Contents

1 Acknowledging RC

3
Documentation covering the use of Research Computing resources.

Here are some quick links into the documentation to get you started.

• Logging In
• Research Computing Filesystems
• Compiling Software
• Batch Jobs
• The Module System
• Frequently Asked Questions (FAQ)

More information is available at https://www.colorado.edu/rc.

If you have any questions, please contact rc-help@colorado.edu.
1.1 Frequently Asked Questions

See our documentation homepage for information about our most common topics.

1. I have a new phone. How do I move my Duo onto it?
2. How do I check how full my Summit directories are?
3. When will my job start?
4. How much memory did my job use?
5. Where is my current fair share priority level at?
6. Why is my job pending with reason ‘ReqNodeNotAvail’?
7. Why do I get the following ‘Invalid Partition’ error when I submit my job?:

```
sbatch: error: Batch job submission failed: Invalid partition name specified.
```
8. How can I check what allocations I belong to?
9. Why do I get the following ‘LMOD’ error when I try to load slurm/summit?:

```
Lmod has detected the following error: The following module(s) are unknown: "slurm/summit"
```
10. *How do I install my own python library?*

### 1.1.1 I have a new phone. How do I move my Duo onto it?

You can add a new device to your duo account by visiting [https://duo.colorado.edu](https://duo.colorado.edu). After a CU authorization page you will be directed to a Duo authentication page. **Ignore the Duo Push prompt and instead click “Settings”:**

![Duo Push Prompt](image)

In this settings side bar click “Add a new device”: 

![Add a new device](image)
Duo will then try to authenticate your account by push notification to verify your identity. Cancel this push notification... 

...and click on “Enter a Passcode”, or “Call Me”.

- If you select “Call Me” the simply recieve the call and press 1.
- If you select “Enter a passcode” then click “Text me new codes” and you will be sent a list of one time passwords. Type in any one of the codes and you will be authenticated. Once you have verified your identity, follow the instructions provided by Duo to add your device.

If you cannot authenticate your account, contact rc-help@colorado.edu for further assistance.

1.1.2 How do I check how full my Summit directories are?

You have three directories allocated to your username ($USER). These include /home/$USER (2 G), /projects/$USER (250 G) and /scratch/summit/$USER (10 T). To see how much space you’ve used in each, from a Summit ‘scompile’ node, type `curc_quota` as follows:

```
[user@shas0100 ~]$ curc-quota
```

(continues on next page)
You can also check the amount of space being used by any directory with the `du -h` command:

```
[janedoe@shas0136 ~]$ du -h /scratch/summit/janedoe/WRF
698M  WRF/run
698M  WRF
```

### 1.1.3 When will my job start?

You can pull up information on your job’s start time using the `squeue` command:

```
squeue --user=your_rc-username --start
```

Note that Slurm’s estimated start time can be a bit inaccurate. This is because Slurm calculates this estimation off the jobs that are currently running or queued in the system. Any job that is added in later with a higher priority may delay your job.

For more information on the `squeue` command, take a look at our Useful Slurm Commands tutorial. Or visit the Slurm page on `squeue`.

### 1.1.4 How much memory did my job use?

You can check how much memory your job used by using the `sacct` command. Simply replace `YYYY-MM-DD` with the date you ran the job:

```
sacct --starttime=YYYY-MM-DD --jobs=your_job-id --format=User,JobName,JobId,MaxRSS
```

If you’d like to monitor memory usage on jobs that are currently running, use the `sstat` command:

```
sstat --jobs=your_job-id --format=User,JobName,JobId,MaxRSS
```

For more information on `sstat` or `sacct` commands, take a look at our Useful Slurm Commands tutorial. Or visit the Slurm reference pages on `sstat` and `sacct`.

### 1.1.5 How can I see my current FairShare priority?

You can check your current fair share priority level using the `sshare` command:

```
sshare -U -l
```

The `sshare` command will print out a table of information regarding your usage and priority on all allocations. The `-U` flag will specify the current user and the `-l` flag will print out more details in the table. The field we are looking for is the `LevelFS`. The LevelFS holds a number from 0 to infinity that describes the fair share of an association in relation to its other siblings in an account. Over serviced accounts will have a LevelFS that’s between 0 and 1. Under serviced accounts will have a LevelFS that’s greater than 1. Accounts that haven’t run any jobs will have a LevelFS of infinity (inf).

For more information on fair share the `sshare` command, take a look at Slurm’s documentation on fair share or check out the Slurm reference page on `sshare`.
1.1.6 Why is my job pending with reason ‘ReqNodeNotAvail’?

The ‘ReqNodeNotAvail’ message usually means that your node has been reserved for maintenance during the period you have requested within your job script. This message often occurs in the days leading up to our regularly scheduled maintenance, which is performed the first Wednesday of every month. So, for example, if you submit a job with a 72 hour wall clock request on the first Monday of the month, you will receive the ‘ReqNodeNotAvail’ error because the node is reserved for maintenance within that 72-hour window. You can confirm whether the requested node has a reservation by typing `scontrol show reservation` to list all active reservations.

If you receive this message, the following solutions are available: 1) submit a shorter job that does not intersect the maintenance window; or 2) wait until after maintenance.

1.1.7 Why do I get an ‘Invalid Partition’ error when I try to submit a job?

This error usually means users do not have an allocation that would provide the service units (SUs) required to submit a job. This can occur if a user has no valid allocation, specifies an invalid allocation, or specifies an invalid partition. Think of SUs as “HPC currency”: you need an allocation of SUs to use the system. Allocations are free. New CU users should automatically get added to a ‘ucb-general’ allocation upon account creation which will provide a modest allocation of SUs for running small jobs and testing/benchmarking codes. However, if this allocation expires and you do not have a new one you will see this error. ‘ucb-general’ allocations are intended for benchmarking and testing, and it is expected that users will move to a project allocation. To request a Project and apply for a Project Allocation visit our allocation site.

1.1.8 How can I check what allocations I belong to?

You can check the allocations you belong to with the `sacctmgr` command. Simply type:

```
sacctmgr -p show associations user=$USER
```

…from a login or compile node. This will print out an assortment of information including allocations and QoS available to you. For more information on sacctmgr, check out the Slurm’s documentation

1.1.9 Why do I get an ‘LMOD’ error when I try to load Slurm?

The slurm/summit module environment can not be loaded from compile or compute nodes. It should only be loaded from login nodes when attempting to switch between Blanca and Summit enviornments. This error can be disregarded, as no harm is done.

1.1.10 How do I install my own python library?

Research Computing provides commonly used Python libraries as modules. This guide covers installing a local Python library (pyDOE) which is not included in the Research Computing modules. One prerequisite assumption is that you are using the new module system. That being said, this guide can be tweaked to be used on the older modules as well.

First login to a login node and then ssh to a compile node.

```
[user@login01 ~]$ ssh scompile
```

Next load the version of Python you’d like to add a library too. For this guide we’ll be using Intel and Python 2.7.11.

```
[user@shas0100 ~]$ ml intel/17.4
[user@shas0100 ~]$ ml python/2.7.11
```
Before installing, create a directory in which to keep your local Python libraries. It is recommended that the /projects directory be used as it has more space.

```
[user@shas0100 ~]$ mkdir /projects/$USER/python_libs
```

You can now install your local python library.

```
[user@shas0100 ~]$ pip install --prefix="/projects/$USER/python_libs" pyDOE
```

In order to use your newly installed library it needs to be added to your PYTHONPATH. Use the following export command:

```
[user@shas0100 ~]$ export PYTHONPATH=$PYTHONPATH:/projects/$USER/python_libs/lib/
˓→python2.7/site-packages/
```

You can quickly check if your install worked with the following:

```
[user@shas0100 ~]$ python -c "import pyDOE"
```

Every time you log out you will need to rerun the above export to use your Python library (don’t forget to load Python as well). Two ways of avoiding this are to add the export command to your bashrc. vim ~/.bashrc. An alternative is to make your own modulefile.

One final item of note is a Python virtualenv. Virtualenvs allow you to keep multiple Python environments with separate versions of packages. There are plenty of guides available online such as this one: http://docs.python-guide.org/en/latest/dev/virtualenvs/. These are especially handy if you have several projects which require different versions of the same Python library.

### 1.2 Logging In

Research Computing offers a variety of resources for researchers to use in their own projects. To get started with Research Computing resources we need the following:

- A Research Computing account
- Duo 2 factor authentication
- The PuTTY application (if you are a Windows user)

Users accessing RC’s resources will be connected to a login node. A login node is a outward facing node within the Research Computing environment that users can connect to from their local machines. Once on a login node, users can perform a limited number of tasks:

- Edit files
- Transfer Data
- Submit Jobs
- Access storage resources

Note that the login nodes should not be used for resource-intensive tasks such as running code. For all other tasks, users should submit batch jobs, interactive jobs, or use the compile nodes.

- **For CSU users, please refer to the CSU login guide.**
- **For RMACC users, please refer to RMACC Access to Summit.**
1.2.1 Getting an account

Although Research Computing resources are free and available to the CU Boulder research community, they are also managed resources with large value and high demand. Therefore, Research Computing mandates that each user obtain a Research Computing account. Please note that Research Computing accounts are separate from your CU accounts. You must sign up for an individual Research Computing account if you wish to access Research Computing resources.

A Research Computing account can be secured quickly and easily by filling out the form here. Once you’ve received an account with Research Computing, you will automatically be sent an invitation for Duo enrollment via email. After you have accepted the Duo invitation, you will be able to authenticate via Duo and log into the system.

1.2.2 Logging in from a Windows Machine

Logging in from a Windows machine requires the additional step of installing the PuTTY ssh client onto your local machine. This application allows users to connect to remote servers with the ssh protocol. Note that there are other ssh clients that allow Windows machines to connect to remote ssh servers; Research Computing recommends PuTTY for reliability and simplicity.

1. Open the PuTTY application on your computer
   • Under “Host Name (or IP address)”, enter login.rc.colorado.edu. Select “SSH” as the connection type. Click on “Open”.
2. Enter your Identikey in response to the “login as” prompt
3. When prompted to enter your password:
   • If you are logging in using Duo Push, simply type your Identikey password. You will then receive an authentication request from the Duo app on your phone. Approve the request.
   • If you are using Duo SMS, Phone Call, or Token login methods, instructions can be found here.
   • Note that as a security feature, PuTTY does not display any text while you type your password

1.2.3 Logging in from a Mac

Logging in with a Mac requires no extra installation on your local machine. Simply utilize the terminal application that is pre-installed with your operating system to access Research Computing resources.

1. Under “File”, open a new finder window. Navigate to the “Applications” folder, then the “Utilities” folder. Open a terminal window and type ssh username@login.rc.colorado.edu, where username is your assigned username. Press enter.
2. Enter your password:
   • If you are logging in using Duo Push, type your Identikey password. You will then receive an authentication request on the Duo app on your phone. Approve the request.
   • If you are using Duo SMS, Phone Call, or Token login methods, instructions can be found here.
1.2.4 Logging in from Linux

Much like with Macs, Linux machines require no additional setup to access Research Computing resources. Simply utilize the your Linux terminal to access Research Computing resources.

1. Open a terminal window from your application menu and type: `ssh username@login.rc.colorado.edu`, where `username` is your research computing username.

2. Enter your password:
   - If you are logging in using Duo Push, simply type your Identikey password. You will then receive an authentication request on the Duo app on your phone. Approve the request.
   - If you are using Duo SMS, Phone Call, or Token login methods, instructions can be found here.

1.2.5 SSH host keys

The first time you log into an RC login node you will be asked to verify the host key. You can refer to the keys published here to confirm that you are connecting to a valid RC login node.

Note that each login node may support more than one type of key, but only one is used (or displayed) by your client at any given time.

```bash
# Fingerprint
# 256 SHA256:MB+601um10c1sPXT4AXbV0rNRywUH4U10IB9oJMvD8Q no comment (ECDSA)
ecdsa-sha2-nistp256
   → AAAAE2Vj2HNhlXNoYTItbmlzdHAYNTYAAAIbmlzdHAYNTYAAABBBPfjz9VZAwSS0329z6RNZQDNrN3vU1Yct1mBRqQAqxmDxBV

# Fingerprint
# 256 SHA256:uNn+9REKipRZ59VZQEK1Bz8xjOCe/9yII8uBqEGFtGOY no comment (ED25519)
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAIAIPub4h8XLU3dXJBNZl1s4pDPOORXoBdSdaKnwFvCMTx3

# Fingerprint
# 2048 SHA256:xZ9xXWtJwBWYqm3ESvqqQ0qnhnImGFtyantrEk no comment (RSA)
ssh-rsa
   → AAAAB3NzaC1yc2EAAADAQABABABABCWQgetVDOqKB2im3HyQZJ72PMYXFJUXR2Z+dzhGfOREABA8V6m0fKCaVzPrBjXSYR4QB
   → 1Up9wo9m2UN8VABE0rwgcEvKfbQriwhkdqjEzEbEN5Fx05iYN2o2vphmi3dUpHaKM2P10bHQKwjmJ3bf7a7gpXWNAhGi2
   → z/
   → v7JKV26jIxXmdq6+VbRCpovnL3pJkoU5e4vaSC4V5kvgfI9G4tj6BEDGsRhXHRcZXk+hLtNp2nj2VsSocWcO
   → Vxn85obSfVnVwV/
```

login.rc.colorado.edu (as of 9 May 2018)

```bash
# Fingerprint
# 256 SHA256:MB+601um10c1sPXT4AXbV0rNRywUH4U10IB9oJMvD8Q no comment (ECDSA)
ecdsa-sha2-nistp256
   → AAAAE2Vj2HNhlXNoYTItbmlzdHAYNTYAAAIbmlzdHAYNTYAAABBBPfjz9VZAwSS0329z6RNZQDNrN3vU1Yct1mBRqQAqxmDxBV

# Fingerprint
# 256 SHA256:uNn+9REKipRZ59VZQEK1Bz8xjOCe/9yII8uBqEGFtGOY no comment (ED25519)
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAIAIPub4h8XLU3dXJBNZl1s4pDPOORXoBdSdaKnwFvCMTx3

# Fingerprint
# 2048 SHA256:xZ9xXWtJwBWYqm3ESvqqQ0qnhnImGFtyantrEk no comment (RSA)
ssh-rsa
   → AAAAB3NzaC1yc2EAAADAQABABABABCWQgetVDOqKB2im3HyQZJ72PMYXFJUXR2Z+dzhGfOREABA8V6m0fKCaVzPrBjXSYR4QB
   → 1Up9wo9m2UN8VABE0rwgcEvKfbQriwhkdqjEzEbEN5Fx05iYN2o2vphmi3dUpHaKM2P10bHQKwjmJ3bf7a7gpXWNAhGi2
   → z/
   → v7JKV26jIxXmdq6+VbRCpovnL3pJkoU5e4vaSC4V5kvgfI9G4tj6BEDGsRhXHRcZXk+hLtNp2nj2VsSocWcO
   → Vxn85obSfVnVwV/
```

blogin01.rc.colorado.edu

A private login node owned by ICS

```bash
# Fingerprint
# 256 SHA256:SL1jPqUgIZ2rMYPY127ssUeg++52wivsqJva9NBKB8vk no comment (ECDSA)
ecdsa-sha2-nistp256
   → AAAAE2Vj2HNhlXNoYTItbmlzdHAYNTYAAAIbmlzdHAYNTYAAABBBNcwOXb6FQ0k0wEtoNHqYur2xc3t+Dwid5JbzRBasCqpb/8+y
   → GSr2SgP14JKJ40ptAPkjdj+Qq9BRd42ExVdE=
```
1.3 Duo 2-factor Authentication

Research Computing utilizes a 2-factor authentication utility called Duo that must be utilized in order to log into RC resources. This document covers installing and logging in with Duo for CU Boulder users only.

- For CSU Users please see CSU’s documentation on Duo 2-factor authentication
- For RMACC Users please see XSEDE’s documentation on Duo 2-factor authentication

1.3.1 Setting up Duo

1. You will receive a Duo invitation when your RC account is created. Open your Duo invitation email (from Duo security or duo.com) on a desktop/laptop computer.

   • Within one business day you should receive a duo invitation. If you didn’t receive an invitation or your invitation has expired, then please contact rc-help@colorado.edu for a new invitation.

2. Login with your CU identikey and follow the instructions to setup

   • Research Computing recommends users utilize Duo Push as Duo push, install the Duo Mobile app and link your account with your device. Duo

3. Login to RC Resources via ssh as described below.

Common Issues

- Duo Invite email may be sent to your Spam folder.
- Do not request a phone call if you want to use the Push app for authentication.
- Duo accounts are purged if unused for 6-9 months.

Please contact us at rc-help@colorado.edu if you encounter any issues with Duo setup.

1.3.2 Logging in with Duo

Duo offers a variety of methods to log into your account. Depending on what you select when setting up your Duo account, you should have several different methods of 2-factor authentication when logging into RC Resources.
Using Duo Push

Duo Push is Research Computing’s recommended method of 2-factor authentication. Because Duo Push is tied to your physical smart device instead of a phone number or account, Duo Push is provides a more secure method of 2-factor authentication than either SMS or phone call.

1. Type `ssh <username>@login.rc.colorado.edu` into the command line. For example, if my username is jodo2018 I would type `ssh jodo2018@login.rc.colorado.edu`

2. Enter your identikey password when prompted.

3. Wait for a push to your phone.

*Note: Duo push is tied to your device so you will need to add or remove your device if you get a new phone.*

Using Duo SMS

If you prefer to not use the Duo app or if you don’t have a smart device, then Duo offers an SMS method of 2-factor authentication:

1. `ssh username@login.rc.colorado.edu`. For example, if my username is jodo2018 I would type `ssh jodo2018@login.rc.colorado.edu`

2. Enter your password when prompted, as `mypassword,sms`. For example, if my password is Ih3artdu0 I would type: `Ih3artdu0,sms`

3. A list of one-time-passwords (OTPs) will be texted to you via SMS. Another login prompt will appear on your screen. Using the first OTP in the list, login with your password as `mypassword,OTP`. For example, if my password is Ih3artdu0 and my OTP is 330456 I would type (without quotes): `Ih3artdu0,330456`.

4. Note that the next time you login, you can either request a new list of OTPs using Step 2 and then enter the first OTP via Step 3, or you can just use the next OTP in the list, and skip directly to Step 3.

Using Duo Phone Call

Duo also provides a phone call solution for 2-factor authorization if you only have a land line, or prefer to not use Push or SMS:

1. Type `ssh username@login.rc.colorado.edu` into the command line. For example, if my username is jodo2018 I would type `ssh jodo2018@login.rc.colorado.edu`

2. Enter your password when prompted, as `mypassword,phone`. For example, if my password is Ih3artdu0 I would type: `Ih3artdu0,phone`

3. Wait for a phone call. Answer the call, select option #1, and you will automatically be logged in.

Using the Duo Token

If you prefer a physical device, similar to the Vasco OTP. You have to arrange with RC staff to acquire a token. There is a small fee for this device. Email rc-help@colorado.edu for assistance.

1. `ssh username@login.rc.colorado.edu`. For example, if my username is jodo2018 I would type `ssh jodo2018@login.rc.colorado.edu`

2. Enter your password when prompted, as `mypassword,6-digit-number`. The 6-digit number is given to you on the Duo token. For example, if my password is Ih3artdu0, and I press the button on the token and it is 123456, I would type: `Ih3artdu0,123456`
1.3.3 Managing Duo Devices

Users can manage their own Duo devices by visiting https://duo.colorado.edu. This can allow users to add or remove activated devices at their discretion.

After a CU authorization page, you will be directed to a Duo authentication page. Do not respond to the Push notification and instead click the “Settings” button:

![Duo Authentication Page]

This will bring up a menu that provides several options on device management and general help.

Clicking “Add a new device” will allow you to add a new smart phone, tablet, or land-line. Simply select the option you wish to add and follow the steps provided by Duo complete setting up your new device.

Selecting “My Settings & Devices” provides a more detailed list of all devices you have registered. From here you can also add a new device, set your default device, and change your default authentication method when you attempt to log in.

1.4 RMACC Access to Summit

If you are at an RMACC institution other than CU Boulder or CSU and would like to use Summit, please see the below steps to gain access.

1.4.1 Getting an XSEDE account

Visit the XSEDE User Portal and use the “Create Account” button. Complete the form and follow the instructions to create an XSEDE account. As part of the process, you will select an XSEDE username, which will be used to access the XSEDE User Portal and the XSEDE SSO Hub.
1.4.2 Configuring Duo

Once your XSEDE account has been created, follow the Multi-Factor Authentication setup instructions on the XSEDE website. Multi-factor authentication with Duo is required for access to the XSEDE SSO Hub, which provides access to RMACC Summit.
1.4.3 Getting authorization for RMACC Summit

Send an email from your institutional email address to rc-help@colorado.edu requesting access to RMACC Summit via XSEDE. Include the full name of your home institution, specific department within the institution, a short summary of why you require access to RMACC Summit, and your XSEDE username. You will receive a response indicating whether your account has been added to the rmacc-summit.colorado.xsede.org project.

1.4.4 Logging in to CU Boulder RC using the XSEDE SSO Hub

Use an ssh client to connect to the XSEDE SSO Hub using your XSEDE credentials. You will be prompted to authenticate both using your XSEDE password and via Duo.

```
ssh -l <your-xsede-username> login.xsede.org
```

From there, use gsissh (available in the SSO Hub environment) to log into rmacc-summit, which serves as an alias for the CU Boulder RC login environment.

```
gsissh rmacc-summit
```

1.5 Blanca

CU Research Computing operates a shared “condo” compute cluster, named Blanca, which consists of nodes owned by individual research groups or departments. Condo partners get significantly prioritized access on nodes that they own and can run jobs on any nodes that are not currently in use by other partners.

An allocation of CPU time is not needed in order to run on Blanca.

If you would like to purchase a Blanca node, please visit the Research Computing website for more details.

1.5.1 Blanca Quick-Start

1. If your group is a Blanca partner, ask your PI or PoC to send an email to rc-help@colorado.edu requesting access for you to their high-priority queue.
2. From a login node, run “module load slurm/blanca” to access the Slurm job scheduler instance for Blanca.
3. Consult the Table and the Examples section below to learn how to direct your jobs to the appropriate compute nodes.
4. If needed, compile your application on the appropriate compute node type.
5. Read the rest of this page thoroughly.

1.5.2 Job Scheduling

All jobs are run through a batch/queue system. Interactive jobs on compute nodes are allowed but these must be initiated through the scheduler. Each partner group has its own high-priority QoS (analogous to a queue) for jobs that will run on nodes that it has purchased. High-priority jobs move to the top of the queue and are thus guaranteed to start running within a few minutes, unless other high-priority jobs are already queued or running ahead of them. High-priority jobs can run for a maximum wall time of 7 days. All partners also have access to a low-priority preemptable QoS that can run on any Blanca nodes that are not already in use by their owners. Low-priority jobs have a maximum wall time of 24 hours.
Blanca uses a separate instance of the Slurm scheduling system from the other RC compute resources. You can use Blanca’s Slurm instance by loading a special module on a login node: “module load slurm/blanca”.

More details about how to use Slurm can be found here.

1.5.3 QoS

Slurm on Blanca uses “Quality of Service”, or QoS, to classify jobs for scheduling. A QoS in this case is analogous to a “queue” in other scheduling systems. Each partner group has its own high-priority QoS called blanca-<group identifier> and can also use the condo-wide low-priority QoS, which is called preemptable.

If you are a new Blanca user, ask your PI or Point of Contact person to request access for you to your group’s high-priority QoS; requests should be made via email to rc-help@colorado.edu. You are only allowed to use a high-priority QoS if you have specifically been added as a member of it, and you can only use the low-priority preemptable QoS if you are also a member of a high-priority QoS. Your PI may also be able to point you to group-specific documentation regarding Blanca.

1.5.4 Node-QoS-Features

Since not all Blanca nodes are identical, you can include node features in your job requests to help the scheduler determine which nodes are appropriate for your jobs to run on when you are using the preemptable QoS.

To determine which nodes exist on the system, type `scontrol show nodes` to get a list.

1.5.5 Node Features Table

1.5.6 Description of features

- **westmere-ex**: Intel processor generation
- **sandybridge**: Intel processor generation
- **ivybridge**: Intel processor generation
- **haswell**: Intel processor generation
- **broadwell**: Intel processor generation
- **avx**: AVX processor instruction set
- **avx2**: AVX2 processor instruction set
- **fdr**: InfiniBand network generation
- **Quadro**: NVIDIA GPU generation
- **Tesla**: NVIDIA GPU generation
- **k2000**: NVIDIA K2000 GPU
- **P100**: NVIDIA P100 GPU
- **localraid**: large, fast RAID disk storage in node
- **rhel7**: RedHat Enterprise Linux version 7 operating system
1.5.7 Examples

Here are examples of Slurm directives that can be used in your batch scripts in order to meet certain job requirements. Note that the “constraint” directive constrains a job to run only on nodes with the corresponding feature.

1. To run a 32-core job for 36 hours on a single blanca-ics node:

```bash
#SBATCH --qos=blanca-ics
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=32
#SBATCH --time=36:00:00
```

1. To run a 56-core job across two blanca-sha nodes for seven days:

```bash
#SBATCH --qos=blanca-sha
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=28
#SBATCH --time=7-00:00:00
#SBATCH --export=NONE
```

1. To run an 8-core job in the low-priority QoS on any node that has broadwell processors and uses the RHEL 7 operating system:

```bash
#SBATCH --qos=preemptable
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --time=4:00:00
#SBATCH --export=NONE
#SBATCH --constraint="broadwell&rhel7"
```

1. To run an 8-core job in the low-priority QoS on any node that has either the AVX or AVX2 instruction set:

```bash
#SBATCH --qos=preemptable
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --time=4:00:00
#SBATCH --export=NONE
#SBATCH --constraint="avx|avx2"
```

1. To start a 2-hr interactive job on one core on a blanca-ceae node, run this at the command line:

```bash
sinteractive --qos=blanca-ceae --export=NONE --time=02:00:00
```

Note that the interactive job won’t start until the resources that it needs are available, so you may not get a command prompt on your assigned compute node immediately.

1.5.8 Important notes

1. To see what modules are available, start an interactive job on a compute node and use `module avail` or `module spider` on it.

2. /home, /projects, and /work (PetaLibrary Active) are available on all Blanca nodes. Scratch I/O can be written to /rc_scratch, which should offer much better performance than /projects. All Blanca nodes also have at least 850 GB of scratch space on a local disk, /local/scratch. For more info on the different RC storage spaces, please see our page on storage.
3. There are no dedicated Blanca compile nodes. To build software that will run on Blanca, start an interactive job on a node like the one on which you expect your jobs to run, and compile your software there. Do not compile on the login nodes!

4. Multi-node MPI jobs that do a lot of inter-process communication do not run well on most Blanca nodes because there is no high-performance low-latency inter-node network except on blanca-ccn.

1.5.9 Blanca Preemptable QOS

(effective 2018-03-01) Each partner group has its own high-priority QoS ("blanca-") for jobs that will run on nodes that it has contributed. High-priority jobs can run for up to 7 days. All partners also have access to a low-priority QoS ("preemptable") that can run on any Blanca nodes that are not already in use by the partners who contributed them. Low-priority jobs will have a maximum time limit of 24 hours, and can be preempted at any time by high-priority jobs that request the same compute resources being used by the low-priority job. The preemption process will terminate the low-priority job with a grace period of up to 120-seconds. Preempted low-priority jobs will then be requeued by default. Additional details follow.

Usage

To specify the preemptable QoS in a job script:

```
#SBATCH --QoS=preemptable
```

To specify the preemptable QoS for an interactive job:

```
$ sinteractive --qos=preemptable <other_arguments>
```

Batch jobs that are preempted will automatically requeue if the exit code is non-zero. (It will be non-zero in most cases.) If you would prefer that jobs not requeue, specify:

```
#SBATCH --no-requeue
```

Interactive jobs will not requeue if preempted.

Best practices

Checkpointing: Given that preemptable jobs can request wall times up to 24 hours in duration, there is the possibility that users may lose results if they do not checkpoint. Checkpointing is the practice of incrementally saving computed results such that – if a job is preempted, killed, canceled or crashes – a given software package or model can continue from the most recent checkpoint in a subsequent job, rather than starting over from the beginning. For example, if a user implements hourly checkpointing and their 24 hour simulation job is preempted after 22.5 hours, they will be able to continue their simulation from the most recent checkpoint data that was written out at 22 hours, rather than starting over. Checkpointing is an application-dependent process, not something that can be automated on the system end; many popular software packages have checkpointing built in (e.g., ‘restart’ files). In summary, users of the preemptable QoS should implement checkpointing if at all possible to ensure they can pick up where they left off in the event their job is preempted.

Requeuing: Users running jobs that do not require requeuing if preempted should specify the `--no-requeue` flag noted above to avoid unnecessary use of compute resources.
Example Job Scripts

Submit a 6-hour preemptable python job on 32 cores without specifying a partition (job will run on any available compute partitions on Blanca, regardless of features, so long as they have at least 16 cores each).

```bash
#!/bin/bash
#SBATCH --time=06:00:00
#SBATCH --qos=preemptable
#SBATCH --job-name=test
#SBATCH --nodes=2
#SBATCH --ntasks=32
#SBATCH --output=test.%j.out

module purge
module load python

python myscript.py
```

Same as Example 1, but specify a specific partition (‘blanca-ccn’) (job will only run on blanca-ccn nodes)

```bash
#!/bin/bash
#SBATCH --time=06:00:00
#SBATCH --qos=preemptable
#SBATCH --partition=blanca-ccn
#SBATCH --job-name=test
#SBATCH --nodes=2
#SBATCH --ntasks=32
#SBATCH --output=test.%j.out

module purge
module load python

python myscript.py
```

Same as Example 1, but specify desired node features, in this case the avx2 instruction set and Redhat V7 OS (job will run on any node meeting these feature requirements, and which has at least 16 cores per node).

```bash
#!/bin/bash
#SBATCH --time=06:00:00
#SBATCH --qos=preemptable
#SBATCH --constraint="avx2&rhel7"
#SBATCH --job-name=test
#SBATCH --nodes=2
#SBATCH --ntasks=32
#SBATCH --output=test.%j.out

module purge
module load python

python myscript.py
```

Other considerations

Grace period upon preemption: When jobs are preempted, a 120 second grace period is available to enable users to save and exit their jobs should they have the ability to do so. The preempted job is immediately sent SIGCONT and SIGTERM signals by Slurm in order to provide notification of its imminent termination. This is followed by the SIGCONT, SIGTERM and SIGKILL signal sequence upon reaching the end of the 120 second grace period. Users
wishing to do so can monitor the job for the SIGTERM signal and, when detected, take advantage of this 120 second grace period to save and exit their jobs.

The ‘blanca’ QoS: Note that as of 1 March, 2018, the “preemptable” qos replaces the previous low-priority QoS, “blanca”, which is no longer active.

## 1.6 Allocations

Allocations are a crucial component of running on an HPC system. They are required to run on RC resources. When you receive an account you will automatically be assigned a ucb-general share. ucb-general is a great place to run smaller jobs or to use to test and benchmark your code. ucb-general users have a lower priority. To receive higher priority, you can apply for an allocation to run on our systems.

In this document, we will describe the allocation process and how Fair Share works to give priority.

### 1.6.1 Why Do I Need An Allocation

Having an account only validates that you are eligible to use RC resources. An allocation allows us to keep track of your use of the system. This is important because:

- We need to make sure we have enough resources to accommodate all of our users
- Helps for reporting to NSF and the CU Research & Innovation Office. Summit is funded through a grant from NSF.

### 1.6.2 How does this process work?

To apply for an allocation, you must do the following:

1. Fill out the template at the linked page in your preferred format. If you need assistance, please always feel free to email rc-help@colorado.edu.
2. Login to RCAMP
3. Fill out some basic information about your request and create a project. This project is a space in which you can link multiple allocations, manage PIs, etc.
4. After creating a project, upload the template to complete your allocation request.
5. A committee will review your proposal and either accept, provide comments on adjustments that need to be made, or deny

### 1.6.3 What is Fair Share?

On some of our older systems, allocations have been treated as a bank account. You apply for an allocation and receive a certain number of hours based on your benchmark tests, and once you’ve used those hours they are gone. If it is Christmas Day and you are ready to run, but have no hours, and the system is under-utilized, our resources and your time are wasted. Fair share improves upon that, and balances out the load usage in a more appropriate way for all users.

Fair share scheduling starts in the allocation application process. When you are running in ucb-general, you receive a pre-determined priority (see below). When you apply for an allocation, you will still ask for a certain number of hours based on the benchmarking you did in ucb-general. Fair share scheduling will then use a complex formula to determine your priority in queue based on these numbers, our determination of your needs, and general utilization.
From this information, you will be assigned a fair share target percentage, and your utilization of your “fair share” on the system will be based on historical usage and how far under or over the fair share target percentage you are.

### 1.6.4 Fair Share Target Percentage

For allocations, your target percentage depends on your priority based on your project proposal (as described above). Everyone not associated with a project shares a target percentage of 13% (20% of the CU fraction). Your jobs are likely to have a lower priority if you are running in ucb-general than if you are running with a target percentage assigned from writing a project allocation proposal.

Everyone therefore has an assigned target percentage, and your priority will depend upon how far under or over that target percentage you are based on a four week average. If you are under (over) your four week target percentage, your priority is increased (decreased). Information on how to check your fair share priority level can be found here.

Finally - just a reminder that this only impacts pending jobs. If there are no other pending jobs and enough resources are available then your job will run regardless of your previous usage!

### 1.7 Node types

Research Computing has several node types available on our resources. Each node type is meant for certain tasks. These node types are relatively common for other HPC centers. We will discuss each node type and its intended use below.

#### 1.7.1 Login nodes

- Four virtual machines
- This is where you are when you log in
- No computation, compiling code, interactive jobs, or long running processes
- Script or code editing
- Job submission

#### 1.7.2 Compile nodes

- Where you compile code, such as Fortran, C, C++
- No heavy computation
- Job submission
- Access these nodes by typing `ssh scompile` from a login node

#### 1.7.3 Compute nodes

This is where jobs that are submitted through the scheduler run.

- Intended for heavy computation
- When run an interactive job will be performing tasks directly on the compute nodes
1.8 Filesystems

All users are allocated space on the /home and /projects filesystems. In addition, separate scratch directories are visible from Summit and Blanca. These scratch directories are hosted on separate, high-performance filesystems designed to support intensive, parallel I/O operations.

Please note that the use of /home or /projects for high-performance I/O may negatively affect the environment for all users. As a result, all compute jobs should write to the appropriate scratch filesystem. Users performing intensive I/O on the /home or /projects filesystems will have their jobs terminated and may have their accounts temporarily disabled.

1.8.1 The Home Filesystem

Every user is allocated 2 GB of space on the /home filesystem in a subdirectory corresponding to their user name (e.g., /home/janedoe). Home directories are backed up frequently and are intended for the use of their owner only; sharing the contents of home directories with other users is strongly discouraged. Your /home directory is a good place to store source code, small compiled programs, and job scripts.

1.8.2 The Projects Filesystem

Each user has access to a 250 GB of space in their subdirectory of /projects (e.g., /projects/janedoe). As with the /home system, these directories are visible from all Research Computing nodes and are regularly backed up. The projects directory is intended to store software builds and smaller data sets.

1.8.3 Scratch Filesystems

Summit users are provided a subdirectory on /scratch/summit, the high-performance parallel scratch filesystem meant for I/O from jobs running on that system (e.g., /scratch/summit/janedoe). By default, each user is limited to a quota of 10 TB worth of storage space and 20M files and directories. Email rc-help@colorado.edu if you need these limits increased. Blanca users should write to /rc_scratch/janedoe instead of /scratch/summit.

Scratch space should be used for all compute jobs run on Summit or Blanca. These high-performance scratch directories are not backed up, and are not appropriate for long-term storage. Data may be purged at any time subject to overall system needs. Files are automatically removed 90 days after their initial creation.

Users requiring longer-term retention of their files should perform regular backups to their local machine if they have not purchased space on the PetaLibrary. Inappropriate use of scratch storage, including attempts to circumvent the automatic file purge policy, may result in loss of access to Research Computing resources.

1.8.4 Monitoring Disk Usage

Disk usage may be checked using the curc-quota command. When run from a Summit compile node, you will see output similar to:

```
[janedoe@shas0136 ~]$ curc-quota
------------------------------------------------------------------------
| Used | Avail | Quota Limit |
------------------------------------------------------------------------
| /home/janedoe 1.7G 339M 2.0G |
| /projects/janedoe 67G 184G 250G |
| /scratch/summit 29G 10211G 10240G |
```
If the command is run from a login node, information concerning /scratch/summit will be omitted.

Note that the space occupied by a particular directory and its subdirectories can be obtained via the `du -h` command:

```
[janedoe@shas0136 ~]$ du -h /scratch/summit/janedoe/WRF
 698M   WRF/run
 698M   WRF
```

### 1.8.5 Backups

Regular backups are performed for all `/home` and `/projects` directories and at a range of cadences. Low-cadence backups are retained for longer periods of time than high-cadence backups. A summary of the backup schedule is provided in the table below.

If disaster strikes and you need access to a previous version of your `/home` or `/projects` directories, change to that directory and look through the `.snapshot` subdirectory. You will see a subdirectory associated with each snapshot of your `/home` or `/projects` directory, named using the time-stamp associated with the snapshot.

### 1.8.6 Workspace Sharing

All users have complete control over their personal directory permissions. While we encourage you to share your `/projects` and `/scratch` directories with collaborators as appropriate, we strongly discourage sharing of your `/home` directory due to the limited space and potentially sensitive information stored there.

Directories may be shared with all Research Computing users or with only a subset of our users. In the latter case, a system administrator will need to add your chosen collaborators to your Linux group. Please email rc-help@colorado.edu if you would like to add users to your Linux group.

In the example that follows, we make our `/projects` directory open to all users and then create subdirectories with select read/write permissions for all users and our chosen collaborators.

First, we make our `/projects` directory world-readable:

```
[janedoe@shas0136 ~]$ chmod a+rx /projects/janedoe
```

Next, we create a subdirectory that is visible to all users and which is read-only:

```
[janedoe@shas0136 ~]$ cd /projects/janedoe
[janedoe@shas0136 ~]$ mkdir world_read
[janedoe@shas0136 ~]$ chmod a+rx world_read
```

For our collaborators, we may want a writeable directory in addition to a read-only directory:

```
[janedoe@shas0136 ~]$ cd /projects/janedoe
[janedoe@shas0136 ~]$ mkdir group_read
[janedoe@shas0136 ~]$ chmod g+rwx group_read
[janedoe@shas0136 ~]$ mkdir group_read_write
[janedoe@shas0136 ~]$ chmod g+rwx group_read_write
```

A similar methodology will need to be followed for all subdirectories you wish to share. If you make a mistake or change your mind, use the – symbol in lieu of + to remove privileges. Note that the x is necessary if you want other users to be able to `cd` into your directory.

---

**1.8. Filesystems** 23
1.9 The modules system

Research Computing uses a module system to load most software into a user’s environment. Most software is not accessible by default and must be loaded in. This allows Research Computing to provide multiple versions of the software concurrently and enables users to switch easily between different versions.

1.9.1 The module Command

Modules should be loaded in job scripts, interactive jobs, or on compile nodes only. They should not be loaded when on one of the login nodes. The login node will restrict the loading of modules, so you won’t be able to access software unless you do so through a job or a compile node.

To see what modules are available to load, ssh into a compile node by typing `ssh scompile` from a login node, and type:

```
module avail
```

This will return a list of modules available to load into the environment. Please note if you run this command on a login node you will not receive a full list of modules present on the system.

To load your chosen modules into the environment type:

```
module load some_module
# example: "module load python"
```

You can specify the version of the software by appending a / with the version number:

```
module load some_module/version
# example: "module load python/3.5.1"
```

The Lmod hierarchical module system provides five layers to support programs built with compiler and library consistency requirements. A module’s dependencies must be loaded before the module can be loaded.

The Layers include:

- Independent programs
- Compilers
- Compiler dependent programs
- MPI implementations
- MPI dependent programs

If you cannot load a module because of dependencies, you can use the `module spider` to find what dependencies you need to load the module.

```
module spider some_module
# example: "module spider openmpi"
```
1.9.2 Loading Modules in a Job Script

Loading a module will set or modify a user’s environment variables. Additionally, modules will enable access to the software package provided by that module. This can be useful in interactive jobs or in job-scripts that are dependent on software like python.

Modules in a job script can be loaded after your `#SBATCH` directives and before your actual executable is called. A sample job script that loads python into the environment is provided below:

```bash
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --time=00:01:00
#SBATCH --qos=debug
#SBATCH --ntasks=1
#SBATCH --job-name=test-job
#SBATCH --output=test-job.%j.out

module purge
module load python/3.5.1

python3 test-program.py
```

1.9.3 Subcommands

The `module` command has a variety of subcommands, outlined in the table below. You may shorten the command to `ml`, but the shortened command may require specialized syntax.

1.10 Data transfer

Research Computing supports several methods of file transfers onto Summit. File transfers from a local machine can be done via two ways: Through Globus or through SSH protocols.

1.10.1 Globus

On Globus, file transfers are handled through an interactive web application. Globus addresses deficiencies in secure copy requests by automating large data transfers, resuming failed transfers, and simplifying the implementation of high performance transfers between computing centers.

Sign into Globus Connect by selecting “University of Colorado at Boulder” from the dropdown menu and by logging in using your CU IdentiKey and password.

- If you’re with an institution outside of the University of Colorado at Boulder that is registered with Globus, sign in with your appropriate credentials.
- If your institution is not registered with Globus, you will need to make an account with Globus.
Files can easily be transferred from Summit to your local computer with Globus.

- Research Computing resources are installed with a Globus endpoint. You can connect to this endpoint by clicking the “endpoint” field and searching for the endpoint: CU Boulder Research Computing. Log into the end point by using your Research computing credentials.

- Your local computer must also have an endpoint. You can easily set up a Globus endpoint by installing Globus Connect Personal on your local machine.

Using the web app, connect your local workstation endpoint with the Research Computing endpoint and transfer files easily using the Globus GUI.
1.10.2 Secure Copy (SCP)

The Secure Copy protocol or `scp` allows users to send and receive data to the server remotely via a terminal command. The command appears as:

```
# Command to copy files from a local workstation to research computing resources
# Replace <path-to-file> with the file you wish to copy
# Replace <username> with your Research Computing username
# Replace <target_directory> with the full path to the directory you would like
# to send the file to.
scp <path-to-file> <username>@login.rc.colorado.edu:<target-path>
```

For more information on secure copy take a look at some of our listed resources or consult the man page with the command:

```
manscp
```

1.10.3 Secure File Transfer Protocol (SFTP)

The Secure File Transfer Protocol is an interactive terminal solution to transfer data to and from research computing resources. SFTP works much like moving files in a terminal, we use a series of file system commands to navigate, move, remove, and copy information from a client’s personal machine to research computing resources. To invoke SFTP type the command:

```
sftp <username>@login.rc.colorado.edu
```

We can then use various commands to traverse and manipulate both file systems. A list of commands are listed below:

1.10.4 Other Options

OIT also offers a file transfer service with a web interface which provides a good way to transfer files to collaborators. Files are uploaded to a server and a link to download the file can be emailed to an on, or off-campus, user: http://oit.colorado.edu/safe-transfer
1.10.5 DTN SSH Host Keys

```
ecdsa-sha2-nistp256
AAAAE2VjZHhlUXN0YTItbm1zdHAYNTYAAAAbm1zdHAYNTYAAABBBQA0dGntTpowZ/
    YdjXxaumHHbw59nTRuUQZDXjnVpZXRtP+Db3liM9ytWCjyjHAEH8FgfKJt/MuPbNcfm1Sg=
```

```
ssh-ed25519
AAAAC3NzaC112D1NTE5AAAAIKXXkp8RQyYvN2MYGYzBpECKwwyB929enmFVz1Jm2LtkG
```

```
ssh-rsa_
    -AAAAAB3NzaC1yc2EAAADADAQABAABAABQC0P4GvSYfG0B+dAEfBqKeJTeTk5bQ1MPzh1N8Zs1koh3fKymm16FuMI5chvP6g
01V0+ZBmPpT00PdkqNCV04Y761CnYH+V2/qC1qenTcEVs8OD7WYZy9Y1hXvXFwu/
    4aQGCMmU00dpKsJ1bAEGXDGxBas0XRV5uekbX6WrTYphr/ayP0j1l17fP4/2qhh2YCQoeh+
    cKGIj2Tg7asP3P3P3B/
    7VFqRPKsN7nLrGCDY8tcdmv6J0A0hmabl2gxyYunxEq+XS1N3gyT4WEy3qblzu601WRO4rJbrPD/uParzN3
```

1.10.6 More reading

- Indiana University Tutorial on SFTP
- Linux Academy’s Tutorial on SSH and SCP
- ssh.com’s Tutorial on SCP and SFTP

1.11 Compiling and linking

Before compiling in the RC environment, first ssh to one of the Summit compile nodes via `ssh scompile`. Next, load those modules corresponding to the compiler, MPI version (if needed), and third-party libraries required by your application. The load order should always be compiler first, MPI second, and third-party libraries last.

For example, suppose your application requires MPI and the HDF5 library. To compile using the Intel compiler and Intel MPI, the sequence of `module` commands would be:

```
module purge
module load intel
module load impi
module load hdf5
```

Supporting library-modules will be loaded as needed, and your environment will be updated so that the appropriate library directories are prepended to your `PATH` and `$LD_LIBRARY_PATH`. The standard compiler variables FC, CC and CXX are set as appropriate for your compiler/MPI combination. These environment variables reference to the Fortran, C, and C++ compilers respectively.

In addition, several environment variables are set that may be useful during the compilation process. These variables possess the prefix CURC and may easily be found by searching your environment for CURC via `env | grep CURC`. This will yield output similar to:

```
[johndoe@shas0137 ~]$ env | grep CURC
CURC_INTEL_BIN=/curc/sw/intel/17.4/bin
CURC_INTEL_INC=/curc/sw/intel/17.4/include
CURC_INTEL_ROOT=/curc/sw/intel/17.4
CURC_INTEL_LIB=/curc/sw/intel/17.4/lib
CURC_HDF5_ROOT=/curc/sw/hdf5/1.10.1/impi/17.3/intel/17.4
```

(continues on next page)
Once the relevant modules are loaded, you are ready to compile. For our HDF5 example, a compilation command that uses the environment variables set by the module system may look like:

```bash
$FC my_program.f90 -I$(CURC_HDF5_INC) -L$(CURC_HDF5_LIB) -lhdf5_fortran -o my_program
```

**Note:** Your run-time environment should reflect your compilation environment. Be sure to include the same sequence of `module` commands in your job script as that used at compile time.

### 1.11.1 Navigating the Software Stack

The RC module system is hierarchical in nature, and available software libraries become visible to the user only after the compiler and MPI implementations that they depend on have been loaded. As noted above, modules should be loaded in the order: compiler, MPI, third-party software. At each stage of the load, executing `module avail` will reveal a list of newly available modules. The `module purge` command can be used to unload all currently loaded modules.

For example, before choosing a compiler, we can view the available compilers with

```bash
[janedoe@shas0136 ~]$ module purge
[janedoe@shas0136 ~]$ module avail
```

This will yield output similar to

```
----------------------------------- Compilers --------------------------------------
gcc/6.1.0  intel/16.0.3 (m)  intel/17.0.0 (m)  intel/17.4 (m,D)  pgi/16.5 ,
```

Several compiler-independent modules will also be displayed. Those modules (e.g., the Julia module) can be loaded at any time, irrespective of the compiler or MPI version in use.

If multiple versions of a package are available, a `D` is used to indicate the default version. When the version number is omitted during the `module load` command, the default version will be used. Considering the output above, the following two commands are equivalent:

```bash
[janedoe@shas0136 ~]$ module load intel
[janedoe@shas0136 ~]$ module load intel/17.4
```

Once the compiler is loaded, MPI-implementations and third-party serial libraries that depend on that compiler appear in the available module list until MPI Implementations and Compiler Dependent Applications:

```bash
[janedoe@shas0136 ~]$ module load intel
[janedoe@shas0136 ~]$ module avail
```

```
----------------------------------- MPI Implementations -----------------------------------
impi/17.3
```

```
---------------------------------- Compiler Dependent Applications ----------------------------------
antlr/2.7.7  gdal/2.2.1  gsl/2.4  hdf5/1.10.1 (D)  jasper/1.900.1
fftware/3.3.4  geos/3.6.2  hdf5/1.8.18  intel_cluster_tools/17.3
```

1.11. Compiling and linking
Choosing an MPI implementation will similarly reveal MPI-dependent software under the header **MPI Dependent Applications**:

```
[janedoe@shas0136 ~]$ module load impi
[janedoe@shas0136 ~]$ module avail
---------------------------------- MPI Dependent Applications --------------------------------
 boost/1.64.0          hdf5/1.8.18          lammps/31Mar17          perfsuite/1.1.4
 fftw/3.3.4           (D)       hdf5/1.10.1           (D)       netcdf/4.4.1.1           (D)       petsc/3.8.0
```

You may search for a particular software package using the `module spider` command. This is typically a two-stage process. First search on the general software name without including any version information. If the software exists on our system, a list of available versions will appear:

```
[janedoe@shas0136 ~]$ module spider hdf5
----------------------------------------------------------------
hdf5:
----------------------------------------------------------------
  Description:  HDF5 Tools and Library
  Versions:
        hdf5/1.8.15
        hdf5/1.8.18
        hdf5/1.10.0
        hdf5/1.10.1
```

Finally, to see which modules must be loaded to make your desired version available, run the `module spider` command again with the version information included:

```
[janedoe@shas0136 ~]$ module spider hdf5/1.10.1
----------------------------------------------------------------
hdf5:  hdf5/1.10.1
----------------------------------------------------------------
  Description:  HDF5 Tools and Library
  You will need to load all module(s) on any one of the lines below before the "hdf5/1.10.1" module is available to load.
        intel/17.4
        intel/17.4  impi/17.3
        [...]
```

### 1.11.2 Compiler and Optimization Recommendations

The Summit and Blanca clusters run on Intel-designed hardware. As such, we **strongly recommend** using the Intel compiler along with Intel’s MPI library when compiling software. For production, we suggest compiling with the `-O2` or `-O3` optimization flags along with the vectorization flags appropriate for the node you plan to run on. For Haswell nodes, this means compiling with the `-xCORE-AVX2` flag. For the Xeon-Phi and Skylake nodes, use `-xCORE-AVX512`.

Compilation commands for a typical Summit Haswell node should resemble:

```
$FC -O3 -xCORE-AVX2 my_program.f90 -o my_program.out
$CC -O3 -xCORE-AVX2 my_program.c -o my_program.out
$CXX -O3 -xCORE-AVX2 my_program.cpp -o my_program.out
```
For the Phi and Skylake nodes, the appropriate commands would be:

```
$FC -O3 -xCORE-AVX512 my_program.f90 -o my_program.out
$CC -O3 -xCORE-AVX512 my_program.c -o my_program.out
$CXX -O3 -xCORE-AVX512 my_program.cpp -o my_program.out
```

### 1.11.3 Linking to the Math Kernel Library (MKL)

The Intel Math Kernel Library (MKL) provides optimized routines for a number of common mathematical operations. Notably, it provides interfaces to the LAPack and BLAS linear algebra libraries as well as the FFTW Fourier transform package.

If you wish to link MKL to your Intel-compiled application, use the `-mkl` flag:

```
$CXX -O3 -xCORE-AVX2 my_program.cpp -o my_program.out -mkl
```

If your application uses FFTW, you will also need to include MKL’s FFTW directory in your compilation command:

```
$CXX -O3 -xCORE-AVX2 -I$CURC_MKL_INC/fftw my_program.cpp -o my_program.out -mkl
```

For the GNU and PGI compilers, the link syntax becomes more complex. The Intel Link Advisor can be used to generate the appropriate linking syntax based on your application’s needs.

For the GNU compiler, linking against sequential MKL libraries, the appropriate Fortran linking syntax is:

```
$FC my_program.f90 -m64 -I$CURC_MKL_INC -o my_program.out -L$CURC_MKL_LIB -Wl,--no-as-needed -lmkl_gf_lp64 -lmkl_sequential -lmkl_core -lpthread -lm -ldl
```

The comparable c/c++ syntax would be:

```
$FC my_program.cpp -m64 -I$CURC_MKL_INC -o my_program.out -L$CURC_MKL_LIB -Wl,--no-as-needed -lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lpthread -lm -ldl
```

Note that if your application uses FFTW, you will must use the FFTW include flag just as with the Intel compiler. See the link advisor or contact rc-help@colorado.edu if you have additional questions about how to link MKL to your application.

### 1.12 The PetaLibrary

Use the PetaLibrary to store, archive and share research data.

- Minimum project size: 2 TB
- 2 classes of storage: active and archive
  - 5 different storage options within these classes
  - See our website for pricing information
- New customers are limited to either
  - 75 TB in Active Storage, or
  - 100 TB in Archive Storage
  - Email rc-help@colorado.edu to request more space if necessary
1.12.1 Accessing the PetaLibrary

Request access to the PetaLibrary by:

- Reading the PetaLibrary Memorandum of Understanding
- Filling out the attached End User Agreement and Order Form (found in the MOU), and emailing those documents to rc-help@colorado.edu

Each person who accesses the PetaLibrary is required to have a Research Computing account and two-factor authentication.

- **To access active storage:** Log in to a Research Computing login node and use the command `/work/`
- **To access archive storage:** Use the command `/archive/

Note that access via the login nodes is not recommended for frequent or large read/writes of archived data.

1.12.2 More information

- [https://www.colorado.edu/rc/resources/petalibrary](https://www.colorado.edu/rc/resources/petalibrary)

1.13 Running applications with Jobs

Because Summit is shared among many researchers, Research Computing manages usage of the system through jobs. Jobs are simply an allotment of resources that can be used to execute processes. Research Computing uses a program named the *Simple Linux Utility for Resource Management*, or *Slurm*, to create and manage jobs.

In order to run a program on Summit, you must request resources from Slurm to generate a job. Resources can be requested from a login node or a compile node. You must then provide commands to run your program on those requested resources. Where you provide you commands depends on whether you are running a batch job or an interactive job.

When you submit a batch job or an interactive job, it will be placed in a queue until resources are available. A detailed guide on the Slurm queue and accounting tools can be found [here](https://www.colorado.edu/rc/resources/petalibrary).

1.13.1 Batch Jobs

The primary method of running applications on Research Computing resources is through a batch job. A **batch job** is a job that runs on a compute node with little or no interaction with the users. You should use batch jobs for:

- Any computationally expensive application that could take hours or days to run
- Any application that requires little or no user input
- Applications that you do not need to monitor extensively

Unlike running an application on your personal machine, you do not call the application you wish to run directly. Instead you create a **job script** that includes a call to your application. Job scripts are simply a set of resource requests and commands. When a job script is submitted all the commands in the job script are executed on a compute node.

Once you’ve created your job script it can be submitted to the Slurm queue with the `sbatch` command followed by your job script name:
sbatch <your-jobscript-name>

If no job script is provided then `sbatch` will take whatever commands follow as standard input. A detailed guide on constructing and submitting Job scripts can be found here.

### 1.13.2 Interactive Jobs

Another method of running applications on Research Computing resources is through an interactive job. As the name would imply, an **interactive job** is a job that allows users to interact with requested resources in real time. Users can run applications, execute scripts, or run other commands directly on a compute node. Interactive jobs should be used for:

- Debugging applications or workflows
- Any application that requires user input at runtime
- Any application with a GUI (Graphical User Interface)

You can request an interactive job by using the `sinteractive` command. Unlike the `sbatch`, resources must be requested via the command line through the use of flags. Though running `sinteractive` without any flags is possible, this will result in default values being used for your jobs. Research Computing highly recommends you provide a `qos` and a `time` parameter to avoid long queue times or accidental overuse of your priority.

```
sinteractive --qos=interactive --time=00:10:00
```

A list of `sinteractive` parameters can be found here

The example above will submit an interactive job that will run a terminal session on one core of one node with the interactive quality of service (QoS) for ten minutes. Once the interactive session has started you can run any interactive terminal application you may need on the command line.

More details on running Interactive Jobs can be found here.

### 1.14 Batch Jobs and Job Scripting

Batch jobs are by far the most common type of job on Summit. Batch jobs are resource provisions that run applications on nodes away from the user and do not require supervision or interaction. Batch jobs are commonly used for applications that run for long periods of time or require little to no user input.

Batch jobs are created from a job script which provide resource requirements and commands for the job.

#### 1.14.1 Job Scripts

Even though it is possible to submit jobs completely from the command line, it is often overly tedious and unorganized to do so. Instead, Research Computing recommends constructing a job script for your batch jobs. A **job script** is set of Linux commands paired with a set of resource requirements that can be submitted to the Slurm job scheduler. Slurm will then generate a job according to the parameters set in the job script. Any commands that are included with the job script will be run within the job.
1.14.2 Submitting a Job Script

Submitting a job script can be done with the `sbatch` command:

```
sbatch <your-job-script-name>
```

Because job scripts specify the desired resources for your job, you won’t need to specify any resources on the command line. You can, however, overwrite or add any job parameter by providing the specific resource as a flag within `sbatch` command:

```
sbatch --partition=sgpu <your-job-script>
```

Running this command would force your job to run on the sgpu partition *no matter what your job script specified.*

1.14.3 Making a Job Script

Although Research Computing provides a variety of different sample scripts users can utilize when running their own jobs, knowing how to draft a job script can be quite handy if you need to debug any errors in your jobs or you need to make substantial changes to a script.

A job script looks something like this:

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --time=00:10:00
#SBATCH --partition=shas-testing
#SBATCH --qos=testing
#SBATCH --output=sample-%j.out
module purge
module load intel
module load mkl

echo "== This is the scripting step! =="
sleep 30
./executable.exe

echo "== End of Job =="
```

Normally job scripts are divided into 3 primary parts: directives, loading software, and user scripting. Directives give the terminal and the Slurm daemon instructions on setting up the job. Loading software involves cleaning out the environment and loading specific pieces of software you need for your job. User scripting is simply the commands you wish to be executed in your job.

1. Directives

A directive is a comment that is included at the top of a job script that tells the shell information about the script. The first directive, the shebang directive, is always on the first line of any script. The directive indicates which shell you want running commands in your job. Most users employ bash as their shell, so we will specify bash by typing:

```
#!/bin/bash
```
The next directives that must be included with your job script are `sbatch` directives. These directives specify resource requirements to Slurm for a batch job. These directives must come after the shebang directive and before any commands are issued in the job script. Each directive contains a flag that requests resource the job would need to complete execution. An `sbatch` directive is written as such:

```
#SBATCH --<resource>=<amount>
```

For example if you wanted to request 2 nodes with an `sbatch` directive, you would write:

```
#SBATCH --nodes=2
```

A list of some useful `sbatch` directives can be found here. A full list of commands can be found in Slurm’s documentation for `sbatch`.

### 2. Software

Because jobs run on a different node than from where you submit, any shared software that is needed must be loaded via the job script. Software can be loaded in a job script just like it would be on the command line. First we will purge all software that may be left behind from a previous job by running the command:

```
module purge
```

After this you can load whatever software you need by running the following command:

```
module load <software>
```

More information about software modules can be found here.

### 3. User Scripting

The last part of a job script is the actual user scripting that will execute when the job is submitted. This part of the job script includes all user commands that are needed to set up and execute the desired task. Any Linux command can be utilized in this step. Scripting can range from highly complex loops iterating over thousands of files to a simple call to an executable. Below is an simple example of some user scripting:

```
echo "== This is the scripting step! =="
touch tempFile1.in
touch tempFile2.in
sleep 30
./executable.exe tempfile1.in tempfile2.in
echo "== End of Job =="
```

### 1.14.4 Examples

Job script to run a 5 minute long, 1 node, 1 core C++ Job:

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --time=00:05:00
```

(continues on next page)
#SBATCH --qos=testing
#SBATCH --partition=shas-testing
#SBATCH --ntasks=1
#SBATCH --job-name=cpp-job
#SBATCH --output=cpp-job.%j.out

module purge
module load gcc
./example_cpp.exe

Job script to run a 7 minute long, 1 node, 4 core C++ OpenMP Job:

#!/bin/bash
#SBATCH --nodes=1
#SBATCH --time=00:07:00
#SBATCH --qos=testing
#SBATCH --partition=shas-testing
#SBATCH --ntasks=4
#SBATCH --job-name=omp-cpp-job
#SBATCH --output=omp-cpp-job.%j.out

module purge
module load gcc
export OMP_NUM_THREADS=4
./example_omp.exe

Job script to run a 10 minute long, 2 node, 24 core C++ MPI Job:

#!/bin/bash
#SBATCH --nodes=2
#SBATCH --time=00:10:00
#SBATCH --qos=testing
#SBATCH --partition=shas-testing
#SBATCH --ntasks=24
#SBATCH --job-name=mpi-cpp-job
#SBATCH --output=mpi-cpp-job.%j.out

module purge
module load intel
module load impi
mpirun -np 24 ./example_mpi.exe

1.14.5 Job Flags

The `sbatch` command supports many optional flags. To review all the options, please visit the Slurm `sbatch` page. Below are a few flags you may want to consider when submitting your job via `sbatch`. 
1.15 Interactive jobs

Interactive jobs allow a user to interact with applications in real time within an HPC environment. With interactive jobs, users request time and resources to work on a compute node directly. Users can then run graphical user interface (GUI) applications, execute scripts, or run other commands directly on a compute node. Common reasons for running interactive jobs include debugging, designing workflows, or preference in using the GUI interface of an application.

1.15.1 General Interactive Jobs

To run an interactive job on Research Computing resources, request an interactive session by utilizing the `sinteractive` command. The `sinteractive` command creates a job with parameters provided through flags run with the command. After moving through the Slurm queue the interactive job will put the user onto the command line of a compute node to interactively use their resource allotment.

Any resource that could be specified in a job script or with `sbatch` can also be used with `sinteractive`. The primary flags we recommend users specify are the `qos` flag and the `time` flag. These flags will specify quality of service and amount of time for your job respectively. The `sinteractive` command is run as follows:

```
sinteractive --qos=interactive --time=00:10:00
```

This will submit an interactive job to the Slurm queue that will start a terminal session that will run on one core of one node with the interactive quality of service for ten minutes. Once the session has started you can run any application or script you may need from the command line. For example, if you type `python` you will open an interactive python shell on a compute node (rather than the login nodes, which is forbidden).

1.15.2 Interactive GUI Applications

To run an interactive GUI application on Summit, we must install an X windows server application and enable X11 forwarding on our personal computer.

Windows setup

On Windows we must first install an X windows server application to allow Summit to forward the GUI information to your local system. For Windows, we will use an application called Xming to accomplish this. Download the Xming here.

Next we must enable X11 forwarding on the PuTTY application. Download and install the PuTTY application here if you have not done so already.
Expand the SSH tab on the left side of the application and click X11.
In the X11 Menu check the “Enable X11 Forwarding” checkbox and type “localhost:0” in the X display location field. Clicking open will open a terminal window where you can login.

macOS setup

Using macOS, we will also need to install an X windows server application to allow Summit to forward GUI information to your local system. For Mac, we will use an application called XQuartz to accomplish this. Download and install XQuartz here.

Opening the application will bring up a terminal window. In this window, you will ssh to login.rc.colorado.edu as you normally would except you’ll include the “-X” flag:

```
ssh -X your_rc-username@login.rc.colorado.edu
```

Running GUI Applications

Once you have logged into Summit with X11 Forwarding enabled, you will be able to initialize a GUI application by starting an interactive job and running your selected application. The X-window server application installed on your local system will display the window generated on the cluster on your local machine.

If you plan on submitting your interactive job from a compile node, you must also enable x11 forwarding when you ssh into scompile:

```
ssh -X scompile
```

From here you will be able to submit your interactive job like normal and X11 forwarding will carry through to the job.
1.16 Useful Slurm commands

Slurm provides a variety of tools that allow a user to manage and understand their jobs. This tutorial will introduce these tools, as well as provide details on how to use them.

1.16.1 Finding queuing information with `squeue`

The `squeue` command is a tool we use to pull up information about the jobs in queue. By default, the `squeue` command will print out the `job ID, partition, username, job status, number of nodes, and name of nodes` for all jobs queued or running within Slurm. Usually you wouldn’t need information for all jobs that were queued in the system, so we can specify jobs that only you are running with the `--user` flag:

```
$ squeue --user=your_rc-username
```

We can output non-abbreviated information with the `--long` flag. This flag will print out the non-abbreviated default information with the addition of a `timelimit` field:

```
$ squeue --user=your_rc-username --long
```

The `squeue` command also provides users with a means to calculate a job’s estimated start time by adding the `--start` flag to our command. This will append Slurm’s estimated start time for each job in our output information. (Note: The start time provided by this command can be inaccurate. This is because the time calculated is based on jobs queued or running in the system. If a job with a higher priority is queued after the command is run, your job may be delayed.)

```
$ squeue --user=your_rc-username --start
```

When checking the status of a job, you may want to repeatedly call the `squeue` command to check for updates. We can accomplish this by adding the `--iterate` flag to our `squeue` command. This will run `squeue` every n seconds, allowing for a frequent, continuous update of queue information without needing to repeatedly call `squeue`:

```
$ squeue --user=your_rc-username --start --iterate=n_seconds
```

Press `ctrl-z` to stop the command from looping and bring you back to the terminal.

For more information on `squeue`, visit [the Slurm page on squeue](#).

1.16.2 Stopping jobs with `scancel`

Sometimes you may need to stop a job entirely while it’s running. The best way to accomplish this is with the `scancel` command. The `scancel` command allows you to cancel jobs you are running on Research Computing resources using the job’s ID. The command looks like this:

```
$ scancel your_job-id
```

To cancel multiple jobs, you can use a comma-separated list of job IDs:

```
$ scancel your_job-id1, your_job-id2, your_jobid3
```

For more information, visit [the Slurm manual on scancel](#).
1.16.3 Learning status information with `sstat`

The `sstat` command allows users to easily pull up status information about their jobs. This includes information about CPU usage, task information, node information, resident set size (RSS), and virtual memory (VM). We can invoke the `sstat` command as such:

```
$ sstat --jobs=your_job-id
```

By default, `sstat` will pull up significantly more information than what would be needed in the command's default output. To remedy this, we can use the `--format` flag to choose what we want in our output. The format flag takes a list of comma separated variables which specify output data:

```
$ sstat --jobs=your_job-id --format=var_1,var_2, ... , var_N
```

A chart of some these variables are listed in the table below:

For an example, let's print out a job's average job id, cpu time, max rss, and number of tasks. We can do this by typing out the command:

```
sstat --jobs=your_job-id --format=jobid,cputime,maxrss,ntasks
```

A full list of variables that specify data handled by `sstat` can be found with the `--helpformat` flag or by visiting the slurm page on `sstat`.

1.16.4 Analyzing past jobs with `sacct`

The `sacct` command allows users to pull up status information about past jobs. This command is very similar to `sstat`, but is used on jobs that have been previously run on the system instead of currently running jobs. We can use a job's id...

```
$ sacct --jobs=your_job-id
```

... or your rc username...

```
$ sacct --user=your_rc-username
```

... to pull up accounting information on jobs run at an earlier time.

By default, `sacct` will only pull up jobs that were run on the current day. We can use the `--starttime` flag to tell the command to look beyond its short-term cache of jobs.

```
$ sacct --jobs=your_job-id --starttime=YYYY-MM-DD
```

To see a non-abbreviated version of `sacct` output, use the `--long` flag:

```
$ sacct --jobs=your_job-id --starttime=YYYY-MM-DD --long
```

**Formatting `sacct` output**

Like `sstat`, the standard output of `sacct` may not provide the information we want. To remedy this, we can use the `--format` flag to choose what we want in our output. Similarly, the format flag is handled by a list of comma separated variables which specify output data:

```
$ sacct --user=your_rc-username --format=var_1,var_2, ... ,var_N
```
A chart of some variables is provided below:

As an example, suppose you want to find information about jobs that were run on March 12, 2018. You want to show information regarding the job name, the number of nodes used in the job, the number of cpus, the maxrss, and the elapsed time. Your command would look like this:

```
$ sacct --jobs=your_job-id --starttime=2018-03-12 --format=jobname,nnodes,ncpus,maxrss,elapsed
```

As another example, suppose you would like to pull up information on jobs that were run on February 21, 2018. You would like information on job ID, job name, QoS, Number of Nodes used, Number of CPUs used, Maximum RSS, CPU time, Average CPU time, and elapsed time. Your command would look like this:

```
$ sacct --jobs=your_job-id --starttime=2018-02-21 --format=jobid,jobname,qos,nnodes,ncpu,maxrss,cputime,avecpu,elapsed
```

A full list of variables that specify data handled by sacct can be found with the --helpformat flag or by visiting the slurm page on sacct.

### 1.16.5 Controlling queued and running jobs using scontrol

The `scontrol` command provides users extended control of their jobs run through Slurm. This includes actions like suspending a job, holding a job from running, or pulling extensive status information on jobs.

To suspend a job that is currently running on the system, we can use `scontrol` with the `suspend` command. This will stop a running job on its current step that can be resumed at a later time. We can suspend a job by typing the command:

```
$ scontrol suspend job_id
```

To resume a paused job, we use `scontrol` with the `resume` command:

```
$ scontrol resume job_id
```

Slurm also provides a utility to hold jobs that are queued in the system. Holding a job will place the job in the lowest priority, effectively “holding” the job from being run. A job can only be held if it’s waiting on the system to be run. We use the `hold` command to place a job into a held state:

```
$ scontrol hold job_id
```

We can then release a held job using the `release` command:

```
$ scontrol release job_id
```

Scontrol can also provide information on jobs using the `show job` command. The information provided from this command is quite extensive and detailed, so be sure to either clear your terminal window, grep certain information from the command, or pipe the output to a separate text file:

```
# Output to console
$ scontrol show job job_id

# Streaming output to a textfile
$ scontrol show job job_id > outputfile.txt

# Piping output to Grep and find lines containing the word "Time"
$ scontrol show job job_id | grep Time
```
For a full primer on grep and regular expressions, visit GNU’s page on Grep
For more information on scontrol, visit the Slurm page on scontrol

1.17 Job Resource Information

Slurm allows the use of flags to specify resources needed for a job. Below is a table describing some of the most common Slurm resource flags, followed by tables describing available Summit partitions and Quality of Service (QoS) options.

1.17.1 Slurm Resource Flags

Job scripts, the `sbatch` command, and the `sinteractive` command support many different resource requests in the form of flags. To review all options for `sbatch`, please visit the Slurm `sbatch` page. Below, we have listed some flags to consider when submitting your job script.

1.17.2 Partitions

On Summit, nodes with the same hardware configuration are grouped into partitions. You will need to specify a partition using `--partition` in your job script in order for your job to run on the appropriate type of node.

These are the partitions available on Summit.

*The `smem` partition is limited to 1 node per job. If you need more memory or cores, please contact rc-help@colorado.edu.

In addition to these partitions, Research Computing also provides specialized partitions for interactive and test jobs. Each of these partitions must be paired with their corresponding Quality of Service (see QoS options below).

*The `shas testing` partition is limited to 24 cores total. These cores can be spread upon multiple nodes but only 24 will be available for the partition.

To run a job longer than 24 hours on the `shas`, `sgpu`, or `sknl` partitions, use the `long QOS`.

More details about each type of node can be found here.

1.17.3 Quality of Service

On Summit, Quality of Service or QoS is used to constrain or modify the characteristics that a job can have. For example, by selecting the `testing` QoS, a user can obtain higher queue priority for a job with the trade-off that the maximum allowed wall time is reduced from what would otherwise be allowed on that partition. We recommend specifying a QoS as well as a partition for every job.

The available QoS’s for Summit are:

*The `normal QOS` is the default QOS if no other is specified.

The `testing` and `interactive QOS` must be paired with a `testing` or `interactive partition`. Jobs that utilize `testing` and `interactive QOS` will fail if paired with a any other partition
1.18 `squeue` status and reason codes

The `squeue` command details a variety of information on an active job’s status including reasons why a job has a particular status. The following tables outline a variety of job state and reason codes you may encounter when using `squeue` to check on your jobs. *Job state codes* describe a job’s current state in queue (e.g. pending, completed). *Job reason codes* describe the reason why the job is in its current state.

1.18.1 Job State Codes

A full list of these Job State codes can be found in Slurm’s documentation.

1.18.2 Job Reason Codes

A full list of these Job Reason Codes can be found in Slurm’s documentation.

1.19 Running jobs on RoCE enabled Nodes

We have some nodes in Blanca that are equipped with Mellanox 10G cards and RoCE v2 enabled switches to enable users to run MPI jobs over the 10G interfaces. While the 10G network is not as performant with regards to latency as Infiniband or Omnipath, you can still get line speed for bandwidth.

In order to take advantage of RoCE on these nodes, you will need to compile your code with a MPI compiler that was built with support for Unified Communication X (UCX). Without UCX a job submitted to these nodes will fail.

1.19.1 Using pre-built modules

You can easily build/rebuild your binaries with support for the 10G RoCE network by building your code with the module keys gcc/8.2.0 and openmpi_ucx/4.0.0. Once you have loaded those software keys you can begin building your code as you normally would.

1.19.2 Build a MPI compiler with support for UCX

First ensure that you have UCX installed on the node you intend to build the MPI on:

```
yum info ucx ucx-devel
```

Then you can move on to building the MPI, in this example we are using the default GNU compiler, and are using the most recent version for OpenMPI.

```
./configure --prefix=/home/job16604/soft/openmpi-4.0.0 --with-ucx
```

After successfully building the MPI, you can then compile your code against it and start running jobs. You do not need to worry about passing any flags or arguments into the MPI command for your job script.

1.19.3 Tips

If you are still having issues trying to run your code you can try passing some flags to MPI
1.20 JupyterHub

JupyterHub is a multi-user server for Jupyter (formerly known as IPython) notebooks. It provides a web service that allows you to create and share documents that contain live code, equations, visualizations and explanatory text. The CU Research Computing environment includes support for parallel computation on local HPC resources.

JupyterHub is available at https://jupyter.rc.colorado.edu. To log in use your RC credentials. If you do not have an RC account, please request an account before continuing.

1.20.1 Start a notebook server

To start a notebook server, select one of the available options in the Select job profile menu and click Spawn. Available options are explained below.

Summit Haswell

These notebook servers start on the shas partition using the normal QOS. The following features are supported by Summit Haswell notebooks:

• Access to standard RC file systems
• Access to RC environment modules
• Python 3 notebooks
• Python 2.7 notebooks
• BASH notebooks
• IPyParallel/IPython clusters

Summit Knight’s Landing

These notebook servers start on the sknl partition using the normal QOS. The following features are supported by Summit KNL notebooks:

• Access to standard RC file systems
• Access to RC environment modules
• Python 3 notebooks
• Python 2.7 notebooks
• BASH notebooks
• IPyParallel/IPython clusters

1.20.2 Stopping a Notebook Server

Use the Stop My Server button in the Control Panel to shut down the Jupyter notebook server when finished.
1.20.3 Troubleshooting

Jupyter notebook servers spawned on RC compute resources log to ~/.jupyterhub-spawner.log. Watching the contents of this file provides useful information regarding any problems encountered during notebook startup or execution.

1.20.4 IPython Clusters

Notebook servers started on RC compute resources can also launch IPython clusters for parallel processing. See IPython Parallel and MPI for Python for general information on parallel processing with IPython clusters.

The RC environment provisions two IPython profiles automatically which can be used as a reference point for starting IPython clusters. These profiles are available in the IPython Clusters tab.

**default**

The default profile is generated automatically by IPython. Engines are dispatched in the same resources as the notebook server, and provide no MPI support or cluster performance.

**example-shas**

Each example-shas engine provides access to a Summit Haswell CPU thread. Multiple engines are aggregated into a single MPI session.

Note: IPython engines on RC cluster resources are provisioned as batch jobs using Slurm, but Jupyter does not yet report queue progress. The exception “NoEnginesRegistered: Can’t build targets without any engines” indicates that the cluster job is still in the queue and not ready to accept work.

1.20.5 Updating Your JupyterHub Config

>This only applies if you used JupyterHub at RC prior to September 2017.

The September 2017 update of our JupyterHub deployment deprecates some of the Jupyter and IPython configuration in your home directory. Updating to the new configuration (which is required for documented features to work properly) is a manual process.

Start by removing the stale Jupyter config from your home directory:

```bash
rm -rf $HOME/.jupyter
```

Remove all IPython profiles for Crestone and Janus:

```bash
cd $HOME/.ipython
rm -rf profile_crestone-*
rm -rf profile_janus-*
```

The new configurations will be automatically applied the next time you start a notebook server.

1.20.6 See Also

- Parallel Programming with Jupyter Notebooks
- RC JupyterHub CHANGELOG
1.21 Parallel programming with Jupyter

This tutorial demonstrates simple parallel processing examples using the CURC JupyterHub web service, in both ipyparallel and MPI for Python.

1.21.1 Prerequisites

Before you begin, you need

- an RC account
- Duo credentials
- access to a CU campus network or VPN
- a web browser

1.21.2 Log in

First, log into the RC JupyterHub service by navigating to https://jupyter.rc.colorado.edu. Enter your RC username and password (where your password may be a combination of a password, OTP, and/or credential prefix).

After authenticating, you should be redirected to your Jupyter notebook server; but this process is a queued process, so you may have to wait if the JupyterHub resources are otherwise fully utilized.

1.21.3 Prepare an IPython cluster

Navigate to the IPython Clusters tab to access a list of available parallel profiles. Each profile represents an IPython cluster you can initialize, with a predefined configuration; the # of engines is the number of processes you will spawn for the cluster.

Any of the RC-provided cluster profiles (though not the default profile) can be used for these examples. Specify 2 engines for the example-shas node, and use the Start button to start the compute cluster.

1.21.4 Creating a parallel-processing notebook

Return to the Files tab and use the New button to create a Python 3 notebook. A new notebook should include an initial Python code cell; but, if necessary, use the Insert menu to insert a new cell, and use the Cell > Cell Type menu to configure the new cell as a Code cell.

This first cell includes code to initialize access to the running cluster using ipyparallel. You can simply paste this code into the cell, all of which executes within the Jupyter notebook.

```python
import ipyparallel

# attach to a running cluster
cluster = ipyparallel.Client(profile='example-shas')
print('profile:', cluster.profile)
print("IDs:", cluster.ids) # Print process id numbers
```

Execute the cell using Shift+Return, which produces output identifying the engine IDs available in the cluster.

```
profile: example-shas
IDs: [0, 1]
```
Note: IPython engines on RC cluster resources are provisioned as batch jobs using Slurm, but Jupyter does not yet report queue progress. If no IDs are listed, or an exception “NoEnginesRegistered: Can’t build targets without any engines” is raised, the cluster job is still in the queue and is not ready to accept work.

1.21.5 IPython Parallel

Once the above code has successfully reported the engine IDs for the cluster, insert a new code cell below the existing code block.

```python
# The %px magic executes a single Python command on the engines specified by the targets attribute of the DirectView instance.
#
%px import socket
%px print("hosts:", socket.gethostname())

# calculate square numbers in parallel. Print result.
squares = cluster[:].map_sync(lambda x: x**2, range(32))
print("squares:", squares)
```

Execute this cell using Shift+Return, which outputs the hostname of the host for each engine, as well as the calculated square numbers in each cell. In this cell, code prepended with %px is executed in each engine; and the squares are calculated using the cluster reference obtained in the previous code block.

MPI for Python

Insert a new code cell below the existing code block to demonstrate message passing using MPI.

```python
%%px
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# passing MPI datatypes explicitly
if rank == 0:
    data = numpy.arange(100, dtype='i')
    numpy.random.shuffle(data)
    comm.Send([data, MPI.INT], dest=1, tag=77)
    print("{}: sent data to 1: {}".format(rank, data))
elif rank == 1:
    data = numpy.empty(100, dtype='i')
    comm.Recv([data, MPI.INT], source=0, tag=77)
    print("{}: received data from 0: {}".format(rank, data))
else:
    print("{}: idle".format(rank))
```

Execute this cell using Shift+Return. This cell is prepended with %%px, which causes the entire cell to execute in parallel on all nodes. Rank 0 generates a random numpy array which is then sent to rank 1.
1.21.6 Shutting down

Both the IPython cluster and notebook server will persist until manually stopped or each reaches its time limit. IPython clusters have a 4-hour time limit by default (though this can be changed by editing the profile in $HOME/.ipython/). The notebook server itself has a time limit of 2 or 12 hours depending on which job profile you select. When you are done using a resource, please shut it down so that the resources can be used for other work.

Return to the “IPython Clusters” tab and press the “Stop” button for example-shas (or whichever profile was used during the example).

Finally, if you are done using Jupyter notebook for now, access the “Control Panel” and press the “Stop My Server” button to stop the Jupyter notebook server. After that, you may press “Logout”, or simply close the browser window.

1.22 EnginFrame

NICE EnginFrame provides a 3d-accelerated remote desktop environment on an Nvidia GPU-equipped compute node. Coupled with the proprietary Desktop Cloud Visualization (DCV) VNC server, the EnginFrame service supports the use of common visualization applications in a typical desktop environment using only a modern web browser.

1.22.1 Accessing EnginFrame

Access to EnginFrame is granted on request. Request access by sending email to rc-help@colorado.edu.

Once access has been granted, EnginFrame is available at https://viz.rc.colorado.edu/.

From the welcome page, select “Views” from the available interfaces (or use this direct link).
Provide your RC login credentials at the login prompt. You will be prompted to use a second authentication factor (e.g., the Duo mobile app) to log in.

1.22.2 Remote desktop

After logging in, select “Remote Desktop” from the list of services in the left sidebar. (Other custom services may be configured for you as well.)

When starting a Remote Desktop session you may customize the resources allocated to the session and other characteristics of the dispatched Slurm job. In most cases the defaults should be sufficient; however, you may need to supply a Slurm account if you are associated with more than one and you do not want to use your default account.
Remote Desktop service

Once the session has started, a thumbnail of the running session appears in the Sessions list. EnginFrame will attempt to open the session automatically, but may be blocked by the browser. In that case, simply select the session thumbnail from the list, or use the “click here” link in the notification text.

Desktop session

With the Remote Desktop session running and open, you should be able to run standard Linux desktop applications, including 3d-accelerated OpenGL applications.
1.22.3 Additional Resources

- https://www.nice-software.com/products/enginframe
- https://www.nice-software.com/products/dcv

1.23 Load Balancer

The CU Research Computing Load Balancer is an effective tool for optimally utilizing multiple processors and nodes on the Summit HPC resource, without the need to learn OpenMP or MPI. This tutorial assumes user knowledge of Slurm job submission, shell scripting, and some python.

1.23.1 Why Use the Load Balancer?

Suppose you have a very simple serial program that crops a photo, and you need to apply it to crop several million photos. You could rewrite the serial program into a parallel program that would utilize multiple processors to more quickly run the program over the entire set of photos (compared to doing one-at-a-time), but this would require some knowledge of parallel programming. Even worse, if your code is in a language that has limited parallelization capabilities, this may not be an option. The easiest solution for this problem is to utilize the Load Balancer.

1.23.2 Using the Load Balancer

The Load Balancer is a tool written by CU Boulder Research Computing that allows shell commands (for example, calls to serial programs) to be distributed amongst nodes and cores on Summit. This means code doesn’t need to be explicitly parallelized for MPI or OpenMP. Additionally, code can be written in any language that can be run from a Linux shell.
Let’s create a simple ‘Hello World’ serial python script to demonstrate the Load Balancer tool. We will call the script `hello_World.py` and it will print “Hello World from process: ” followed by a command line argument:

```python
import sys

print “Hello World from process: “, sys.argv[1]
```

Now we will create a list of calls to the python script that will be distributed to multiple cores. (Each compute node has one or more discrete compute processor; most modern processors are made up of multiple compute “cores”, each of which can operate independently and simultaneously.)

Instead of slowly typing out commands one-at-a-time, we will use a bash shell script to create our commands. In a text editor, create a bash shell script called `create_hello.sh`, that has the following text:

```bash
#!/bin/bash

for i in {1..4}
   do
      echo “python hello_World.py $i;” >> lb_cmd_file
   done
```

Next run the bash script by first changing permissions of the script to be executable by typing: `chmod +x create_hello.sh` and then by typing: `./create_hello.sh` at the terminal prompt. It will create a file called `lb_cmd_file` that contains 4 calls to our `hello_World.py` script:

```
python hello_World.py 1;
python hello_World.py 2;
python hello_World.py 3;
python hello_World.py 4;
```

Now create a job script called `run_hello.sh` that will submit all instances of your python script in `lb_cmd_file` using the Load Balancer. Within the script, before using Load Balancer, we need to load the Intel C++ compiler, Intel MPI, Python, and the Load Balancer utility itself. We load Intel and Intel MPI because the Load Balancer uses `mpirun` to parallelize commands. Your job script should look something like this:

```bash
#!/bin/bash

#SBATCH --nodes=1
#SBATCH --time 00:02:00
#SBATCH --qos=testing
#SBATCH --partition shas-testing
#SBATCH --ntasks=4
#SBATCH --job-name lbPythonDemo
#SBATCH --output loadbalance.out

module purge
module load intel
module load impi
module load python
module load loadbalance

mpirun lb lb_cmd_file
```

Submitting this script via `sbatch` will run the commands we stored in `lb_cmd_file` in parallel. A successful job will result in output that looks something like this:
1.23.3 Additional Resources

- https://github.com/ResearchComputing/lb
- https://github.com/t-brown/lb

1.24 Gaussian

Important: Gaussian is available on Summit only to members of universities that have purchased Gaussian licenses. It cannot be run by other Summit users. Please note and abide by the licensing, rights, and citation information shown at the top of your Gaussian output files.

This document describes how to submit G16 jobs efficiently on Summit. It does not attempt to teach how to use Gaussian for solving science/engineering questions.

Good general instructions can be found at here; however some minor modifications are needed when running on Summit.

1.24.1 Environment

To set up your shell environment to use G16, load a Gaussian software module (e.g. module load gaussian/16_avx2). Nearly all necessary environment variables are configured for you via the module. You do not need to source g16.login or g16.profile.

However, it is important to specify GAUSS_SCRDIR to tell G16 where to put its large scratch files. These should always be on one of Summit’s scratch storage systems (either /scratch/local [accessed via the SLURM_SCRATCH env variable] or /scratch/summit/$USER.) If GAUSS_SCRDIR is not set, then the scratch files will be created in whatever directory G16 is run from; if this directory is in /projects or /work, then your job’s performance will be dramatically reduced.

1.24.2 Running G16

If you create a Gaussian input file named h2o_dft.com then you can execute it simply via g16 h2o_dft. Output from the computation will go to a file called h2o_dft.log.

1.24.3 Memory

The default dynamic memory request in G16 is frequently too small to support the amount of memory that needs to be allocated to efficiently support computations on even modest-sized molecules. If too little memory is requested, the job can crash. Thus, use the ‘-m’ flag in your g16 command line (e.g. -m=48gb) to specify at least of 50% of the amount of memory your Slurm job has requested.
1.24.4 Parallel jobs

**Single-node (SMP) parallelism**

Many G16 functions scale well to 8 or more cores on the same node. You can specify how many cores to use via the `-p` flag to g16 (e.g. `-p=24`). This value should correspond to the number of cores that your Slurm job has requested. You should test your G16 computations with several different core counts to see how well they scale, as there may be diminishing returns beyond a certain number of cores.

**Example SMP BASH script:**

```bash
#!/bin/bash
#SBATCH --job-name g16-test
#SBATCH -p shas
#SBATCH --qos normal
#SBATCH --nodes 1
#SBATCH --ntasks-per-node 24
#SBATCH --time 00:50:00
#SBATCH --output g16-test.%j.out
module load gaussian/16_avx2
# Always specify a scratch directory on a fast storage space (not /home or /projects!)
export GAUSS_SCRDIR=/scratch/summit/$USER/$SLURM_JOBID
# or export GAUSS_SCRDIR=$SLURM_SCRATCH
# alternatively, to use the local SSD; max 159GB available
# the next line prevents OpenMP parallelism from conflicting with Gaussian's internal SMP parallelization
export OMP_NUM_THREADS=1
mkdir $GAUSS_SCRDIR  # only needed if using /scratch/summit
date  # put a date stamp in the output file for timing/scaling testing if desired
g16 -m=50gb -p=24 my_input.com
date
```

**Multi-node (Linda) parallelism**

In order to run on more than 24 cores in the “shas” partition on Summit, your job will need to span multiple nodes using the Linda network parallel communication model. We advise using one Linda worker per node, with multiple SMP cores per node. For example, your g16 flags might include

```
-p=24 -s=ssh -w=shas0521-opa,shas0604-opa,shas0622-opa
```

which tells G16 to use 24 cores on each of three shas nodes, and to set up the connections between nodes using ssh.

Since you don’t know in advance what nodes your job will be assigned to, you will have to determine the arguments for `-w` at runtime via commands in your Slurm batch script. See the batch script example below.

Not all G16 computations scale efficiently beyond a single node! According to the G16 documentation: “HF, CIS=Direct, and DFT calculations are Linda parallel, including energies, optimizations, and frequencies. TDDFT energies and gradients and MP2 energies and gradients are also Linda parallel. Portions of MP2 frequency and CCSD calculations are Linda parallel.” As with SMP parallelism, testing the scaling of your Linda parallel computation is very important.
Linda Parallel

#!/bin/bash

#SBATCH --job-name g16-test
#SBATCH -p shas
#SBATCH --qos normal
#SBATCH --nodes 2
#SBATCH --ntasks-per-node 1
#SBATCH --cpus-per-task 24
#SBATCH --time 00:50:00
#SBATCH --output g16-test.%j.out

module load gaussian/16_avx2

for n in `scontrol show hostname | sort -u`; do
  echo $n-opa
done | paste -s -d, > tsnet.nodes.$SLURM_JOBID

# Always specify a scratch directory on a fast storage space (not /home or /projects!)
export GAUSS_SCRDIR=/scratch/summit/$USER/$SLURM_JOBID

# the next line prevents OpenMP parallelism from conflicting with Gaussian's internal parallelization
export OMP_NUM_THREADS=1

# the next line increases the verbosity of Linda output messages
export GAUSS_LFLAGS="-v"

mkdir $GAUSS_SCRDIR  # only needed if using /scratch/summit

date  # put a date stamp in the output file for timing/scaling testing

g16 -m=20gb -p=24 -w=`cat tsnet.nodes.$SLURM_JOBID` my_input.com

date

rm tsnet.nodes.$SLURM_JOBID

G16 on GPU

Please read http://gaussian.com/running/?tabid=5 carefully to determine whether the K80 GPUs in Summit’s “sgpu” partition will be effective for your calculations. In many cases, SMP parallelization across all of the cores in a shas node will provide better speedup than offloading computational work to a GPU.

G16 on Knight’s Landing

We do not recommend running Gaussian16 on RMACC Summit’s “sknl” partition.

1.24.5 Sample input file

Here’s an input file that can be used for both SMP and Linda parallel testing:

```
#P b3lyp/6-31g* test stable=(opt,qconly)

Gaussian Test Job 135:
Fe=O perpendicular to ethene, in triplet state.
```

(continues on next page)
1.25 Matlab

Research Computing (RC) provides a large suite of software on RC resources. In this tutorial we will learn how to run Matlab on these resources. The tutorial assumes you are familiar with Matlab and basic Linux terminal commands.

There are two basic ways to run Matlab (or many other kinds of software) on RC resources. The first is through an interactive job, and the second is through a batch job. An interactive job allows one to work in real-time with Matlab. Two reasons you may want to do this would be if you are actively debugging your code, or if you would like to use the GUI (in this instance, the Matlab Desktop). However, there might be other reasons you would like to work interactively with Matlab.

The second way to run Matlab on RC resources is by submission of a batch job. This allows the job to run in the background when resources become available. You may choose to use this method if you have a large job that may wait in the queue for awhile, or if you are not debugging or in need of a GUI. Both ways to work with Matlab are below.

1.25.1 Submitting Matlab Interactive Jobs

Running Matlab interactive jobs on RC resources is both a simple and easy task to accomplish. In this tutorial we will learn how to launch Matlab as an interactive job. For more information on launching interactive jobs check out our interactive jobs tutorial

Begin by launching an interactive job by loading slurm/summit into your environment and running the sinteractive command.

```
module load slurm/summit
sinteractive
```

From here you will load the Matlab module into your environment.

```
module load matlab
```

Lastly we will run Matlab from the terminal.
matlab -nosplash

Note: You can run Matlab without the gui using the `--nodesktop` command when launching Matlab.

### 1.25.2 Submitting Matlab Batch Jobs

Here, we will learn how to run a Matlab script in a non-interactive batch job. For more general information on batch job scripts on Summit, please see our tutorial on batch jobs.

Let’s begin by constructing a small Matlab script that prints ‘hello world’ to the user. The Matlab script we will use for the purposes of this tutorial is called `hello_world.m` and contains only one line, the Matlab command:

```matlab
fprintf('Hello world\n')
```

Which simply prints “Hello world” when called.

Next, we will construct our batch script that will enable us to submit this job. The batch script organizes the variety of flags slurm needs to submit a job and specifies the software commands we want to execute. An advantage of batch scripts is that they are easily reusable and adaptable for other similar tasks.

We will submit this job using a bash script titled: `slurm_hello.sh`, which contains the following lines:

```bash
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --time=0:01:00
#SBATCH --qos=testing
#SBATCH --partition=shas-testing
#SBATCH --ntasks=1
#SBATCH --job-name=Matlab_Hello_World
#SBATCH --output=Matlab_Hello_World.out
module purge
module load matlab
matlab -nodesktop -nodisplay -r 'clear; hello_world;'
```

This file has a few basic parts:

1. The first line specifies that it is a bash shell script, and ensures the rest of the lines will be interpreted in the correct shell.

2. The lines beginning with `#SBATCH` specify the Slurm parameters that will be used for this job. These lines are viewed as comments by bash, but will be read by Slurm. Of particular note is the `--output` parameter which specifies the file where stderr and stdout (including the output from our Matlab script) will be written. For a description of the Slurm parameters, please see the general Slurm documentation here.

3. The lines beginning with `module purge` remove any unneeded software and ensure that the appropriate Matlab module is loaded on the compute node.

4. The final line calls Matlab and instructs it to run our script. This entire line includes commands that are specific to Matlab; the `nodesktop` and `nodisplay` flags ensure that the Matlab Desktop will not open, and the `r` flag will run the script `hello_world`. The `clear` command forces Matlab to clear any existing variables, and is simply included as good coding practice.

You have now completed your batch script. After saving the script and exiting your text editor, submit the job as follows:
Research Computing University of Colorado Boulder

sbatch slurm_hello.sh

Once the job has run, the output of the Matlab script, “Hello world” will be shown in Matlab_Hello_World.out. That’s it!

1.25.3 Parallel Matlab on Summit

To fully utilize the multi-core capabilities of Summit to speed up jobs, most code must first be parallelized. Matlab has many built in tools to accomplish this task. In this tutorial we will parallelize our “Hello World” program.

Let’s begin with the Matlab script we created above called hello_world.m. First we will modify the fprintf line so that it includes a variable ‘i’ that will print out the iteration of the parallel loop.

```
fprintf(“Hello World from process %i”, i)
```

Next, we need to encapsulate the print statement in a parallel ‘for’ loop. Matlab uses the construct `parfor` to separate the task into multiple threads. In order to utilize the `parfor` command one must ensure that the Parallel Computing Toolbox is available as part of the Matlab software package. RC has this available and thus no additional action is required on your part if you are utilizing RC resources.

The order of runtime in the loop is not guaranteed, so the output may not be in sequential order. The loop is formatted as such:

```
parfor(int i = initial_Value:final_Value, maximum_amount_of_threads)
```

For example, let’s use `parfor` to implement an 5-iteration loop with a maximum of 4 processors in our script (new lines are highlighted here in blue):

```
parfor(int i = 1:5, 4)
    fprintf("Hello, World from process %i", i)
end
```

Now all we have left to do is modify our batch script to specify that we want to run 4 tasks on the node (we can use up to 24 cores on each ‘shas’ node on Summit). We can also change the name of the job and the output file if we choose.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --time=0:01:00
#SBATCH --qos=testing
#SBATCH --partition=shas-testing
#SBATCH --ntasks=4
#SBATCH --job-name=Matlab_Parallel_Hello
#SBATCH --output=Matlab_Parallel_Hello.out
module purge
module load matlab
matlab -nodesktop -nodisplay -r 'clear; hello_world;'
```

Now we submit the job using the `sbatch` command shown above, and our output in Matlab_Parallel_Hello.out will be as follows (the process order may be different in your output):

```
```

1.25. Matlab

59
Hello World from process 4
Hello World from process 1
Hello World from process 2
Hello World from process 3

Viola! You have run a parallel job on Summit with Matlab!

RC Matlab currently does not support parallelization across nodes, only across cores on one node.

1.26 GNU Parallel

GNU Parallel is an effective tool for optimally using multiple cores and nodes on RMACC Summit to run lots of independent tasks without the need to learn OpenMP or MPI. This tutorial assumes user knowledge of Slurm job submission, shell scripting, and some Python.

1.26.1 Why Use GNU Parallel?

Suppose you have a very simple serial program that crops a photo, and you need to apply it to crop several million photos. You could rewrite the serial program into a parallel program that would use multiple processors to more quickly run the program over the entire set of photos (compared to doing one-at-a-time), but this would require some knowledge of parallel programming. If your code is in a language that has limited parallelization capabilities, this may not even be an option. The easiest solution for this problem is to use GNU Parallel.

1.26.2 Using GNU Parallel

GNU Parallel is provided as a software module on RMACC Summit. It allows shell commands (for example, calls to serial programs) to be distributed amongst nodes and cores on RMACC Summit. This means code doesn’t need to be explicitly parallelized for MPI or OpenMP. Additionally, code can be written in any language that can be run from a Linux shell.

Let’s create a simple ‘Hello World’ serial python script to demonstrate the GNU Parallel tool. We will call the script hello_World.py and it will print “Hello World from task: ” followed by a command line argument:

```python
import sys

print “Hello World from task: “, sys.argv[1]
```

Now create a job script called run_hello.sh that will use GNU Parallel to submit as many instances of your python script as you want. Before running GNU Parallel in our script, we need to load the Python and GNU Parallel modules. Your job script should look something like this:

```bash
#!/bin/bash

#SBATCH --time 00:02:00
#SBATCH --qos=testin
#SBATCH --partition shas-testing
#SBATCH --ntasks=4
#SBATCH --job-name gpPythonDemo
#SBATCH --output gnuparallel.out

module purge
module load python
```

(continues on next page)
Note the last three lines of the script. We customize the GNU Parallel `parallel` command by creating a variable called `$my_parallel` that delays the submission of each task by 0.2 seconds (`--delay 0.2`) which mitigates bottlenecks for tasks that have heavy I/O when they start, and which specifies the number of tasks to run simultaneously (`-j $SLURM_NTASKS`). The environment variable `$SLURM_NTASKS` is set by Slurm at runtime and contains the number of `--ntasks` cores requested in the `#SBATCH` directives near the top of the job script (in this case the value is 4). We then customize the `srun` command so that it properly allocates the GNU parallel tasks to the allocated cores (`--export=all --exclusive -N1 -n1 --cpus-per-task=1 --cpu-bind=cores`). Note that the use of `srun` will also ensure that GNU parallel runs properly for cases where we request cores across multiple nodes (e.g., if we request `--ntasks=100`). Finally, we invoke GNU Parallel to run our python script 20 times using the customized `parallel` and `srun` commands we just created, `$my_parallel` and `$my_srun` respectively. Submitting this script via `sbatch` will run the commands. A successful job will result in output that looks something like this:

```
Hello World from task: 1
Hello World from task: 2
Hello World from task: 3
Hello World from task: 4
Hello World from task: 5
Hello World from task: 6
Hello World from task: 7
Hello World from task: 8
Hello World from task: 9
Hello World from task: 10
Hello World from task: 11
Hello World from task: 12
Hello World from task: 13
Hello World from task: 14
Hello World from task: 15
Hello World from task: 16
Hello World from task: 17
Hello World from task: 18
Hello World from task: 19
Hello World from task: 20
```

In this example the 20 invocations of your python script will run across the 4 cores requested; as each core finishes one task, the next remaining task will be executed on that core until all 20 have finished. The printed output above may or may not be in order depending on how quickly each task completes.

Tip: For sufficiently-large workflows one can add the `--joblog` and `--resume` flags in `$my_srun`. These flags will enable GNU Parallel to keep track of tasks it has run successfully and, if needed, rerun tasks that failed or were not executed. Additional details can be found in the links below.

### 1.26.3 Additional Resources

- [https://www.gnu.org/software/parallel/](https://www.gnu.org/software/parallel/)
- [https://github.com/ResearchComputing/HTC_Short_Course_Spring_2019](https://github.com/ResearchComputing/HTC_Short_Course_Spring_2019)
1.27 Containerization on Summit

When installing software, you may come across applications that have complex chains of dependencies that are challenging to compile and install. Some software may require very specific versions of libraries that may not be available on Summit or conflict with libraries needed for other applications. You may also need to move between several workstations or HPC platforms, which often requires reinstalling your software on each system. Containers are a good way to tackle all of these issues and more.

1.27.1 Containerization Fundamentals

Containers build upon an idea that has long existed within computing: hardware can be emulated through software. Virtualization simulates some or all components of computation through a software application. Virtual machines use this concept to generate an entire operating system as an application on a host system. Containers follow the same idea, but at a much smaller scale and contained within a system’s kernel.

Containers are portable compartmentalizations of some or all of the following: An operating system, software, libraries, data, and workflows. Containers offer:

- Portability: containers can run on any system equipped with its specified container manager.
- Reproducibility: because containers are instances of prebuilt isolated software, software will always execute the same every time.

Containers distinguish themselves through their low computational overhead and their ability to utilize all of a host system’s resources. Building containers is a relatively simple process that starts with a container engine.

1.27.2 Docker

Docker is by far the most popular container engine, and can be used on any system where you have administrative privileges. Because of this need for administrative privileges, Docker containers cannot be built or run directly on Research Computing resources. To utilize a Docker container on Research Computing resources please build a singularity image using a Docker image as a base.

See the documentation on Singularity (below) if you wish to run a Docker container on RMACC Summit or Blanca.

1.27.3 Singularity

Singularity is a containerization software package that does not require users to have administrative privileges when running containers, and can thus be safely used on Research Computing resources such as RMACC Summit and Blanca. Singularity is preinstalled on Research Computing resources, so all that is needed to run Singularity containers is to load the Singularity module on a compute node on RMACC Summit or Blanca:

```
module load singularity/3.0.2
```

Much like Docker, Singularity is a containerization software designed around compartmentalization of applications, libraries, and workflows. This is done through the creation of Singularity images which can be run as ephemeral Singularity containers. Unlike Docker, however, Singularity does not manage images, containers, or volumes through a central application. Instead, Singularity generates saved image files that can either be mutable or immutable based on compression.
Singularity Hub

Singularity Hub is a container registry that allows users to pull images from a server and into a system with Singularity installed. Singularity Hub uses Github to host image recipes, builds images in the cloud from these recipes, and places the resulting images in the Singularity Hub registry.

**Note:** You do not need an account with Github if you only wish to pull Singularity images.

https://singularity-hub.org/

Singularity Hub has a variety of useful prebuilt images for different software packages and workflows so be sure to check if the software you need is already available.

**Note:** As of 2019, there are presently two Singularity container registries. The former is Singularity Hub, described above, which is managed by Stanford University and Lawrence Berkeley National Laboratory. The latter is the Sylabs Singularity Container Library, which was created in late 2018 when Singularity was spun off into the private company Sylabs. Below we provide documentation on how to pull images from either repository, and on how to build images on Singularity Hub via Github, and in the Sylabs Singularity Container Library using their “Remote Builder” functionality.

Pulling Singularity Images

Because we cannot build our own Singularity images on HPC systems, we must instead bring our images over from another location. Pulling images from public repositories is often the easiest solution to this problem.

We can use the `singularity pull` command to remotely download our chosen image file. The command requires the container registry we would like to use, followed by the repository’s name:

```
singularity pull <container-registry>://<repository-name>
```

A container registry is simply a server that manages uploaded containers. Some examples of these container registries include Docker Hub, Singularity Hub, and the Singularity Container Library.

**Pull from Docker Hub:**

```
singularity pull docker://another:example
```

**Pull from Singularity Hub:**

```
singularity pull shub://example:repo
```

**Pull from Singularity Container Library (Singularity version 3.0 and greater):**

```
singularity pull library://example:repo
```

Lastly we can rename the Singularity image file pulled from a repository by utilizing the `-n/--name` flag.

```
singularity pull -n ExampleContainer.sif shub://example:tag
```

Example:

Pulling the Docker image of the latest tag of ubuntu can be done with the following command:

```
singularity pull docker://ubuntu:latest
```

1.27. Containerization on Summit
Running a Singularity image as a container

Singularity images can be run as containers much like Docker images. Singularity commands, however, follow a bit more nuanced syntax depending on what you’d like to do. After pulling your image from either Docker Hub or Singularity Hub, you can run the image by using the `singularity run` command. Type:

```
singularity run <image-name>
```

Running a Singularity container will execute the container’s default program that is specified in container definition file. To execute specific programs in your container, we can use the `singularity exec` command, and then specify the program:

```
singularity exec <image-name> <program>
```

Much like specifying an application in Docker, this will allow a user to execute any program that is installed within your container. Unlike Docker however, you do not need to specify a shell application to shell into the container. We can simply use the `singularity shell` command:

```
singularity shell <image-name>
```

Example:

Say we have a Singularity image that contains python 3.7 as the default software, and we want to run python from the container. We can do this with the command:

```
singularity run python-cont.img
```

If the default application for the image is not python we could run python as follows:

```
singularity exec python-cont.img python
```

File Access

By default most user-owned files and directories are available to any container that is run on RMACC Summit and Blanca (this includes files in /home/$USER, /projects/$USER, /scratch/summit/$USER and /rc_scratch/$USER). This means that normally a user will not need to bind any folders to the container’s directory tree. Furthermore, a container will also have access to the files in the same folder where it was initialized.

Sometimes, however, certain folders that are not bound by default may be necessary to run your application. To bind any additional folders or files to your Singularity container, you can utilize the `-B` flag in your `singularity run`, `exec`, and `shell` commands. To bind an additional folder to your Singularity container, type:

```
singularity run -B /source/directory:/target/directory sample-image.img
```

Additionally you can bind directories by utilizing the `SINGULARITY_BINDPATH` environment variable. Simply export a list of directory pairs you would like to bind to the your container:

```
export SINGULARITY_BINDPATH=/source/directory1:/target/directory1,\  /source/directory2:/target/directory2
```

Then run, execute, or shell into the container as normal.

Building a Singularity image

Important: You cannot build Singularity images directly on Summit. If you cannot build an image on your local machine you will need to build it on Singularity Hub or Sylabs Remote Builder.
Singularity Build

Just like Docker, Singularity allows a user to build images using a definition file. The file is saved with the name “Singularity” and contains instructions on how to prepare a Singularity image file. Just like a Dockerfile, this file has a variety of directives that allow for the customization of your image. A sample image would look something like this:

```plaintext
Bootstrap: shub
From: ubuntu

%help
    I am help text!

%setup
    apt-get update
    apt-get install nano
    apt-get install gcc

%runscript
    echo "hello! I am a container!"
```

Once you have written your Singularity recipe, you can build the application either remotely (see below) or locally with the singularity build command. To build a Singularity image locally, type:

```
sudo singularity build <img-name.img> <recipe-name.def>
```

Again, it is important to note that if you build an image locally as described above, you must build your image on a computer that you have administrative privileges on. If you do not have administrative privileges you will not be able to build the container in this manner. Fortunately, there are other ways to build containers remotely, which are discussed next.

Building Images Remotely with Singularity Hub

To build images with Singularity Hub, you must first create a Github account at https://github.com/join if you do not have one already. After completing this step log into your github account and create an empty repository.

After creating your repository, upload a Singularity definition file named “Singularity” to the repository. This is all we need to generate our Singularity image.

Now, log into Singularity Hub with your Github credentials and navigate to “My Container Collections” and click the link “Add a Collection.” From here a list of Github repositories you contribute to will be listed. Simply click the button on the repository you wish to add to Singularity Hub.

Your container should build automatically if you have a recipe file named “Singularity” within your repository. By default Singularity Hub will attempt to build any time something is pushed to the github repository. This can be changed in the settings tab in the containers build page. If the build fails the first time, revise the Singularity recipe and the build will initiate again.

More on building containers: https://www.sylabs.io/guides/3.0/user-guide/build_a_container.html

Building Images Remotely with the Singularity Remote Builder

With Singularity 3.0, users have the ability to build containers remotely through Sylabs remote builder. Unlike Singularity Hub though, the Singularity remote builder can be utilized directly on the command line from RMACC Summit or Blanca without needing to upload to a repository.

To begin using Singularity Remote Builder, navigate to your home directory and run the commands:
Now on your local machine, navigate to: https://cloud.sylabs.io/auth

and log into Sylabs with your Google, Github, Gitlab, or Microsoft account. Once you have logged into Sylabs, provide a label for your token under the field “Create A New Access Token” and click “Create a new Token.” This will generate a large string that will be read by Singularity on RMACC Summit or Blanca.

Now on RMACC Summit or Blanca run the command:

```
  echo "<your-token>" > sylabs-token
```

After this you can now build containers through the Sylabs remote builder on RMACC Summit or Blanca. Simply load Singularity 3.0.2 into your module stack and run the command:

```
  singularity build --remote <desired-image-name> <your-recipe>
```

**Building MPI-enabled Singularity images**

MPI-enabled Singularity containers can be deployed on RMACC Summit with the caveat that the MPI software within the container stays consistent with MPI software available on the system. This requirement diminishes the portability of MPI-enabled containers, as they may not run on other systems without compatible MPI software. Regardless, MPI-enabled containers can still be a very useful option in many cases.

Here we provide an example of using a gcc compiler with OpenMPI. RMACC Summit uses an Omni-Path interconnect (a low latency network fabric that enables MPI to be efficiently implemented across nodes). In order to use a Singularity container with OpenMPI (or any MPI) on Summit, there are two requirements:

Singularity container needs to have Omni-Path libraries installed inside. OpenMPI needs to be installed both inside and outside of the Singularity container. More specifically, the SAME version of OpenMPI needs to be installed inside and outside (at least very similar, you can sometimes get away with two different minor versions, ex: 2.1 and 2.0).

The following Singularity recipe ensures that OpenMPI 2.0.1 is installed in the image, which matches the openmpi/2.0.1 module that is available on RMACC Summit. This recipe can be used as a template to build your own MPI-enabled container images for RMACC Summit and can be found at: https://github.com/ResearchComputing/core-software/tree/master/singularity

Once you’ve built the container with one of the methods outlined above, you can place it on RMACC Summit and run it on a compute node. The following is an example of running a gcc/OpenMPI container with Singularity on RMACC Summit. The syntax is a normal MPI run where multiple instances of a Singularity image are run. The following example runs mpi_hello_world with MPI from a container.

```
  ml gcc/6.1.0
  ml openmpi/2.0.1
  ml singularity/3.0.2
  mpirun -np 4 singularity exec openmpi.sif mpi_hello_world"
```

Note that it is also possible to build intel/IMPI containers for use on RMACC Summit, which are likely to have enhanced performance on Summit’s intel architecture compared to gcc/OpenMPI containers. If you would like assistance building MPI-enabled containers contact rc-help@colorado.edu.
1.28 Coding best practices

Programmers employ numerous tactics to ensure readable and organized code. These include:

1. using naming conventions for variables;
2. placing whitespaces, indentations and tabs within code;
3. adding comments throughout to aid in interpretation.

In this tutorial we will examine these concepts.

1.28.1 Variable Naming Conventions

Variable naming is an important aspect in making your code readable. Naming variables follow a simple idea: Create variables that describe their function and which follow a consistent theme throughout your code. Let’s take a look at some naming conventions.

**Multiword Delimited**

This convention is to separate words in a variable name without the use of white space. White space within variables is usually difficult for programming languages to interpret. Because of this variables must be delimited in some way. Here are several delimiting conventions commonly used in code:

**Snakecase:** Words are delimited by an underscore.

```plaintext
variable_one
variable_two
```

**Pascalcase:** Words are delimited by capital letters.

```plaintext
VariableOne
VariableTwo
```

**Camelcase:** Words are delimited by capital letters, except the initial word.

```plaintext
variableOne
variableTwo
```

These conventions are by no means binding, but instead examples of how many programmers format their code. Consistency and readability are key ideas that should be utilized in the naming of variables.

**Hungarian Notation**

This notation describes the variable type or purpose at the start of the variable name, followed by a descriptor that indicates the variable’s function. The Camelcase notation is used to delimit words. Here are a few examples of Hungarian Notation:

```plaintext
arrDistrubuteGroup //Array called “Distribute Group”
sUserName //String called “User Name”
iRandomSeed //Integer called “Random Seed”
```

Regardless of how you choose to name your variables, always ensure that your naming conventions are consistent throughout the code. Consistency allows others to more easily understand your code.
1.28.2 Function and Class Naming conventions

Much like variable naming conventions, functions and classes should also follow a similar structure of descriptive titles delimited with the conventions described above. An important aspect of naming is to ensure your classes, functions, and variables can be distinguished from each other. For example, one could use Camelcase and Pascalcase for functions and classes respectively, while reserving Snakecase or Hungarian notation for variable names. Distinguishing functions, classes, and variables with different naming conventions can greatly aid other users of your code, and can eliminate the need for large sections of comments that would otherwise be needed.

1.28.3 Whitespace and Tabbing

Whitespace and tabbing are critical for organizing code. Whitespace is essentially any bit of space in your code that is not taken up by physical characters. Tabbing is one way to create whitespace in consistent units using the ‘tab’ key. Many languages ignore whitespace and tabbing all together, so it is important to ensure your code is correctly tabbed and you utilize whitespace to segment your code into neat blocks. Whitespace and tabbing are often used to clarify nested loops and logical statements. Let’s look at some examples of C code that demonstrate the effect of whitespace.

Minimal Whitespace:

```c
#include <stdio.h>
int main(int argc, char const *argv[]) { int loop_Sum = 0; for(int i = 0; i < 50; i++){ loop_Sum += 1; } printf("%d
", loop_Sum); return 0; }
```

Liberal use of Whitespace:

```c
#include <stdio.h>
int main(int argc, char** argv){
    int loop_Sum = 0;
    for(int i = 0; i < 50; i++){
        loop_Sum += 1;
    }
    printf("%d\n", loop_Sum);
    return 0;
}
```

It is important to note that like variable naming, whitespace can be utilized in various different styles and approaches. Just remember to use whitespace and tabbing in a consistent and intuitive manner throughout your code.

1.28.4 Commenting your code

Commenting may be the most important way to organize and segment code. Comments are sections of code that the compiler ignores, but humans can read, and thus can be used to label code and divide it into logical segments to aid users. For example, one can label loops, scopes, functions, and other code snippets. Lines with comments in code are preceded by a symbol that tells the compiler to ignore that line when compiling, for example “#”, “!” or “;” (the symbol depends on the programming language). Let’s look at some C++ code that uses no comments vs. comments.

No Comments:

```c
#include <stdio.h>
#include <vector>
```

(continues on next page)
using namespace std;

int main(int argc, char** argv) {
    vector<int> multiples;

    for(int i = 0; i < 50; i++) {
        if(i % 5 == 0) {
            multiples.push_back(i);
        }
    }

    for(int i = 0; i < multiples.size(); i++) {
        printf("%d is a multiple of 5\n", multiples[i]);
    }

    return 0;
}

Comments:

#include <stdio.h>
#include <vector>

using namespace std;

int main(int argc, char** argv) {
    //Declare a vector to store
    vector<int> multiples;

    //Iterate from 0 to 50
    for(int i = 0; i <= 50; i++) {
        //If iterator is a multiple of 5 add it to the vector
        if(i % 5 == 0) {
            multiples.push_back(i);
        }
    }

    //Print all items that are a multiple of 5
    for(int i = 0; i < multiples.size(); i++) {
        printf("%d is a multiple of 5\n", multiples[i]);
    }

    return 0;
}

Note that it is entirely possible to “over comment” code. Code should be designed in an efficient, consistent and intuitive manner such that comments enhance user understanding but are not needed to describe the entire code.

### 1.29 Fundamentals of parallel programming

In this tutorial we will explore the differences between serial and parallel computation, and look at how parallel programs work in general. We will also assess a couple of parallel program solutions that utilize the multiprocessor environment of a supercomputer.
Useful Links:

1.29.1 Why Parallel?

Assume you are attempting to assemble a 10,000-piece jigsaw puzzle* on a rainy weekend. The number of pieces is staggering, and instead of a weekend it takes you several weeks to finish the puzzle. Now assume you have three friends helping with the puzzle – it goes much faster, and you are able to finish the puzzle over the weekend. This principle is the central idea behind parallel computation. You can dramatically cut down on computation by splitting one large task into smaller tasks that multiple processors can perform all at once. With parallel processes a task that would normally take several weeks can potentially be reduced to several hours.

- Puzzle analogy for describing parallel computing adopted from Henry Neeman’s Supercomputing in Plain English tutorial series.

1.29.2 Serial and Parallel Processes

A serial process is simply a process that is run entirely by one core of one processor. This means tasks are run one after another as they appear in code. This is analogous to you doing the jigsaw puzzle on your own. A parallel process is a process that is divided among multiple cores in a processor or set of processors. Each sub process can have its own set of memory as well as share memory with other processes. This is analogous to doing the puzzle with the help of friends. A supercomputer has a large network of nodes with many cores, and is designed for parallel computing. However, without the use of parallel software design, we cannot fully utilize supercomputing resources.

1.29.3 How parallel computation works

Parallel computation connects multiple processors to memory that is either pooled or connected via high speed networks. Here are three different types of parallel computation.

Shared Memory Model:
In a shared memory model all processors have access to a pool of common memory that they can freely use.

Distributed Memory Model:
In a distributed memory model a separate segment of memory is available to each processor. Because memory isn’t shared inherently, information that must be shared between processes is sent over a network.

Distributed/Shared Model:
A split distributed/shared model is a hybrid between a shared and distributed model and has the properties of both. Each separate set of processors sharing a set of common memory is called a node.

Summit utilizes a hybrid distributed/shared model: there are 380 nodes, each having 24 cores.
1.29.4 Tools for Parallel Programming

Two common solutions for creating parallel code are OpenMP and MPI. Both solutions are limited to the C++ or Fortran programming languages. (Though other languages may be extended with C++ or Fortran code to utilize OpenMP or MPI.)

**OpenMP**

OpenMP (“Open Multi-Processing”) is a compiler-side application programming interface (API) for creating code that can run on a system of threads. No external libraries are required in order to parallelize your code. OpenMP is often considered more user friendly with thread safe methods and parallel sections of code that can be set with simple scoping. OpenMP is, however, limited to the amount of threads available on a node – in other words, it follows a shared memory model. On Summit, this means that no more than 24 processors can be utilized with programs parallelized using OpenMP.

**MPI**

MPI (“Message Passing Interface”) is a library standard for handling parallel processing. Unlike OpenMP, MPI has much more flexibility in how individual processes handle memory. MPI is also compatible with multi-node structures, allowing for very large, multi-node applications (i.e, distributed memory models). MPI is, however, often considered more difficult to learn. Regardless, learning the library provides a user with the ability to maximize processing ability. MPI is a library standard, meaning there are several libraries based on MPI that you can use to develop parallel code. Two solutions available on Summit are OpenMPI and Intel MPI.

1.30 Using MPI with C

Parallel programs enable users to fully utilize the multi-node structure of supercomputing clusters. Message Passing Interface (MPI) is a standard used to allow several different processors on a cluster to communicate with each other. In this tutorial we will be using the Intel C++ Compiler, GCC, IntelMPI, and OpenMPI to create a multiprocessor ‘hello world’ program in C++. This tutorial assumes the user has experience in both the Linux terminal and C++.

**Resources:**

- http://www.dartmouth.edu/~rc/classes/intro_mpi/intro_mpi_overview.html
- http://mpitutorial.com/tutorials/
- http://condor.cc.ku.edu/~grobe/docs/intro-MPI-C.shtml
- https://computing.llnl.gov/tutorials/mpi/

1.30.1 Setup and “Hello, World”

Begin by logging into the cluster and using ssh to log in to a compile node. This can be done with the command:

```
ssh scompile
```

Next we must load MPI into our environment. Begin by loading in your choice of C++ compiler and its corresponding MPI library. Use the following commands if using the GNU C++ compiler:

**GNU C++ Compiler**
module load gcc
module load openmpi

Or, use the following commands if you prefer to use the Intel C++ compiler:

**Intel C++ Compiler**

module load intel
module load impi

This should prepare your environment with all the necessary tools to compile and run your MPI code. Let’s now begin to construct our C++ file. In this tutorial, we will name our code file: `hello_world_mpi.cpp`

Open `hello_world_mpi.cpp` and begin by including the C standard library `<stdio.h>` and the MPI library `<mpi.h>`, and by constructing the main function of the C++ code:

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv){
    return 0;
}
```

Now let’s set up several MPI directives to parallelize our code. In this ‘Hello World’ tutorial we’ll be utilizing the following four directives:

**MPI_Init()**:

This function initializes the MPI environment. It takes in the addresses of the C++ command line arguments `argc` and `argv`.

**MPI_Comm_size()**:

This function returns the total size of the environment via quantity of processes. The function takes in the MPI environment, and the memory address of an integer variable.

**MPI_Comm_rank()**:

This function returns the process id of the processor that called the function. The function takes in the MPI environment, and the memory address of an integer variable.

**MPI_Finalize()**:

This function cleans up the MPI environment and ends MPI communications.

These four directives should be enough to get our parallel ‘hello world’ running. We will begin by creating two variables, `process_Rank`, and `size_Of_Cluster`, to store an identifier for each of the parallel processes and the number of processes running in the cluster respectively. We will also implement the `MPI_Init` function which will initialize the mpi communicator:

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv){
    int process_rank, size_Of_Cluster
    MPI_Init(&argc, &argv);
    return 0;
}
```
Let’s now obtain some information about our cluster of processors and print the information out for the user. We will use the functions `MPI_Comm_size()` and `MPI_Comm_rank()` to obtain the count of processes and the rank of a process respectively:

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv){
    int process_Rank, size_Of_Cluster;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Cluster);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);
    printf("Hello World from process %d of %d\n", process_Rank, size_Of_Cluster);
    return 0;
}
```

Lastly let’s close the environment using `MPI_Finalize()`:

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv){
    int process_Rank, size_Of_Cluster;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Cluster);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);
    printf("Hello World from process %d of %d\n", process_rank, size_Of_Cluster);
    MPI_Finalize();
    return 0;
}
```

Now the code is complete and ready to be compiled. Because this is an MPI program, we have to use a specialized compiler. Be sure to use the correct command based off of what compiler you have loaded.

**OpenMPI**

```
mpic++ hello_world_mpi.cpp -o hello_world_mpi.exe
```

**Intel MPI**

```
mpiicc hello_world_mpi.cpp -o hello_world_mpi.exe
```

This will produce an executable we can submit to Summit as a job. In order to execute MPI compiled code, a special command must be used:

```
mpirun -np 4 ./hello_world_mpi.exe
```

The flag `-np` specifies the number of processor that are to be utilized in execution of the program.

In your job submission script, load the same compiler and OpenMPI choices you used above to compile the program, and submit the job with slurm to run the executable. Your job submission script should look something like this:

**OpenMPI**

```bash
1.30. Using MPI with C    73
```
#!/bin/bash
#SBATCH -N 1
#SBATCH --ntasks 4
#SBATCH --job-name parallel_hello
#SBATCH --partition shas-testing
#SBATCH --time 0:01:00
#SBATCH --output parallel_hello_world.out
#SBATCH --qos=testing

module purge
module load gcc
module load openmpi

mpirun -np 4 ./hello_world_mpi.exe

Intel MPI

#!/bin/bash
#SBATCH -N 1
#SBATCH --ntasks 4
#SBATCH --job-name parallel_hello
#SBATCH --partition shas-testing
#SBATCH --time 0:01:00
#SBATCH --output parallel_hello_world.out
#SBATCH --qos=testing

module purge
module load intel
module load impi

mpirun -np 4 ./hello_world_mpi.exe

It is important to note that on Summit, there is a total of 24 cores per node. For applications that require more than 24 processes, you will need to request multiple nodes in your job submission. Our output file should look something like this:

Hello World from process 3 of 4
Hello World from process 2 of 4
Hello World from process 1 of 4
Hello World from process 0 of 4

Ref: http://www.dartmouth.edu/~rc/classes/intro_mpi/hello_world_ex.html

1.30.2 MPI Barriers and Synchronization

Like many other parallel programming utilities, synchronization is an essential tool in thread safety and ensuring certain sections of code are handled at certain points. MPI_Barrier is a process lock that holds each process at a certain line of code until all processes have reached that line in code. MPI_Barrier can be called as such:

MPI_Barrier(MPI_Comm comm);

To get a handle on barriers, let’s modify our “Hello World” program so that it prints out each process in order of thread id. Starting with our “Hello World” code from the previous section, begin by nesting our print statement in a loop:
```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv){
    int process_Rank, size_Of_Cluster;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Cluster);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);

    for (int i = 0, i < size_Of_Cluster, i++){
        printf("Hello World from process %d of %d\n", process_Rank, size_Of_Cluster);
    }

    MPI_Finalize();
    return 0;
}
```

Next, let's implement a conditional statement in the loop to print only when the loop iteration matches the process rank.

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv){
    int process_Rank, size_Of_Cluster;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Cluster);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);

    for (int i = 0, i < size_Of_Cluster, i++){
        if (i == process_Rank){
            printf("Hello World from process %d of %d\n", process_Rank, size_Of_Cluster);
        }
    }

    MPI_Finalize();
    return 0;
}
```

Lastly, implement the barrier function in the loop. This will ensure that all processes are synchronized when passing through the loop.

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv){
    int process_Rank, size_Of_Cluster;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Cluster);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);

    for (int i = 0, i < size_Of_Cluster, i++){
        if (i == process_Rank){
            printf("Hello World from process %d of %d\n", process_Rank, size_Of_Cluster);
        }
    }

    MPI_Finalize();
    return 0;
}
```
{ }  
    MPI_Barrier(MPI_COMM_WORLD);  
}  
    MPI_Finalize();  
    return 0;  
}

Compiling and submitting this code will result in this output:

Hello World from process 0 of 4  
Hello World from process 1 of 4  
Hello World from process 2 of 4  
Hello World from process 3 of 4

### 1.30.3 Message Passing

Message passing is the primary utility in the MPI application interface that allows for processes to communicate with each other. In this tutorial, we will learn the basics of message passing between 2 processes.

Message passing in MPI is handled by the corresponding functions and their arguments:

```c
MPI_Send(void* message, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm, communicator);
MPI_Recv(void* data, int count, MPI_Datatype datatype, int from, int tag, MPI_Comm comm, MPI_Status* status);
```

The arguments are as follows:

**MPI_Send**

- `void* message;` //Address for the message you are sending.
- `int count;` //Number of elements being sent through the address.
- `MPI_Datatype datatype;` //The MPI specific data type being passed through the address.
- `int dest;` //Rank of destination process.
- `int tag;` //Message tag.
- `MPI_Comm comm;` //The MPI Communicator handle.

**MPI_Recv**

- `void* message;` //Address to the message you are receiving.
- `int count;` //Number of elements being sent through the address.
- `MPI_Datatype datatype;` //The MPI specific data type being passed through the address.
- `int from;` //Process rank of sending process.
- `int tag;` //Message tag.
- `MPI_Comm comm;` //The MPI Communicator handle.
- `MPI_Status* status;` //Status object.

Let’s implement message passing in an example:

**Example**

We will create a two-process process that will pass the number 42 from one process to another. We will use our “Hello World” program as a starting point for this program. Let’s begin by creating a variable to store some information.
```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv){
    int process_Rank, size_Of_Cluster, message_Item;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Cluster);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);
    MPI_Finalize();
    return 0;
}
```

Now create if and else if conditionals that specify appropriate process to call `MPI_Send()` and `MPI_Recv()` functions. In this example we want process 1 to send out a message containing the integer 42 to process 2.

```c
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv){
    int process_Rank, size_Of_Cluster, message_Item;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Cluster);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);

    if(process_Rank == 0){
        message_Item = 42;
        printf("Sending message containing: %d\n", message_Item);
    } else if(process_Rank == 1){
        printf("Received message containing: %d\n", message_Item);
    }
    MPI_Finalize();
    return 0;
}
```

Lastly we must call `MPI_Send()` and `MPI_Recv()`. We will pass the following parameters into the functions:

```c
MPI_Send(
    &message_Item, /*Address of the message we are sending.*/
    1, /*Number of elements handled by that address.*/
    MPI_INT, /*MPI_TYPE of the message we are sending.*/
    1, /*Rank of receiving process*/
    1, /*Message Tag*/
    MPI_COMM_WORLD /*MPI Communicator*/
);

MPI_Recv(
    &message_Item, /*Address of the message we are receiving.*/
    1, /*Number of elements handled by that address.*/
    MPI_INT, /*MPI_TYPE of the message we are sending.*/
    0, /*Rank of sending process*/
    1, /*Message Tag*/
    MPI_COMM_WORLD /*MPI Communicator*/
    MPI_STATUS_IGNORE /*MPI Status Object*/
);
```

1.30. Using MPI with C
Let's implement these functions in our code:

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char ** argv) {
    int process_Rank, size_Of_Cluster, message_Item;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Cluster);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);

    if(process_Rank == 0){
        message_Item = 42;
        MPI_Send(&message_Item, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
        printf("Message Sent: %d\n", message_Item);
    }

    else if(process_Rank == 1){
        MPI_Recv(&message_Item, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Message Received: %d\n", message_Item);
    }

    MPI_Finalize();
    return 0;
}
```

Compiling and submitting our code with 2 processes will result in the following output:

```
Message Sent: 42
Message Received: 42
```

### 1.30.4 Group Operators: Scatter and Gather

Group operators are very useful for MPI. They allow for swaths of data to be distributed from a root process to all other available processes, or data from all processes can be collected at one process. These operators can eliminate the need for a surprising amount of boilerplate code via the use of two functions:

**MPI_Scatter:**

- `void* send_Var;` //Address of the variable that will be scattered.
- `int send_Count;` //Number of elements that will be scattered.
- `MPI_Datatype send_Type;` //MPI Datatype of the data that is scattered.
- `void* recv_Var;` //Address of the variable that will store the scattered data.
- `int recv_Count;` //Number of data elements that will be received per process.
- `MPI_Datatype recv_Type;` //MPI Datatype of the data that will be received.
- `int root_Process;` //The rank of the process that will scatter the information.
- `MPI_Comm comm;` //The MPI_Communicator.

**MPI_Gather:**

- `void* send_Var;` //Address of the variable that will be sent.
- `int send_Count;` //Number of data elements that will sent.
- `MPI_Datatype send_Type;` //MPI Datatype of the data that is sent.
- `void* recv_Var;` //Address of the variable that will store the received data.
- `int recv_Count;` //Number of data elements per process that will be received.

(continues on next page)
In order to get a better grasp on these functions, let's go ahead and create a program that will utilize the scatter function. Note that the gather function (not shown in the example) works similarly, and is essentially the converse of the scatter function. Further examples which utilize the gather function can be found in the MPI tutorials listed as resources at the beginning of this document.

**Example**

We will create a program that scatters one element of a data array to each process. Specifically, this code will scatter the four elements of an array to four different processes. We will start with a basic C++ main function along with variables to store process rank and number of processes.

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
    int process_Rank, size_Of_Comm;
    return 0;
}
```

Now let's setup the MPI environment using `MPI_Init, MPI_Comm_size, MPI_Comm_rank, and MPI_Finalize`:

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
    int process_Rank, size_Of_Comm;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Comm);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);
    MPI_Finalize();
    return 0;
}
```

Next let's generate an array named `distro_Array` to store four numbers. We will also create a variable called `scattered_Data` that we shall scatter the data to.

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
    int process_Rank, size_Of_Comm;
    int distro_Array[4] = {39, 72, 129, 42};
    int scattered_Data;
} (continues on next page)
Now we will begin the use of group operators. We will use the operator scatter to distribute `distro_Array` into `scattered_Data`. Let’s take a look at the parameters we will use in this function:

```c
MPI_Scatter(&distro_Array, 1, MPI_INT, &scattered_Data, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Let’s see this implemented in code. We will also write a print statement following the scatter call:

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv){
    int process_Rank, size_Of_Comm;
    int distro_Array[4] = {39, 72, 129, 42};
    int scattered_Data;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Comm);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);

    MPI_Scatter(&distro_Array, 1, MPI_INT, &scattered_Data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    printf("Process has received: %d \n", scattered_Data);
    MPI_Finalize();
    return 0;
}
```

Running this code will print out the four numbers in the distro array as four separate numbers each from different processors (note the order of ranks isn’t necessarily sequential):

```
Process has received: 39
Process has received: 72
Process has received: 129
Process has received: 42
```
1.31 Using MPI with Fortran

Parallel programs enable users to fully utilize the multi-node structure of supercomputing clusters. Message Passing Interface (MPI) is a standard used to allow different nodes on a cluster to communicate with each other. In this tutorial we will be using the Intel Fortran Compiler, GCC, IntelMPI, and OpenMPI to create a multiprocessor programs in Fortran. This tutorial assumes the user has experience in both the Linux terminal and Fortran.

Helpful MPI tutorials:
- http://www.dartmouth.edu/~rc/classes/intro_mpi/intro_mpi_overview.html
- http://condor.cc.ku.edu/~grobe/docs/intro-MPI.shtml
- https://computing.llnl.gov/tutorials/mpi/

1.31.1 Setup and “Hello World”

Begin by logging into the cluster and using ssh to log in to a Summit compile node. This can be done with the command:

```
ssh scompile
```

Next we must load MPI into our environment. Begin by loading in the Fortran compiler and OpenMPI. Use the following commands if using the GNU Fortran compiler:

**GNU Fortran Compiler**

```plaintext
module load gcc
module load openmpi
```

Or, use the following commands if you prefer to use the Intel Fortran compiler:

**Intel Fortran Compiler**

```plaintext
module load intel
module load impi
```

This should prepare your environment with all the necessary tools to compile and run your MPI code. Let’s now begin to construct our Fortran program. In this tutorial, we will name our program file: `hello_world_mpi.f90`

Open `hello_world_mpi.f90` and begin by including the mpi library `mpi.h`, and titling the program `hello_world_mpi`

```fortran
PROGRAM hello_world_mpi
include 'mpif.h'
```

Now let’s set up several MPI directives to parallelize our code. In this ‘Hello World’ tutorial we will be calling the following four functions from the MPI library:

`MPI_INIT()`:
This function initializes the MPI environment. It takes in the an error handling variable.

`MPI_COMM_SIZE()`:
This function returns the total size of the environment in terms of the quantity of processes. The function takes in the MPI environment, an integer to hold the commsize, and an error handling variable.

`MPI_COMM_RANK()`:
This function returns the process id of the process that called the function. The function takes in the MPI environment, an integer to hold the comm rank, and an error handling variable.

**MPI_FINALIZE()**: This function cleans up the MPI environment and ends MPI communications. These four directives are enough to get our parallel 'hello world' program running. We will begin by creating three integer variables, `process_Rank`, `size_Of_Cluster`, and `ierror` to store an identifier for each of the parallel processes, store the number of processes running in the cluster, and handle error codes respectively. We will also implement the `MPI_Init` function which will initialize the mpi communicator:

```fortran
PROGRAM hello_world_mpi
include 'mpif.h'
integer process_Rank, size_Of_Cluster, ierror
call MPI_INIT(ierr)
```

Let's now obtain some information about our cluster of processors and print the information out for the user. We will use the functions `MPI_Comm_size()` and `MPI_Comm_rank()` to obtain the count of processes and the rank of a given process respectively:

```fortran
PROGRAM hello_world_mpi
include 'mpif.h'
integer process_Rank, size_Of_Cluster, ierror
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
print *, 'Hello World from process: ', rank, 'of ', size
```

Lastly let's close the environment using `MPI_Finalize()`:

```fortran
PROGRAM hello_world_mpi
include 'mpif.h'
integer rank, size, ierror, tag
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
print *, 'Hello World from process: ', rank, 'of ', size
call MPI_FINALIZE(ierr)
END PROGRAM
```

Now the code is complete and ready to be compiled. Because this is an MPI program, we have to use a specialized compiler. The compilation command will be one of the following:

**GNU Fortran Compiler**

```bash
mpif90 hello_world_mpi.f90 -o hello_world_mpi.exe
```

**Intel Fortran Compiler**
mpiifort hello_world_mpi.f90 -o hello_world_mpi.exe

This will produce an executable we can submit to Summit as a job. In order to execute MPI compiled code, a special command must be used:

mpirun -np 4 ./hello_world_mpi.exe

The flag -np specifies the number of processor that are to be utilized in execution of the program. In your job submission script, load the same compiler and OpenMPI choices you used above to create and compile the program, and submit the job with slurm to run the executable. Your job submission script should look something like this:

**GNU Fortran Compiler**

```bash
#!/bin/bash
#SBATCH -N 1
#SBATCH --ntasks 4
#SBATCH --job-name parallel_hello
#SBATCH --partition shas-testing
#SBATCH --time 0:01:00
#SBATCH --output parallel_hello_world.out
#SBATCH --qos=testing

module purge
module load gcc
module load openmpi

mpirun -np 4 ./hello_world_mpi.exe
```

**Intel Fortran Compiler**

```bash
#!/bin/bash
#SBATCH -N 1
#SBATCH --ntasks 4
#SBATCH --job-name parallel_hello
#SBATCH --partition shas-testing
#SBATCH --time 0:01:00
#SBATCH --output parallel_hello_world.out
#SBATCH --qos=testing

module purge
module load intel
module load impi

mpirun -np 4 ./hello_world_mpi.exe
```

It is important to note that on Summit, there are 24 cores per node. For applications that require more than 24 processes, you will need to request multiple nodes in your job submission (i.e., "-N ").

Our output file should look something like this (note the order of ranks isn’t necessarily sequential):

```
Hello World from process 3 of 4
Hello World from process 2 of 4
Hello World from process 1 of 4
Hello World from process 0 of 4
```

Ref: http://www.dartmouth.edu/~rc/classes/intro_mpi/hello_world_ex.html
1.31.2 MPI Barriers and Synchronization

Like many other parallel programming utilities, synchronization is an essential tool in thread safety and ensuring certain sections of code are handled at certain points. MPI_BARRIER is a process lock that holds each process at a certain line of code until all processes have reached that line. MPI_BARRIER can be called as such:

```fortran
call MPI_BARRIER(MPI_com comm, integer ierror);
```

To get a handle on barriers, let's modify our “Hello World” program so that it prints out each process in order of thread id. Starting with our “Hello World” code from the previous section, begin by putting our print statement in a loop:

```fortran
PROGRAM hello_world_mpi
include 'mpif.h'

integer rank, size, ierror

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)

DO i = 0, 3, 1
   print *, 'Hello World from process: ', rank, 'of ', size
END DO

call MPI_FINALIZE(ierr)
END PROGRAM
```

Next, let’s implement a conditional statement in the loop to print only when the loop iteration matches the process rank.

```fortran
PROGRAM hello_world_mpi
include 'mpif.h'

integer rank, size, ierror

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)

DO i = 0, 3, 1
   IF (i == rank) THEN
      print *, 'Hello World from process: ', rank, 'of ', size
   END IF
END DO

call MPI_FINALIZE(ierr)
END PROGRAM
```

Lastly, implement the barrier function in the loop. This will ensure that all processes are synchronized when passing through the loop.

```fortran
PROGRAM hello_world_mpi
include 'mpif.h'

integer rank, size, ierror

call MPI_INIT(ierr)

call MPI_BARRIER(MPI_com comm, integer ierror);

call MPI_FINALIZE(ierr)
END PROGRAM
```
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

DO i = 0, 3, 1
   IF (i == rank) THEN
      print *, 'Hello World from process: ', rank, ' of ', size
   END IF
   call MPI_BARRIER(MPI_COMM_WORLD, i_error)
END DO

call MPI_FINALIZE(ierror)
END PROGRAM

Compiling and submitting this code will result in the following output (note the ranks are now sequential):

Hello World from process 0 of 4
Hello World from process 1 of 4
Hello World from process 2 of 4
Hello World from process 3 of 4

1.31.3 Message Passing

Message passing is the primary utility in the MPI application interface that allows for processes to communicate with each other. Next, we will learn the basics of message passing between two processes. Message passing in MPI is handled by the corresponding functions and their arguments:

call MPI_SEND(integer message, integer count, MPI_Datatype datatype, integer dest, integer tag, MPI_Comm comm, integer ierror);
call MPI_RECV(integer data, integer count, MPI_Datatype datatype, integer from, integer tag, MPI_Comm comm, MPI_Status* status, integer ierror);

The arguments are as follows:

**MPI_SEND:**

- **integer message:** !Variable storing message you are sending.
- **integer count:** !Number of elements being sent through the array.
- **MPI_Datatype datatype:** !The MPI-specific data type being passed through the array.
- **integer dest:** !Process rank of destination process.
- **integer tag:** !Message tag.
- **MPI_Comm comm:** !The MPI Communicator handle.
- **integer ierror:** !An error handling variable.

**MPI_RECV:**

- **integer message:** !Variable storing message you are receiving.
- **integer count:** !Number of elements being sent through the array.
- **MPI_Datatype datatype:** !The MPI-specific data type being passed through the array.
- **integer from:** !Process rank of sending process.
- **integer tag:** !Message tag.
- **MPI_Comm comm:** !The MPI Communicator handle.
- **MPI_Status* status:** !Status object.
- **integer ierror:** !An error handling variable.

Let’s implement message passing in an example:
1.31.4 Example

We will pass the number 42 from one process to another. We will use our “Hello World” program as a starting point for this program. Let’s begin by renaming our program and creating a variable to store some information.

```fortran
PROGRAM send_recv_mpi
include 'mpif.h'

integer rank, size, ierror, message_Item

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
call MPI_FINALIZE(ierr)
END PROGRAM
```

Now create ‘if’ and ‘else if’ conditionals that specify the appropriate processes to call `MPI_SEND()` and `MPI_RECV()` functions. In this example we want process 1 to send out a message containing the integer 42 to process 2.

```fortran
PROGRAM send_recv_mpi
include 'mpif.h'

integer rank, size, ierror, message_Item

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

IF (rank == 0) THEN
    message_Item = 42
    print *, "Sending message containing: ", message_Item
ELSE IF (rank == 1) THEN
    print *, "Received message containing: ", message_Item
END IF

call MPI_FINALIZE(ierr)
END PROGRAM
```

Lastly we must call `MPI_SEND()` and `MPI_RECV()`. We will pass in the following parameters into the functions:

```fortran
MPI_SEND(  
    message_Item,  !Variable storing the message we are sending.
    1,  !Number of elements handled by the array.
    MPI_INT,  !MPI_TYPE of the message we are sending.
    1,  !Rank of receiving process
    1,  !Message Tag
    MPI_COMM_WORLD  !MPI Communicator
    ierr  !Error Handling Variable
)

MPI_RECV(  
    message_Item,  !Variable storing the message we are receiving.
    1,  !Number of elements handled by the array.
    MPI_INT,  !MPI_TYPE of the message we are sending.
    0,  !Rank of sending process
    1,  !Message Tag
)
```

(continues on next page)
Research Computing University of Colorado Boulder

(continued from previous page)

<table>
<thead>
<tr>
<th>MPI_COMM_WORLD</th>
<th>!MPI Communicator</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_STATUS_IGNORE</td>
<td>!MPI Status Object</td>
</tr>
<tr>
<td>ierror</td>
<td>!Error Handling Variable</td>
</tr>
</tbody>
</table>

Lets implement these functions in our code:

```fortran
PROGRAM send_recv_mpi
include 'mpif.h'

integer rank, size, ierror, message_Item

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

IF(rank == 0) THEN
    message_Item = 42
    call MPI_SEND(message_Item, 1, MPI_INT, 1, 1, MPI_COMM_WORLD, ierror)
    print *, "Sending message containing: ", message_Item
ELSE IF(rank == 1) THEN
    call MPI_RECV(message_Item, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierror)
    print *, "Received message containing: ", message_Item
END IF

call MPI_FINALIZE(ierr)
END PROGRAM
```

Compiling and submitting a batch job with our code that requests 2 processes (--ntasks 2) will result in the following output:

```
Sending message containing: 42
Received message containing: 42
```

### 1.3.1.5 Group Operators: Scatter and Gather

Group operators are very useful for MPI. They allow for swaths of data to be distributed from a root process to all other available processes, or data from all processes can be collected at one process. These operators can eliminate the need for a surprising amount of boilerplate code via two functions:

**MPI_Scatter:**

```fortran
integer send_Var       !Variable storing the values that will be scattered.
integer send_Count     !Number of elements that will be scattered.
MPI_Datatype send_Type !MPI Datatype of the data that is scattered.
integer recv_Var       !Variable that will store the scattered data.
integer recv_Count     !Number of data elements that will be received per process.
MPI_Datatype recv_Type !MPI Datatype of the data that will be received.
integer root_Process   !The rank of the process that will scatter the information.
MPI_Comm comm          !The MPI Communicator.
integer ierror         !An error handling variable.
```

**MPI_Gather:**

1.31. Using MPI with Fortran
In order to get a better grasp on these functions, let’s go ahead and create a program that will utilize the scatter function. Note that the gather function (not shown in the example) works similarly, and is essentially the converse of the scatter function. Further examples which utilize the gather function can be found in the MPI tutorials listed as resources at the beginning of this document.

Example

We will create a new program that scatters one element of a data array to each process. Specifically, this code will scatter the four elements of a vector array to four different processes. We will start with a Fortran header along with variables to store process rank and number of processes.

```fortran
PROGRAM scatter_mpi
  include 'mpif.h'
  integer rank, size, ierror, message_Item
END PROGRAM
```

Now let’s setup the MPI environment using `MPI_Init`, `MPI_Comm_size`, `MPI_Comm_rank`, and `MPI_Finalize`:

```fortran
PROGRAM scatter_mpi
  include 'mpif.h'
  integer rank, size, ierror, message_Item
  call MPI_INIT(ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_FINALIZE(ierr)
END PROGRAM
```

Next let’s generate an array named `distro_Array` to store four numbers. We will also create a variable called `scattered_Data` to which we will scatter the data.

```fortran
PROGRAM scatter_mpi
  include 'mpif.h'
  integer rank, size, ierror, message_Item
  integer scattered_Data
  integer, dimension(4) :: distro_Array
  distro_Array = (/39, 72, 129, 42/)  
```

(continues on next page)
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_FINALIZE(ierr)
END PROGRAM

Now we will begin the use of group operators. We will use the operator scatter to distribute distro_Array into scattered_Data. Let’s take a look at the parameters we will use in this function:

MPI_Scatter(
    distro_Array, !Array we are scattering from.
    1, !Number of items we are sending each processor
    MPI_INT, !MPI Datatype of scattering array.
    scattered_Data, !Variable to which we are receiving scattered data.
    1, !Amount of data each process will receive.
    MPI_INT, !MPI Datatype of receiver array.
    0, !Process ID that will distribute the data.
    MPI_COMM_WORLD !MPI Communicator.
    ierr !Error Handling Variable )

Let’s implement this in the code. We will also write a print statement following the scatter call:

PROGRAM scatter_mpi
include 'mpif.h'

integer rank, size, ierr, message_Item
integer scattered_Data
integer, dimension(4) :: distro_Array
distro_Array = (/39, 72, 129, 42/)
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_Scatter(distro_Array, 1, MPI_INT, scattered_Data, 1, MPI_INT, 0, MPI_COMM_WORLD, ierr);
print *, "Process ", rank, " received: ", scattered_Data
call MPI_FINALIZE(ierr)
END PROGRAM

Running this code will print out the four numbers in the distro array as four separate numbers each from different processes (note the order of ranks isn’t necessarily sequential):

Process 1 received: 39
Process 0 received: 72
Process 3 received: 129
Process 2 received: 42
1.32 Using OpenMP with C

Because Summit is a cluster of CPUs, the most effective way to utilize these resources involves parallel programming. Probably the simplest way to begin parallel programming involves the utilization of OpenMP. OpenMP is a Compiler-side solution for creating code that runs on multiple cores/threads. Because OpenMP is built into a compiler, no external libraries need to be installed in order to compile this code. These tutorials will provide basic instructions on utilizing OpenMP on both the GNU C++ Compiler and the Intel C++ Compiler.

This guide assumes you have basic knowledge of the command line and the C++ Language.

Resources:

Much more in depth OpenMP and MPI C++ tutorial:


Helpful Slides from the Aristotle University of Thessaloniki:


1.32.1 Parallel “Hello, World” Program

In this section we will learn how to make a simple parallel hello world program in C++. Let’s begin with the creation of a program titled: parallel_hello_world.cpp. From the command line run the command:

```
nano parallel_hello_world.cpp
```

We will begin with include statements we want running at the top of the program:

```
#include <stdio.h>
#include <omp.h>
```

These flags allow us to utilize the stdio and omp libraries in our program. The `<omp.h>` header file will provide openmp functionality. The `<stdio.h>` header file will provide us with print functionality.

Let’s now begin our program by constructing the main function of the program. We will use `omp_get_thread_num()` to obtain the thread id of the process. This will let us identify each of our threads using that unique id number.

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    printf("Hello from process: %d\n", omp_get_thread_num());
    return 0;
}
```

Let’s compile our code and see what happens. We must first load the compiler module we want into our environment. We can do so as such:

GCC:

```
module load gcc
```

Or

Intel:
module load intel

From the command line, where your code is located, run the command:

**GCC:**

```
g++ parallel_hello_world.cpp -o parallel_hello_world.exe -fopenmp
```

Or

**Intel:**

```
icc parallel_hello_world.cpp -o parallel_hello_world.exe -qopenmp
```

This will give us an executable we can submit as a job to Summit. Simply submit the job specifying slurm to run the executable. Your submission script should look something like this:

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --time=0:01:00
#SBATCH --qos=testing
#SBATCH --partition=shas-testing
#SBATCH --ntasks=4
#SBATCH --job-name=CPP_Hello_World
#SBATCH --output=CPP_Hello_World.out

./parallel_hello_world.exe
```

Our output file should look like this:

```
Hello from process: 0
```

As you may have noticed, we only get one thread giving us a Hello statement. How do we parallelize the print statement? We parallelize it with `#pragma`. The `#pragma omp parallel { ... }` directive creates a section of code that will be run in parallel by multiple threads. Let’s implement it in our code:

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    #pragma omp parallel
    {
        printf("Hello from process: %d\n", omp_get_thread_num());
    }
    return 0;
}
```

We must do one more thing before achieving parallelization. To set the amount of threads we want OpenMP to run on, we must set an Linux environment variable to be specify how many threads we wish to use. The environment variable: `OMP_NUM_THREADS` will store this information. Changing this variable does not require recompilation of the the program, so this command can be placed in either the command line or on your job script:

```
export OMP_NUM_THREADS=4
```

Important to note: this environment variable will need to be set every time you exit your shell. If you would like to make this change permanent you will need to add these lines to your `.bash_profile` file in your home directory:

```
1.32. Using OpenMP with C  91
```
OMP_NUM_THREADS=4;
export OMP_NUM_THREADS

Now let’s re-compile the code and run it to see what happens:

GCC
g++ parallel_hello_world.cpp -o parallel_hello_world.exe -fopenmp

Or

Intel
icc parallel_hello_world.cpp -o parallel_hello_world.exe -qopenmp

Resubmit our job script and we should end with an output file similar to this one:

Hello from process: 3
Hello from process: 0
Hello from process: 2
Hello from process: 1

Don’t worry about order of processes that printed, the threads will print out at varying times.

1.32.2 Private vs. Shared Variables

Memory management is a quintessential component of any parallel program that involves data manipulation. In this section, we will learn about the different variable types in OpenMP as well as a simple implementation of these types into the program we made in the previous section.

OpenMP has a variety of tools that can be utilized to properly describe how the parallel program should handle variables. These tools come in the forms of shared and private variable type classifiers.

- Private types create a copy of a variable for each process in the parallel system.
- Shared types hold one instance of a variable for all processes to share.

To indicate private or shared memory, declare the variable before your parallel section and annotate the pragma omp directive as such:

```c
#pragma omp shared(shar_Var1) private(priv_Var1, priv_Var2)
```

Variables that are created and assigned inside of a parallel section of code will be inherently be private, and variables created outside of parallel sections will be inherently public.

Example

Let’s adapt our ‘Hello World’ code to utilize private variables as an example. Starting with the code we left off with, let’s create a variable to store the thread id of each process.

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
  int thread_id;
```
#pragma omp parallel
{
    printf("Hello from process: %d\n", omp_get_thread_num());
    return 0;
}

Now let’s define `thread_id` as a private variable. Because we want each task to have a unique thread id, using the `private(thread_id)` will create a separate instance of `thread_id` for each task.

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    int thread_id;

    #pragma omp parallel private(thread_id)
    {
        printf("Hello from process: %d\n", omp_get_thread_num());
    }
}
```

Lastly, let’s assign the thread id to our private variable and print out the variable instead of the `omp_get_thread_num()` function call:

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    int thread_id;

    #pragma omp parallel private(thread_id)
    {
        thread_id = omp_get_thread_num();
        printf("Hello from process: %d\n", thread_id );
    }
    return 0;
}
```

Compiling and submitting our code will result in a similar result to our original hello world:

```
Hello from process: 3
Hello from process: 0
Hello from process: 2
Hello from process: 1
```

### 1.32.3 Barrier and Critical Directives

OpenMP has a variety of tools for managing processes. One of the more prominent forms of control comes with the `barrier`:

```
#pragma omp barrier
```

… and the `critical` directives:
The barrier directive stops all processes for proceeding to the next line of code until all processes have reached the barrier. This allows a programmer to synchronize sequences in the parallel process.

A critical directive ensures that a line of code is only run by one process at a time, ensuring thread safety in the body of code.

**Example**

Let’s implement an OpenMP barrier by making our ‘Hello World’ program print its processes in order. Beginning with the code we created in the previous section, let’s nest our print statement in a loop which will iterate from 0 to the max thread count. We will retrieve the max thread count using the OpenMP function: `omp_get_max_threads()`.

Our ‘Hello World’ program will now look like:

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    //define loop iterator variable outside parallel region
    int i;
    int thread_id;

    #pragma omp parallel
    {
        thread_id = omp_get_thread_num();

        //create the loop to have each thread print hello.
        for(i = 0; i < omp_get_max_threads(); i++){
            printf("Hello from process: %d\n", thread_id);
        }
    }
    return 0;
}
```

Now that the loop has been created, let’s create a conditional that requires the loop to be on the proper iteration to print its thread number:

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    int i;
    int thread_id;

    #pragma omp parallel
    {
        thread_id = omp_get_thread_num();

        //create the loop to have each thread print hello.
        for(i = 0; i < omp_get_max_threads(); i++){
            if(i == thread_ID){
                printf("Hello from process: %d\n", thread_id);
            }
        }
    }
    return 0;
}
```

(continues on next page)
Lastly, to ensure one process doesn’t get ahead of another, we need to add a barrier directive in the code. Let’s implement one in our loop:

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
  int i;
  int thread_id;

  #pragma omp parallel
  {
    thread_id = omp_get_thread_num();

    for( int i = 0; i < omp_get_max_threads(); i++)
    {
      if(i == omp_get_thread_num())
      {
        printf("Hello from process: %d\n", thread_id);
      }
      #pragma omp barrier
    }
  }
  return 0;
}
```

Compiling and submitting our code should order our print statements as such:

```
Hello from process: 0
Hello from process: 1
Hello from process: 2
Hello from process: 3
```

### 1.32.4 Work Sharing Directive: omp for

OpenMP’s power comes from easily splitting a larger task into multiple smaller tasks. Work-sharing directives allow for simple and effective splitting of normally serial tasks into fast parallel sections of code. In this section we will learn how to implement omp for directive.

The directive `omp for` divides a normally serial loop into a parallel task. We can implement this directive as such:

```c
#pragma omp for { ... }
```

**Example**

Let’s write a program to add all the numbers between 1 and 1000. Begin with a main function and the stdio and omp headers:

```c
#include <stdio.h>
#include <omp.h>
```

(continues on next page)
int main(int argc, char** argv){
    return 0;
}

Now let’s go ahead and setup variables for our parallel code. Let’s first create variables partial_Sum and total_Sum to hold each thread’s partial summation and to hold the total sum of all threads respectively.

#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    int partial_Sum, total_Sum;
    return 0;
}

Next let’s begin our parallel section with pragma omp parallel. We will also set partial_Sum to be a private variable and total_Sum to be a shared variable. We shall initialize each variable in the parallel section.

#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    int partial_Sum, total_Sum;

    #pragma omp parallel private(partial_Sum) shared(total_Sum)
    {
        partial_Sum = 0;
        total_Sum = 0;
    }
    return 0;
}

Let’s now set up our work sharing directive. We will use the #pragma omp for to declare the loop as to be work sharing, followed by the actual C++ loop. Because we want to add all number from 1 to 1000, we will initialize our loop at one and end at 1000.

#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    int partial_Sum, total_Sum;

    #pragma omp parallel private(partial_Sum) shared(total_Sum)
    {
        partial_Sum = 0;
        total_Sum = 0;

        #pragma omp for
        {
            for(int i = 1; i <= 1000; i++){
                partial_Sum += i;
            }
        }
        return 0;
    }
Now we must join our threads. To do this we must use a critical directive to create a thread safe section of code. We do this with `#pragma omp critical` directive. Lastly we add partial sum to total sum and print out the result outside the parallel section of code.

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    int partial_Sum, total_Sum;

    #pragma omp parallel private(partial_Sum) shared(total_Sum)
    {
        partial_Sum = 0;
        total_Sum = 0;

        #pragma omp for
        {
            for(int i = 1; i <= 1000; i++)
            {
                partial_Sum += i;
            }
        }

        //Create thread safe region.
        #pragma omp critical
        {
            //add each threads partial sum to the total sum
            total_Sum += partial_Sum;
        }

    }

    printf("Total Sum: %d\n", total_Sum);
    return 0;
}
```

This will complete our parallel summation. Compiling and submitting our code will result in this output:

```
Total Sum: 500500
```

### 1.33 Using OpenMP with Fortran

Because Summit is a cluster of CPUs, parallel programming is the most effective way to utilize these resources. Probably the simplest way to begin parallel programming is utilization of OpenMP. OpenMP is a Compiler-side solution for creating code that runs on multiple cores/threads. Because OpenMP is built into a compiler, no external libraries need to be installed in order to compile this code. These tutorials provide basic instructions on utilizing OpenMP on both the GNU Fortran Compiler and the Intel Fortran Compiler.

This guide assumes you have basic knowledge of the command line and the Fortran Language.

**Resources:**

- Much more in depth OpenMP Fortran tutorial:
1.33.1 Parallel “Hello, World” Program

In this section we will learn how to make a simple parallel hello world program in Fortran. Let’s begin with creation of a program titled: parallel_hello_world.f90. From the command line run the command:

```
nano parallel_hello_world.f90
```

We will begin with the program title and the use statement at the top of the program:

```
PROGRAM Parallel_Hello_World
USE OMP_LIB
```

These flags allow us to utilize the omp library in our program. The ‘USE OMP_LIB’ line of code will provide openmp functionality.

Let’s now begin our program by constructing the main body of the program. We will use `OMP_GET THREAD_NUM()` to obtain the thread id of the process. This will let us identify each of our threads using that unique id number.

```
PROGRAM Parallel_Hello_World
USE OMP_LIB
PRINT *, "Hello from process: ", OMP_GET THREAD_NUM()
END
```

Let’s compile our code and see what happens. We must first load the compiler module we want into our environment. We can do so as such:

**GNU Fortran**

```
module load gcc
```

Or

**Intel Fortran**

```
module load intel
```

From the command line, where your code is located, run the command:

**GNU Fortran**

```
gfortran parallel_hello_world.f90 -o parallel_hello_world.exe -fopenmp
```

Or

**Intel Fortran**

```
ifort parallel_hello_world.f90 -o parallel_hello_world.exe -qopenmp
```

This will give us an executable we can submit as a job to Summit. Simply submit the job specifying slurm to run the executable. Your submission script should look something like this:

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --time=0:01:00
#SBATCH --qos=testing
#SBATCH --partition=shas-testing
```

(continues on next page)
Our output file should look like this:

```
Hello from process: 0
```

As you may have noticed, we only get one thread giving us a Hello statement.

How do we parallelize the print statement? We parallelize it with \texttt{omp parallel}!

The \texttt{!$OMP PARALLEL} and \texttt{!$OMP END PARALLEL} directives creates a section of code that is run from all available threads.

```fortran
PROGRAM Parallel_Hello_World
USE OMP_LIB

!$OMP PARALLEL
  PRINT *, "Hello from process: ", OMP_GET_THREAD_NUM()
!$OMP END PARALLEL
END
```

We must do one more thing before achieving parallelization. To set the amount of threads we want OpenMP to run on, we must set an Linux environment variable to be specify how many threads we wish to use. The environment variable: \texttt{OMP_NUM_THREADS} will store this information. Changing this variable does not require recompilation of the the program, so this command can be placed in either the command line or on your job script:

```
export OMP_NUM_THREADS=4
```

Important to note: this environment variable will need to be set every time you exit your shell. If you would like to make this change permanent you will need to add these lines to your .bash_profile file in your home directory:

```
OMP_NUM_THREADS=4;
export OMP_NUM_THREADS
```

Now let’s re-compile the code and run it to see what happens:

**GNU Fortran**

```
gfortran parallel_hello_world.f90 -o parallel_hello_world.exe -fopenmp
```

Or

**Intel Fortran**

```
ifort parallel_hello_world.f90 -o parallel_hello_world.exe -qopenmp
```

Resubmit our job script and we should end with an output file similar to this one:

```
Hello from process: 3
Hello from process: 0
Hello from process: 2
Hello from process: 1
```

(Note don’t worry about order of processes that printed, the threads will print out at varying times.)

1.33. Using OpenMP with Fortran
1.33.2 Private vs. Shared Variables

Memory management is a quintessential component of any parallel program that involves data manipulation. In this section, we will learn about the different variable types in OpenMP as well as a simple implementation of these types into the program we made in the previous section.

OpenMP has a variety of tools that can be utilized to properly indicate how the parallel program should handle variables. These tools come in the forms of shared and private variable classifiers.

- Private classifiers create a copy of a variable for each process in the parallel system.
- Shared classifiers hold one instance of a variable for all processes to share.

To indicate private or shared variables, declare the variable before your parallel section and annotate the omp directive as such:

```c
!$OMP PARALLEL SHARED(shar_Var1) PRIVATE(priv_Var1, priv_Var2)
```

Variables that are created and assigned inside of a parallel section of code will be inherently be private, and variables created outside of parallel sections will be inherently public.

**Example:**

Let’s adapt our ‘Hello World’ code to utilize private variables as an example. Starting with the code we left off with, let’s create a variable to store the thread id of each process. We will also change the name of the program as good coding practice.

```fortran
PROGRAM Parallel_Stored_Hello
USE OMP_LIB

INTEGER :: thread_id

!$OMP PARALLEL

    PRINT *, "Hello from process: ", OMP_GET_THREAD_NUM()

!$OMP END PARALLEL

END
```

Now let’s define `thread_id` as a private variable. Because we want each task to have a unique thread id, using the `private(thread_id)` will create a separate instance of `thread_id` for each task.

```fortran
PROGRAM Parallel_Stored_Hello
USE OMP_LIB

INTEGER :: thread_id

!$OMP PARALLEL PRIVATE(thread_id)

    PRINT *, "Hello from process: ", OMP_GET_THREAD_NUM()

!$OMP END PARALLEL

END
```

Lastly, let’s assign the thread id to our private variable and print out the variable instead of the `OMP_GET_THREAD_NUM()` function call:
1.33.3 Barrier and Critical Directives

OpenMP has a variety of tools for managing processes. One of the more prominent forms of control comes with the barrier:

```fortran
!$OMP BARRIER
```

...and the critical directives:

```fortran
!$OMP CRITICAL
...
!$OMP END CRITICAL
```

The barrier directive stops all processes for proceeding to the next line of code until all processes have reached the barrier. This allows a programmer to synchronize processes in the parallel program.

A critical directive ensures that a line of code is only run by one process at a time, ensuring thread safety in the body of code.

Example

Let’s implement an OpenMP barrier by making our ‘Hello World’ program print its processes in order. Beginning with the code we created in the previous section, let’s nest our print statement in a loop which will iterate from 0 to the max thread count. We will retrieve the max thread count using the OpenMP function:

```fortran
OMP_GET_MAX_THREADS()
```

Our ‘Hello World’ program will now look like:

```fortran
PROGRAM Parallel_Ordered_Hello
USE OMP_LIB
INTEGER :: thread_id

!$OMP PARALLEL PRIVATE(thread_id)

    thread_id = OMP_GET_THREAD_NUM()
    PRINT *, "Hello from process: ", thread_id

!$OMP END PARALLEL

END
```

Compiling and submitting our code will result in a similar result to our original hello world:

```
Hello from process: 3
Hello from process: 0
Hello from process: 2
Hello from process: 1
```
Now that the loop has been created, let’s create a conditional that will stop a process from printing its thread number until the loop iteration matches its thread number:

```fortran
PROGRAM Parallel_Ordered_Hello
USE OMP_LIB

INTEGER :: thread_id

!$OMP PARALLEL PRIVATE(thread_id)

    thread_id = OMP_GET_THREAD_NUM()
    DO i=1,OMP_GET_MAX_THREADS()
        IF (i == thread_id) THEN
            PRINT *, "Hello from process: ", thread_id
        END IF
    END DO

!$OMP END PARALLEL

END
```

Lastly, to ensure one process doesn’t get ahead of another, we need to add a barrier directive in the code. Let’s implement one in our loop.

```fortran
PROGRAM Parallel_Ordered_Hello
USE OMP_LIB

INTEGER :: thread_id

!$OMP PARALLEL PRIVATE(thread_id)

    thread_id = OMP_GET_THREAD_NUM()
    DO i=1,OMP_GET_MAX_THREADS()
        IF (i == thread_id) THEN
            PRINT *, "Hello from process: ", thread_id
        END IF
    END DO
    !$OMP BARRIER

!$OMP END PARALLEL

END
```

Compiling and submitting our code should order our print statements as such:
1.33.4 Work Sharing Directive: omp do

OpenMP’s power comes from easily splitting a larger task into multiple smaller tasks. Work-sharing directives allow for simple and effective splitting of normally serial tasks into fast parallel sections of code. In this section we will learn how to implement omp do directive. The directive omp do divides a normally serial for loop into a parallel task. We can implement this directive as such:

```
!$OMP DO
...
!$OMP END DO
```

**Example**

Let’s write a program to add all the numbers between 1 and 1000. Begin with a program title and the omp_lib header:

```
PROGRAM Parallel_Do
USE OMP_LIB
END
```

Now let’s go ahead and setup variables for our parallel code. Let’s first create variables `partial_Sum` and `total_Sum` to hold each thread’s partial summation and to hold the total sum of all threads respectively.

```
PROGRAM Parallel_Hello_World
USE OMP_LIB

INTEGER :: partial_Sum, total_Sum

END
```

Next let’s begin our parallel section with `!$OMP PARALLEL`. We will also set `partial_Sum` to be a private variable and `total_Sum` to be a shared variable. We shall initialize each variable in the parallel section.

```
PROGRAM Parallel_Hello_World
USE OMP_LIB

INTEGER :: partial_Sum, total_Sum

!$OMP PARALLEL PRIVATE(partial_Sum) SHARED(total_Sum)
  partial_Sum = 0;  total_Sum = 0;
!$OMP END PARALLEL

END
```

Let’s now set up our work sharing directive. We will use the `!$OMP DO` to declare the loop to be work sharing, followed by the actual Fortran loop. Because we want to add all number from 1 to 1000, we will initialize out loop at
Now we must join our threads. To do this we must use a critical directive to create a thread safe section of code. We do this with the !$OMP CRITICAL directive. Lastly we add partial sum to total sum and print out the result outside the parallel section of code.

This will complete our parallel summation. Compiling and submitting our code will result in this output:

Total Sum: 500500

1.34 University of Utah videos

These videos were made by the Center for High Performance Computing (CHPC) at the University of Utah. These videos include information that is useful for CU Research Computing users, but some of the information is specific to
resources at the University of Utah. If you are a CU Boulder user with questions about these videos, please contact rc-help@colorado.edu.

1.34.1 Introduction to Linux

This video provides an introduction to using Linux on a high-performance computing resource.

1.34.2 Introduction to CHPC Modules

This video demonstrates the use of modules on a CHPC resource. CHPC uses the Lmod system, which CU Research Computing also uses.

1.34.3 Slurm Basics

This video provides a description of the clusters, partitions and types of accounts available at CHPC. It also demonstrates use of the “squeue” and “srun” commands, as well as the use of the “sbatch” command to submit a batch job.

1.34.4 Slurm Batch Scripting

This video provides information about the use of modules and the use of “sbatch” commands in Slurm Batch Scripting.

1.34.5 Slurm Interactive Jobs

This video demonstrates how to run interactive jobs and provides advice on how to determine whether an interactive job best meets your needs.

1.35 Facilities, equipment, and other resources

1.35.1 Expertise

Research Computing at CU Boulder consists of a small group of computational scientists, high-performance computing specialists, and system administrators with the mission to provide leadership in developing, deploying, and operating an integrated cyberinfrastructure. This cyberinfrastructure consists of high-performance computing, storage, and high speed networking that supports and encourages research, collaboration and discovery. The groups also contribute to the educational mission of the university by providing workshops and training on cyberinfrastructure related topics as well as 1:1 consulting.
1.35.2 Compute

RC operates the joint CSU-CU-Boulder Summit supercomputer, funded by NSF under Grant No. AC- 1532236. The system has peak performance of over 400 TFLOPS. The 380 general compute nodes have two Intel Haswell CPUs with 12 cores each, 128 GB of RAM and a local SSD. Additionally, the system has 10 GPU nodes containing two NVIDIA K80 GPUs, 5 high-memory nodes with about 1 TB of main memory and in a second deployment in December 2016 20 Xeon Phi (“Knight’s Landing”) nodes with 72 real cores supporting 288 threads for development and benchmarking. All nodes are connected through a high-performance network based on Intel Omni-Path with a bandwidth of 100 Gb/s and a latency of 0.4 microseconds. 1 PB of high-performance storage is provided using the IBM GPFS file system. This system is available to CU-Boulder researchers and collaborators, as well as 10% of cycles are provided to members of the Rocky Mountain Advanced Computing Consortium.

The RC Condo Computing service offers researchers the opportunity to purchase and own compute nodes that will be operated as part of a cluster, named “Blanca.” The aggregate cluster is made available to all condo partners while maintaining priority for the owner of each node.

1.35.3 Networking

The current CU Boulder network is a 40 Gbps fiber core with Cat 5 or higher wiring throughout campus. RC has created an 80 Gbps Science-DMZ to connect the Summit supercomputer to storage and to bring individual dedicated 10 Gbps circuits to various locations as needed. CU Boulder participates in I2 (the Internet 2 higher education, government, and vendor research computing consortium) and is an active member of the Front-Range gigapop and other networks. RC has started to provide campus researchers with a leading-edge network that meets their needs and facilitates collaboration, high performance data exchange, access to co-location facilities, remote mounts to storage, and real-time communications.

1.35.4 File Transfer

For moving large amounts of data Research Computing has several nodes dedicated to GridFTP file transfer. RC’s GridFTP servers support both the Globus Connect web environment and basic GridFTP via the command line.

OIT also offers a file transfer service with a web interface, which provides a good way to transfer files to collaborators. Files are uploaded to a server and a link to download the file can be emailed to an on or off-campus user.

1.35.5 Storage

Each researcher using the computational resources at CU Boulder has a home directory with 2GB and a project space consisting of 250 GB of storage. Additional storage is provided as part of a storage condominium at a cost of $65 per TB for single copy storage. Tape and HSM are additional storage options that are available for archive data.

1.35.6 PetaLibrary

The two main categories of service offered to customers of the PetaLibrary are Active storage for data that needs to be accessed frequently and Archive storage for data that is accessed infrequently. Active data is always stored on disk and is accessible to researchers on compute resources managed by RC. Archive storage consists of a two-level hierarchical storage management (HSM) solution, with disk storage for data that is more likely to be accessed and tape for data that is less likely to be accessed frequently. The cost for the research is $65/ TB/year for disk and $35/ TB/year for archival storage.
1.36 CHANGETLOG

1.36.1 Wednesday, 17 October 2018

• Configured Slurm to reject invalid jobs during submit, rather than accept and pend forever

1.36.2 Wednesday, 16 May 2018

• Shut down legacy login nodes

1.36.3 Wednesday, 9 May 2018

• Moved login.rc.colorado.edu to a new set of login nodes

1.36.4 Tuesday, 27 February 2018

• Reduced memory limits on Summit

1.36.5 Tuesday, 20 February 2018

• Installed version control clients on Summit in response to a request for Mercurial

1.36.6 Monday, 19 February 2018

• Reduced default per-core memory requests on Summit

1.36.7 25 September 2017

• /work/ mounts are now supported for JupyterHub Virtual Notebooks.
• JupyterHub Notebooks can now be spawned on Summit, and the Crestone spawn option has been deprecated.
• JupyterHub Ippparallel profiles have been updated for Summit. To remove stale Janus/Crestone profiles, please follow the procedure described in the JupyterHub User Guide
• JupyterHub Cluster notebooks now use lmod instead of virtualenv for software dependencies.