TUC Labs

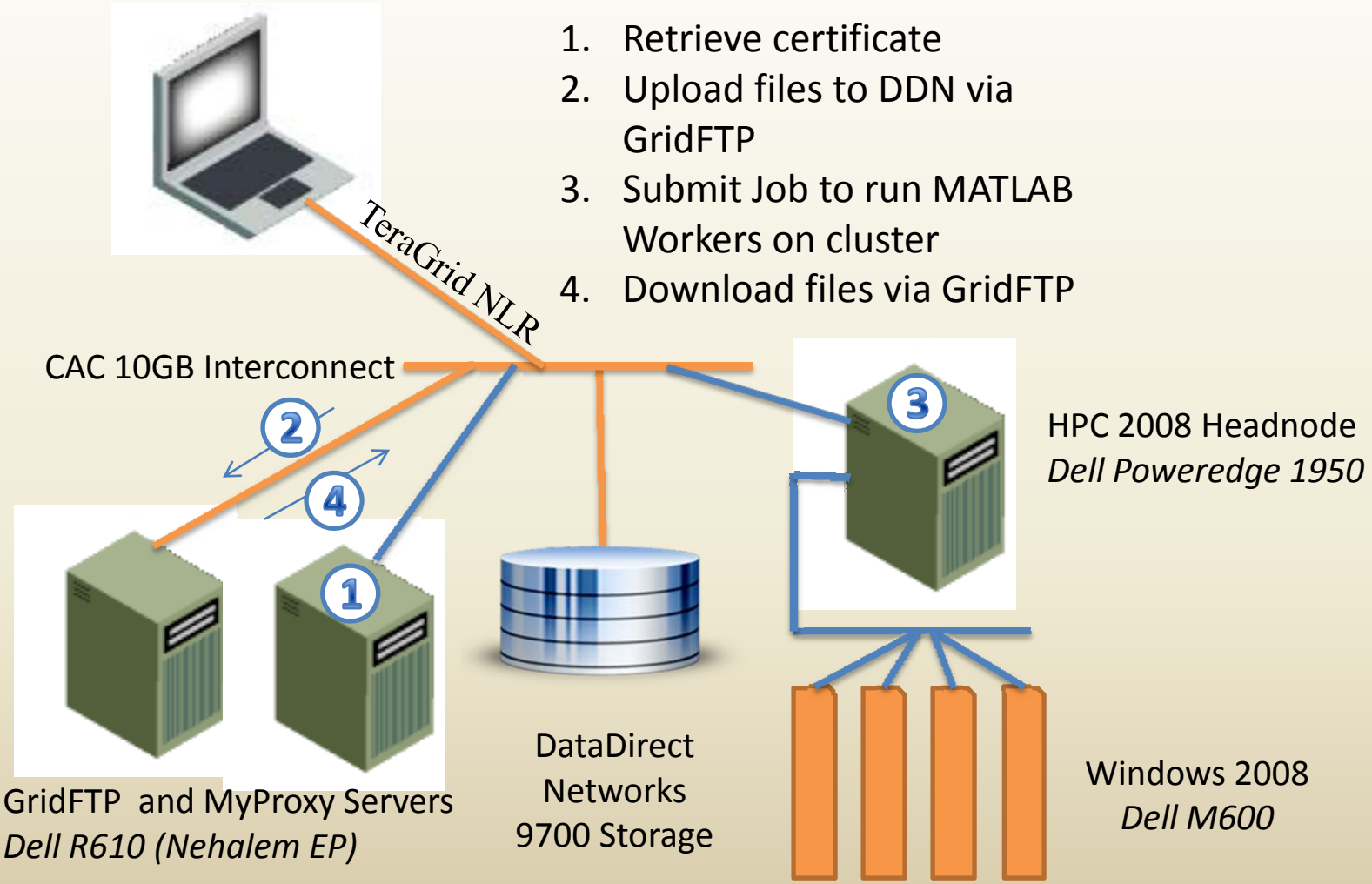
- Files for the labs are located at:

  http://www.cac.cornell.edu/matlab/downloads/labs.zip

- Solutions are provided in the solutions directory if you need help or would like to see how we solved a particular problem.
- Labs:
  - File Transfer – Learn about file and path dependencies as well as how to use the gridFTP object
  - MatlabPoolJob – Practice running a remote pool on TUC
  - Debugging – Practice solving problems using debugging strategies
  - Mex (Optional) – Learn how to mex C and C++ files on TUC.

File Transfer Lab

1. Retrieve certificate
2. Upload files to DDN via GridFTP
3. Submit Job to run MATLAB Workers on cluster
4. Download files via GridFTP



Why Transfer Files?

- The *submit* operation submits the job to a server where functions must be available.

```
j = createJob(sched);
createTask(j,@rand,1,{3,3});
submit(j);
waitForState(j);
A = getAllOutputArguments(j);
```

- This works as-is because *rand* is a built-in function to MATLAB. It's always in the MATLAB path.

Why Transfer Files?

- As soon as we call a custom function and/or require a data file, we must move that file.

```
j = createJob(sched);
createTask(j,@myfunction,1,{3,3});
submit(j);
waitForState(j);
a = getAllOutputArguments(j);
```

- This will fail as **myfunction.m** isn't going to exist on TUC in the MATLAB path. We must somehow transfer this file and get it added to the path.

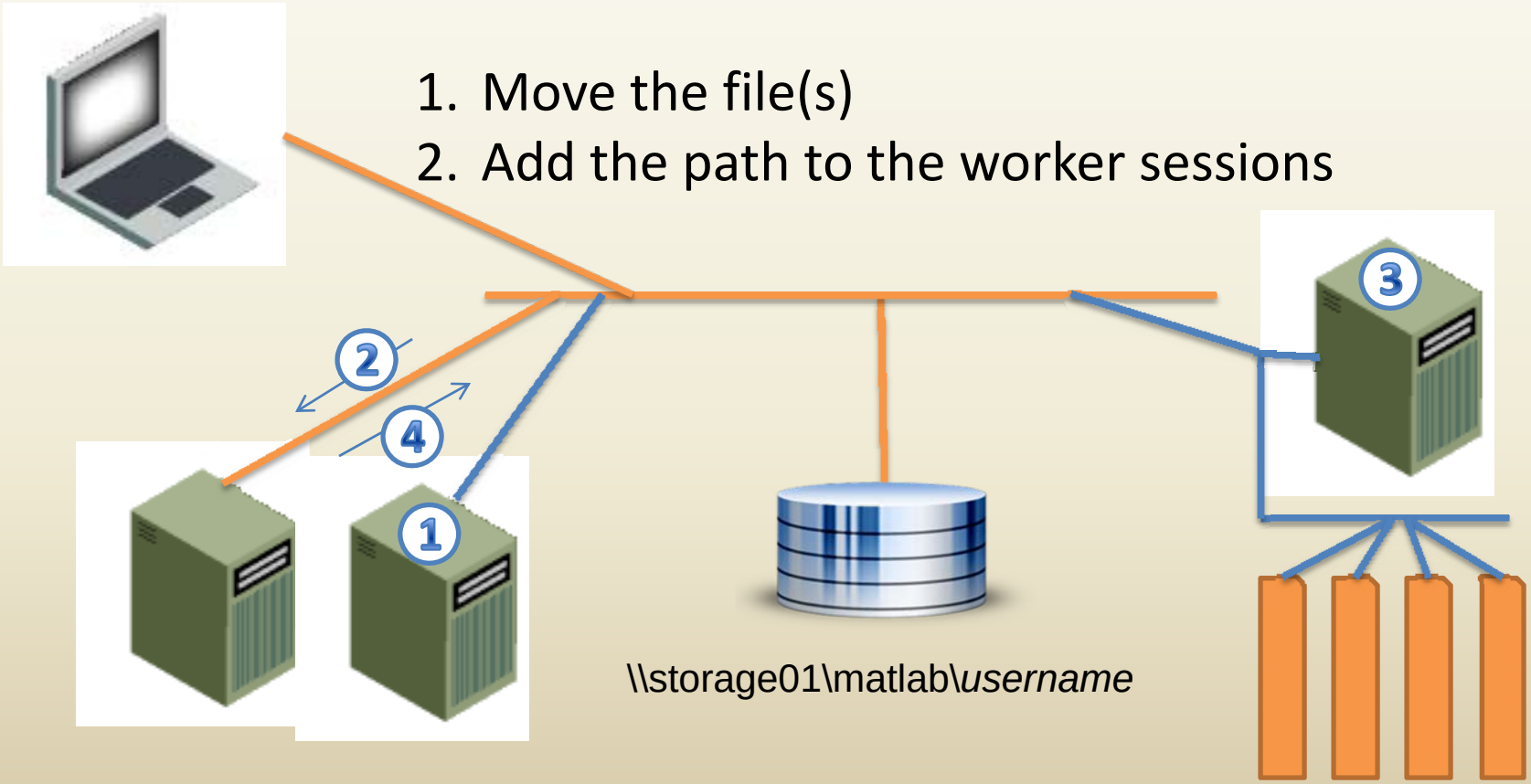
Use FileDependencies

- Simple solution: have MATLAB move files for you using the *FileDependencies* property.
- Specify directories and files the worker will need. All files and directory structure will be copied.
- Best for smaller projects with only a couple of files (file transfer occurs for each worker running a task for that particular job on a machine).

set(j,'FileDependencies',{'/home/src/myfunction.m','/home/data/dfile.mat');

Move the Files Yourself

1. Move the file(s)
2. Add the path to the worker sessions



1. Move the Files

- First move the file(s) needed by the job:

  sendFileToCAC('filename.m');

- sendFileToCAC('filename') – super simple method for dumping a single file into your home directory on TUC.
- sendDirToCAC('mydir','tucDir') – Recursively move a directory and its contents to TUC.

2. Add the path

- On your laptop/workstation, you commonly issue addpath('path/to/file') statements.
- The same is true for the MATLAB on TUC, but with some caveats and there is some slightly different syntax to accomplish the same thing.
- The task function must be on the startup path of MATLAB. You may enter addpath and cd statements into your task function, but first your function must be available.
- We will use *PathDependencies* to make our task function available.

PathDependencies

- *PathDependencies* is a property of the Job object that allows you to issue addpath statements on TUC before calling your task. Specify the path dependency in your job submission script:

```
set(j,'PathDependencies',{'\\storage01\matlab\username'});
```

- This sets a property on the job, the property can only be set once. The value is a cell array and can contain as many entries as you like, but all entries should be absolute paths.

```
paths = cell(1,2);
paths{1} = ‘’\\storage01\matlab\username\src\module1’;
paths{2} = ‘\\storage01\matlab\username\src\module2’;
set(j,'PathDependencies',paths);
```

PathDependencies

```
function [j,a] = addpath_submit(sched)
%This is a simple distributed job submission example.

sendFileToCAC('addpath_remote.m');

j = createJob(sched);

createTask(j,@addpath_remote,1,{5});
set(j,'PathDependencies',{'\\storage01\matlab\naw47'});
submit(j);

waitForState(j);

a = getAllOutputArguments(j);
```

Scripted Solution

- Both send*ToCAC methods are primarily for one time use, best for moving a big directory of data files, testing, or copying a single file.
- Otherwise, you should use the more flexible gridFTP interface. This allows you to interactively move files to TUC as well as write scripts that move files.
- For projects that involve more than one or two source files, we recommend writing a “prep” function which ensures that the most up-to-date functions are available on TUC.

Example PrepFunction

```
function out = orno_prep(cleanup)

if nargin == 0 || cleanup == 0
    out = 1;
    ftp=gridFTP();
    %This will print an error on subsequent runs
    ftp.createDir('SoundXcorrToolv2.1');
    ftp.put('SoundCorrParallel.m',['SoundXcorrToolv2.1/' 'SoundCorrParallel.m']);
    files = dir('Sound Xcorr Tool v2.1\private');
    for i =3:length(files)
        %just cheat and skip entries 1 and 2 which are . and ..
        fn = files(i).name;
        ftp.put(fullfile('Sound Xcorr Tool v2.1\private',fn), ['SoundXcorrToolv2.1/',fn]);
    end
    ftp.close();
else
    ftp=gridFTP();
    ftp.deleteeverything('SoundXcorrToolv2.1');
    ftp.close();
end
```

Then add a pathDependency to the job in the submission script:

```
set(j,'PathDependencies',{[userHome '\SoundXcorrToolv2.1']});
```

Lab

Addpath_submit.m is a simple batch script for submitting a job with a custom function. The task function addpath_remote.m requires 2 source files (calcLatLongDistance.m and degrees2Radians.m) and a data file (airports_boardings.txt).

1) Use FileDependencies to modify the job script to upload the source and data files to storage01 so the function will complete successfully.
2) Write an addpath_prep() function that uses gridFTP to upload the needed files to your directory on storage01. Add *PathDependencies* to the job to allow the function to complete successfully.

MATLAB Pool Lab

MATLAB Pools

MATLAB pools offer a friendly way to parallelize code using OpenMP-style parallelization. Once the pool is started parallel code is identified by two keywords.

parfor – A parallel-for loop where each computation is independent from all of the others

spmd – Single Process Multiple Data, for working with (typically large) composite data objects

MATLAB Pools

Typical pool code that will run on your local machine:

```
matlabpool local 4

%Read the file
load(fname);
%fname contains a string array called IDs which contains names of all the
%spectra contained.  We use that to load everything.  The file format is a
%bit arcane, so there's a bunch of eval nonsense to load things.
[num_spectra,c] = size(IDs);
[c,spectra_length] = eval(['size(' IDs(1,:) ')']);
%assemble the bits into a matrix
mat = zeros(num_spectra,spectra_length);
for i = 1:num_spectra
    eval(['mat(i,:) = ' IDs(i,:) ';']);
end
peakList = cell(num_spectra,1);
pe = cell(num_spectra,1);

parfor i = 1:size(mat,1)
    [peakList{i},pe{i}] = peakpick(mat(i,:),30,5E-5);
end

matlabpool close
```

MATLAB Pools

To run this code on TUC, modify the pool code by commenting out the matlabpool open and close statements. Then call the modified task function of a matlabpooljob.

This is a special-case ParallelJob that starts a pool for you, so that your task code executes inside of the pool.

Creating a MatlabPoolJob is identical to creating a ParallelJob and all of the same properties can be set on it.

MatlabPoolJob on TUC

```
function [j,a] = findallpeaks_submit(sched)
%This is a simple distributed job submission example.


j = createMatlabPoolJob(sched);
set(j, 'MaximumNumberOfWorkers', 8);
set(j, 'MinimumNumberOfWorkers', 8);
createTask(j,@findAllPeaks_TUCPool,3,{'spec_data.mat'});
set(j,'PathDependencies',{'\\storage01\matlab\naw47\workshop\findpeaks'});
submit(j);

waitForState(j);

a = getAllOutputArguments(j);
getErrors(j);
```

MATLAB Pools Lab

Convert "findAllPeaks_pool" to run on TUC

Part 1

a) Remove the MATLAB pool and graphing commands.

b) Create a submission script using either File or Path Dependencies to add the MATLAB and data files.

c) Files needed for this job: findAllPeaks_pool.m, peakpick.m, spec_data.mat)

Part 2 (re-enable the graphing functionality)

a) Add the graphics commands to the submission script, by parsing the results from getAllOutputArguments().

b) Put the graphics commands into findAllPeaks_pool, and pass the resulting figure in getAllOutputArguments()

Debugging

Debugging

- Debugging a remote process is always difficult; TUC is no different. Errors must be caught, captured and returned to the client machine to resolve.
- MATLAB usually captures any errors in the task function and stores it as a MException in the task output. The contrib function getErrors "pretty prints" these errors for you.

Getting Errors

- SimpleError.m contains a simple error in the task function. Submit SimpleErrorSubmit.m and examine the error (Remember the JobID):

```
>> [j,a] = SimpleErrorSubmit(sched);
>> getErrors(j);
>> ts = get(j,'Tasks')
>> ex = get(ts,'Error')
>> ex.message
>> ex.stack(1)
```

Manual Retrieval

- The stacktrace information that we looked at previously is stored in the task output files. Each task stores its own errors, so distributed jobs will have an exception for each task.
- Sometimes a job will hang or fail in such a way that the files don't get downloaded correctly. In this case, you'll want to retrieve those files manually.

>> downloadJob(sched,j);

Manual Retrieval

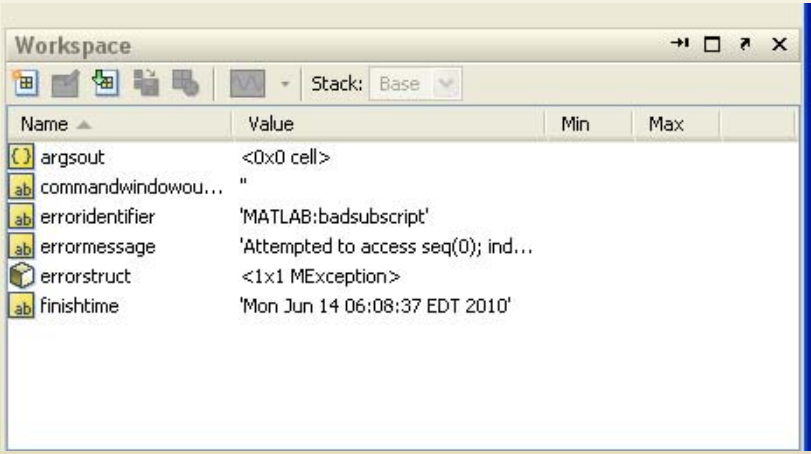
- For large parallel jobs, you might be able to shortcut downloadJob, which will download all of the files.
- For parallel jobs, the error will almost always be in either all Tasks or the master node (Task1), which means that you can just grab Task1.out.mat and it will be very likely you'll get the exception you need.

```
ftp = gridFTP();
ftp.get('Job4/Task1.out.mat');
ftp.close();
```

Manual Retrieval

- Examine the Task1.out.mat file from the SimpleError.m job you just ran. You'll need the DataLocation and the JobID.

```
>> DL = get(sched,'DataLocation');
>> cd([DL '/Job73'])
>> load Task1.out.mat
```



Printf

- The previous examples showed how to capture errors but won't help you catch numeric problems or understand things like how long your code is running or where a long running job is.
- There are two ways to get this information.
- CaptureCommandWindowOutput – This task property tells MATLAB to return output from fprintf statements and statements without a semicolon to the client.

```
alltasks = get(j, 'Tasks');
set(alltasks, 'CaptureCommandWindowOutput', true);
```

- LittleJohnLog/Qpeek – Used to examine a long-running job as it runs.

Printf

- Verbose.m and VerboseSubmit.m contain both LittleJohnLog and fprintf statements. Notice that you must request the tasks to return the command window output at task creation.
- Run the jobs and make sure you can retrieve the output manually as well as using the contrib getOutput(job) function.

```
at = get(j,'Tasks');
out = get(at,'CommandWindowOutput');
getOutput(j);
```

Debug Lab

- The Travelling Salesman Problem is a classic minimization problem. A salesman has a fixed set of cities that he must visit (each only one time); in what order should he visit these cities to minimize the total distance travelled?
- ga_run.m solves this problem using a genetic algorithm (GA) for the airport dataset that we worked with in the file transfer lab. This is a relatively small dataset with about 150 locations.
- ga_run2.m is a buggy version of this that solves the problem for a larger dataset (cities.txt). Both functions take the same arguments and return the same output arguments, so ga_submit.m should not need to be modified. Examine the output from getErrors and getOutput in order to find and fix the problems and help our intrepid salesman out.

Mex Lab

Mex

- Mex is a method for extending MATLAB with C, C++, and Fortran code. A special Mex-function needs to be included that handles the conversion of MATLAB's input types to the appropriate language-specific types, and vice versa for the output types. Then the code is compiled, typically from inside MATLAB using the “mex” command.
- TUC is a Win64 system and Mex functions must be compiled for that architecture. If you don’t have a Win64 system then you’ll need to mex your source files on TUC.

Mex

- There are several difficulties with mex-ing files on TUC, so a helper script is included in the LittleJohn distribution (examples/cacMex.m). There is also an example submission script (examples/cacMexSubmitter.m) to help guide you through its use.
- cacMex takes two arguments, a list of the files that need to be compiled (the files that you would put on the mex line on your local machine) and a complete list of all files needed for the compile to succeed (include files, .h files, etc).

Mex Lab

- A simple C++ file (jcread.cpp) has been provided that will read the .jdx files included with the lab.
- Create a submission script (modify cacMexSubmitter.m) to compile the jcread.cpp. The command to compile this on your local machine (provided an appropriate compiler is installed) is simply:

  >> mex jcread.cpp
- Once you have the file submit a simple job that uses jcread.mexw64. You'll want to use readjdx.m which takes a cell array of files to read and returns an x-vector and y-matrix.