



Open Science Grid

Backpacking with Code: Software Portability for DHTC

Wednesday morning, 9:00 am

Christina Koch (ckoch5@wisc.edu)

Research Computing Facilitator

University of Wisconsin - Madison

Goals for this session

- Understand the basics of...
 - how software works
 - where software is installed
 - how software is accessed and run
- ...and the implications for DHTC
- Describe what it means to make software “portable”
- Learn about and use two software portability techniques:
 - Build portable code
 - Use wrapper scripts



Motivation

running a piece of software is like cooking a meal in a kitchen



The problem



Running
software on
your own
computer =
cooking in your
own kitchen

The problem

In your own kitchen:

- You have all the pots and pans you need
- You know where everything is
- You have access to all the cupboards

On your own computer:

- The software is installed, you know where it is, and you can access it.



The problem



Running on a shared computer =
cooking in someone else's kitchen.

The problem

In someone else's kitchen:

- You are guaranteed some things...
- ...but others may be missing
- You don't know where everything is
- Some of the cupboards are locked

On a shared computer:

- Your software may be missing, unfindable, or inaccessible.



The solution

- Think like a backpacker
- Take your software with you
 - Install anywhere
 - Run anywhere
- This is called making software *portable*





Software

- How do we make software portable?
- First we have to understand:
 - What software is and how it works
 - Where software lives
 - How we run it

How software works

- A software program can be thought of as a list of instructions or tasks that can be run on a computer
- A launched program that is running on your computer is managed by your computer's operating system (OS)
- The program may make requests (access this network via wireless, save to disk, use another processor) that are mediated by the OS
- A single program may also depend on other programs besides the OS

How software works*

*Not to scale

Program
(software, code,
executable, binary)

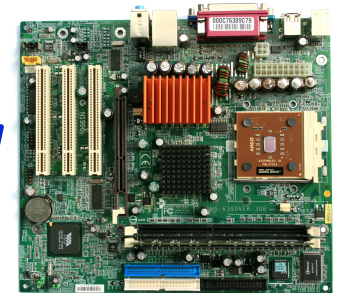


runs
own
tasks

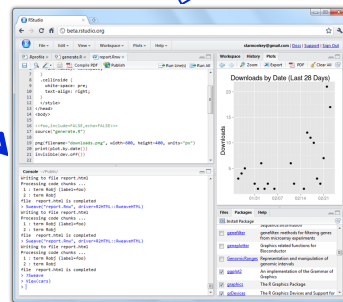
**Operating
System**



translates
program's
request



launches to



makes
requests

monitors
running
programs

depends on

Running Program
(process, instance)



Hardware
(processors,
memory, disk)

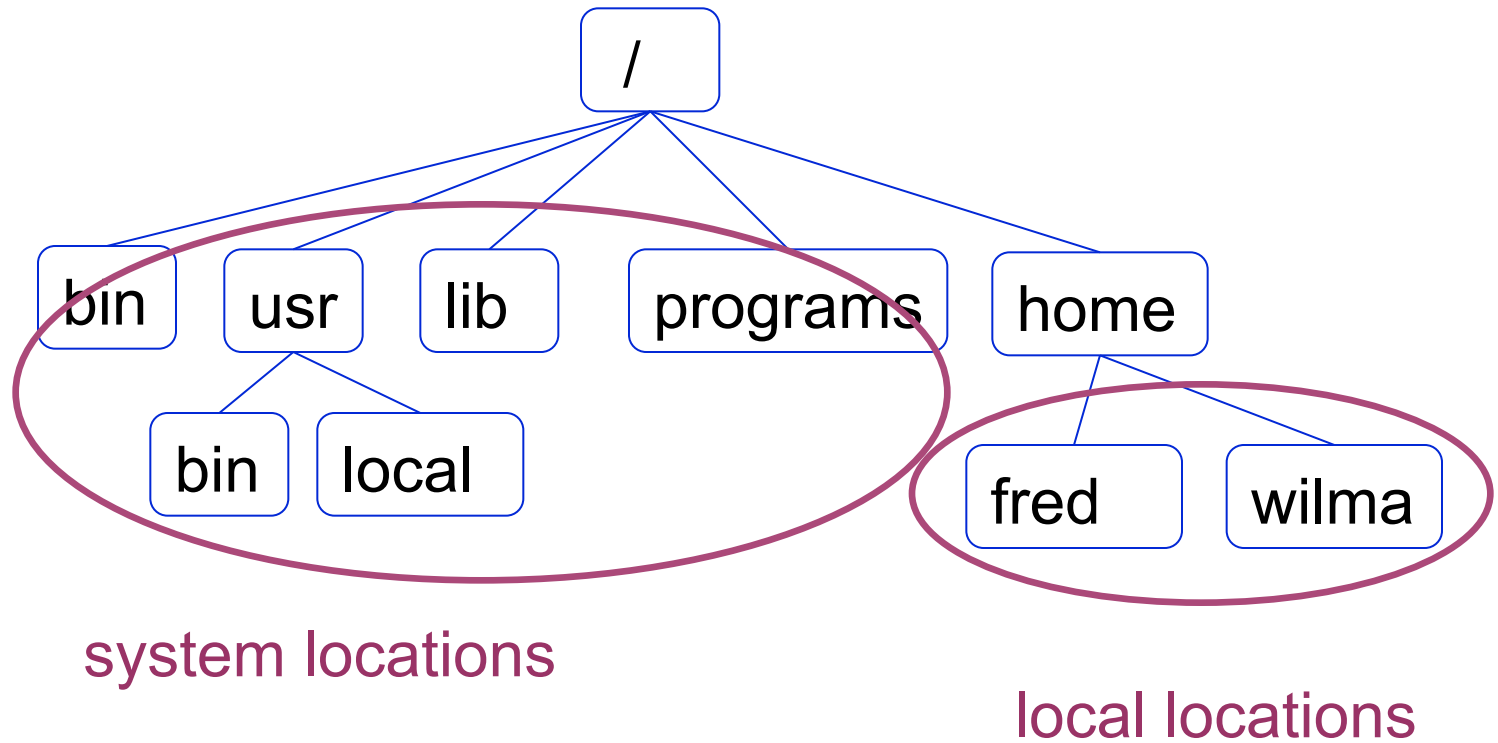
How software works

Implications for DHTC:

- Software must be able to run on target operating system (usually Linux)
- Request specific OS as job requirement
- Know what else your software depends on

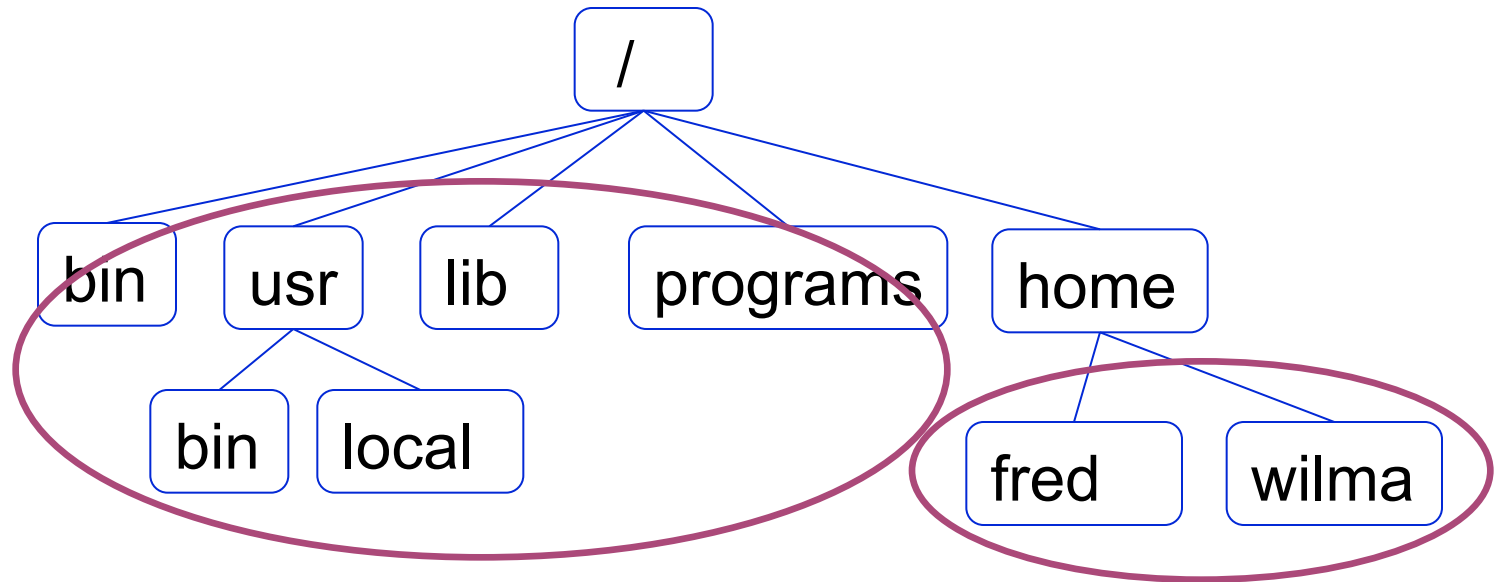
Location, location, location

- Where can software be installed?



Location, location, location

- Who can install the software?

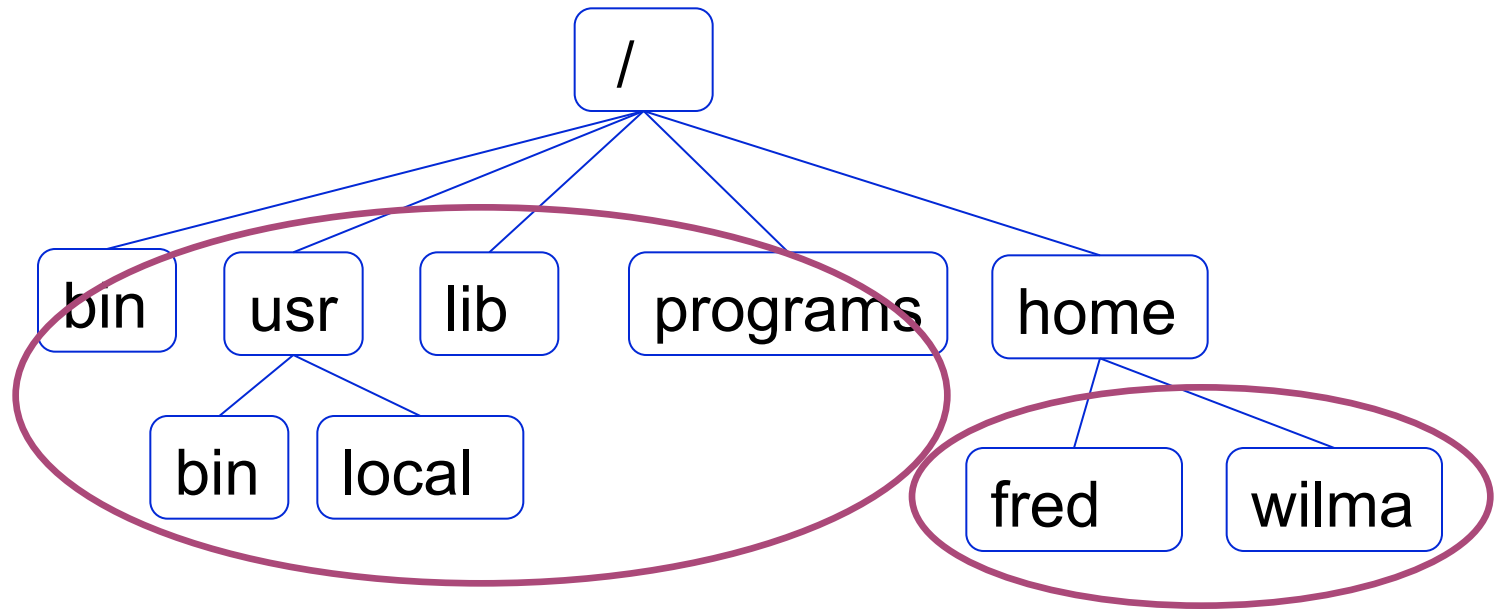


Usually requires
administrative privileges

Owner of the
directory

Location, location, location

- Who can access the software?



Anyone on the system

The local user can control who has access

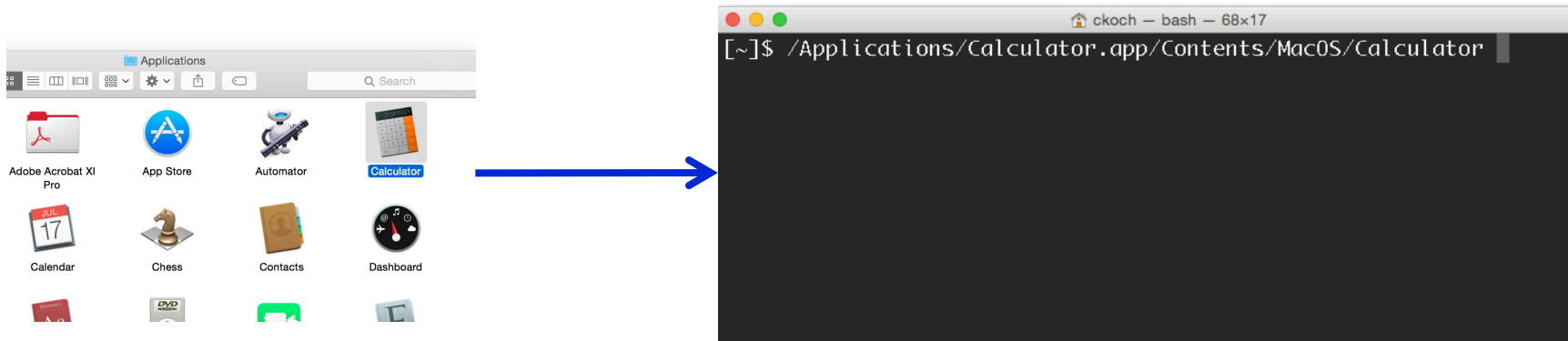
Location, location, location

Implications for DHTC:

- Software **MUST** be able to install to a local location
- Software must be installable without administrative privileges

Location and running software

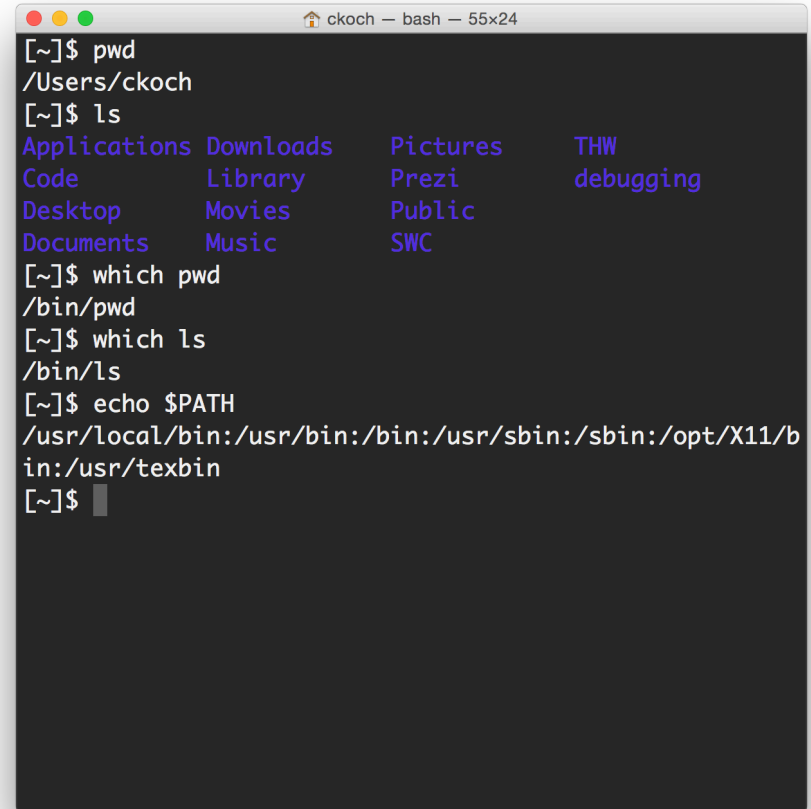
Instead of graphic interface... command line



- All DHTC jobs must use software that can be run from the command line.
- To run a program on the command line, your computer needs to know where the program is located in your computer's filesystem.

Common command line programs

- Common command line programs like `ls` and `pwd` are in a system location called `/bin`
- Your computer knows their location because `/bin` is included in your `PATH`



```
ckoch — bash — 55x24
[~]$ pwd
/Users/ckoch
[~]$ ls
Applications  Downloads  Pictures  THW
Code          Library    Prezi     debugging
Desktop       Movies     Public
Documents     Music      SWC
[~]$ which pwd
/bin/pwd
[~]$ which ls
/bin/ls
[~]$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/X11/bin:/usr/texbin
[~]$
```

- The PATH is a list of locations to look for programs

Other programs on command line

- Other programs may be installed in locations not listed in the PATH. You can access them by:

Adding their location to the PATH, then running

```
ckoch — bash — 53x13
[~]$ export PATH=/Users/ckoch/Code/python/bin/:$PATH
[~]$ echo $PATH
/Users/ckoch/Code/python/bin:/usr/local/bin:/usr/bin
:/bin:/usr/sbin:/sbin:/opt/X11/bin:/usr/texbin
[~]$ python2.7 --version
Python 2.7.7
[~]$
```

Using an relative or absolute path to the software

```
ckoch — bash — 53x13
[~]$ Code/python/bin/python2.7 --version
Python 2.7.7
[~]$ /Users/ckoch/Code/python/bin/python2.7 --version
Python 2.7.7
[~]$
```

Command line

Implications for DHTC:

- Software must have ability to be run from the command line
- Multiple commands are okay, as long as they can be executed in order within a job
- There are different ways to “find” your software on the command line: relative path, absolute path, and PATH variable

Portability requirements

Based on the previous slides, we now know that in order to make software portable for DHTC, the software:

- Must work on target operating system (probably Linux)
- Must be accessible to your job (placed or installed in job's working directory)
- Must be able to run without administrative privileges
- Must be able to run from the command line, without any interactive input from you

Returning to our scenario:

In a DHTC situation, we are:

- Using someone else's computer
 - Software may not be installed
 - The wrong version may be installed
 - We can't find/run the installed software

Therefore:

- We need to bring along and install/run software ourselves

Portability methods

There are two primary methods to make code portable:

- Use a single compiled binary
 - Typically for code written in C, C++ and Fortran
- “Install” with every job
 - Can’t be compiled into a single binary
 - Interpreted languages (Matlab, Python, R)



Method 1

USE A COMPILED BINARY

What is compilation?

Source code

```
function isLoggedin() {  
    global $username, $password  
    if ($username && $password) {  
        $pass = md5(GetPassword);  
        return ($password == $pass) ? TRUE : FALSE;  
    }  
}
```

Binary



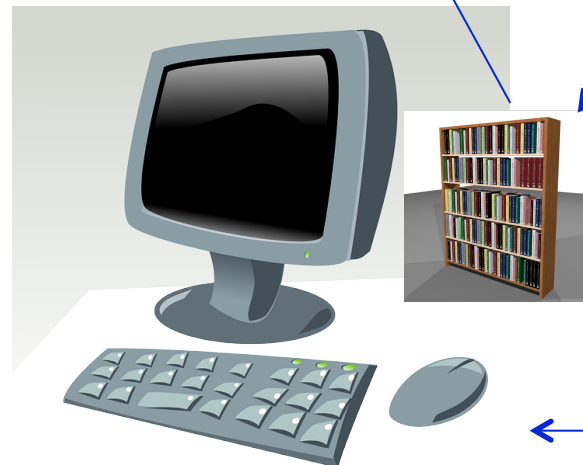
compiled + linked

compiler
and OS

libraries

uses

run on



Static linking

Source code

```
stPassw  
"");  
= file("login.dat");  
$i = 0; $i < count($users); $i  
$line = $users[$i];  
if (ereg("^$username(\\.)*", trim($line  
// User gevonden, Password is mo  
$pass = $regs[1];  
break; // Stop met de 'for'-l  
)  
return $pass;  
}  
function isLoggedIn($username, $password)  
{  
global $username && $password;  
if ($username == md5(GetPassword($  
return ($password == $p  
return FALSE;
```

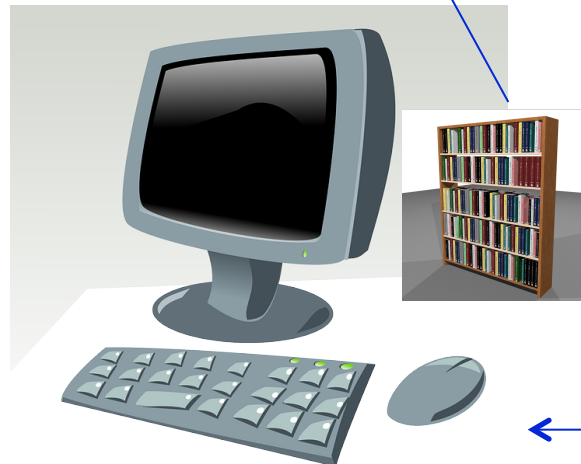
compiled + static linking



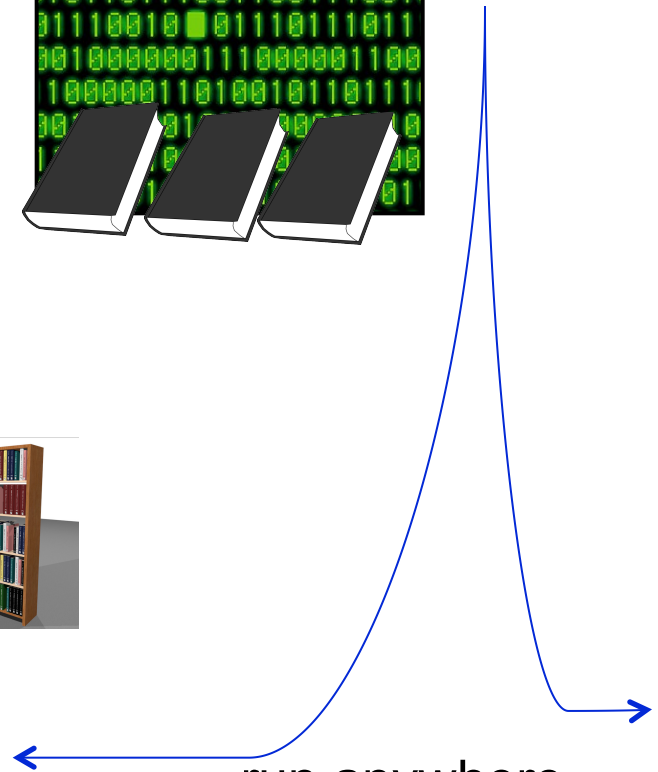
compiler
and OS

libraries

Static binary



run anywhere



Compilation (command line)

```
ckoch — ckoch5@submit-5:~/osg/code/compile — ssh — 69x21
$ ls
hello.c
$ gcc hello.c -o hello_dynamic
$ ls
hello.c hello_dynamic
$ file hello_dynamic
hello_dynamic: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.18, not stripped
$ gcc -static hello.c -o hello_static
$ ls
hello.c hello_dynamic hello_static
$ file hello_static
hello_static: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, for GNU/Linux 2.6.18, not stripped
$
```

Static compilation workflow

Option 1
compile

```
spass="login.de"
if [ $(cat /dev/urandom | fold -n 10 | tr -dc 'a-z0-9' | fold -w 10 | paste | sha1sum | cut -d ' ' -f 1) = $spass ]; then
  echo "Password is correct"
else
  echo "Wrong password"
  break // stop met de 'for'
fi
return $spass
}
function tslog($dir) {
  global $username && $password
  if [ $username && $password ]
  then
    return $(spass "$dir" "$username" "$password")
  else
    return FALSE
  fi
}
```

Option 2
download

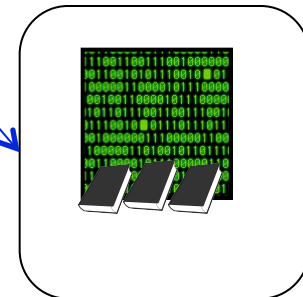
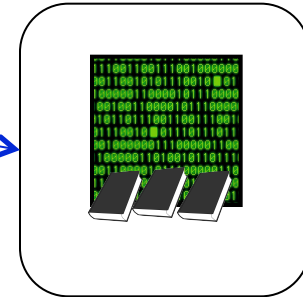
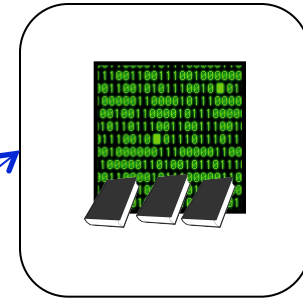


Submit server

Static binary



Execute server





Method 2

USE WRAPPER SCRIPTS

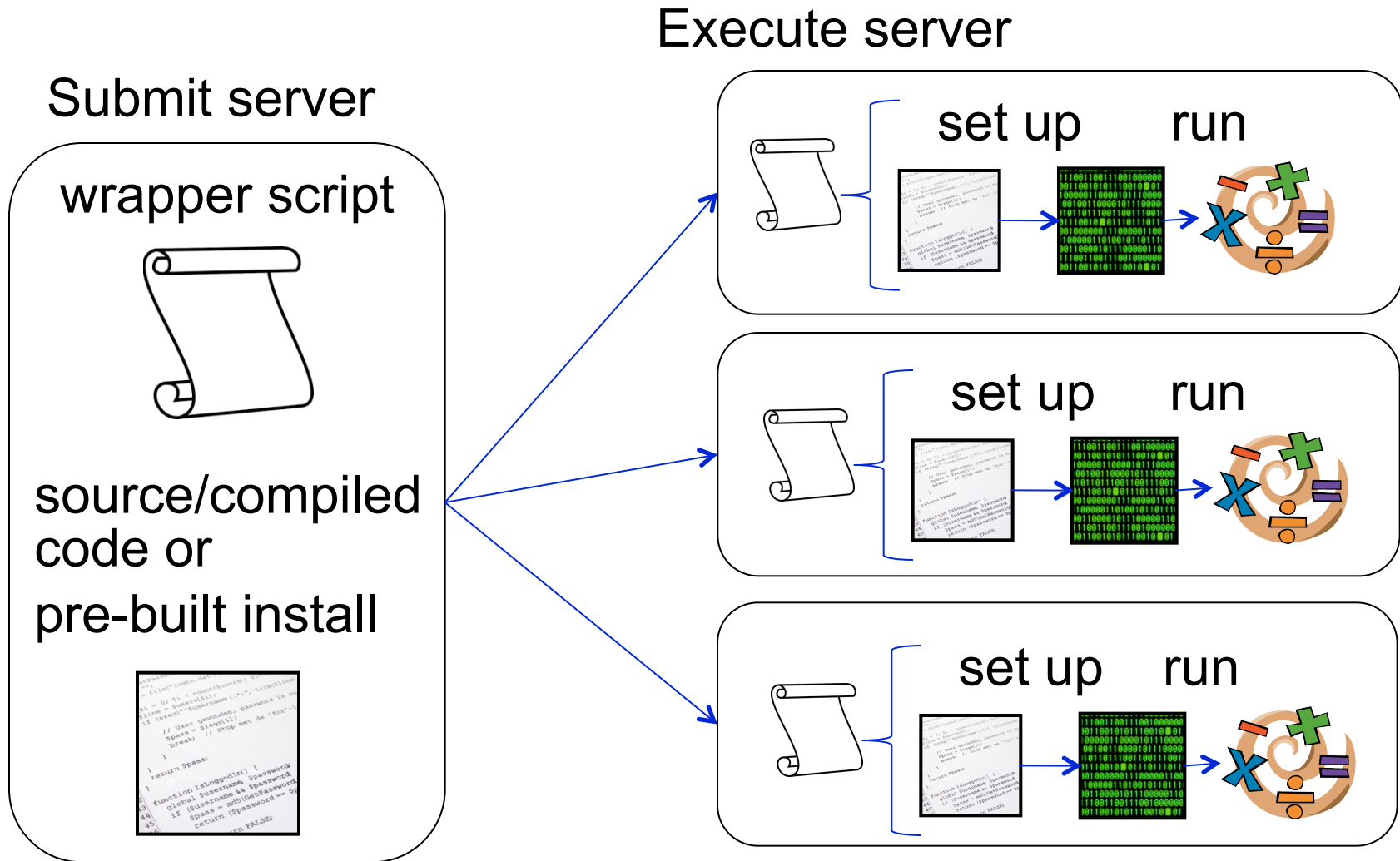
Set up software with every job

- Good for software that:
 - Can't be statically compiled
 - Uses interpreted languages (Matlab, Python, R)
 - Any software with instructions for local installation
- Method: write a wrapper script
 - Contains a list of commands to execute
 - Typically written in bash or perl (usually common across operating systems/versions)

Wrapper scripts

- Set up software in the working directory
 - Unpack pre-built OR
 - Install on the fly OR
 - Just use normal compiled code
- Run software
- Besides software: manage data/files in the working directory
 - Move or rename output
 - Delete installation files before job completion

Wrapper script workflow



When to pre-build?

Pre-built installation

- Install once, use in multiple jobs
- Faster than installing from source code within the job
- Jobs must run on a computer similar to where the program was built

Install with every job

- Computers must have appropriate tools (compilers, libraries) for software to install
- Can run on multiple systems, if these requirements are met
- Longer set-up time

Preparing your code

- Where do you compile code? Pre-build code? Test your wrapper script?
- Guiding question: how computationally intensive is the task?
 - Computationally intensive (takes more than a few minutes, as a rule of thumb)
 - Run as interactive job, on a private computer/server, or with a queued job
 - Computationally light (runs in few minutes or less)
 - Run on submit server (or above options, if desired)



Exercises

- Software is a compiled binary
 - Exercise 1.1: statically compile code and run (C code)
 - Exercise 1.2: download and run pre-compiled binary (BLAST)



Exercises

- Introduction to using wrapper scripts
 - Exercise 1.3: use a wrapper script to run previously downloaded software (BLAST)
- Portable installation and wrapper scripts
 - Exercise 1.4: create a pre-built software installation, and write a wrapper script to unpack and run software (OpenBUGS)

Questions?

- Feel free to contact me:
 - ckoch5@wisc.edu
- Now: Hands-on Exercises
 - 9:30-10:30am
- Next:
 - 10:30-10:45am: Break
 - 10:45am-12:15pm: Other research software considerations: licenses, interpreted languages, and containers
 - 12:15-1:15pm: Lunch