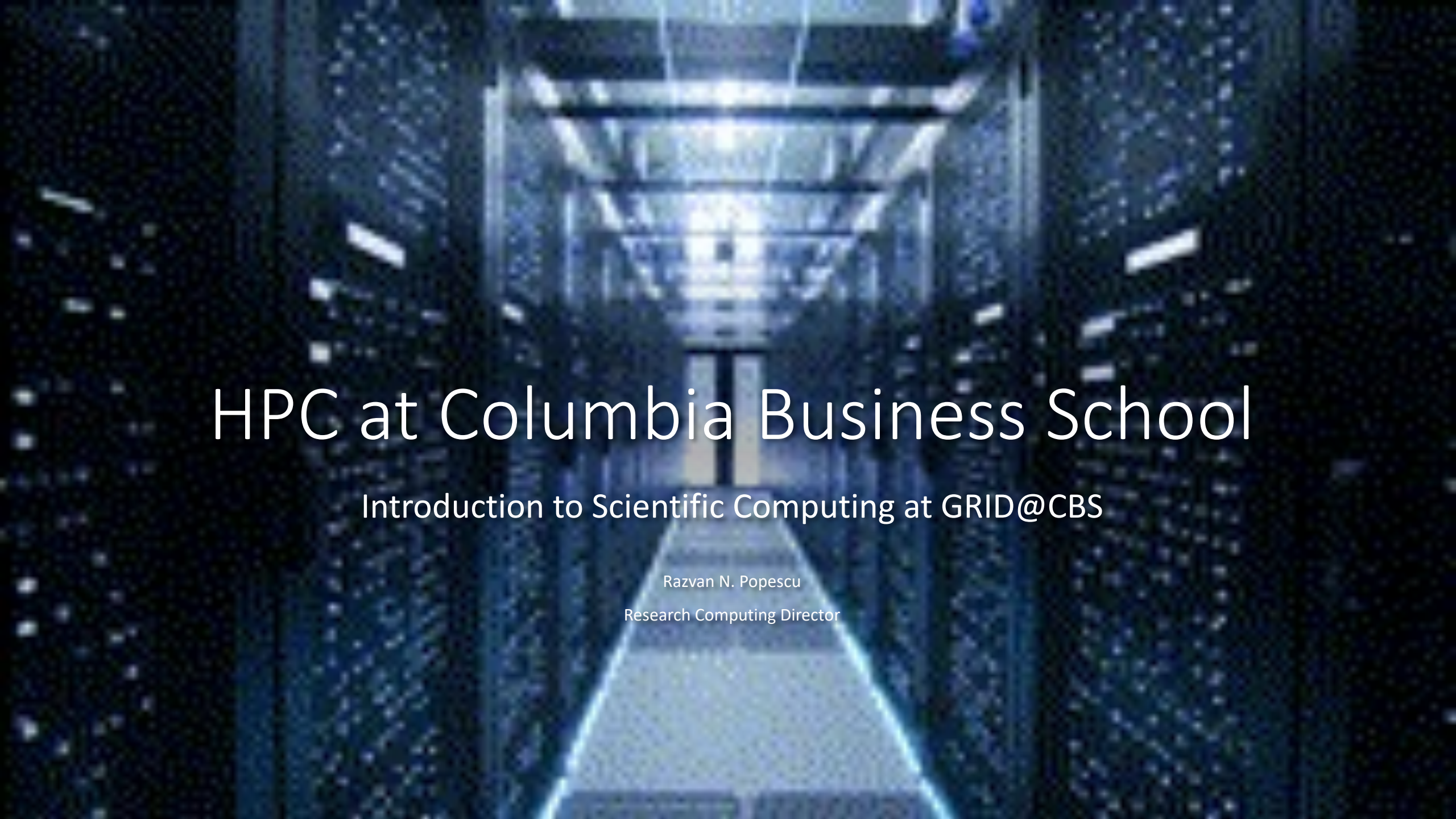
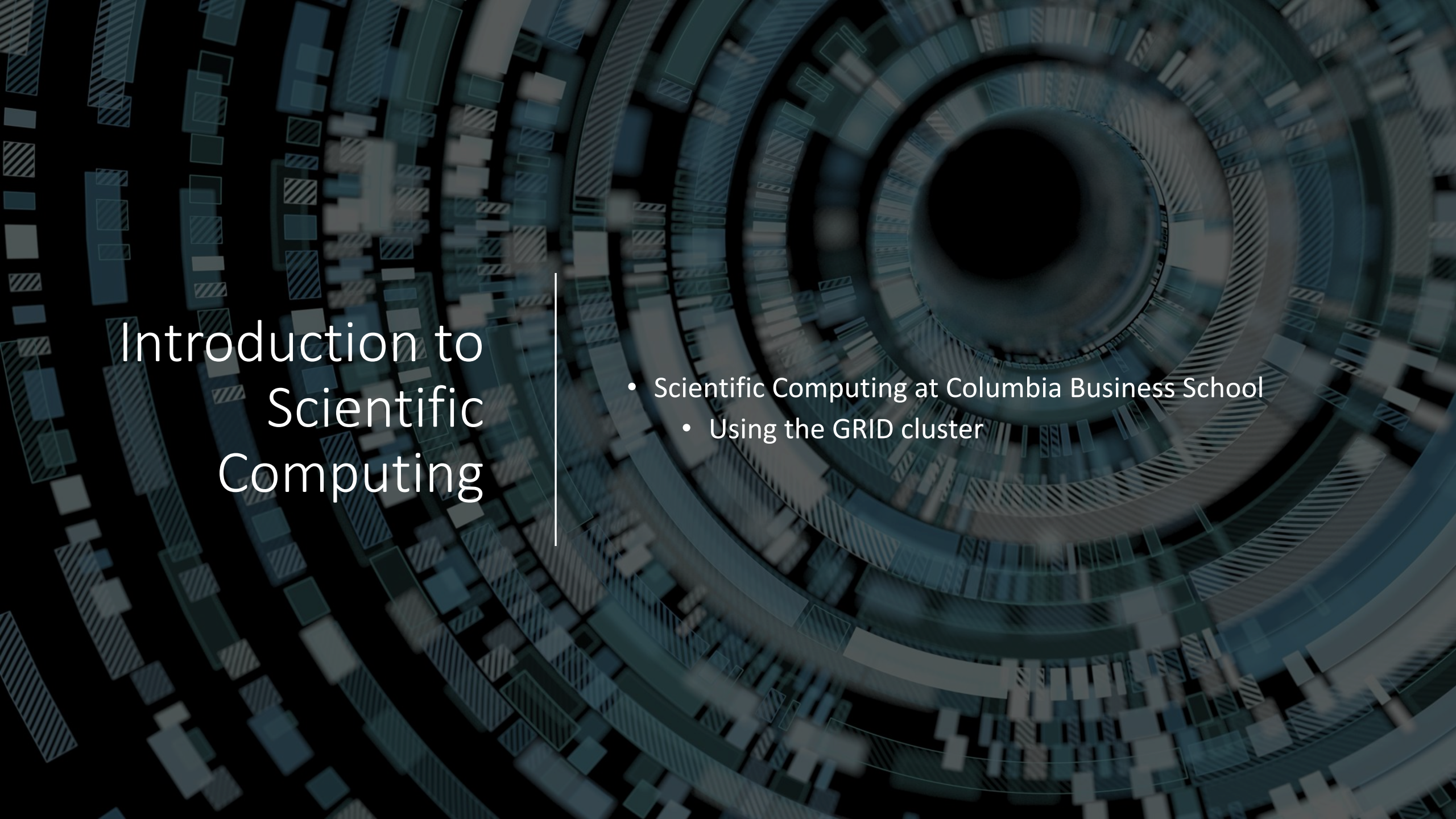


HPC at Columbia Business School

Introduction to Scientific Computing at GRID@CBS

Razvan N. Popescu

Research Computing Director



Introduction to Scientific Computing

- Scientific Computing at Columbia Business School
  - Using the GRID cluster

Scientific computing

From IBM mainframes to minicomputers and supercomputers…

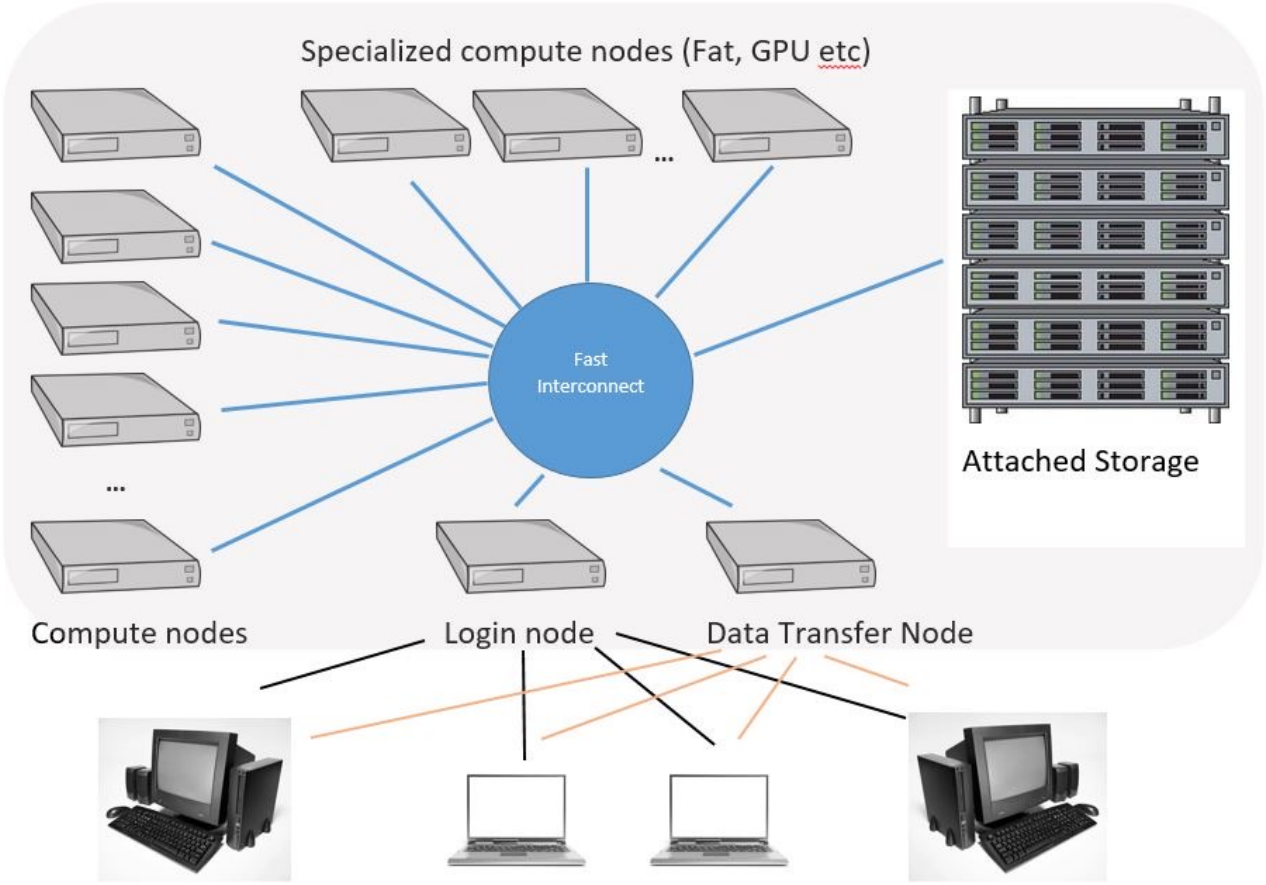
Scientific computing

To HPC clusters, the cloud

… and the GRID@CBS

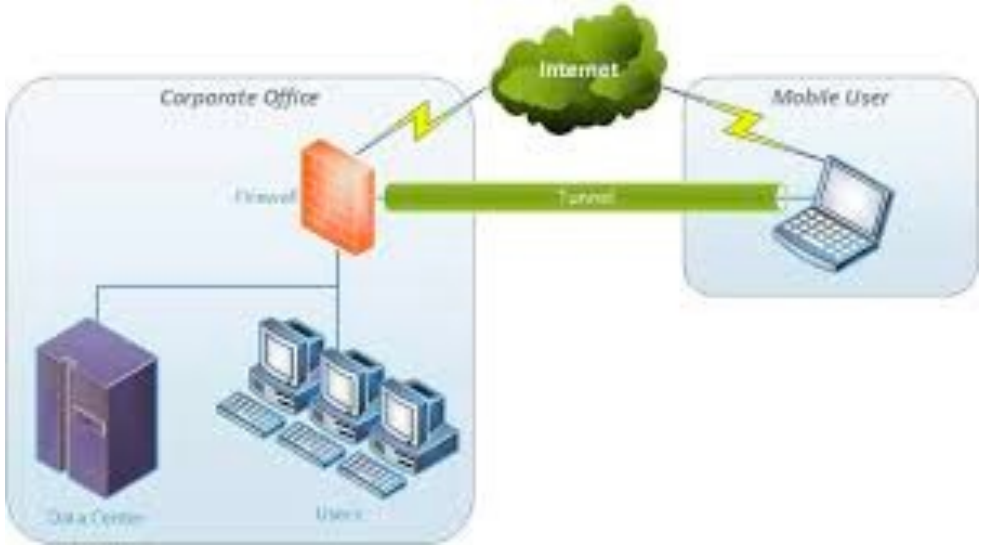
Typical (simple) HPC cluster architecture

- Pool of compute nodes (homogeneous or heterogeneous)
- Specialized compute nodes (high mem, GPU, FPGA, etc.)
- Shared (high-performance) storage
- Interface nodes:
  - Login/Interactive node(s)
  - Data transfer node(s)
  - *(Admin node)*
- *(Infrastructure services):*
  - ***(Scheduler)***
  - *(Backup)*
  - *(Identity management)*
  - *(Logging and monitoring)*



Connecting to the GRID@CBS cluster

COLUMBIA-VPN



- Provides a mean for **secure user access** to Columbia internal services.
- Protects internal CBS systems without requiring complex, error-prone, firewall configurations.

- Creates a ***virtual presence*** **on Columbia internal networks**, similar to being connected to (secure) campus infrastructure.

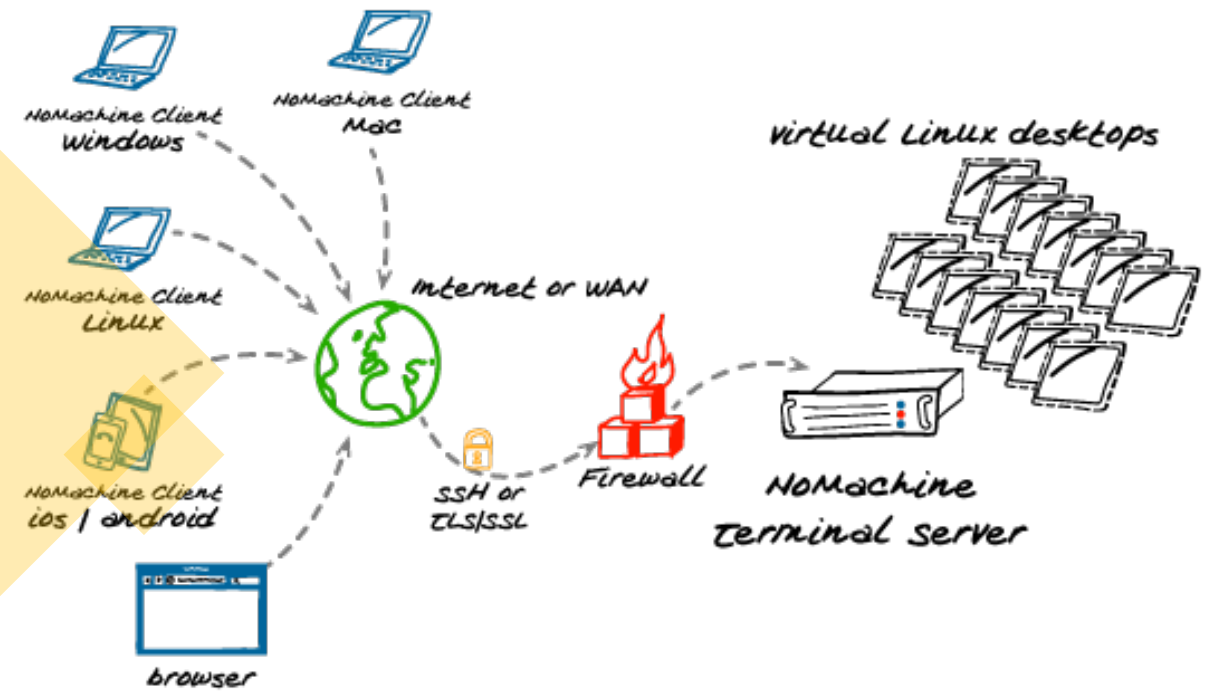
- Due to its ability to grant high-value privileges, it requires dual-factor authentication.

Connecting to COLUMBIA-VPN

- References:
  - CBS-ITG's "CBS Virtual Private Networking
    - https://www8.gsb.columbia.edu/itg/vpn
  - CUIT VPN (scroll down to "How to connect with CUIT VPN using Duo MFA"):
    - https://cuit.columbia.edu/remote-access-services#/cu_accordion_item-11433
- Step 1: Setup and activate Duo MFA
  a) Install and pair the mobile Duo app, as described in these videos: https://cuit.columbia.edu/mfa#/text-1841
  b) (optional) Test the MFA setup: https://cas.columbia.edu/cas-duo-enroll/duoAuth
- Step 2: Connect to CUIT VPN gateway
  a) Go to https://vpn.cc.columbia.edu to download and install the Cisco AnyConnect VPN client. At the second password prompt, type "PUSH". Accept the mobile Duo MFA challenge on the phone setup at step 1.
  b) Open Cisco AnyConnect client and type **vpn.cc.columbia.edu** in the window. Click "Connect"
  c) Enter your UNI, password, and the word **PUSH** in the field "Duo Action". Click "OK" and accept the mobile Duo MFA challenge.
  d) ***Alternative*** *** to step 2c** above: *if you have problems receiving the Duo authentication challenge* on your mobile phone:
    - Open the Duo app and copy the displayed 6 digits numeric code
    - Enter it in the "Duo Action" field **instead of** the word "PUSH"

Connecting to the GRID Interactive Service ("login node")

- The *GRID Interactive Service* is composed of 3 login servers, but presents a single access point:
  research.gsb.columbia.edu
- Supports 2 types of interfaces:
  - Virtual desktop
  - Command line terminal
- Requires specialized clients:
  - Nomachine client:
    - Provides both virtual desktop and command line terminal interf.
    - **Supports graphical applications** (Rstudio, MATLAB, etc.)
  - SSH client {Mac Terminal, PuTTY, etc.}
    - Lightweight
    - Provides only command line terminal interface
    - NO SUPPORT for graphical applications



Nomachine client

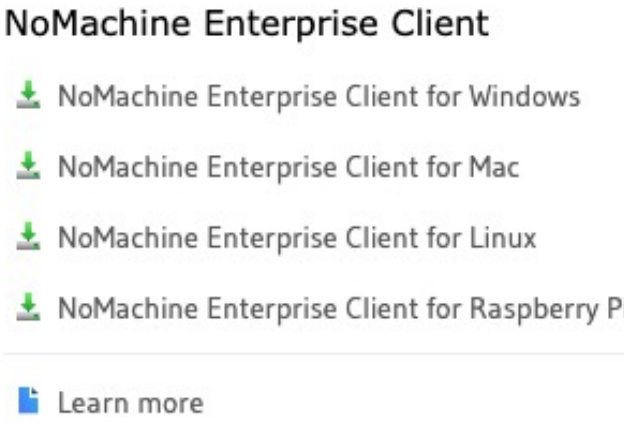
- Multi-platform enhanced visual client optimized for remote access to graphical applications.
- Supports **virtual desktops** and (rootless) **custom applications** (e.g. terminal emulators).
- **Provides session persistency!** (can disconnect/reconnect to active sessions without losing progress)
- Best for all graphical applications (client-side caching and traffic compression).

Connecting to the GRID via Nomachine client

1. Download and install the Nomachine **Enterprise Client** from:

   https://www.nomachine.com/download-enterprise

2. Create a new connection template:
   - Give it a friendly name
   - Host: **research.gsb.columbia.edu**
   - Protocol: **NX** (Port: 4000)
   - Authentication: Password



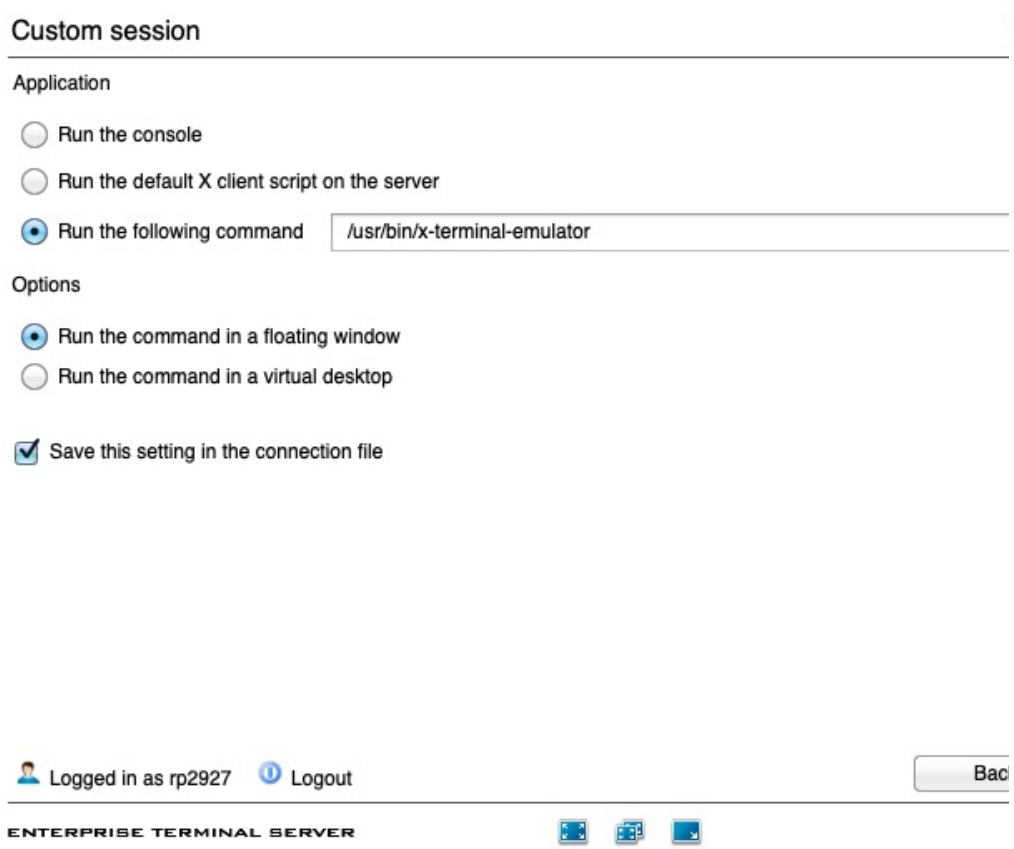
Connecting via Nomachine
-- Virtual Desktop --

3. Double click the connection icon just created at step 2 and fill in the authentication information (UNI & password)
4. Double click on “Create a new virtual desktop”
   - (BUG?!) Accept the defaults when prompted… they won’t provide the desired config anyway…
5. A virtual desktop running on one of the three servers will appear
   - CTRL+ALT+0 and select the DISPLAY config:
     - Select “Resize remote display”
     - You can now extend your desktop’s effective area by resizing the window
   - **To terminate the session**, click your name on the top right corner and select “Log Out…”. Confirm on the popup window.
   - **To disconnect the session** *but leave it active for further reconnection*, close the desktop window from the (X) corner buttons.

Connecting via Nomachine
-- Command Line Terminal --

3. Double click the connection icon just created at step 2 and fill in the authentication information (UNI & password)
4. Double click on “Create a new custom session”
    1. Select “Run the following command”…
    2. and enter “/usr/bin/x-terminal-emulator” in the field
    3. Check “Save this setting in the connection file”
5. A command line terminal will appear in a floating window
    - **To terminate the session**, type “exit” or CTRL-d, or close the terminal window from the corner buttons.
    - **To disconnect the session** *but leave it active for further reconnection*, close the **Nomachine** control window from the (x) corner buttons.

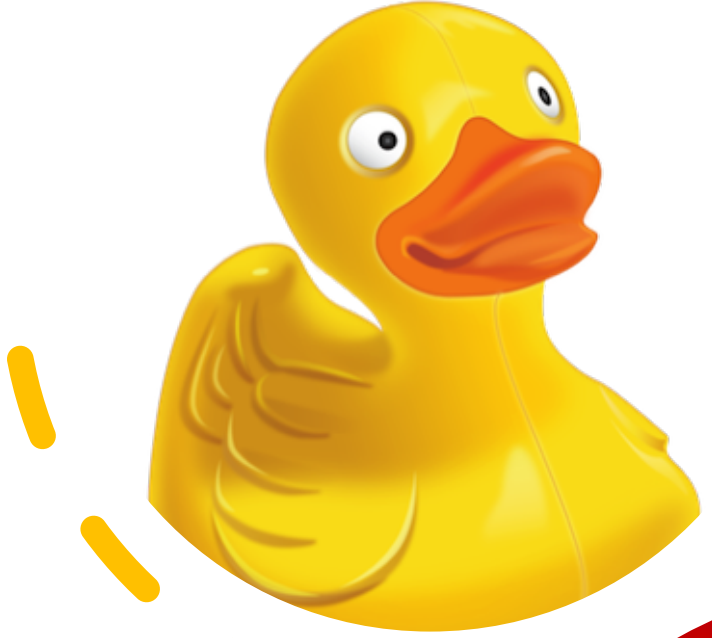
- *Mac users need to install XQuartz (www.xquartz.org) and restart!!*



Connecting to the GRID via SSH client

- Supports only command line terminals (no virtual desktops)
  - *Supports graphical applications via X11 tunnel. GUI responsiveness is net inferior to Nomachine client, and thus we recommend the use of the latter solution when GUI support is required.*
- Mac platforms can use the native “Terminal” application:
  - Connect via “ssh UNI@research.gsb.columbia.edu”
- Windows platforms should use PuTTY (www.putty.org)
  - Host: research.gsb.columbia.edu
  - Port: 22 (default)

Transferring data



- GRID@CBS uses a dedicated data transfer portal to avoid interference with interactive work.
  - Hostname: **researchfiles.gsb.columbia.edu**
  - (Protocol: **SFTP** ; Port: **22**)
- Use your favorite data transfer (sftp) client:
  - Filezilla (https://filezilla-project.org/)
  - Cyberduck (https://cyberduck.io/)
  - WinSCP (https://winscp.net/)

- *For high-volume data transfers use GLOBUS (globus.org)*

File formats across platforms

- Various “platforms” (Windows, Mac, Linux) employ different formats for certain files. This will create incompatibilities and generate errors when files are moved between platforms. Certain file formats can be easily converted, some cannot.

A. Executables (applications and compiled code) are platform specific and *cannot* be converted. One needs to install the appropriate version of the application (Win, Mac, Linux; 32/64bit) or rebuild from source code. Attention to (compiled) modules for modular applications (e.g. Python, R)

B. Text files (source code, configs, scripts) might use a different end-of-line (EOL) terminator. Conversion is relatively easy.

   - Frequent mistake: **script files will fail**, errors indicating unexpected “^M” characters at the end of the line.

Beware of Windows EOL characters

- Windows uses a different line terminator that will confuse Linux. The line terminator is an invisible character; thus, you'll not see any difference by simply listing the file!
- **Script files will behave oddly when executed.**
- **Error log files will show "^M" characters at the end of the quoted lines!**
- **The "file" command will indicate the file has "CRLF line terminators"**

Use the command "dos2unix" to convert text/script files imported from Windows systems.

```
(live)rp2927@researchint01:/shared/share_B9334$ cat test.py
#!/apps/anaconda3/bin/python
print('Testing...')
quit()

(live)rp2927@researchint01:/shared/share_B9334$ ./test.py
-bash: ./test.py: /apps/anaconda3/bin/python^M: bad interpreter: No such file or directory
(live)rp2927@researchint01:/shared/share_B9334$
(live)rp2927@researchint01:/shared/share_B9334$ file test.py
test.py: a /apps/anaconda3/bin/python script, ASCII text executable, with CRLF line terminators
(live)rp2927@researchint01:/shared/share_B9334$
(live)rp2927@researchint01:/shared/share_B9334$ cat -v test.py
#!/apps/anaconda3/bin/python^M
print('Testing...')^M
quit()^M
^M
(live)rp2927@researchint01:/shared/share_B9334$ dos2unix test.py
dos2unix: converting file test.py to Unix format...
(live)rp2927@researchint01:/shared/share_B9334$
(live)rp2927@researchint01:/shared/share_B9334$ file test.py
test.py: a /apps/anaconda3/bin/python script, ASCII text executable
(live)rp2927@researchint01:/shared/share_B9334$
(live)rp2927@researchint01:/shared/share_B9334$ cat -v test.py
#!/apps/anaconda3/bin/python
print('Testing...')
quit()

(live)rp2927@researchint01:/shared/share_B9334$ ./test.py
Testing...
(live)rp2927@researchint01:/shared/share_B9334$
```

Data analysis -- A typical workflow

- Requirements:
  - Data transfers
  - File management
  - File editing
  - Running user code or applications (interactively)
  - Run code or applications in batch mode
  - Reading state, logs and code output

Preparing input data

- Stage in
- File management

Developing and debugging code

- Import or create analysis code
- Debug code using *small data samples*

Running production analysis

- Prepare batch run scripts (Debug control scripts)
- Run analysis on full datasets (batch run)

Analyzing results

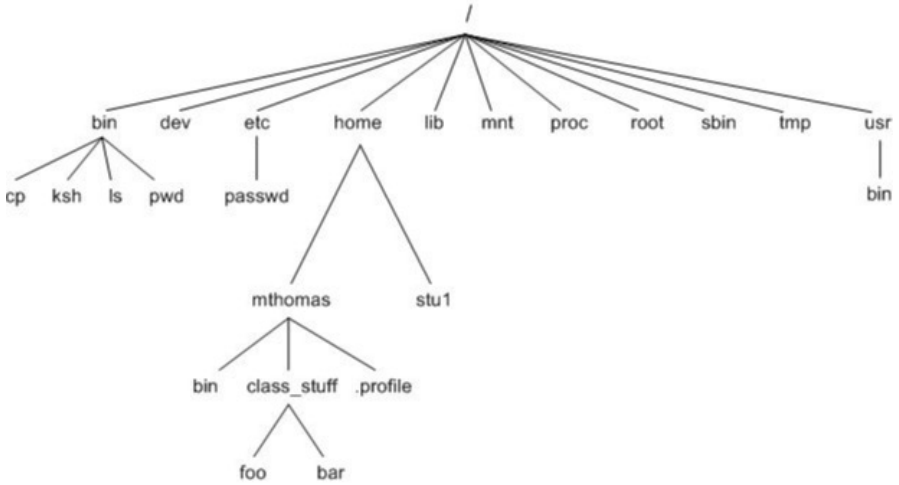
…Re-iterate from code revision (step #2)…

Export final results

File management on Linux systems

File management on Linux systems

- The Linux file namespace:
  - Hierarchical namespace composed of…
  - Nested folders (directories) and regular files (…and links)
  - Root is (named): "/"
- The Linux filename:
  - A sequence of nested folders, starting at *root* (/)…
  - **Separated by (forward) slashes "/"** …
  - Ending with the file or directory name
  - File and directory names are arbitrary strings of alphanumeric characters.
    - ***Case sensitive!***
    - ***No concept of type or version built into the system***
    - ***Do not use special characters (`~!$%&*()/\{}[];) and avoid blank space***.



/user/rp2927/B9334/HelloWorld.sh
/shared/B9334/data/test01
/scratch/rp2927/src/Stata15/xstata-mp.tar.gz

- Remote filesystems and namespace mounting:
  - To be accessible, all storage "containers" must be *mounted* into the common namespace tree
  - Uniform access model for local and remote data

File management on Linux systems – Absolute and Relative file references

- The “current working directory” is a useful convenience… and an important concept
  - Using (long) full file paths is tedious and error prone
    - E.g. /shared/share_powell/credit/market_data/data_by_company/company_monthly_join/2020/07/IBM/sec_filling
  - Declare a “current directory” and refer to files relative to that location
    - Refer to the **current directory as “.”** and complete the filename reference **relative** to it:
      - E.g. if CWD is: /shared/share_powell/credit/market_data/data_by_company/company_monthly_join/2020/07/IBM
      - Use ./sec_filling or ./policy/environmental to access the respective sub-directories and files therein
  - Commands:
    - **cd** (“change (working) directory”)
    - **pwd** (“print working directory”)
  - Special notations:
    - Current directory: “.”
    - Parent directory: “..”
    - Home directory: “~”

Browsing the namespace

…without a visual browser

- Moving (the current directory) around:
  - Down: **cd** *subfolder* OR **cd** *subdir1/subdir2/subdir3*
  - Up: **cd ..** OR **cd ../../..**
- Listing directory content:
  - **ls** *subdir*
  - **ls** *subdir1/subdir2*
  - **ls** (OR **ls .**) Lists the current directory
  - **ls ..** Lists the parent directory
  - **(ls –larth** *subdir***)** (ls options: long, all, by time, reverse order, human units)

Files and directories. Attributes and mode.

- File/dir attributes
  - List using the “long” format (ls –l)
  - List “hidden” files (filenames starting with “.”) (ls –a)
- Mode (file protection)
  - 12bit mask – format: “f/d/l UUU GGG OOO”
    - f/d/l = {-|d|l} file type (file/directory/link)
    - U/G/O = user/group/other
    - rwx = Read Write Execute
- Command: **chmod** (“change mode”)
  - **chmod** *mode FileDirName*
    - *mode* = [ugo]{+-=}{rwx}
    - Ex.: **chmod +x exec.sh** **(memorize it !!)**

Mode	(links)	Owner	Group	Size	Timestamp	Name

```
(live)rp2927@researchint01:~$ ls -la
total 6345
drwx------  79 rp2927 rp2927   16384 Jan 12 11:45  .
drwxrwxr-x 875 root   root     32768 Jan  9 04:49  ..
drwxrwxr-x   4 rp2927 rp2927    4096 Jul  3  2019  accounts
drwxrwxr-x   3 rp2927 rp2927    4096 Jul  6  2018  .anaconda
drwxrwxr-x   3 rp2927 rp2927    4096 Jul  6  2018  .astropy
drwxrwxr-x   2 rp2927 rp2927    4096 Jan  7 21:23  B9334
-rw-------   1 rp2927 rp2927    7739 Jan 12 11:33  .bash_history
-rw-------   1 rp2927 rp2927      18 Jul  3  2018  .bash_logout
-rw-------   1 rp2927 rp2927     260 Oct  8 15:57  .bash_profile
-rw-------   1 rp2927 rp2927     797 Nov 17 21:21  .bashrc
-rw-r--r--   1 rp2927 rp2927    1329 Dec 14  2018  _bashrc_anaconda3
drwxrwxr-x   3 rp2927 rp2927    4096 Oct 26 18:22  bin
drwxrwxr-x  23 rp2927 rp2927    4096 Oct  7 18:28  .cache
drwxrwxr-x   2 rp2927 rp2927    4096 Mar 25  2019  compare_env
drwxrwxr-x   4 rp2927 rp2927    4096 Sep  6  2018  .conda
-rw-rw-r--   1 rp2927 rp2927      66 Nov 17 21:22  .condarc
drwxr-xr-x  30 rp2927 rp2927    4096 Dec  2 12:32  .config
-rw-rw-r--   1 rp2927 rp2927      51 Oct 15  2019  cplex.log
-rwxrwxr-x   1 rp2927 rp2927    8464 Feb  7  2020  crash
-rw-rw-r--   1 rp2927 rp2927      31 Feb  7  2020  crash.c
drwx------   3 rp2927 rp2927    4096 Sep 12  2018  .dbus
drwxr-xr-x   2 rp2927 rp2927    4096 Nov 13 20:20  Desktop
```

Creating/destroying content

- Make a new directory (**mkdir**)
  - **mkdir** *dirName*
- Delete (remove) directory (**rmdir**)
  - **rmdir** *dirName* (will work only if dir is empty!)
- Create a new file
  - **touch** *fileName*
  - Or use a file editor
- Remove a file (**rm**)
  - **rm** *fileName*
- Copy a file (**cp**)
  - **cp** *oldFileName newFileName*
- Move (rename) a file (**mv**)
  - **mv** *oldFileName newFileName*

Displaying file content

…without a file browser (!)

- Displaying the content of a file uses a strangely named command: **cat** (“concatenate and print”)
  - **cat** *fileName*
- Paging through screenfuls of file content has an even more bizarre name choice: **less** (OR **more**) (Geek humor: “less is more”)
  - **less** *fileName*
  - (**more** *fileName* ; “more” is the older version of this Unix utility)
  - “less” Subcommands:
    - <space> page forward ; <b> page backward
    - keyboard arrows move one line at a time
    - <q> exit
    - </string> find ; <n> next match forward ; <N> next match backward
    - <G> go to end ; <g> go to beginning

Accessing local system documentation

- And the most important command is…
  - Print the documentation of a command, tool or utility – the “manual” (**man**)
    - **man** *command*
  - Examples:
    - **man** *ls*
    - **man** *rm*
    - **man** *man*
    - **man** *bash* --- *This command will display the manual of the bash shell (a must read)* ---

File editors

- Vi
- Nano
- Emacs
- [your favorite here]

Overview of nano's shortcuts

The editor's keystrokes and their functions

File handling

Ctrl+S	Save current file
Ctrl+O	Offer to write file ("Save as")
Ctrl+R	Insert a file into current one
Ctrl+X	Close buffer, exit from nano

Editing

Ctrl+K	Cut current line into cutbuffer
Alt+6	Copy current line into cutbuffer
Ctrl+U	Paste contents of cutbuffer
Alt+T	Cut until end of buffer
Ctrl+]	Complete current word
Alt+3	Comment/uncomment line/region
Alt+U	Undo last action
Alt+E	Redo last undone action

Search and replace

Ctrl+Q	Start backward search
Ctrl+W	Start forward search
Alt+Q	Find next occurrence backward
Alt+W	Find next occurrence forward
Alt+R	Start a replacing session

Deletion

Ctrl+H	Delete character before cursor
Ctrl+D	Delete character under cursor
Alt+Bsp	Delete word to the left
Ctrl+Del	Delete word to the right
Alt+Del	Delete current line

Operations

Ctrl+T	Execute some command
Ctrl+J	Justify paragraph or region
Alt+J	Justify entire buffer
Alt+B	Run a syntax check
Alt+F	Run a formatter/fixer/arranger
Alt+:	Start/stop recording of macro
Alt+;	Replay macro

Moving around

Ctrl+B	One character backward
Ctrl+F	One character forward
Ctrl+←	One word backward
Ctrl+→	One word forward
Ctrl+A	To start of line
Ctrl+E	To end of line
Ctrl+P	One line up
Ctrl+N	One line down
Ctrl+↑	To previous block
Ctrl+↓	To next block
Ctrl+Y	One page up
Ctrl+V	One page down
Alt+\	To top of buffer
Alt+/	To end of buffer

Special movement

Alt+G	Go to specified line
Alt+]	Go to complementary bracket
Alt+↑	Scroll viewport up
Alt+↓	Scroll viewport down
Alt+<	Switch to preceding buffer
Alt+>	Switch to succeeding buffer

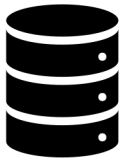
Information

Ctrl+C	Report cursor position
Alt+D	Report word/line/char count
Ctrl+G	Display help text

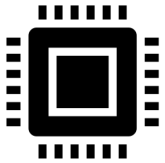
Various

Alt+A	Turn the mark on/off
Tab	Indent marked region
Shift+Tab	Unindent marked region
Alt+N	Turn line numbers on/off
Alt+P	Turn visible whitespace on/off
Alt+V	Enter next keystroke verbatim
Ctrl+L	Refresh the screen
Ctrl+Z	Suspend nano

Storage at GRID@CBS

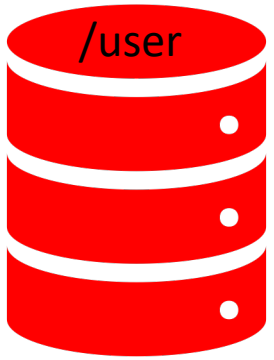


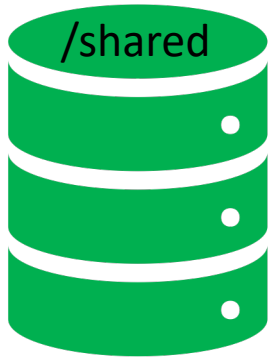
- Disk space vs. memory (RAM) space
  - Disk space:
    - Stores **files**: code, input/output data, logs, config, etc.
    - **Persistent**
    - Controlled by (disk space) **quota**
      - Check quota using the "***lsquota***" command
  - Memory (RAM):
    - Array of bytes that the executables use to perform computations.
    - Contains all code variables (structures) *during code execution*.
    - **Volatile**
    - Created when jobs run, *extended during execution as needed* (!) and destroyed at termination.
    - Controlled by the operating system and *limited by the job scheduler*.

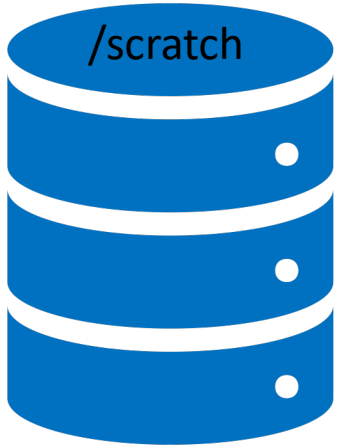


Disk Storage at GRID@CBS

- Disk space: three main ”areas”:
  - Home directory
    - Primary individual disk space – 100GB
    - /user/UNI
  - Shared folders (project space)
    - Created for project/group collaborations
    - Shared access by given user group
    - E.g. /shared/share_B9334
  - Scratch
    - *TEMPORARY* individual workarea – 500GB
    - /scratch/UNI *(created by user)*
    - No protection of any kind!!
    - **Files not used in 1mo will be automatically deleted**
- Separate quotas apply
- Home & Shared are protected by backup and snapshots (NOT scratch!!)







Configuration (startup) files

- Immediately after login the shell reads in a few configuration files to setup your environment, before prompting you for commands.
  - ~/.bash_profile for interactive shells
  - ~/.bashrc for non-interactive shells

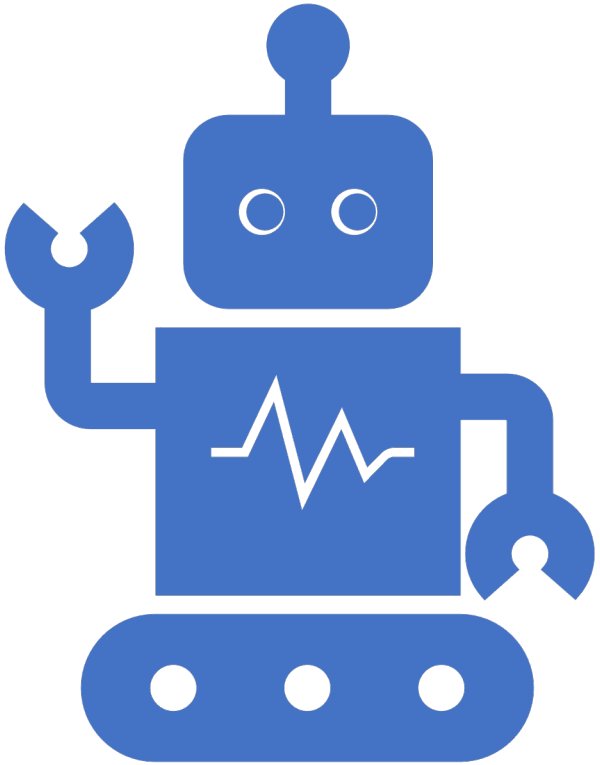
```
(live)rp2927@researchint01:~$ cat .bashrc
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

# Uncomment the following line if you don't like systemctl's auto-paging feature:
# export SYSTEMD_PAGER=

# User specific aliases and functions
alias l='ls -lh'
alias la='ls -lah'
alias lt='ls -larth'
alias sgr='sge_run'
alias sgrt='sge_run --grid_q=test.q'


# >>> conda initialize >>>
# !! Contents within this block are managed by 'conda init' !!
__conda_setup="$('/apps/anaconda3/bin/conda' 'shell.bash' 'hook' 2> /dev/null)"
if [ $? -eq 0 ]; then
    eval "$__conda_setup"
else
    if [ -f "/apps/anaconda3/etc/profile.d/conda.sh" ]; then
        . "/apps/anaconda3/etc/profile.d/conda.sh"
    else
        export PATH="/apps/anaconda3/bin:$PATH"
    fi
fi
unset __conda_setup
# <<< conda initialize <<<

(live)rp2927@researchint01:~$
```

```
(live)rp2927@researchint01:~$ cat .bash_profile
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/.local/bin:$HOME/bin

export PATH

# Add the SGE man pages to the path
export MANPATH=:/opt/n1ge/man
(live)rp2927@researchint01:~$
```

Running applications

Running applications (interactively)

- Simply type the application name
  - R
  - matlab
  - stata
- How about graphical interfaces?
  - Add “x” in front of the application name (if the application has a graphical interface version!)
    - xmatlab
    - xstata
    - (no xR !)
  - Must use the Nomachine client
- Is it so simple?
- These command names don’t really exist. What is really happening here?
- What is that “Connection to research… closed” message?

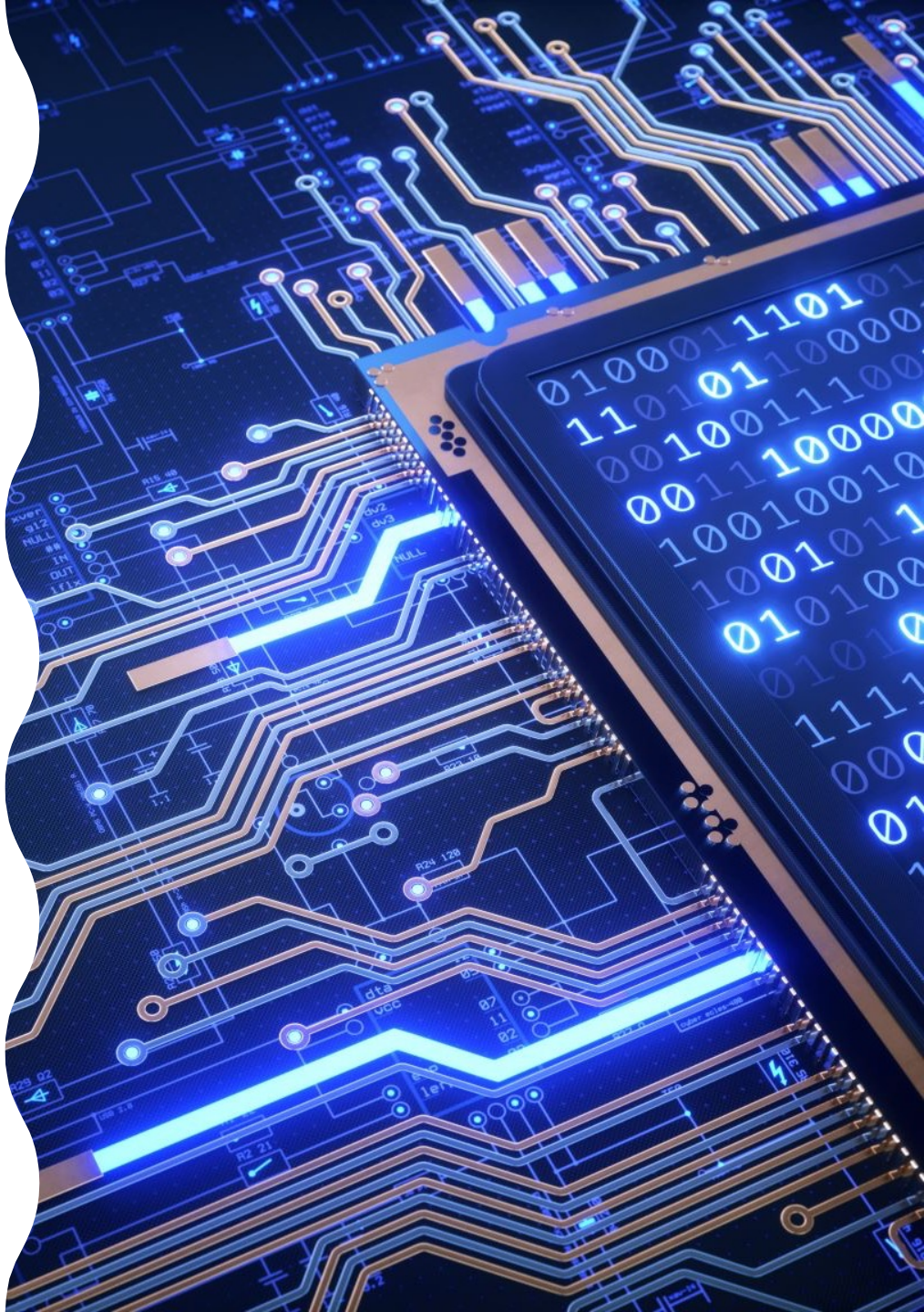


How does the system know how to respond to our commands?

- We type a string of words/characters on the command line:
  - *name options parameters*
  - Ex. stata –b do filename &
- What is the system programmed to do with that?
  - It takes the first word ("name"), *searches for a file with that name* and executes it!
  - Passes the whole command line content to that executable.
- Who controls all that?
  - The "shell" – the program that runs the conversation that occurs at the command prompt as soon as you logged in!
  - By default, we run the "bash" shell.
- Where is the shell searching a match for the first command word?
  - Unless told explicitly(!)… the shell uses an *environment variable* named **PATH**, *containing an ordered list of directories*.
  - Ex.: **echo $PATH**
    /apps/anaconda3/condabin:/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games:/apps/bin:/opt/n1ge/bin/lx24-amd64:/apps/wrappers:/user/rp2927/.local/bin:/user/rp2927/bin
- Why do we care?
  - Because we might need to write simple shell scripts to control our work (especially in batch mode!), and because the PATH will be manipulated by Python private environment activations (and trouble might come out of it).

Inside “Running Applications”

- If the shell searches a match for an arbitrary string in an ordered list of locations, we can do two interesting things (among many):
  - Add new commands – e.g. the “x” versions of GUI applications
  - **Overload** existing commands(!) – in fact, all the application invocations you’ve seen so far are “fake”!
- **(drumroll…) The *“wrapper”* script**
  - A special script that “intercepts” all GRID application invocations
  - *Expands the interface adding GRID specific options*
  - Runs the payload remotely on a lightly used machine (remember the “connection to research… closed” message?)



GRID@CBS Command Syntax

Start with the *publicly available standard application invocation* and **insert all GRID options after the application/commandName** and before any application options or parameters.

gridOptions

applicName *applicOptions* *applicParameters*

applicName *gridOptions* *applicOptions* *applicParameters*

GRID@CBS Options

- **All GRID options start with "--grid_"**
- Resource requests:
  - --grid_ncpus=n Number of CPU cores (default n=1)
  - --grid_mem=nG Amount of memory (default: 4GB)
  - --grid_long Unlimited execution time
- Execution mode:
  - --grid_submit=batch|interactive (default: interactive)
- Notification:
  - --grid_email="*myemail@columbia.edu*" Will send an email notification at job completion (batch).
- Array jobs:
  - --grid_array=<start>-<end>[:<step>][/maxConc]

Examples

- matlab -nodesktop -nojvm -batch myscript
- matlab --grid_mem=20G -nodesktop -nojvm -batch myscript
- stata --grid_mem=15G
- stata --grid_mem=25G -b do myfile
- xstata --grid_ncpus=8 --grid_mem=40G
- R --grid_ncpus=4 --grid_mem=50G
- Rscript --grid_mem=50G myscript

Running user code

Don’t we already know?

- We have seen already a few methods:
  - matlab --grid_mem=20G -nodesktop -nojvm -batch myscript
  - stata --grid_mem=25G -b do myfile
  - Rscript --grid_mem=50G myscript
- How about Python?
  - Maybe “python myScript.py”?...
    - No…. the default system configuration interferes with us
  - How about a made-up command then?...
    - Yes!... “anapy3 --grid_mem=25G myScript.py”
- How about my C++ code or my bash, Ruby, Julia, etc. script?...
  - Hmm…
- Can we do better? How about a comprehensive approach?!

Running an arbitrary user script

- All starts with the shell!
  - The shell (bash) provides a powerful and versatile programming environment, provisioned with logic test, loops, I/O control, integer arithmetic and more.
    - Consider using it to control your analysis workflow
- A shell special feature: the “shebang” (*I can’t stand this name*):
  - Magic character sequence starting with “#!” *on the first line of the shell script*
  - The shell interprets the rest of the line as describing the absolute location of an executable to start
  - The shell starts this executable (interpreter), ignores processing the rest of the file, and instead passes it to this executable as input
- Make sure the script is executable!! (remember chmod?)
    - *chmod +x scriptName*

Shebang examples

- #!/bin/bash
  - Runs the bash shell. A bit redundant unless you run on a system with a different default shell)
- #!/apps/anaconda3/bin/python
  - Runs the system installed Anaconda Python3 interpreter
- #!/user/rp2927/.conda/envs/tf_gpu/bin/python
  - Runs the Python interpreter from my private environment named tf_gpu
- #!/apps/bin/Rscript
  - Starts the Rscript interpreter for running R code

Running arbitrary code/script

- We can now run arbitrary interpreted (Python, Julia, ruby, etc.) or compiled (C/C++/Fortran) code, but …
- If we would invoke the compiled code or script name, as we did with applications, the code or script will execute locally on the interactive nodes, which is highly undesirable, counter-productive and even disruptive.
  - (Remember, the common applications were wrapped and transparently modified to run remotely)
- Moreover, the GRID options won't be recognized!
- To use all the facilities and resources of the cluster, we need a special tool

Executing arbitrary code, scripts or applications on GRID@CBS

- To use all the facilities and resources of the cluster, use:

grid_run

- grid_run is a general-purpose handler useful for **both scripts and user-built executables** (compiled code), as well as for arbitrary applications.
- grid_run accepts all GRID options
- grid_run uses the cluster scheduler to dispatch the workload to a lightly used node
- (grid_run is in fact the same wrapper code that processes installed applications)

Syntax:

grid_run *gridOptions* ***userScriptCodeApp*** *userOptions userParameters*

*The location of ***userScriptCodeApp*** must be explicit!

Examples – Bash script

Bash script: test.sh

```
#!/bin/bash
HOST=$(hostname)
echo Running on $HOST at $(date)
echo Weather is $1
```

grid_run --grid_mem=10G ./test.sh fair

```
Running on research01 at Tue 12 Jan 2021 06:57:47 PM EST
Weather is fair
```

*grid_run will attempt to locate the userScript (here: test.sh) using the same rules as the running shell: either via an explicit location or using the PATH. As user scripts are rarely present in the directories included in the PATH (unless the user customized it), it is required that the grid_run invocation contain an **explicit location** for the userScript!! (here: **./** = the local directory). Otherwise, a "file not found" exception will occur.*

Examples – Python script

Python script: test.py

```
#!/apps/anaconda3/bin/python
print('Testing…')
```

```
grid_run  --grid_mem=5G  ./test.py
```

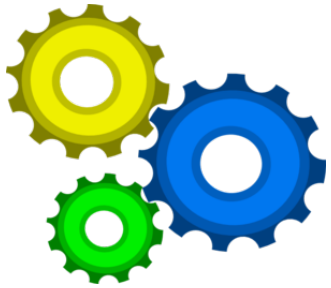
```
Testing...
```

Running production workloads (batch)

Batch computing

- A classic workflow originating in the mainframe era
  - **Submit** a bundle of payloads (**job requests**) and let the system organize (**schedule**) its execution
  - **Unattended execution**
  - **Fully automated**
- Must prepare a script that fully controls each job
- Must include all input parameters, input and output filenames
- All messages produced by the job execution (outputs and errors) will be captured in log files
- The scheduler (SGE) will organize the execution of all submitted jobs, optimizing GRID resource utilization and global response time.

Batch Workflow



Prepare	Submit	Monitor	Inspect	Repeat
Prepare control script(s)	Submit batch job(s)	Monitor progress (or delete jobs)	Inspect logs	(repeat if necessary)

Batch Workflow: #1 Prepare

2 cases:

A. You're using an application (MATLAB, R, Stata, etc.)

B. You're using your own script (Python) or compiled code (C++)

A. Use an application

Prepare an application specific script using the application specific language

B. Use own code (Python or C/C++)

1. Prepare your code in a source code file
2. If C/C++: compile it
3. If Python: add the shebang line indicating the Python interpreter you want to use
4. Make the result of steps 2 or 3 executable (*chmod +x filename*)

Batch Workflow: #2 Submit

Use the same syntax but include the --grid_submit=batch GRID option

- Submit the job to the scheduler by using the GRID option: --grid_submit=batch
  The syntax used so far for running applications or user code remains the same!

Submit an application

- ***applicName*** *--grid_submit=batch otherGridOptions applicOptions applicParams*

Submit own code (Python or C/C++)

- ***grid_run*** *--grid_submit=batch otherGridOptions userScript userScriptOptions userScriptParams*

```
(live)rp2927@researchint01:/shared/share_B9334/Razvan$
(live)rp2927@researchint01:/shared/share_B9334/Razvan$ grid_run  --grid_submit=batch  ./HelloWorld.sh
Your job 6585326 ("HelloWorld.sh") has been submitted
(live)rp2927@researchint01:/shared/share_B9334/Razvan$ qstat
job-ID  prior   name       user         state submit/start at     queue                          slots ja-task-ID
-----------------------------------------------------------------------------------------------------------------
6585326 0.00000 HelloWorld rp2927       qw    01/26/2022 13:50:04                                  1
(live)rp2927@researchint01:/shared/share_B9334/Razvan$
(live)rp2927@researchint01:/shared/share_B9334/Razvan$ qstat
job-ID  prior   name       user         state submit/start at     queue                          slots ja-task-ID
-----------------------------------------------------------------------------------------------------------------
6585326 0.56000 HelloWorld rp2927       r     01/26/2022 13:50:07 debian.q@research117.grid.gsb      1
(live)rp2927@researchint01:/shared/share_B9334/Razvan$
(live)rp2927@researchint01:/shared/share_B9334/Razvan$
(live)rp2927@researchint01:/shared/share_B9334/Razvan$ qstat
(live)rp2927@researchint01:/shared/share_B9334/Razvan$
(live)rp2927@researchint01:/shared/share_B9334/Razvan$
```

Batch Workflow: #3 Monitor

- The scheduler provides specific tools for monitoring and controlling its activity:
  - **Get job status (qstat)**
  - **Delete a job (qdel)**
  - ***Get cluster info (qhost)**
  - (suspend, resume, modify, resubmit, etc. – not used here)
- **When submitted each job receives a jobID**

Batch Workflow Monitoring

- Requesting status (qstat)
  - qstat (list the status of all the jobs *owned by the caller*)
  - qstat -j <jobID> (list the **detailed** status of the job with the provided jobID)
  - qstat -u "*" (list **all** jobs)
- States:
  - r – running
  - qw – queued waiting
  - t – transfering
  - d – deleting
- Find reason for job waiting
  - qstat -j <jobID>

If job is in "qw" state, the reason for waiting will be displayed at the end of a long list of job parameters. Unfortunately, it is presented in in quite a non-obvious way. (Send it to support for interpretation)

Batch Workflow: #3’ Deleting

- Delete a job (**qdel**)
  - qdel <jobID>

  - *qdel -u $(whoami)* *Delete **all** jobs owned by the caller.*
    Use with much care!!

Batch Workflow: #4 Inspect logs

- The logs are created in the **same directory from where the job was submitted!**
- The scheduler will collect all the messages generated by the job in 4 log files (not including data/output files created by your code):
  - *<jobName>.o<jobID>* *stdout (standard output)* *Regular program output*
  - *<jobName>.e<jobID>* *stderr (standard error)* *Error messages*
  - *<jobName>.po<jobID>* *Empty*
  - *<jobName>.pe<jobID>* *Empty*
- Find *jobName* and *jobID* in the ***qstat*** output
- The logs are essential for understanding what happened to the job. **Always check them!** Support will always ask for them, as well.

Batch Workflow: Log files examples

```
[rp2927@researchgrid B9334]$
[rp2927@researchgrid B9334]$ cat HelloWorld.sh
#!/bin/bash
echo "Running on node: $(hostname)"
echo "Working directory: $(pwd)"
echo "Is there anybody out there?..."
echo -e "Here are the task control vars:\n$(env | grep SGE_TASK)"
mistaken_command
exit
[rp2927@researchgrid B9334]$
```

```
[rp2927@researchgrid B9334]$ qstat
[rp2927@researchgrid B9334]$
[rp2927@researchgrid B9334]$ ls -lrt
total 32
-rwxrwxr-x 1 rp2927 578103723 207 Feb  7 10:26 HelloWorld.sh
-rw-r--r-- 1 rp2927 578103723   0 Feb  7 10:26 bash-bat.po4949853
-rw-r--r-- 1 rp2927 578103723   0 Feb  7 10:26 bash-bat.pe4949853
-rw-r--r-- 1 rp2927 578103723 234 Feb  7 10:26 bash-bat.o4949853
-rw-r--r-- 1 rp2927 578103723  61 Feb  7 10:26 bash-bat.e4949853
[rp2927@researchgrid B9334]$
[rp2927@researchgrid B9334]$ cat bash-bat.o4949853
Running on node: research48
Working directory: /user/user1/rp2927/B9334
Is there anybody out there?...
Here are the task control vars:
SGE_TASK_STEPSIZE=undefined
SGE_TASK_LAST=undefined
SGE_TASK_FIRST=undefined
SGE_TASK_ID=undefined
[rp2927@researchgrid B9334]$
[rp2927@researchgrid B9334]$ cat bash-bat.e4949853
./HelloWorld.sh: line 6: mistaken_command: command not found
[rp2927@researchgrid B9334]$
```

Batch Resource Limits

- Different behavior of interactive and batch execution modes with regard to momentary resource availability. If the requested resources are not available:
  - Interactive jobs will timeout after 5 sec, returning an error
  - Batch jobs will wait in queue (indefinitely)
- Regular user limits (under periodic admin adjustment)
  - 1000GB RAM
  - 500cores@regular ;; 100cores@largeMem(50GB) ;; 100cores@longJobs
  - Runtime(wallclock): 480hr@regular,largeMem ;; 100days@longJobs
- --grid_mem
  - Insufficient mem allocation will get your job killed!
  - Excessive mem allocation will waste the resource creating shortages for your colleagues
  - Increase the request in small increments. Default 4GB, although certain programs have a higher default
  - For multi-CPU jobs, the request above is the TOTAL mem allocation (not per CPU)
- --grid_ncpus
  - Ncpu is limited by the number of cores present on an actual worker node*
  - Using more threads than the number of allocated cores will decrease the performance.

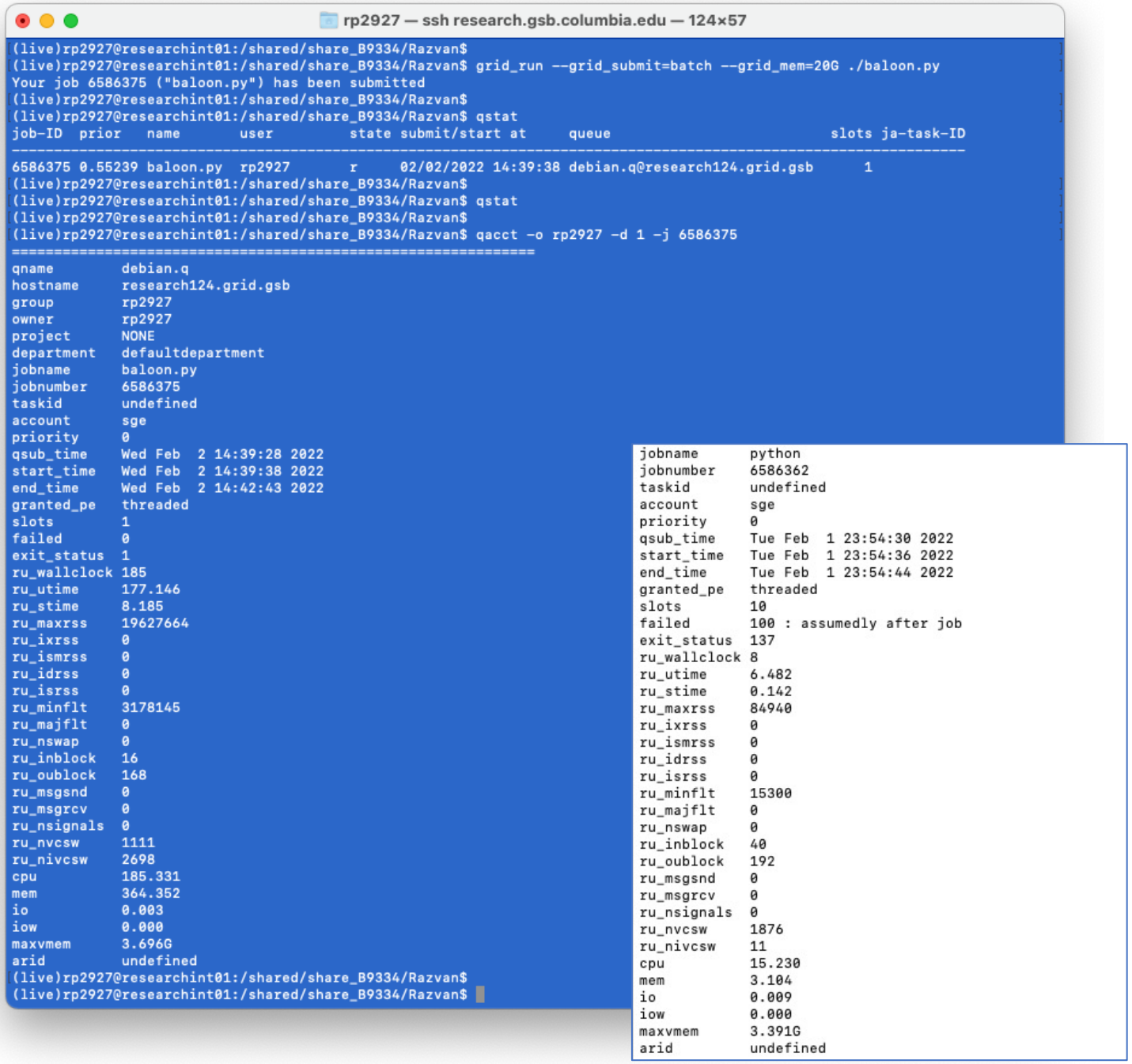
**The number of cores per server varies.*

Deeper down the rabbit hole…

My job had disappeared with empty or absent logs, and no other trace…!

>>> Most probably the job was killed by the scheduler for insufficient memory allocation, but can we confirm?...

1. *(If you can, find the (7digit) job ID from the submit command reply, from an email notification or from the job's log files names – it is the number appended to the log file name following the type suffix {o, e, po, pe}, and use it below as argument to the "-j" option)*
2. Use "**qacct -o *<UNI>* -d *<N>* -j**" to display your recent accounting records.
   - *UNI* is your account, while the "-d <N>" option limits the search to the last N days.
   - Attention: *use a reasonable N interval, just long enough to capture the time of the event* and do not omit it, as the accounting log is huge going back a few years. Also, do not omit your UNI or you'll end up with everyone's records.
3. Use job parameters like qsub/start/end time and *jobname* to identify the record of the job you are interested in.
4. Look at the fields "failed" and "exit_status":
   1. A job killed by the scheduler (for mem overreach) will have an **exit status** of **137** and **failed** code of **100**
      1. Failed code 100 = "Assumedly after job" (job ran, killed by signal)
      2. Exit code 137 = (128 + kill signal) => kill -9 (SIGKILL)
   2. A properly completed job will have both "failed" and "exit_status" codes equal to 0. (If the job terminates abnormally due to some program error, the exit status might be different (e.g. 1), but the "failed" code will remain 0.



More about the accounting record…

- Jobname
- Timestamps
- No. slots
- (!) Exit codes (Failed, Exit_status)
- (!) Resource Usage:
  - ru_wallclock (sec)
  - **ru_maxrss** (max resident set size in kB)
- ** Ignore maxvmem (!) ☹

Selecting the right amount of memory

- Unfortunately, it is not a straightforward task…
- It depends on the number and (size) types of objects your code had created and handles at any given time.
- The size of your input data provides a hint, but what it's done with the inputs matters even more.
- It varies during the program's execution! (thus, the job might start and be running for awhile, but be killed later into the run)
- We need to size the job submit grid_mem option using the MAX value of the "virtual memory" (VmPeak)
- Given the complexity of estimating beforehand the VmPeak, we recommend a trial-and-error approach, followed if all possible by a downsizing based on post-run information.
    1. Start with an educated guess and increase the value of grid_mem option until the job runs successfully. Use reasonably small increments (10-20%?)
    2. Once the job ran successfully read the value of VmPeak from /proc/$PPID/status to find the maximum size of the virtual memory attained during the run and scale grid_mem down to it.

       import os
       os.system('grep VmPeak /proc/$PPID/status')
- Make all efforts to not waste system memory – it's a shared resource!

Batch Workflow: Large scale production

Loosely coupled parallel jobs

What if:

"I have hundreds of input files to be analyzed with the same algorithm"	"I need to run the same analysis with hundreds of parameter values"	"I need to run a distributed optimization scanning a parameter space with hundreds of points"	"I just sent out 100s of jobs and I discovered a mistake. How do I delete them?"

Array jobs: bundle a set of jobs (tasks) and treat it as a single processing job (create, delete, change, etc.)

SGE Array Jobs

- A set of tasks, handled unitarily, accessing an environment variable iterator
- Iterator takes integer values between two limits, with given step
- (can limit the number of concurrently running jobs)
- Create with GRID option:
  - --grid_array=<first>-<last>[:<step>][/<conc>]
  - --grid_SGE_TASK_ID= =<first>-<last>
- Three env variables are available to each task:
  - SGE_TASK_ID *Takes integer values from <first> to <last>, by step <step>*
  - SGE_TASK_FIRST *<first>*
  - SGE_TASK_LAST *<last>*

Array Jobs -- Example

Submit an array job using one version of the required Grid option.
Observe the job array index in the last column of qstat

```
[rp2927@researchgrid B9334]$
[rp2927@researchgrid B9334]$ sge_run                --grid_submit=batch --grid_SGE_TASK_ID=1-5  ./HelloWorld.sh
Your job-array 4949855.1-5:1 ("bash-bat") has been submitted
[rp2927@researchgrid B9334]$ qstat
job-ID  prior   name       user         state submit/start at     queue                          slots ja-task-ID
-----------------------------------------------------------------------------------------------------------------
4949855 0.50236 bash-bat   rp2927       r     02/07/2019 10:56:50 B9334@research47.gsb.columbia.     1 1
4949855 0.50118 bash-bat   rp2927       t     02/07/2019 10:56:51 B9334@research47.gsb.columbia.     1 2
4949855 0.50079 bash-bat   rp2927       t     02/07/2019 10:56:51 B9334@research47.gsb.columbia.     1 3
4949855 0.50059 bash-bat   rp2927       t     02/07/2019 10:56:51 B9334@research47.gsb.columbia.     1 4
4949855 0.50047 bash-bat   rp2927       t     02/07/2019 10:56:51 B9334@research47.gsb.columbia.     1 5
[rp2927@researchgrid B9334]$ qstat
[rp2927@researchgrid B9334]$
[rp2927@researchgrid B9334]$
```

Array Jobs -- Logs

The array jobs will generate one set of 4 logs per task

```
[rp2927@researchgrid B9334]$ ls -lrt
total 176
-rwxrwxr-x 1 rp2927 578103723 191 Feb  7 10:51 HelloWorld.sh
-rw-r--r-- 1 rp2927 578103723   0 Feb  7 10:53 bash-bat.po4949855.1
-rw-r--r-- 1 rp2927 578103723   0 Feb  7 10:53 bash-bat.pe4949855.1
-rw-r--r-- 1 rp2927 578103723   0 Feb  7 10:53 bash-bat.po4949855.2
-rw-r--r-- 1 rp2927 578103723   0 Feb  7 10:53 bash-bat.pe4949855.2
-rw-r--r-- 1 rp2927 578103723   0 Feb  7 10:53 bash-bat.po4949855.3
-rw-r--r-- 1 rp2927 578103723   0 Feb  7 10:53 bash-bat.pe4949855.3
-rw-r--r-- 1 rp2927 578103723   0 Feb  7 10:53 bash-bat.e4949855.1
-rw-r--r-- 1 rp2927 578103723   0 Feb  7 10:53 bash-bat.po4949855.4
-rw-r--r-- 1 rp2927 578103723   0 Feb  7 10:53 bash-bat.pe4949855.4
-rw-r--r-- 1 rp2927 578103723   0 Feb  7 10:53 bash-bat.e4949855.2
-rw-r--r-- 1 rp2927 578103723 202 Feb  7 10:53 bash-bat.o4949855.1
-rw-r--r-- 1 rp2927 578103723   0 Feb  7 10:53 bash-bat.po4949855.5
-rw-r--r-- 1 rp2927 578103723 202 Feb  7 10:53 bash-bat.o4949855.2
-rw-r--r-- 1 rp2927 578103723   0 Feb  7 10:53 bash-bat.pe4949855.5
-rw-r--r-- 1 rp2927 578103723   0 Feb  7 10:53 bash-bat.e4949855.3
-rw-r--r-- 1 rp2927 578103723 202 Feb  7 10:53 bash-bat.o4949855.3
-rw-r--r-- 1 rp2927 578103723   0 Feb  7 10:53 bash-bat.e4949855.4
-rw-r--r-- 1 rp2927 578103723 202 Feb  7 10:53 bash-bat.o4949855.4
-rw-r--r-- 1 rp2927 578103723   0 Feb  7 10:53 bash-bat.e4949855.5
-rw-r--r-- 1 rp2927 578103723 202 Feb  7 10:53 bash-bat.o4949855.5
[rp2927@researchgrid B9334]$
```

Task index

```
[rp2927@researchgrid B9334]$
[rp2927@researchgrid B9334]$ cat bash-bat.o4949855.2
Running on node: research47
Working directory: /user/user1/rp2927/B9334
Is there anybody out there?...
Here are the task control vars:
SGE_TASK_STEPSIZE=1
SGE_TASK_LAST=5
SGE_TASK_FIRST=1
SGE_TASK_ID=2
[rp2927@researchgrid B9334]$
[rp2927@researchgrid B9334]$
```

SGE Array Jobs – Timing and synchronization

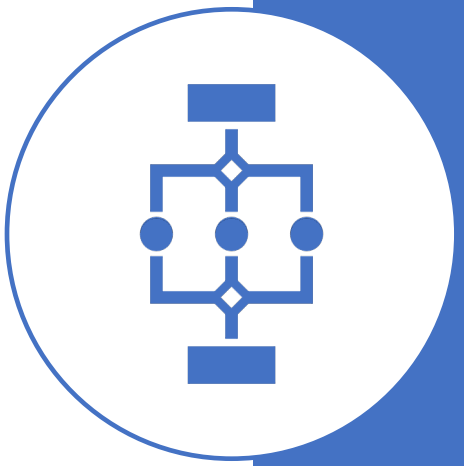
The array tasks will be started in sequence: <first>/1, 2, 3 … <last>

However, task completion time will depend not only on the computational effort required by each parameter set, but also on the machine speed and the average load of the worker where they will be instantiated. (True for both heterogeneous and homogeneous clusters)

Consequently, the task termination time is unpredictable, and the array tasks completion times will be **out of sequence!**

SGE Job Synchronization

- Schedule a parallel computation workflow composed of:
  1. A preparer job (e.g. split input data in equally sized sub-inputs)
  2. An array job (main workload: process the sub-inputs in parallel)
  3. A finisher job (e.g. summarize output of individual tasks from the array job)
- It is essential that each step does not start until the previous one completes!
  - Use GRID "hold" option: "--grid_hold=<jobID>", where *<jobID>* is the jobId of the ***precursor*** job.
    - ***grid_run*** *<gridOptions> <scriptName> <scriptParams>*
    - E.g. ***grid_run*** *--grid_submit=batch --grid_mem=25G --grid_hold=615237 ./myScript.sh*



Python (Anaconda)

Development with Python Code

Caution!

- Several Python interpreters are installed.
  - The **system default** (/usr/bin/python) – used by the OS. Older version without data science packages. DO NOT USE!
  - The Anaconda system installation (/apps/anaconda3/bin/python). Great choice!
  - Private environment. Most flexible choice! *(and most complex)*

- Preconfigured commands invoking Anaconda Python
  - anapy3
  - ipython3
  - spyder
  - notebook
  - jupyterlab

Obs.: All the commands above accept GRID options. E.g.:

- *anapy3 --grid_mem=50G*

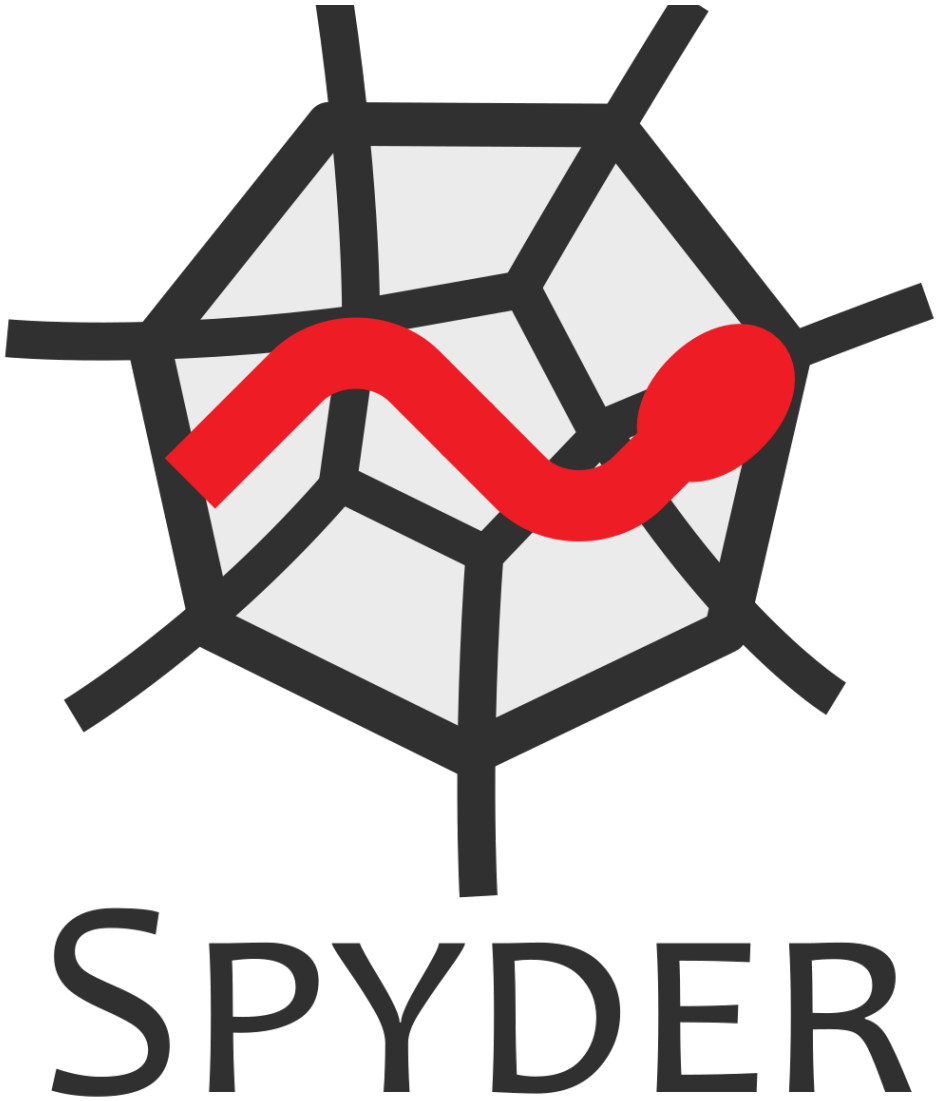
- The *grid_run* general handler using a script that includes a (shebang) properly defined interpreter

Python Interpreters and IDEs

- Command line interpreter:
  - *anapy3*
- IPython:
  - *ipython3*
- Spyder (requires graphic support):
  - *spyder*

Obs.: All the commands above accept GRID options. E.g.:

- *spyder --grid_mem=50G*



Jupyter Notebook / Lab

(https://wiki.gsb.columbia.edu/research/index.php/Jupyter_Notebook/Lab)

- Project Jupyter Notebook and Lab interfaces are available on the CBS cluster, using a **server/client model** where the computation is performed on a Grid node and the presentation is done by a web browser running on the local system. Consequently, the startup requires two steps.

1. Launch, typing "***notebook***" or "***jupyterlab***"
2. Connect the local web browser to the Jupyter server:
   - Copy the *URL containing "gsb.columbia.edu"* displayed at step 1, and paste it into a *local web browser*.
   - DO NOT USE Firefox from the virtual desktop!!

Obs.: All the commands above accept GRID options. E.g.:

- *notebook --grid_mem=50G*

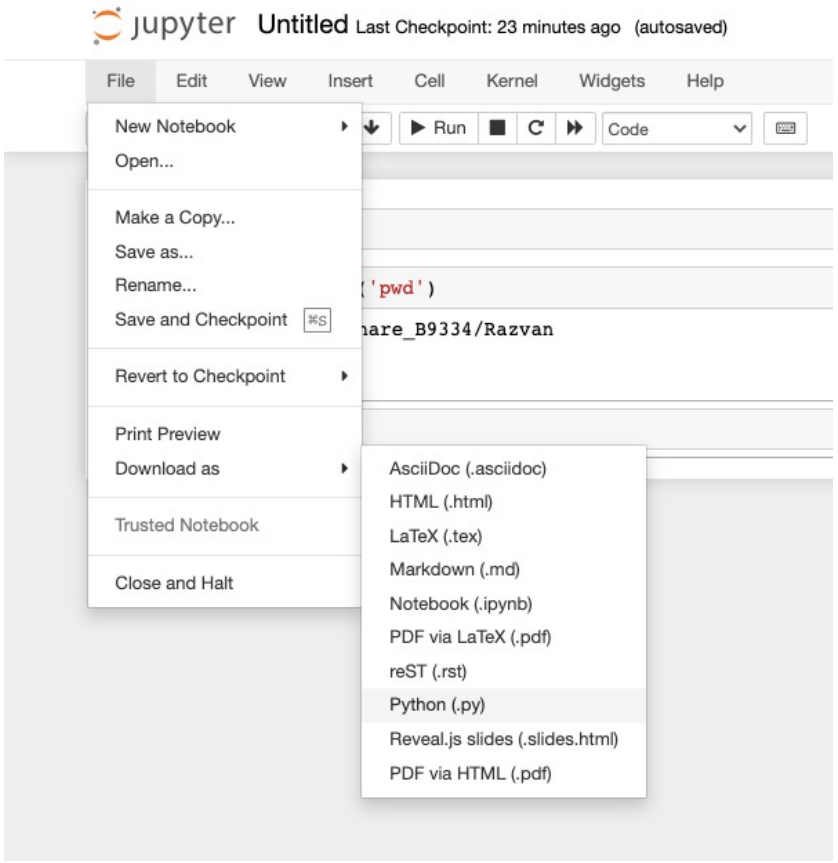
From Notebook development to production

Develop the prototype of a new analysis in a Jupyter Notebook and then run the result as a production (batch) job.

1. Develop the code within a Jupyter notebook using the launch method mentioned before ("notebook…grid_options…"). Request an appropriate amount of resources (CPU, RAM) using GRID options on the "notebook" command.
2. Export the notebook as a Python (.py) script (Menu: File > Download as > Python (.py))
3. Transfer the saved script to the GRID
4. Edit the script's shebang (!) changing:
   from "#!/usr/bin/env python"
   to "#!/apps/bin/anaconda3/bin/python"
5. Run/submit the script using *grid_run* (--grid_submit=batch)

Method 2 (simpler conversion):

1. Run "ipython3 nbconvert --to python <notebookName>"
2. Perform steps 4 and 5 as above.



Running Python code (non-interactive/batch mode)

- Prefix the file containing the Python code with the (shebang) line corresponding to **the Python interpreter you want to use**:
  - #!/apps/anaconda3/bin/python *(avoid #!/bin/env python)*
- Run with "grid_run":
  - ***grid_run*** *--grid_mem=20G ./myScript.py*
- *Would this work?*
  - *anapy3 --grid_mem=20G ./myScript.py*
- *What is the difference between the two invocations?*

Python modules and environments

Python distributions (Anaconda)

- Baseline: Anaconda Python
  - Distribution of Python optimized for scientific computing
    - Python interpreter
    - Set of (~350) packages focused on data science
    - Package manager (conda)
  - Centrally installed on the GRID@CBS cluster
    - *Unique copy* maintained by ResSupport -- *upgraded periodically*
    - Follows the tested releases by Anaconda (bi-annualy)
      - Changes (bi-annually)
      - Is restricted to the content of the tested bundles (meta-packages) released by Anaconda

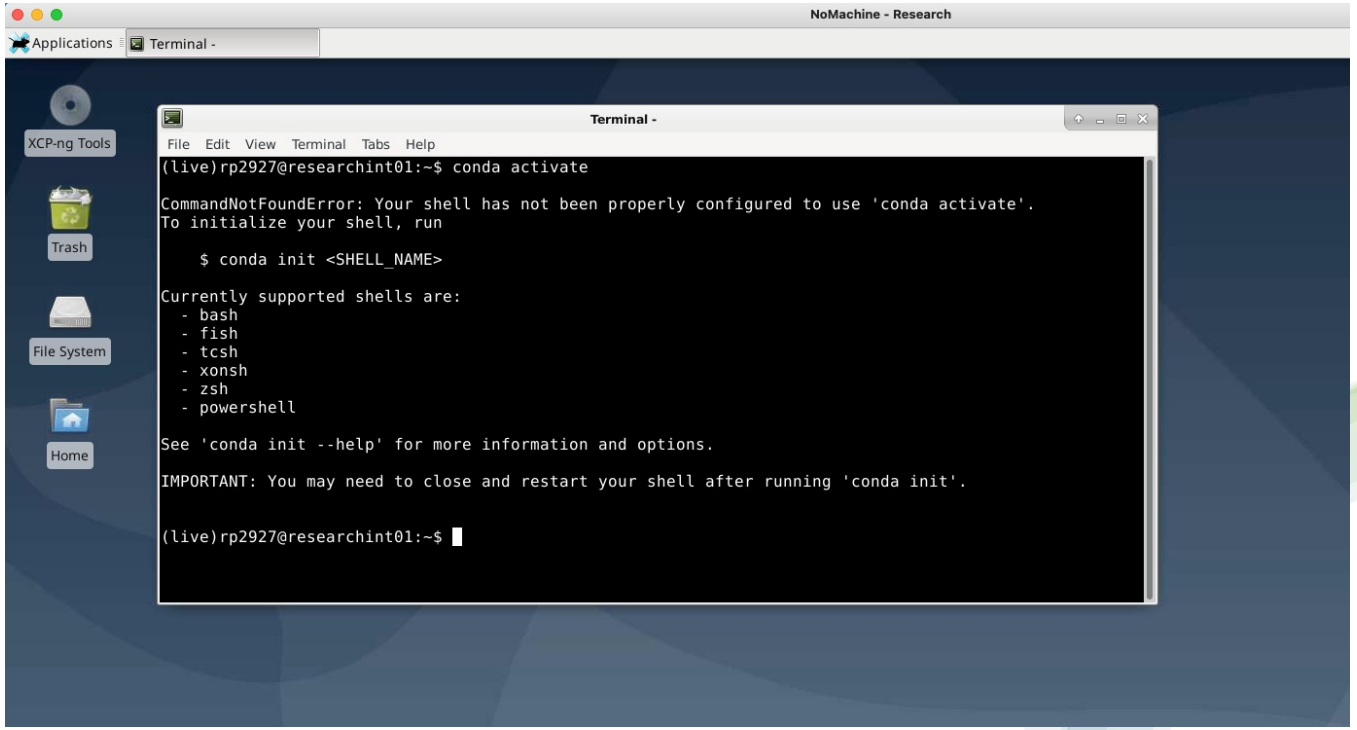
Caution: Variability! The list of installed packages (and their corresponding versions) will change. Otherwise stable code might suddenly stop working.

Conda – Anaconda package manager

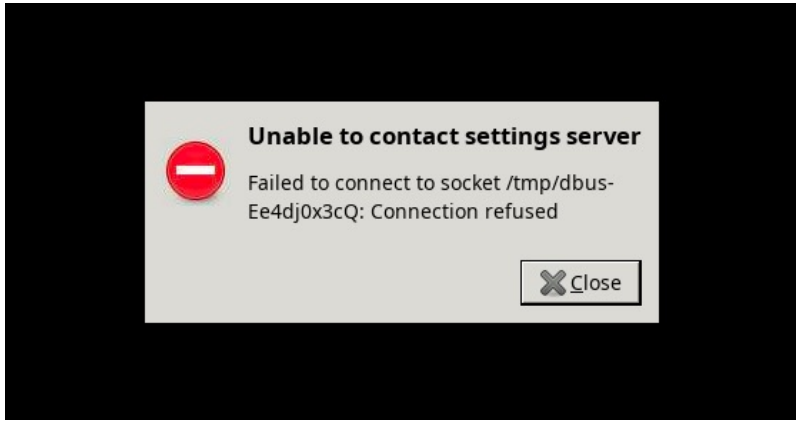
- Versatile package manager created by the Anaconda project
  - Used to manage Python packages and environments
    - Install Python packages, managing channels, resolving dependencies and conflicts
      - ***conda*** *install <packageName>[=<version>]*
    - Create and manage private environments
      - ***conda*** *create –n <envName> <packageName>[=<version>]*
      - *E.g. conda create –n myNewEnv python=3.7 keras*
    - Activate/deactivate Python environments
      - ***conda*** *activate <envName>*
      - *E.g. conda activate myNewEnv*
      - *E.g. conda activate* — Activates the “base” environment (the system Anaconda install)
    - Inspect state
      - ***conda*** *info* — Displays current state (configuration) – includes current env.
      - ***conda*** *list* — Displays the list of packages from the current environment

Anaconda activation and the Nomachine virtual desktop

Anaconda (manual) activation (“conda activate”) *within the virtual desktop* fails and requires to setup automatic activation (“conda init”).



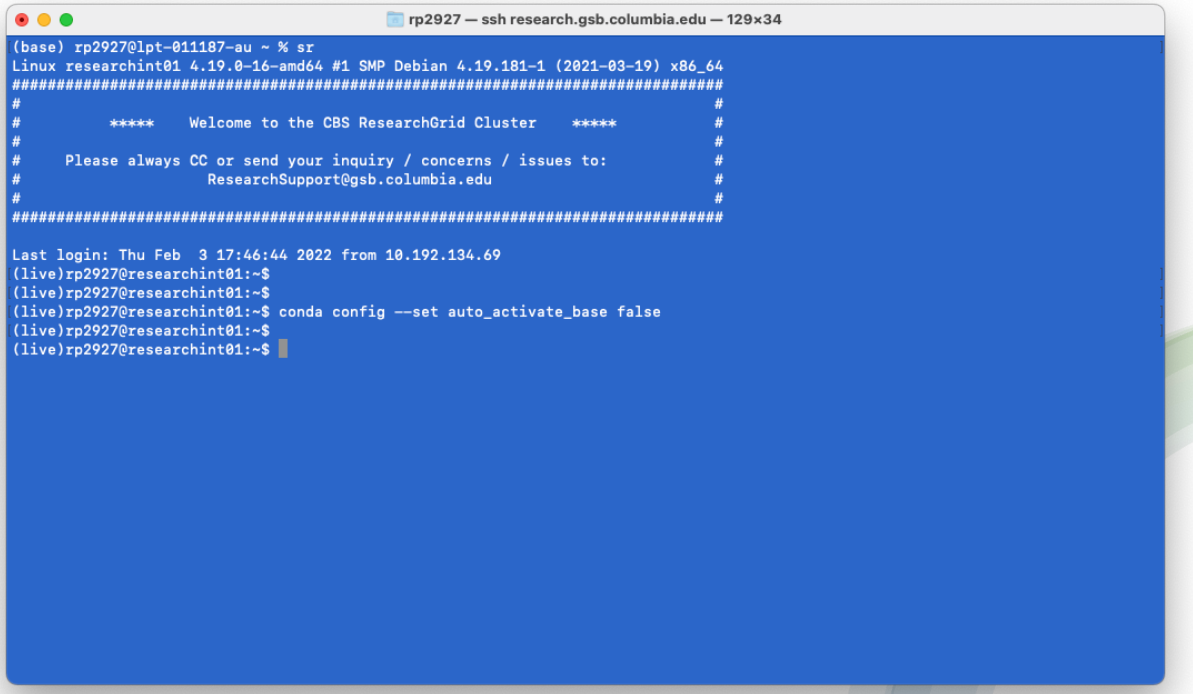
Anaconda activation and the Nomachine virtual desktop



- Unfortunately, Anaconda automatic activation breaks the Nomachine virtual desktop.

Anaconda activation and the Nomachine virtual desktop

- Compromise: let conda setup the shell but disable automatic base environment activation
  - Use a floating terminal or ssh via PuTTY or MacTerminal!...
  - **conda config --set auto_activate_base false**
- *Preferably: perform this additional step immediately after running "conda init", to avoid being locked-out from the virtual desktop in the first place!*
- **Will need to activate the base env manually, when needed!**



Python workflow

A. Start coding using the system provided Anaconda distribution
- Use GRID predefined commands to invoke the interpreter/IDE interactively (anapy3, spyder, notebook)
- Or run your code unattended using ***grid_run*** and scripts that employ the shebang:
  - **#!/apps/anaconda3/bin/python**

B. Problem 1: An import statement fails; I need a module that is not available. How do I add a Python package that is not provided by the system Anaconda?

1) Use the *conda* package manager to install it?!?
   - No, conda won't be able to write in the (protected) system Anaconda location! An explicit error will be thrown.
2) We need a different installer that can be steered to install in a personal location: ***pip***
3) SOLUTION:
   i. First, run "*conda activate*" to activate the base (system) Anaconda and establish the context that ***pip*** will use to resolve dependencies and conflicts. (Also, this step will make the ***pip*** command available)
   ii. Then, install the required packages: "***pip*** *install <packageName>[==<version>]* "

Obs.1: By default, pip will install new packages in ***~/.local/lib/pythonX.X/site-packages*** *which is a location part of the default Python search path (sys.path). Therefore no special action is required to help Python find the new additions. However, if this fails, or if you install in a different location, use the PYTHONPATH variable to include the installation location in the Python search path (add to ~/.bashrc):*

- *export PYTHONPATH=/your/package/installation/location*

Obs.2: ***The local package library is Python version specific, therefore when a new interpreter version is deployed all local packages need to be reinstalled****.*

Python virtual environments

- Python (virtual) environment: a self-contained **context** providing a working software infrastructure that includes interpreter, libraries, scripts and tools, in support of Python software development.
- Purpose?!
  - Provide **multiple**, **self-contained**, **stable** Python development contexts
  - Reliability: remove dependency on shared elements that can change outside user control
  - Maintainability: select compatible, known-to-work, infrastructure components (packages) and supply a reliable method to control them, including the ability to replicate contexts (portability!)
- Manage via:
  - Conda {create, install/upgrade, activate/deactivate}
  - (… Implemented using env variables (PATH))

Python workflow

Problem 2: *"Everything was working fine before, but now my code is crashing!"*

- Changes made to the system Anaconda packages won't update your personal repos!
  - Unidirectional dependencies. Disjoint validation domains.
  - Potential for breaking user code! (introducing regression)

1) Must decouple from the system Anaconda. **Create a private virtual environment**.
   - ***conda*** *create –n <myNewEnv> [<packageName>[=<version>]]*
2) SOLUTION:
   i. First, create a new environment using "***conda*** *create –n <myNewEnv> [<packageName>[=<version>]]*"
   ii. Next, activate the new environment: "***conda*** *activate <myNewEnv>*"
   iii. Then, install all the required packages: "***conda*** *install <packageName>[=<version>]*"

Python workflow – working with private envs

If a private virtual environment is used, the guidance given so far for working with Anaconda Python must be adapted!

A. Running interactively:
   - The preconfigured commands {*anapy3*, *ipython3*, *spyder*, *notebook*, etc.} won't work!
   - *?... conda activate <myNewEnv>*
   - *?... [i]python*
     - *Caution: the command above starts the interpreter locally, on the interactive node. Do not use for heavy computation, to avoid disrupting your colleagues.*
   - ***!... grid_run can help but must be adapted**:*
     - *E.g.: grid_run <grid_options> /user/UNI/.conda/envs/myNewEnv/bin/spyder*
     - *E.g.: grid_run <grid_options> /user/UNI/.conda/envs/myNewEnv/bin/jupyter notebook --ip=\$(hostname).gsb.columbia.edu*

B. Running code:
   - Replace the shebang line with the location of the python interpreter from your private environment and run with *grid_run*:
     - E.g.: ***#!/user/<myUNI>/.conda/envs/<myNewEnv>/bin/python***

Running R

- Same as with Anaconda Python, there is a (shared) system R installation. The GUI frontend "**Rstudio**" is also available.
- Invoke by simply typing: "***R***" or "***Rstudio***"
- Execute (batch) R code using "**Rscript**":
  - ***Rscript*** *[<gridOptions>] <scriptName> <scriptParams>*
- *… or using grid_run and the* ***#!/apps/bin/Rscript*** *shebang*

Obs.: All the commands above accept GRID options. E.g.:

- *R --grid_mem=50G*

Installing (user) R packages

https://wiki.gsb.columbia.edu/research/index.php/R_(new)

- As the system R installation cannot be altered by users, adding new R packages must be done with the help of a user library.
- The software will automatically detect your intention and prompt you to create a personal lib at the proper location, if it doesn't already exist.

Installing R packages in user libraries [edit]

It is frequently much more convenient to install R packages in a personal library rather than ask the system administrators for a system-wide installation. In addition, a system installation will certainly change, updating various packages based on requests from other members of the community, while a personal/user library will always be in a known state. The downside of the latter is that a certain amount of maintenance is required.

In order to find packages, R looks at the variables R_LIBS_USER and R_LIBS_SITE. R_LIBS_USER is by default set to:

```
R_LIBS_USER=${R_LIBS_USER-'~/R/x86_64-pc-linux-gnu-library/3.6'}
```

Therefore the default location of a user library is:

```
/user/<userName>/R/x86_64-pc-linux-gnu-library/3.6
```

You can always check the library search list, within an R session, using:

```
> .libPaths()
```

A typical answer would be:

```
> .libPaths()
[1] "/user/<yourUserNameHere>/R/x86_64-pc-linux-gnu-library/3.6"
[2] "/apps/R-3.6.0/lib64/R/library"
```

If you've never installed any local package the first path won't be listed, as it was never created. So let's install a package in the user library.

Simply open an R session and type:

```
>   install.packages("packageName")
```

As you don't have write access to the system library, you'll be asked whether you want to install it in a personal library.

```
> install.packages("packageName")
Warning in install.packages("packageName") :
  'lib = "/apps/R-3.6.0/lib64/R/library"' is not writable
Would you like to use a personal library instead? (yes/No/cancel) y
Would you like to create a personal library
'~/R/x86_64-pc-linux-gnu-library/3.6'
to install packages into? (yes/No/cancel) y
--- Please select a CRAN mirror for use in this session ---
Secure CRAN mirrors
[...]
```

That's it. Next time you'll need to install other packages the initial dialog won't exist as the system will detect your personal library and will use that default target:

```
> install.packages("stringi")
Installing package into '/user/<userName>/R/x86_64-pc-linux-gnu-library/3.6'
(as 'lib' is unspecified)
trying URL 'https://cloud.r-project.org/src/contrib/....[...]
```

Obs. Keep in mind that the packages will be built for the R version you're running when installing them (here R 3.6 from the new cluster). If you're using multiple R versions certain packages, built for other versions, might fail. To avoid that outcome the personal library default location is version specific (observe the "3.6" in its path) and therefore multiple libraries will have to be maintained.

BASH

THE BOURNE-AGAIN SHELL

```
#!/bin/bash

# Sample script that de-serializes a given iterator, here SGE_TASK_ID
# to generate a parameter space scan, using a set of parameter vectors hardcoded below.
# Extend this example with as many vectors as you need, by folowing the in-line instructions.
# (SGE_TASK_ID should run from 1 to the volume of the parameter space: product of dimensions)
# (R.P.  Jan 2021)


# Define your vectors of parameters, starting with the one that cycles the fastest
A=(1 2 3 4)
B=(i j k l m n)
C=(foo bar baz)


# Calculate the lengths of the vectors
L_A=${#A[@]}
L_B=${#B[@]}
L_C=${#C[@]}
# Add vectors using the syntax:  L_V=${#V[@]} , replacing V with the name of the vector


# Decompose the given iterator value to obtain the indices of the vectors. Lowercase variables are *indices* for the corresponding vectors.
x=$SGE_TASK_ID
x=$((x-1))

a=$((x%L_A))
x=$((x/L_A))

b=$((x%L_B))
x=$((x/L_B))

c=$((x%L_C))
x=$((x/L_C))
# Add similar calculations for additional vectors. Replace V with the name of the respective vector, and repeat as necessary.
#   v=$((x%L_V))
#   x=$((x/L_V))


# (optional) Print the values of the indices for verification
echo $a " " $b " " $c

# (optional) Print the values of the parameters
echo "A = ${A[a]} ;  B = ${B[b]} ;  C = ${C[c]}"

# (optional) Execute user code with given parameter set
yourCodeOrScriptName  ${A[a]}  ${B[b]}  ${C[c]}
```

Example of job control script – parameter space scan

Support

researchsupport@gsb.columbia.edu