the average number of "hops" that tile-to-memory requests require compared to all-to-all mode, which can reduce latency and congestion on the mesh.
- **Sub-NUMA 4 (variation: Sub-NUMA 2).** This mode, abbreviated **SNC-4**, divides the chip into four NUMA nodes so that it acts like a four-socket processor. SNC-4 aims to optimize coherency-related on-chip communication by confining this communication to a single NUMA node when it is possible to do so. To achieve any performance benefit, this requires explicit manual memory management by the programmer/user (in particular, allocating memory within the NUMA node that will use that memory). Stampede2 does not have nodes in this cluster mode.





TACC's early experience with the KNL suggests that there is little reason to deviate from Intel's recommended default memory and cluster modes. Cache-quadrant tends to be a good choice for almost all workflows; it offers a nice compromise between performance and ease of use for the applications we have tested. Flat-quadrant is the most promising alternative and sometimes offers moderately better performance, especially when memory requirements per node are less than 16GB. We have not yet observed significant performance differences across cluster modes, and our current recommendation is that configurations other than cache-quadrant and flat-quadrant are worth considering only for very specialized needs. For more information see [Managing Memory](#) and [Best Known Practices...](#)

## Managing Memory

By design, any application can run in any memory and cluster mode, and applications always have access to all available RAM. Moreover, regardless of memory and cluster modes, there are no code changes or other manual interventions required to run your application safely. However, there are times when explicit manual memory management is worth considering to improve performance. The Linux `numactl` (pronounced "NUMA Control") utility allows you to specify at runtime where your code should allocate memory.

When running in flat-quadrant mode, launch your code with [simple numactl settings](#) to specify whether memory allocations occur in DDR or MCDRAM. See [TACC Training Materials](#) for additional information.

```
numactl      --membind=0  ./a.out    # launch a.out (non-MPI); use DDR (default)
ibrun numactl --membind=0  ./a.out    # launch a.out (MPI-based); use DDR (default)

numactl      --membind=1  ./a.out    # use only MCDRAM
numactl      --preferred=1 ./a.out    # (RECOMMENDED) MCDRAM if possible; else DDR
numactl      --hardware           # show numactl settings
numactl      --help              # list available numactl options
```

Example: Controlling memory in flat-quadrant mode: `numactl` options

Intel's new `memkind` library adds the ability to manage memory in source code with a special memory allocator for C code and a corresponding attribute for Fortran. This makes possible a level of control over memory allocation down to the level of the individual data element. As this library matures it will likely become an important tool for those who need fine-grained control of memory.

When you're running MPI codes in the flat-quadrant queue, the `mem_affinity` script simplifies memory management by calling `numactl` "under the hood" to make plausible NUMA (Non-Uniform Memory Access) policy choices. For MPI and hybrid applications, the script attempts to ensure that each MPI process uses MCDRAM efficiently. To launch your MPI code with `mem_affinity`, simply place `mem_affinity` immediately after `ibrun`:

```
ibrun mem_affinity a.out
```

It's safe to use `mem_affinity` even when it will have no effect (e.g. cache-quadrant mode). Note that `mem_affinity` and `numactl` cannot be used together.

On Stampede2 the keyword `tacc_affinity` was originally an older name for what is now the `mem_affinity` script. To ensure backward compatibility, `tacc_affinity` is now a symbolic link to `mem_affinity`. Note that `mem_affinity` and the symbolic link `tacc_affinity` do not pin MPI tasks.

## Best Known Practices and Preliminary Observations (KNL)

**Hyperthreading.** It is rarely a good idea to use all 272 hardware threads simultaneously, and it's certainly not the first thing you should try. In most cases it's best to specify no more than 64-68 MPI tasks or independent processes per node,

and 1-2 threads/core. One exception is worth noting: when calling threaded MKL from a serial code, it's safe to set `OMP_NUM_THREADS` or `MKL_NUM_THREADS` to 272. This is because MKL will choose an appropriate thread count less than or equal to the value you specify. See [Controlling Threading in MKL](#) for more information. In any case remember that the default value of `OMP_NUM_THREADS` is 1.

### When measuring KNL performance against traditional processors, compare node-to-node rather than core-to-core.

KNL cores run at lower frequencies than traditional multicore processors. Thus, for a fixed number of MPI tasks and threads, a given simulation may run 2-3x slower on KNL than the same submission ran on Stampede1's Sandy Bridge nodes. A well-designed parallel application, however, should be able to run more tasks and/or threads on a KNL node than is possible on Sandy Bridge. If so, it may exhibit better performance per KNL node than it does on Sandy Bridge.

**General Expectations.** From a pure hardware perspective, a single Stampede2 KNL node could outperform Stampede1's dual socket Sandy Bridge nodes by as much as 6x; this is true for both memory bandwidth-bound and compute-bound codes. This assumes the code is running out of (fast) MCDRAM on nodes configured in flat mode (450 GB/s bandwidth vs 75 GB/s on Sandy Bridge) or using cache-contained workloads on nodes configured in cache mode (memory footprint < 16GB). It also assumes perfect scalability and no latency issues. In practice we have observed application improvements between 1.3x and 5x for several HPC workloads typically run in TACC systems. Codes with poor vectorization or scalability could see much smaller improvements. In terms of network performance, the Omni-Path network provides 100 Gbits per second peak bandwidth, with point-to-point exchange performance measured at over 11 GBytes per second for a single task pair across nodes. Latency values will be higher than those for the Sandy Bridge FDR Infiniband network: on the order of 2-4 microseconds for exchanges across nodes.

**MCDRAM in Flat-Quadrant Mode.** Unless you have specialized needs, we recommend using `mem_affinity` or launching your application with `numactl --preferred=1` when running in flat-quadrant mode (see [Managing Memory](#) above). If you mistakenly use `--membind=1`, only the 16GB of fast MCDRAM will be available. If you mistakenly use `--membind=0`, you will not be able to access fast MCDRAM at all.

**Task Affinity.** If you're running one threaded, MPI, or hybrid application at a time, default affinity settings are usually sensible and often optimal. If you run more than one threaded, MPI, or hybrid application at a time, you'll want to pay attention to affinity. For more information see the appropriate sub-sections under [Launching Applications](#).

**MPI Initialization.** Our preliminary scaling tests with Intel MPI on Stampede2 suggest that the time required to complete MPI initialization scales quadratically with the number of MPI tasks (lower case `-n` in your Slurm submission script) and linearly with the number of nodes (upper case `-N`).

**Tuning the Performance Scaled Messaging (PSM2) Library.** When running on KNL with MVAPICH2, set the environment variable `PSM2_KASSIST_MODE` to the value `none` per the [MVAPICH2 User Guide](#). Do not use this environment variable with IMPI; doing so may degrade performance.

## Programming and Performance: SKX and ICX

**Hyperthreading.** It is rarely a good idea to use all the hardware threads simultaneously, and it's certainly not the first thing you should try. In most cases it's best to specify no more than 48 MPI tasks or independent processes per SKX node (80 per ICX node), and 1-2 threads/core. One exception is worth noting: when calling threaded MKL from a serial code, it's safe to set `OMP_NUM_THREADS` or `MKL_NUM_THREADS` to 96 for SKX or 160 for ICX. This is because MKL will choose an appropriate thread count less than or equal to the value you specify. See [Controlling Threading in MKL](#) for more information. In any case remember that the default value of `OMP_NUM_THREADS` is 1.

**Clock Speed.** The published nominal clock speed of the Stampede2 SKX processors is 2.1GHz and for the ICX processors it is 2.3GHz. But [actual clock speed varies widely](#): it depends on the vector instruction set, number of active cores, and other factors affecting power requirements and temperature limits. At one extreme, a single serial application using the `AVX2` instruction set may run at frequencies approaching 3.7GHz, because it's running on a single core (in fact a single hardware thread). At the other extreme, a large, fully-threaded MKL `dgemm` (a highly vectorized routine in which all cores operate at nearly full throttle) may run at 1.4GHz.

**Vector Optimization and `AVX2`.** In some cases, using the `AVX2` instruction set may produce better performance than `AVX512`. This is largely because cores can run at higher [clock speeds](#) when executing `AVX2` code. To compile for `AVX2`,

replace the [multi-architecture flags](#) described above with the single flag `-xCORE-AVX2`. When you use this flag you will be able to build and run on any Stampede2 node.

**Vector Optimization and 512-Bit ZMM Registers.** If your code can take advantage of wide 512-bit vector registers, you may want to try [compiling for SKX and ICX](#) with (for example):

```
-xCORE-AVX512 -qopt-zmm-usage=high
```

The `qopt-zmm-usage` flag affects the algorithms the compiler uses to decide whether to vectorize a given loop with `AVX512` intrinsics (wide 512-bit registers) or `AVX2` code (256-bit registers). When the flag is set to `-qopt-zmm-usage=low` (the default when compiling for SKX and ICX using `CORE-AVX512`), the compiler will choose `AVX2` code more often; this may or may not be the optimal approach for your application. The `qopt-zmm-usage` flag is available only on Intel compilers newer than 17.0.4. Do not use `$TACC_VEC_FLAGS` when specifying `qopt-zmm-usage`. This is because `$TACC_VEC_FLAGS` specifies `AVX2-CORE` as the base architecture, and the compiler will ignore `qopt-zmm-usage` unless the base target is a variant of `AVX512`. See the recent [Intel white paper](#), the [compiler documentation](#), the compiler man pages, and the notes above for more information.

**Vector Optimization and `COMMON-AVX512`.** We have encountered a few complex packages that currently fail to build or run when compiled with `CORE-AVX512` (native SKX or ICX). In all cases so far, these packages build and run well on all KNL, SNL and ICX when compiled as a single-architecture binary with `-xCOMMON-AVX512`.

**Task Affinity.** If you run one MPI application at a time, the `ibrun` MPI launcher will spread each node's tasks evenly across an SKX or ICX node's two sockets, with consecutive tasks occupying the same socket when possible.

**Hardware Thread Numbering.** Execute `lscpu` or `lstopo` on an SKX or ICX node to see the numbering scheme for hardware threads. Note that hardware thread numbers alternate between the sockets: even numbered threads are on NUMA node 0, while odd numbered threads are on NUMA node 1. Furthermore, the two hardware threads on a given core have thread numbers that differ by exactly 48 for SKX and 80 for ICX (e.g. threads 3 and 51 are on the same core on SKX nodes).

**Tuning the Performance Scaled Messaging (PSM2) Library.** When running on SKX with `MVAPICH2`, setting the environment variable `PSM2_KASSIST_MODE` to the value `none` may or may not improve performance. For more information see the [MVAPICH2 User Guide](#). Do not use this environment variable with `IMPI`; doing so may degrade performance. The `ibrun` launcher will eventually control this environment variable automatically.

## File Operations: I/O Performance

This section includes general advice intended to help you achieve good performance during file operations. See [Navigating the Shared File Systems](#) for a brief overview of Stampede2's Lustre file systems and the concept of striping. See [TACC Training material](#) for additional information on I/O performance.

Follow the advice in [Good Conduct](#) to avoid stressing the file system.

**Stripe for performance.** If your application writes large files using MPI-based parallel I/O (including [MPI-IO](#), [parallel HDF5](#), and [parallel netCDF](#)), you should experiment with stripe counts larger than the default values (2 stripes on `$SCRATCH`, 1 stripe on `$WORK`). See [Striping Large Files](#) for the simplest way to set the stripe count on the directory in which you will create new output files. You may also want to try larger stripe sizes up to 16MB or even 32MB; execute `man lfs` for more information. If you write many small files you should probably leave the stripe count at its default value, especially if you write each file from a single process. Note that it's not possible to change the stripe parameters on files that already exist. This means that you should make decisions about striping when you create input files, not when you read them.

**Aggregate file operations.** Open and close files once. Read and write large, contiguous blocks of data at a time; this requires understanding how a given programming language uses memory to [store arrays](#).

**Be smart about your general strategy.** When possible avoid an I/O strategy that requires each process to access its own files; such strategies don't scale well and are likely to stress a Lustre file system. A better approach is to use a single process to read and write files. Even better is genuinely parallel MPI-based I/O.

Use **parallel I/O libraries**. Leave the details to a high performance package like [MPI-IO](#) (built into MPI itself), [parallel HDF5](#) (`module load phdf5`), and [parallel netCDF](#) (`module load pnetcdf`).

When using the Intel Fortran compiler, **compile with** "`-assume buffered_io`". Equivalently, set the environment variable `FORT_BUFFERED=TRUE`. Doing otherwise can dramatically slow down access to variable length unformatted files. More generally, direct access in Fortran is typically faster than sequential access, and accessing a binary file is faster than ASCII.

## References

- [Bash Users' Startup Files: Quick Start Guide](#)
- [idev](#) documentation
- [GNU documentation](#)
- [Intel software documentation](#)
- [Lmod's online documentation](#)
- [Multi-Factor Authentication at TACC](#)
- [Sharing Project Files on TACC Systems](#)
- [Slurm online documentation](#)
- [TACC Analysis Portal](#)

## Help Desk

TACC Consulting operates from 8am to 5pm CST, Monday through Friday, except for holidays. You can [submit a help desk ticket](#) at any time via the TACC User Portal with "Stampede2" in the Resource field. Help the consulting staff help you by following these best practices when submitting tickets.

- **Do your homework** before submitting a help desk ticket. What does the user guide and other documentation say? Search the internet for key phrases in your error logs; that's probably what the consultants answering your ticket are going to do. What have you changed since the last time your job succeeded?
- **Describe your issue as precisely and completely as you can:** what you did, what happened, verbatim error messages, other meaningful output. When appropriate, include the information a consultant would need to find your artifacts and understand your workflow: e.g. the directory containing your build and/or job script; the modules you were using; relevant job numbers; and recent changes in your workflow that could affect or explain the behavior you're observing.
- **Subscribe to Stampede2 User News**. This is the best way to keep abreast of maintenance schedules, system outages, and other general interest items.
- **Have realistic expectations.** Consultants can address system issues and answer questions about Stampede2. But they can't teach parallel programming in a ticket, and may know nothing about the package you downloaded. They may offer general advice that will help you build, debug, optimize, or modify your code, but you shouldn't expect them to do these things for you.
- **Be patient.** It may take a business day for a consultant to get back to you, especially if your issue is complex. It might take an exchange or two before you and the consultant are on the same page. If the admins disable your account, it's not punitive. When the file system is in danger of crashing, or a login node hangs, they don't have time to notify you before taking action.

## Revision History

"Last Update" at the top of this document is the date of the most recent change to this document. This revision history is a list of non-trivial updates; it excludes routine items such as corrected typos and minor format changes.

- 06/04/24 Stampede2 is decommissioned.
- 09/14/22 XSEDE project ends. Replace Globus with Grid Community Toolkit.
- 03/07/22 Intel Ice Lake nodes introduced. New `icx-normal` queue.
- 04/24/18 Changes to Table 1 and Table 5 associated with new `long` queue.
- 04/03/18 Stampede1 decommissioned; removed/revised references to Stampede1 as appropriate.
- 03/26/18 Corrected and relocated material on `qopt-zmm-usage`.
- 02/23/18 New functionality associated with `task\_affinity`, `tacc\_affinity`, and `mem\_affinity` (scripts related to MPI task pinning and KNL memory management).
- 11/30/17 Initial release supporting Phase 2 (SKX).
- 08/02/17 Removed references and links to Stampede2 Transition Guide (now deprecated).
- 06/12/17 Initial public release.