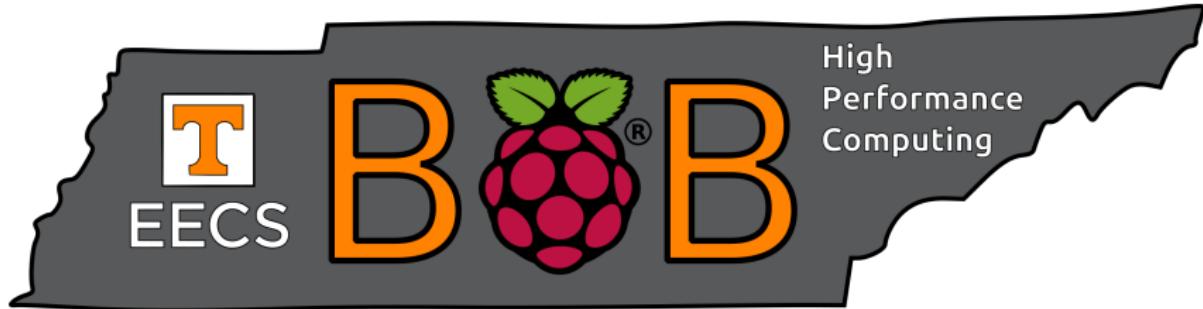


Big Orange Bramble+ BOB and ALICE



December 2, 2016

Overview

Hardware Overview

Software Stack Overview

Enclosure

Daughter Card

Monitor Node

Fan & Reset Controller

Pine64 Issues

Slurm

Accounting

Benchmarking

Facial Recognition

FDS

WRF

Gillespie Algorithm

Hardware Overview

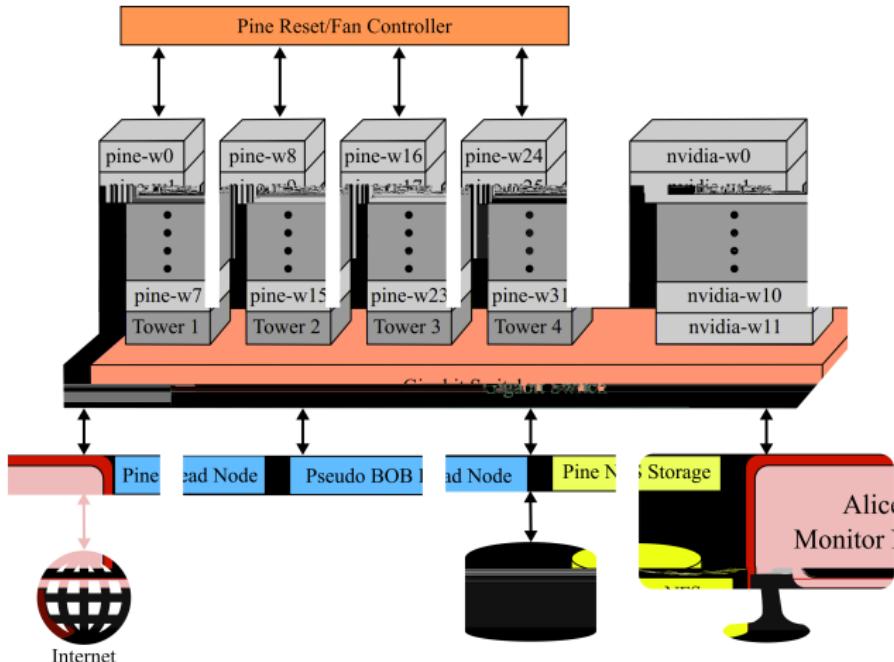


Figure 1: Hardware Diagram

Software Stack Overview

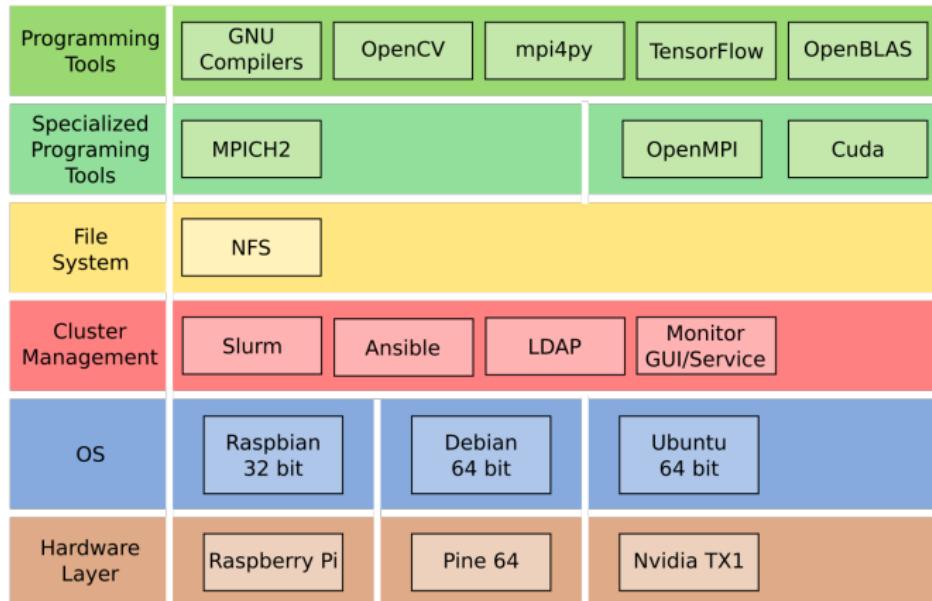


Figure 2: Software Stack

Enclosure



Figure 3: Enclosure

Rack Enclosure on Casters

Pull out shelf for keyboard

One rack shelf for 32
Pine64s

3 rack shelves for NVIDIA
TX1s

12 PWM controllable 12V
Fans

Daughter Card Review

Needed a way to measure power input to nodes.
Convert analog measurements to digital packets.
Send information to a Monitor Node.

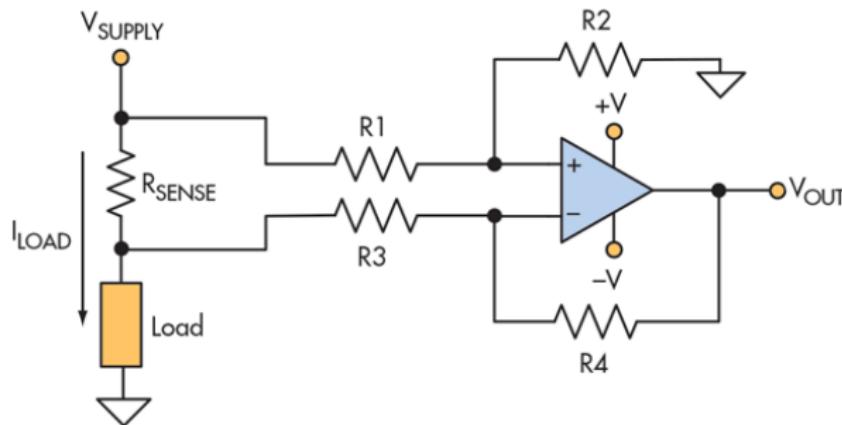


Figure 4: Current Sense Technique

Daughter Card Review

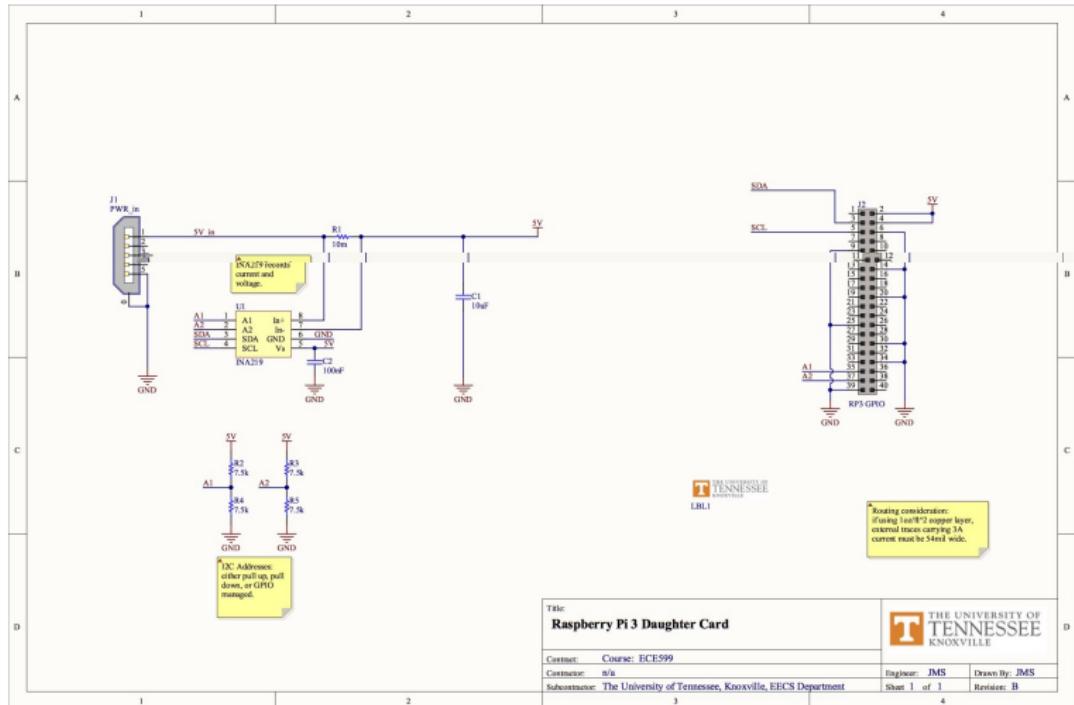


Figure 5: Daughter Card Schematic

Daughter Card Updates

PI-2 port on Pine64 allows for compatibility with BOB DC
BOB DC did not use pull-up or pull-down resistors on I2C
SCL or SDA lines. Simple mod.

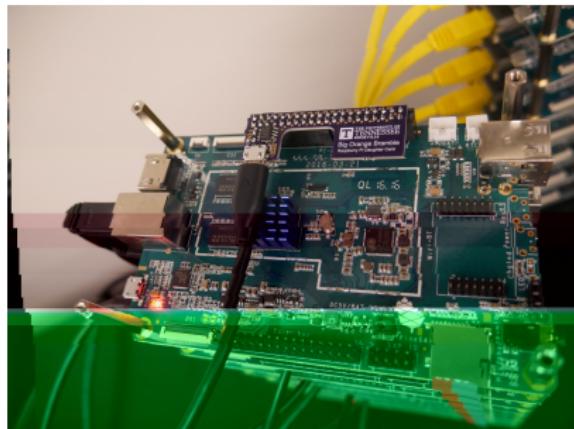


Figure 6: Mounted DC

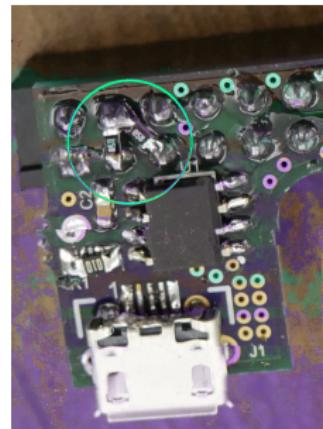


Figure 7: Pull-up Resistors

Daughter Card Firmware

Same Adafruit C++ library for the INA219 was used

The Python library overlay for this Adafruit C++ library had to be completely rewritten

- The original Python library was written by a hobbyist
- It took advantage of several assumptions, many of which included that this library would only be used with a Raspberry Pi 1 or 2
- The rewritten library should be more generic and allow for any Pi-like board with some flavor of Linux to utilize its functionality and is made available on our BitBucket account

Monitor node continues to handle communication with all daughter cards

Monitor Node Backend

Separate Raspberry Pi with Touchscreen

Python monitoring script runs as a service on each node

Each node sends:

- CPU temperature
- CPU load
- CPU frequency
- SoC core voltage

Nodes with daughter cards also send:

- Supply current
- Supply voltage

Information is sent via UDP packets

Information sent from nodes every 2 seconds

Monitor Node GUI

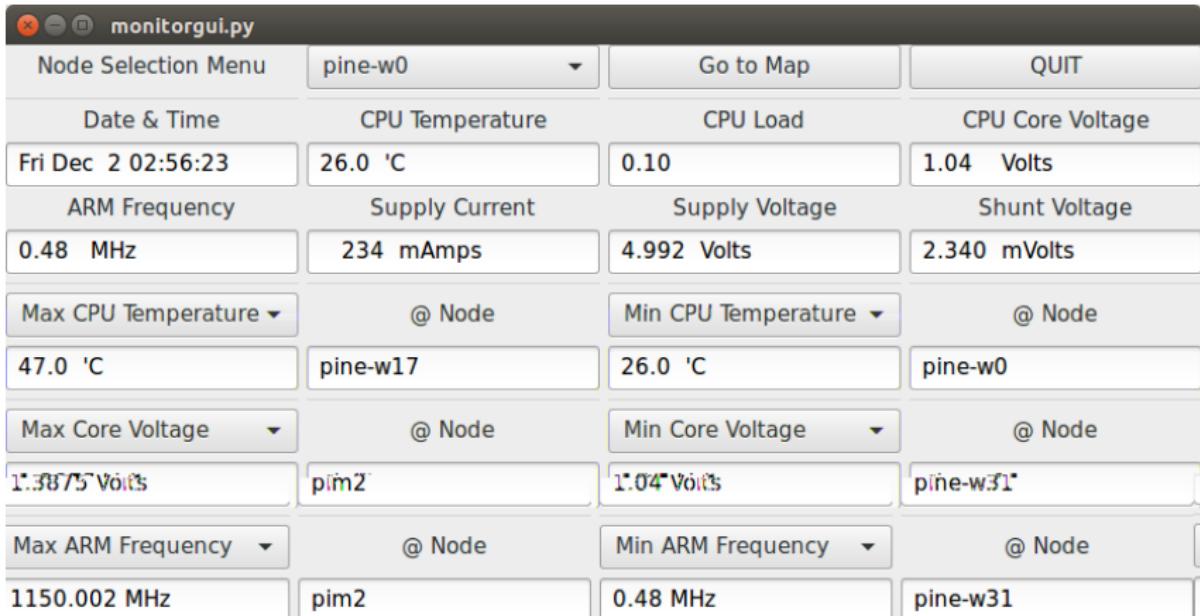


Figure 8: Main Monitor Page

Monitor Node GUI



Figure 9: Heat Map Page

Fan & Pine Reset Controller

Arduino Mega 2560 can control fan
PWM duty cycle, from Daughter Card
feedback through Monitor Node.
Additionally provides wiring to toggle
RST on Pine64s

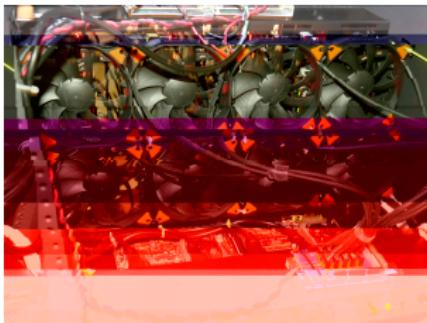


Figure 10: Fans

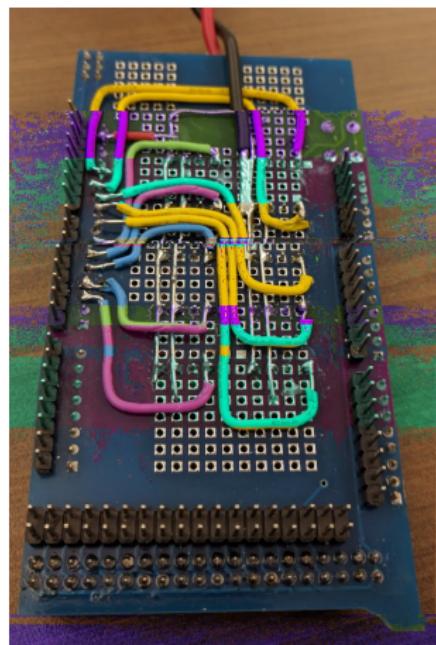


Figure 11: Arduino Shield for Controller

Fan & Pine Reset Controller

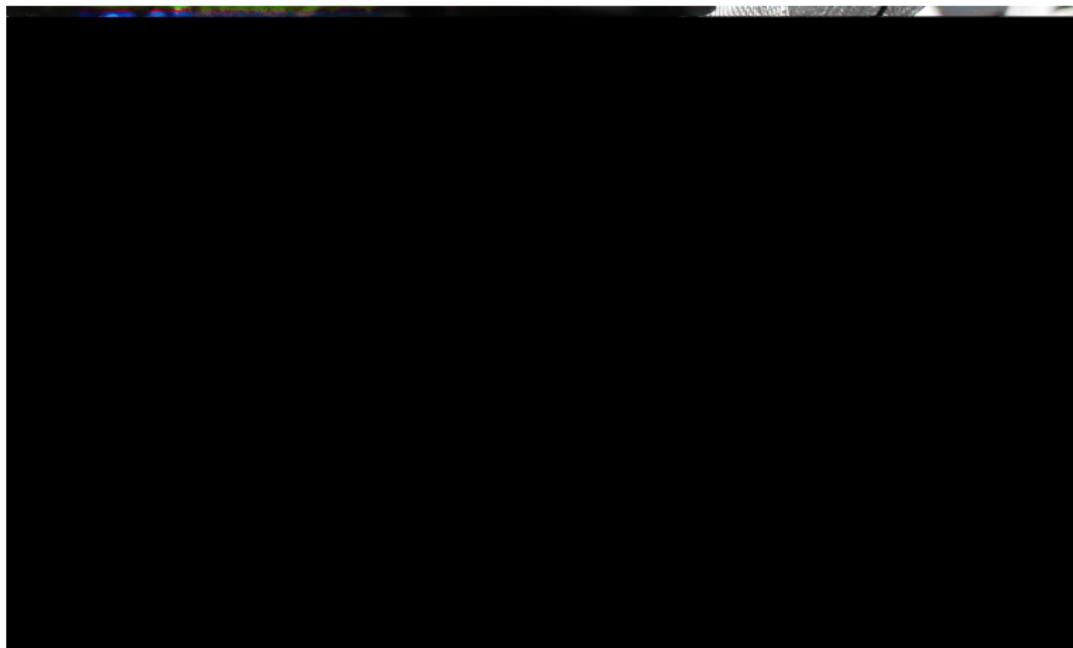


Figure 12: Controller

Pine64 Issues

Issues with HDMI/DVI Display

Reboot Issues

Networking Issues

Software Difficulties

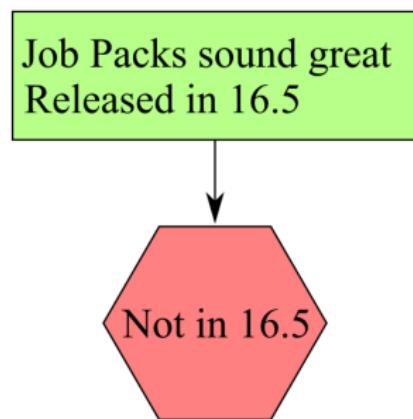
Cache Issues

HPL Issues - Failed Residual Calculation

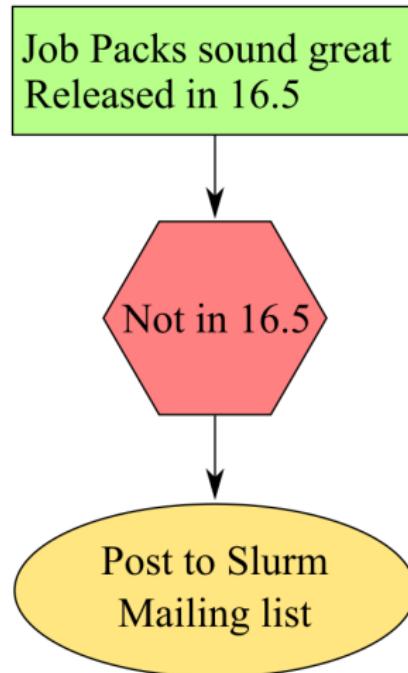
Slurm Job Pack Investigation

Job Packs sound great
Released in 16.5

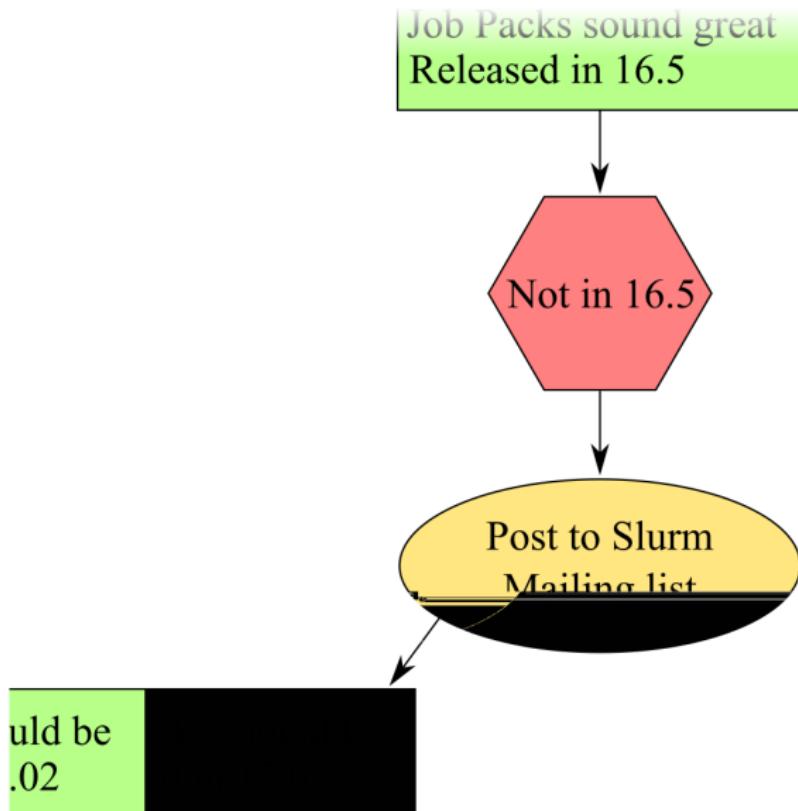
Slurm Job Pack Investigation



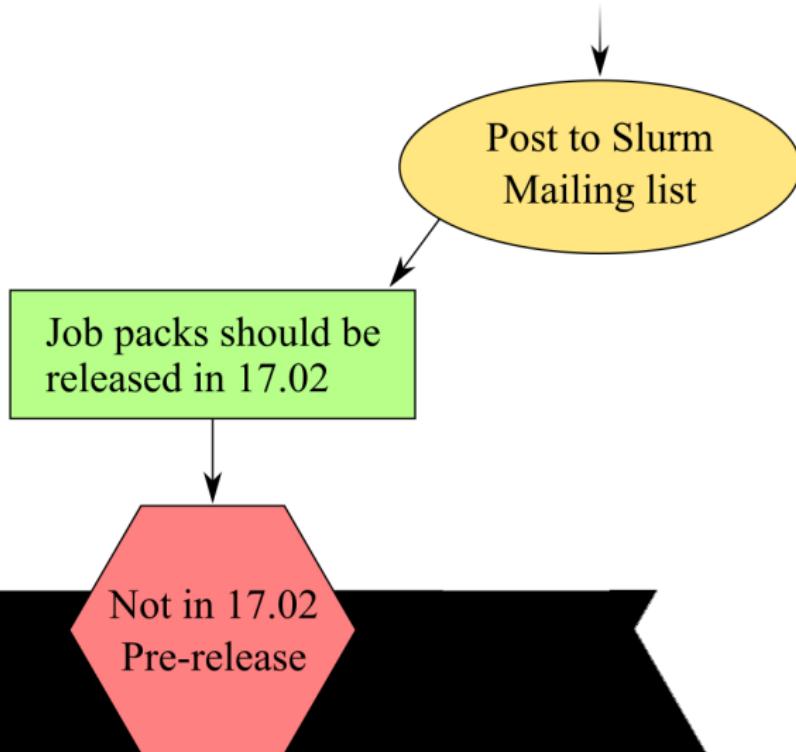
Slurm Job Pack Investigation



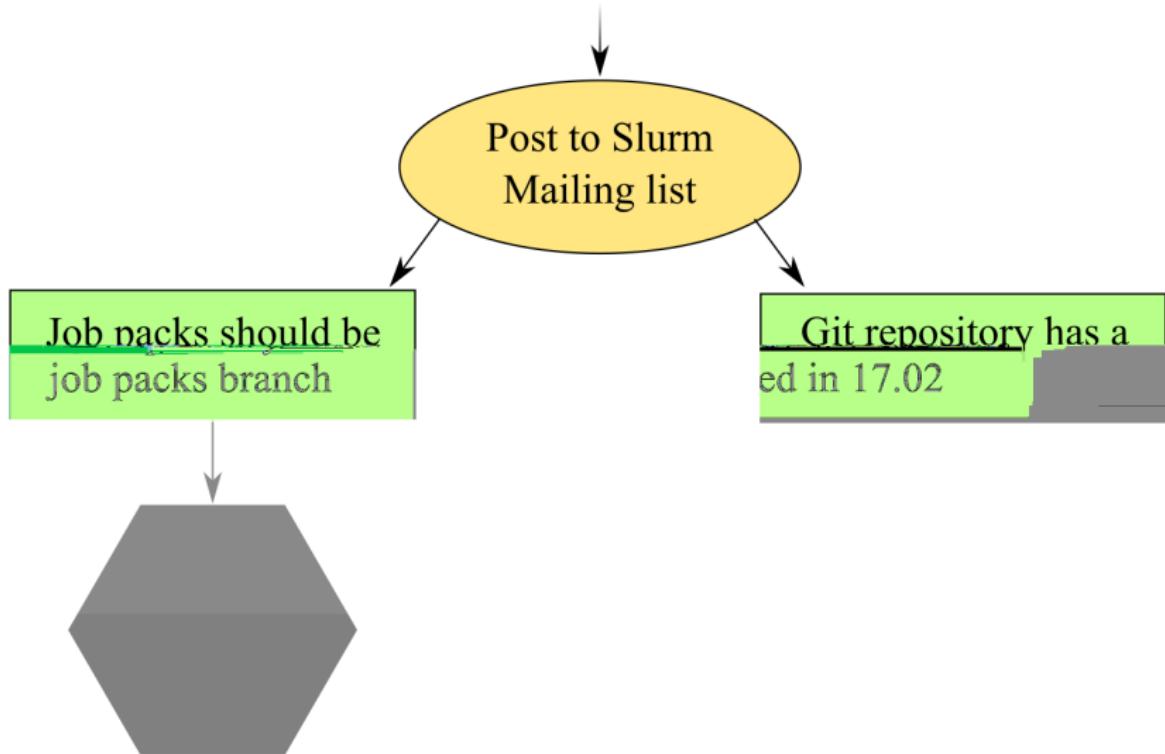
Slurm Job Pack Investigation



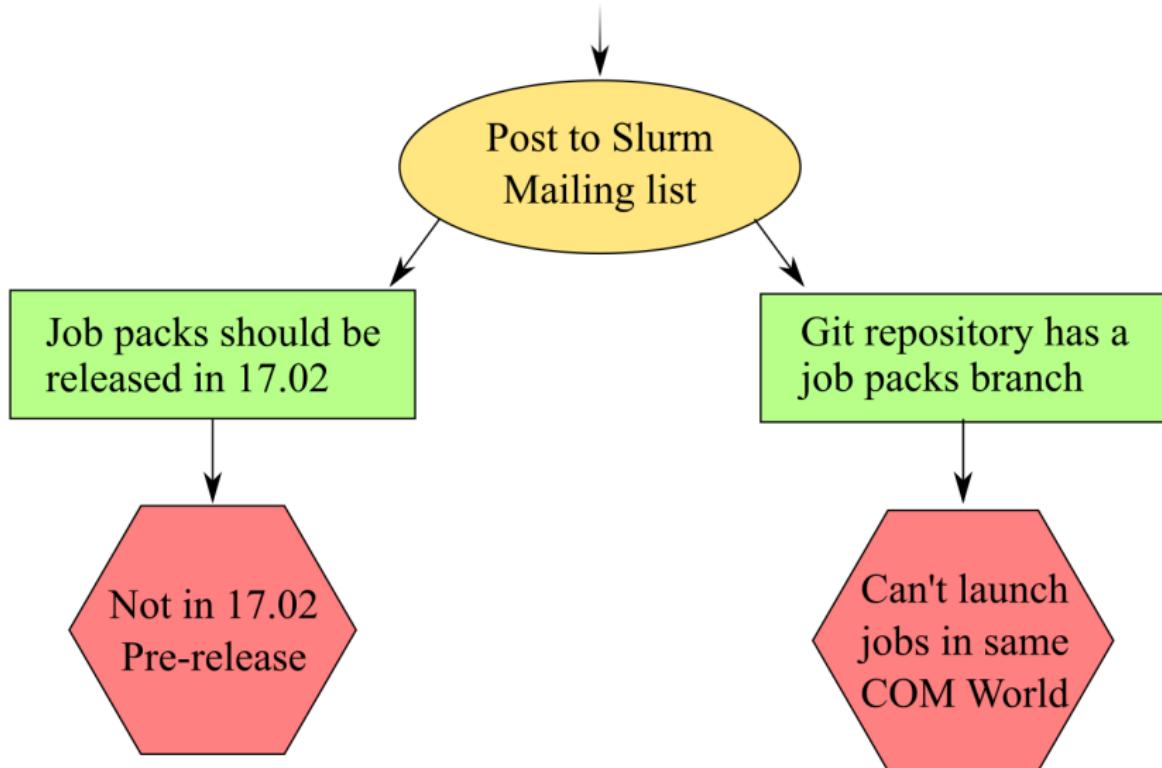
Slurm Job Pack Investigation



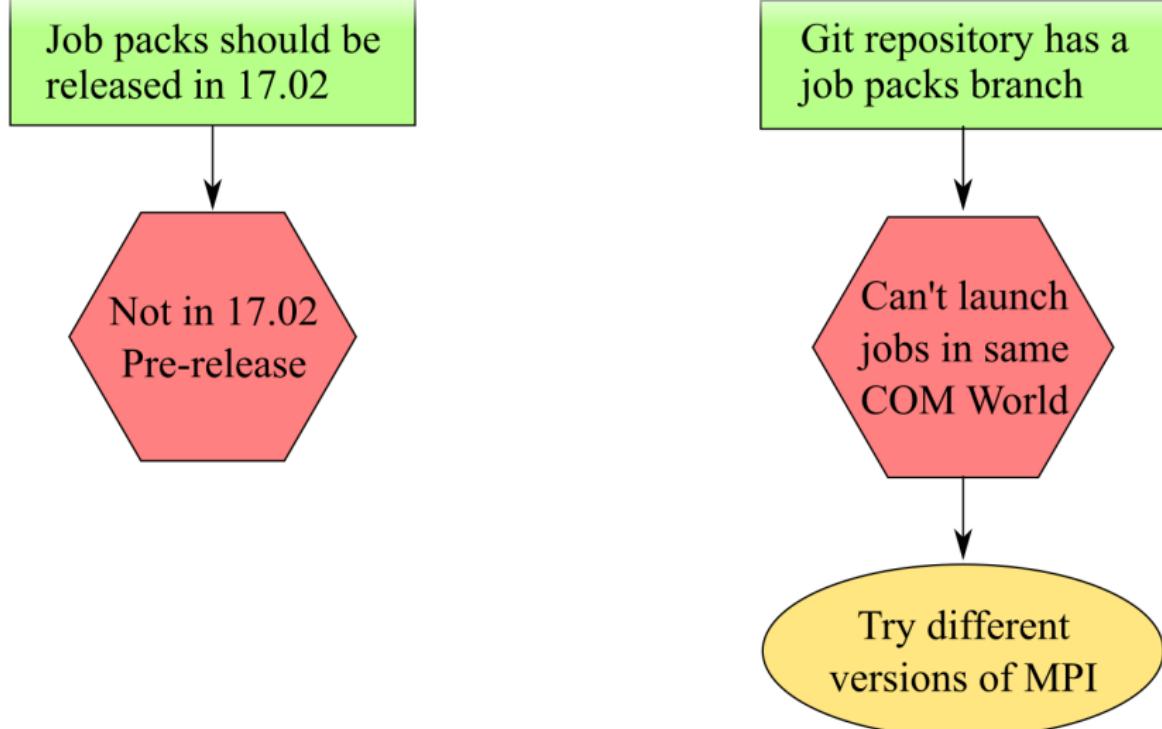
Slurm Job Pack Investigation



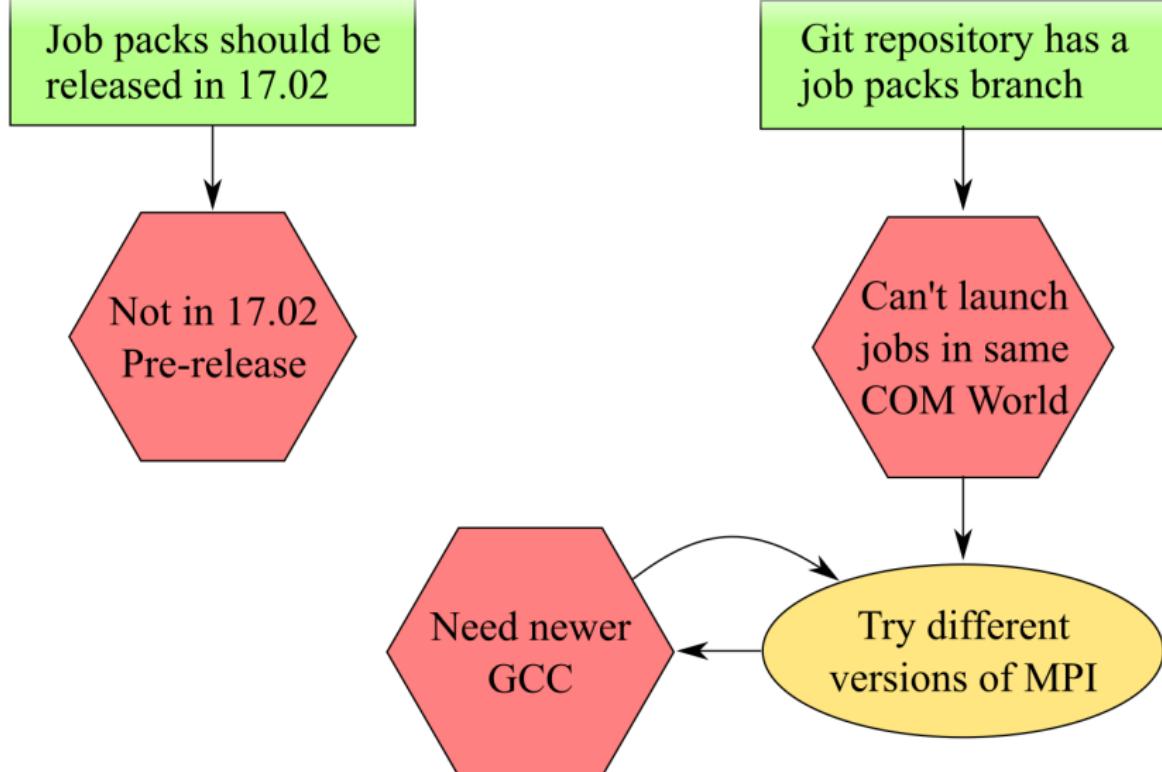
Slurm Job Pack Investigation



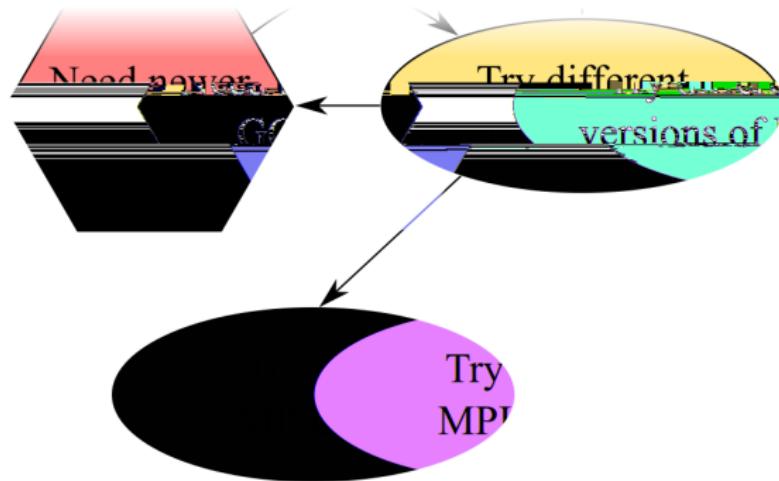
Slurm Job Pack Investigation



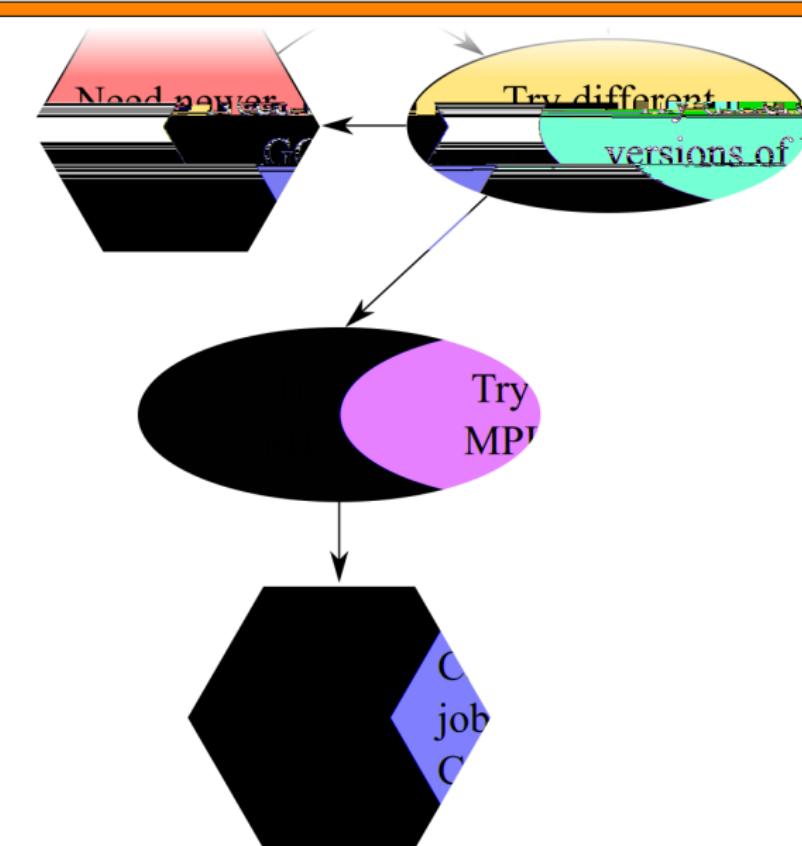
Slurm Job Pack Investigation



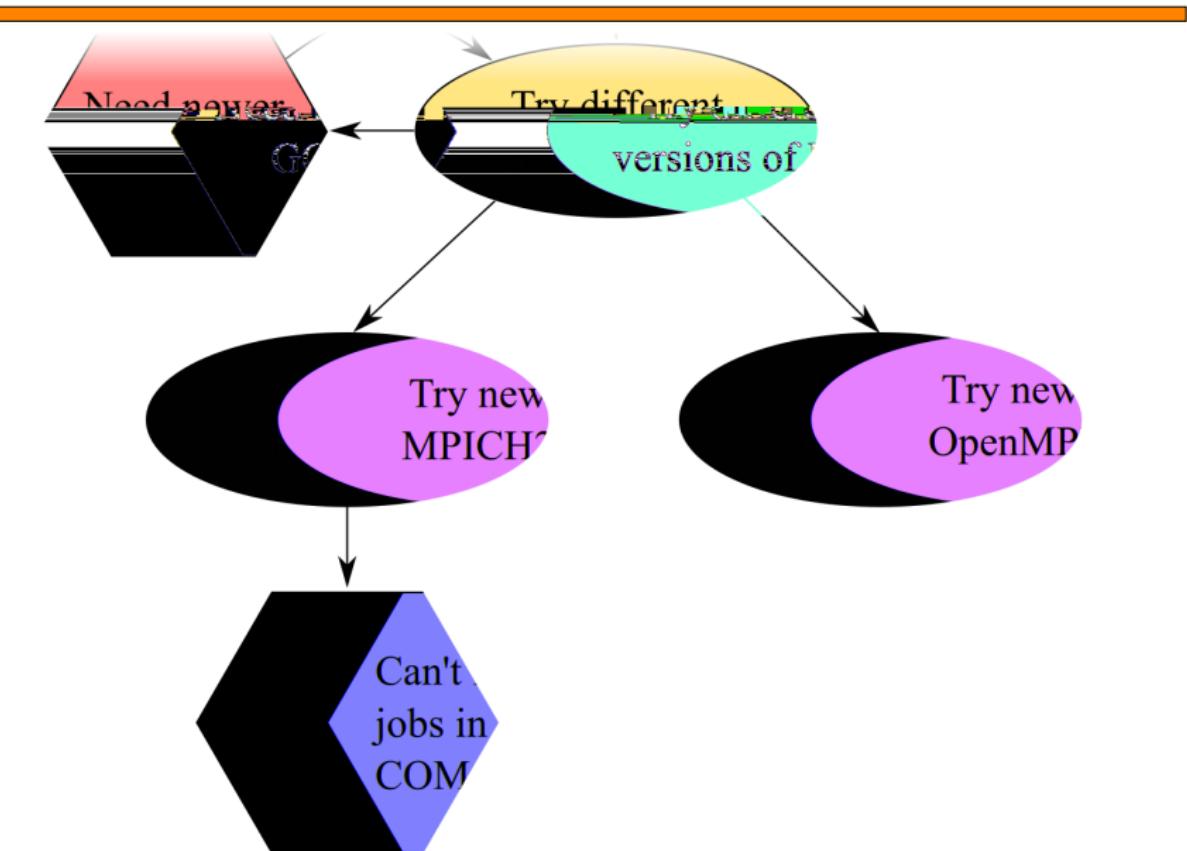
Slurm Job Pack Investigation



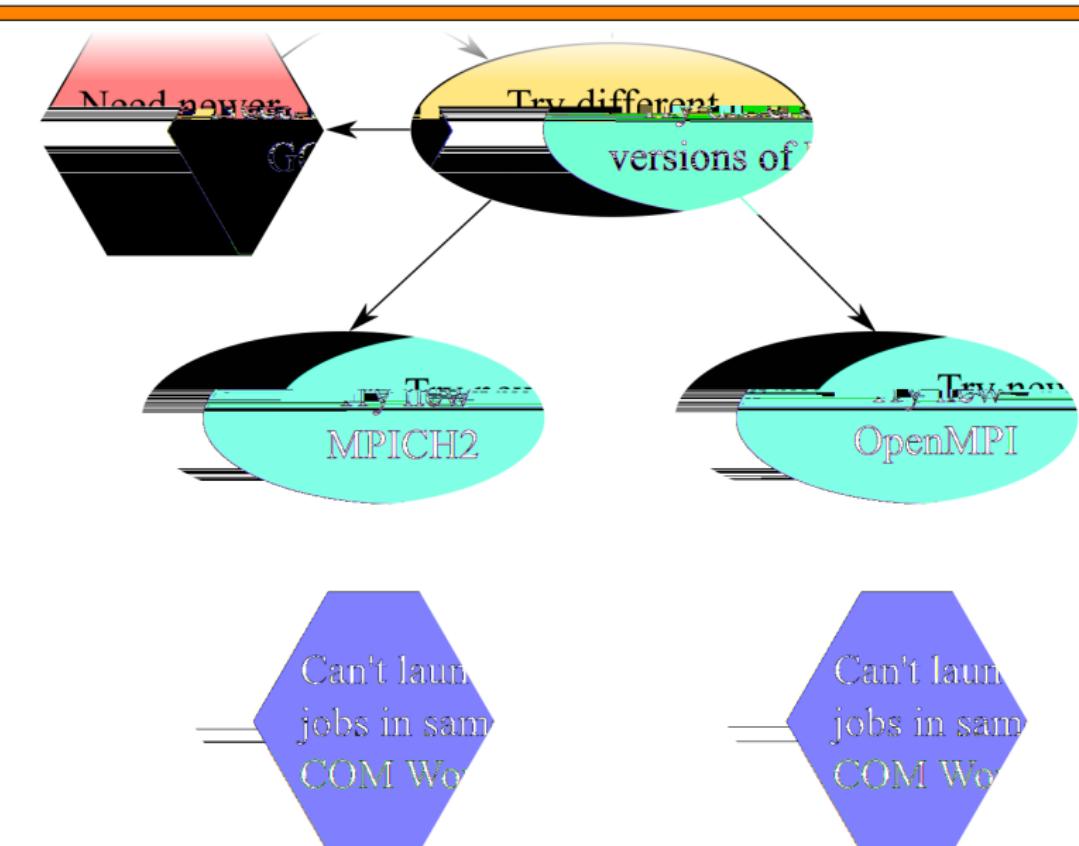
Slurm Job Pack Investigation



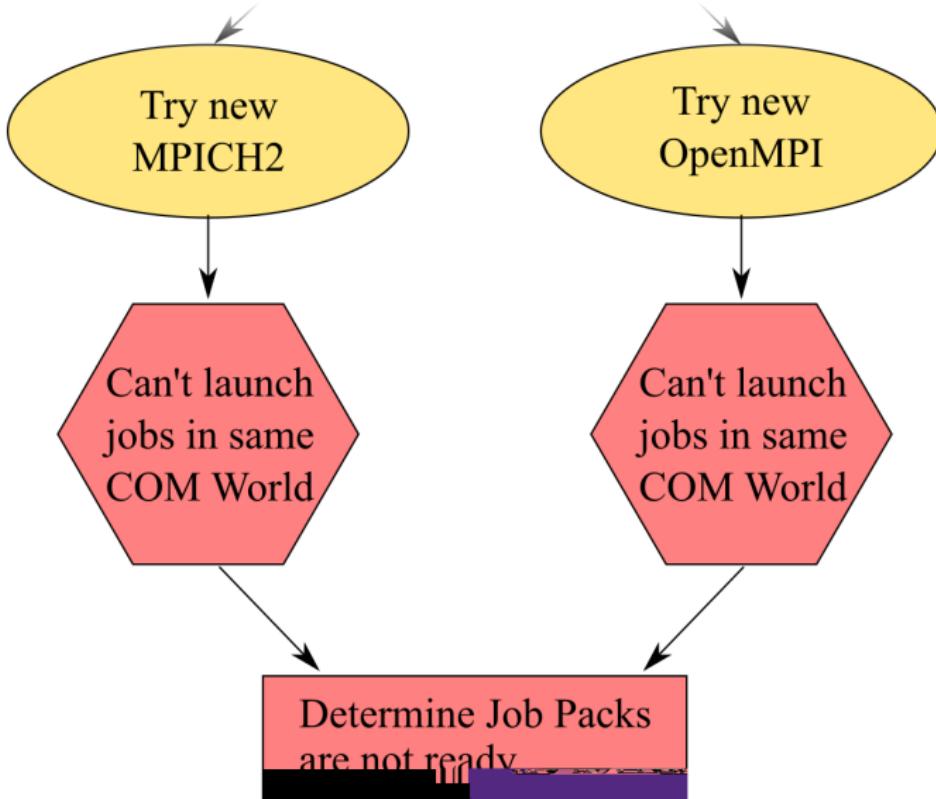
Slurm Job Pack Investigation



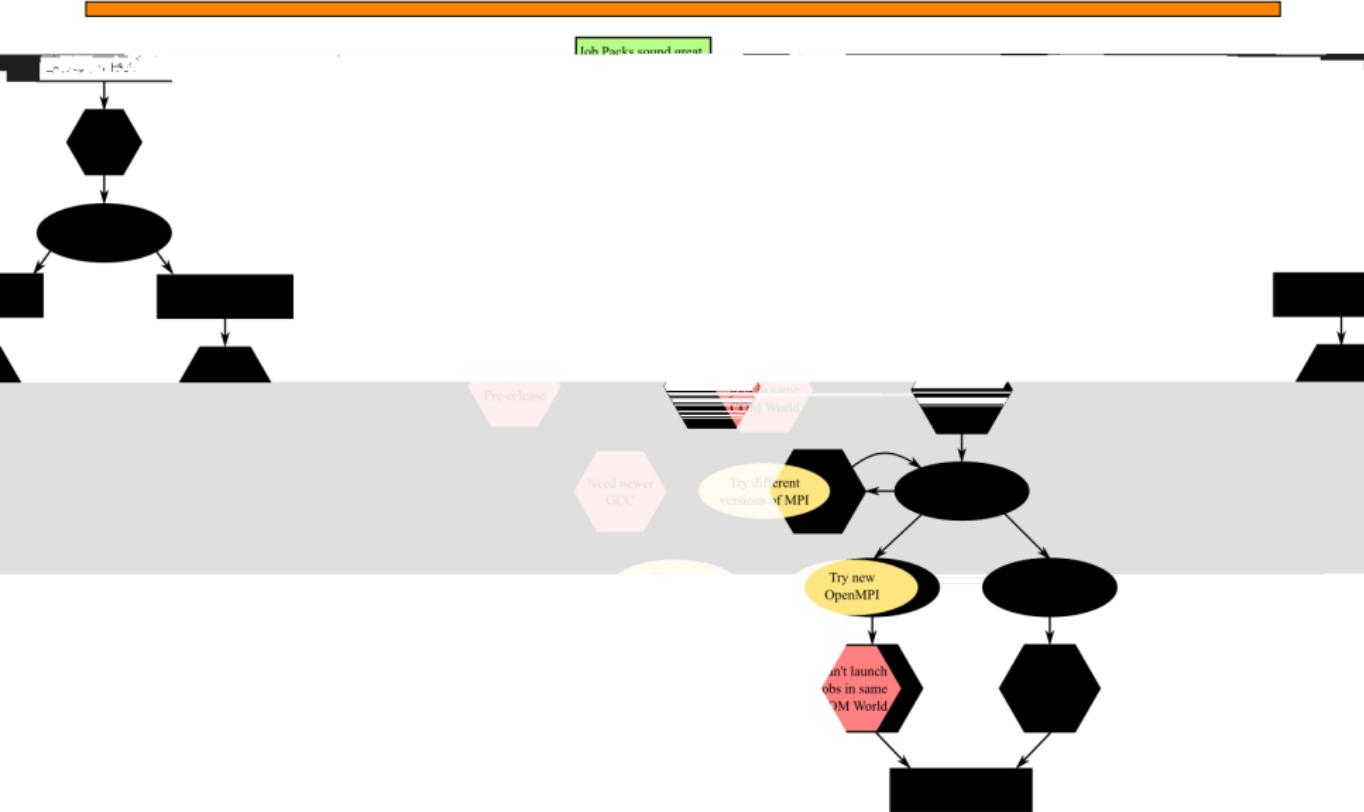
Slurm Job Pack Investigation



Slurm Job Pack Investigation



Slurm Job Pack Investigation



Accounting On Alice Investigation

MySQL is used to store Slurm's accounting information.

MySQL apt package has missing dependencies and will not install on the pine 64s.

Apt has mysql-server-5.6 which can install.

Learned how to configure Slurm for accounting and how to setup the database for use with Slurm.

Discovered that Slurm needs to be reconfigured to detect the MySQL installation.

Slurm can't detect MySQL without MySQL-dev tools. Main MySQL-dev has broken dependencies and 5.6 does not have dev package.

HPL Algorithm

$Ax = b$ solved by LU Decomposition

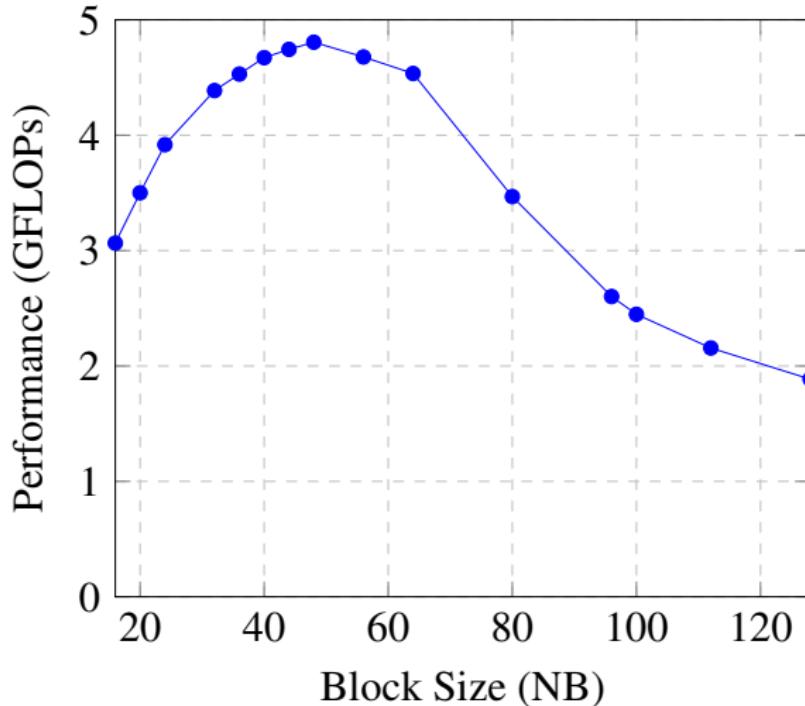
Matrix is of order N, divided into submatrices of order NB

Process grid of P rows by Q columns

A							
0		1		2			
0	1	2	3	4	5	6	7
A_{11}	A_{12}	A_{13}	A_{14}	A_{15}	A_{16}	A_{17}	A_{18}
A_{21}	A_{22}	A_{23}	A_{24}	A_{25}	A_{26}	A_{27}	A_{28}
A_{31}	A_{32}	A_{33}	A_{34}	A_{35}	A_{36}	A_{37}	A_{38}
A_{41}	A_{42}	A_{43}	A_{44}	A_{45}	A_{46}	A_{47}	A_{48}
A_{51}	A_{52}	A_{53}	A_{54}	A_{55}	A_{56}	A_{57}	A_{58}
A_{61}	A_{62}	A_{63}	A_{64}	A_{65}	A_{66}	A_{67}	A_{68}
A_{71}	A_{72}	A_{73}	A_{74}	A_{75}	A_{76}	A_{77}	A_{78}
A_{81}	A_{82}	A_{83}	A_{84}	A_{85}	A_{86}	A_{87}	A_{88}

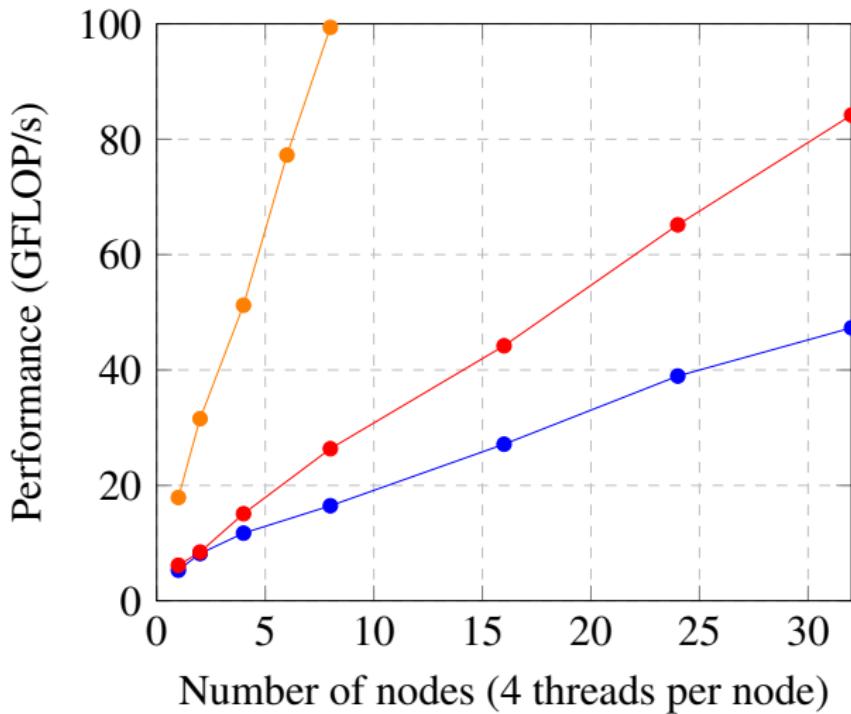
Figure 13: HPL Matrix

HPL - Pine64



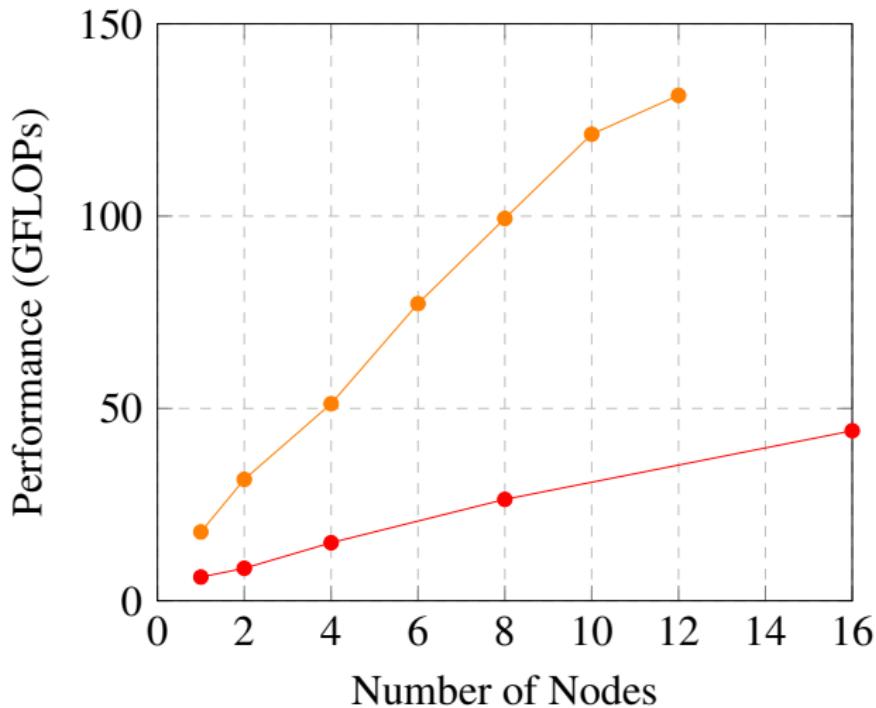
Revealed L2 cache is likely not functioning properly

HPL - Pine64



Blue = Pine64, Red = BOB, Orange = TX1

HPL - nVidia TX1



Orange = TX1 CPU, Red = BOB

HPL - CUDA notes

Two variations utilizing CUDA were found

Github project from David Martin

nVidia provided code for Intel & Fermi

GPU offers only 16 GFLOPs theoretical for double precision

Large overhead associated with both implementations due to memory/data management

Future goal: HPL for a zero-copy unified memory architecture

Future goal: Single precision CUDA-enabled HPL (500 SP GFLOPs theoretical per GPU)

HPL - Performance Summary

Pine64 demonstrated poor scaling due to block size and networking

Currently only getting 47 GFLOPs at 29% scaling efficiency with 32 nodes

nVidia TX1 demonstrated solid CPU performance

Achieving 131 GFLOPs at 61% scaling efficiency with 12 nodes

For reference, BOB achieved 148.8 GFLOPs at 37.7% scaling efficiency with 64 nodes

HPCG

High Performance Conjugate Gradient

Complementary to HPL to evaluate performance

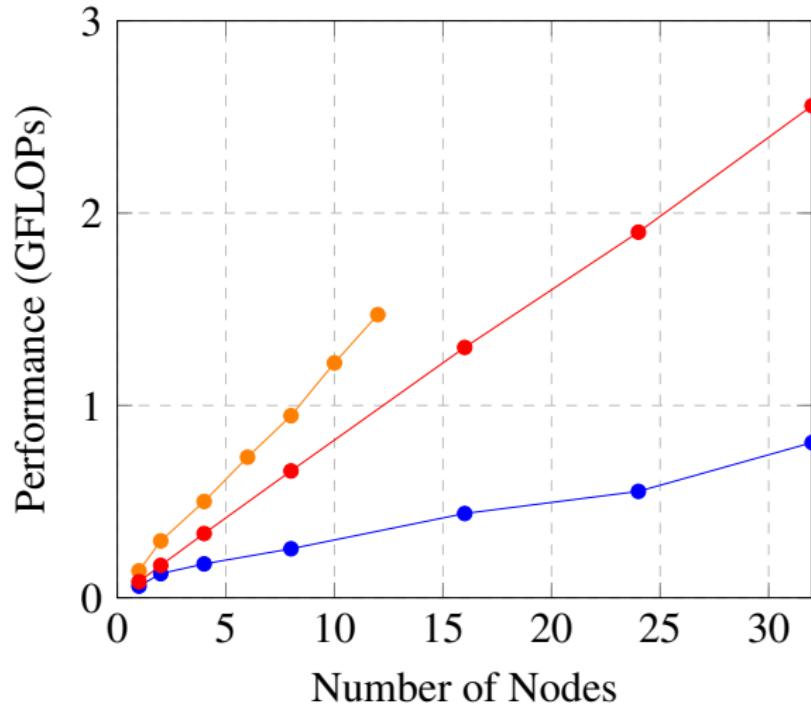
Greater emphasis on memory access speed

Intends to model more realistic workloads, not peak performance

Provides a “lower bound” to go with HPL’s “upper bound” on performance

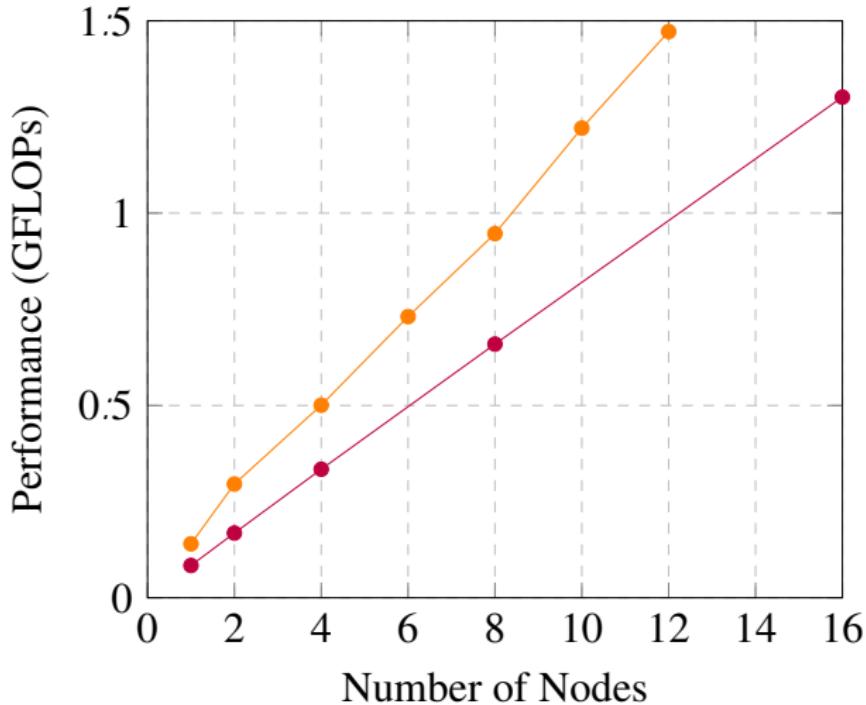
Solves $Ax = b$ with an sparse matrix conjugate gradient method

HPCG - Pine64



Blue = Pine64, Red = BOB, Orange = TX1

HPCG - nVidia TX1



Red = BOB, Orange = TX1

HPCG - Performance Summary

Pine64 yet again demonstrates poor scaling

Achieved 807 MFLOPs at 41.5% scaling efficiency

Actually a better showing than HPL - achieved 1.7% of HPL

nVidia TX1 performs somewhat better than BOB - but less so than in HPL

Achieved 1.472 GFLOPs at 87.15% scaling efficiency (1.1% of HPL)

For reference, BOB achieved 5.07949 GFLOPs at 94.5% scaling efficiency (3.4% of HPL)

nVidia TX1 CUDA Notes

Tegra X1 SoC offers very good performance for an embedded system

GPU is optimized for half-precision performance (1 TFLOP theoretical)

GPU is ill-suited for double precision tasks

CUDA should utilize zero-copy memory because the SoC shares its 4GB between the CPU and GPU already

MPI should then be able to directly access the same memory from the CPU and have good performance

Facial Recognition- Overview

OpenCV

- Software package, initiated in 1999/2000
- Objective: real-time computer vision

Process

- Detect faces in images.
- Detect faces with live video input.
- Machine learning and facial prediction.
- Parallelization.

Facial Recognition- Overview

```
cascade = "cascades/haarcascades/haarcascade_frontalface_default.xml"
c = cv2.CascadeClassifier(cascade)

def open_images(im):
    ri = cv2.imread(im)
    gi = cv2.cvtColor(ri, cv2.COLOR_BGR2GRAY)
    return gi

def detect_faces(im):
    gi = open_images(im)
    flag = cv2.CASCADE_SCALE_IMAGE
    faces = c.detectMultiScale(gi, scaleFactor=1.1, minNeighbors=10, minSize=(100, 100), flags=fl
    print "Number of detected faces: {}".format(len(faces))
    return show_image(gi, faces)

if __name__ == "__main__":
    images = ["image.jpg"]
    im = Pool(4).map(detect_faces, images)
```

Image processing internally parallelized with TBB library.
I/O data parallelism with Pool().

Facial Prediction- Overview

Process:

- Read in images. (Potentially on the order of hundreds)
- **Resize each image consistently!!!**
 - Drawback: Undefined effect on initial image resolution.
 - Solution: Minimalize resize factor.
- Separate into two numpy arrays:
 - Samples: Image vectors.
 - Labels: Assign label to each image vector for classification.
- Train algorithm(s) on image datasets.
- Predict on face using trained data.

Implemented Algorithms

- Classification/Supervised learning:
 - KNN
 - Random Forest
 - Ada Boost
 - SVM
 - Normal Bayes**

Facial Prediction- Results

Approach

```
0, Elias , 39, 40, 41, 45, 46, 47, 126, 127
1, Greg , 80, 81, 82, 83
2, Aaron , 84, 85, 87, 88, 89, 90, 91, 92
3, Jordan , 93, 94, 95, 96, 97, 98
4, Caleb , 99, 100, 101, 102, 103, 104
5, Dean , 105, 107, 108, 111
6, Parker , 112, 113, 118, 119
7, Kelley , 120, 121, 122, 123, 124, 125
```

- Image file with image ranges corresponding to each label.

Results

- Dependent on camera.
- Dependent on training data.
- Successfully predicts on high-resolution images.
- Struggles with low-resolution images.

Onto parallelization!!!!

Facial Prediction- Parallelization

Approaches

- Point-to-point communication

- Master process responsible for distributing jobs.

- Interconnect between nodes too slow.

- Overhead from constant communication.

- Dynamic process management

- Each process gets copy of data.

- Upon job completion, worker broadcasts to surrounding nodes.

- Suffers from slow interconnect speeds.

- Collective communication

- Slices data into even junks.

- Scatters among workers.

- Worker performs processing on individual chunk.

- Master process gathers all processed images.

Facial Prediction- Collective Comm.

```
if __name__ == "__main__":
    comm = MPI.COMM_WORLD
    size = comm.Get_size()
    rank = comm.Get_rank()
    samples = []
    labels = []
    fl = []
    s = []
    l = []

    if rank == 0:
        data = glob("Pics/*.jpg")
        slices = [[] for i in range(size)]
        for i, slice in enumerate(data):
            slices[i % size].append(slice)
    else:
        data = None
        slices = None

    sd = comm.scatter(slices, root=0)

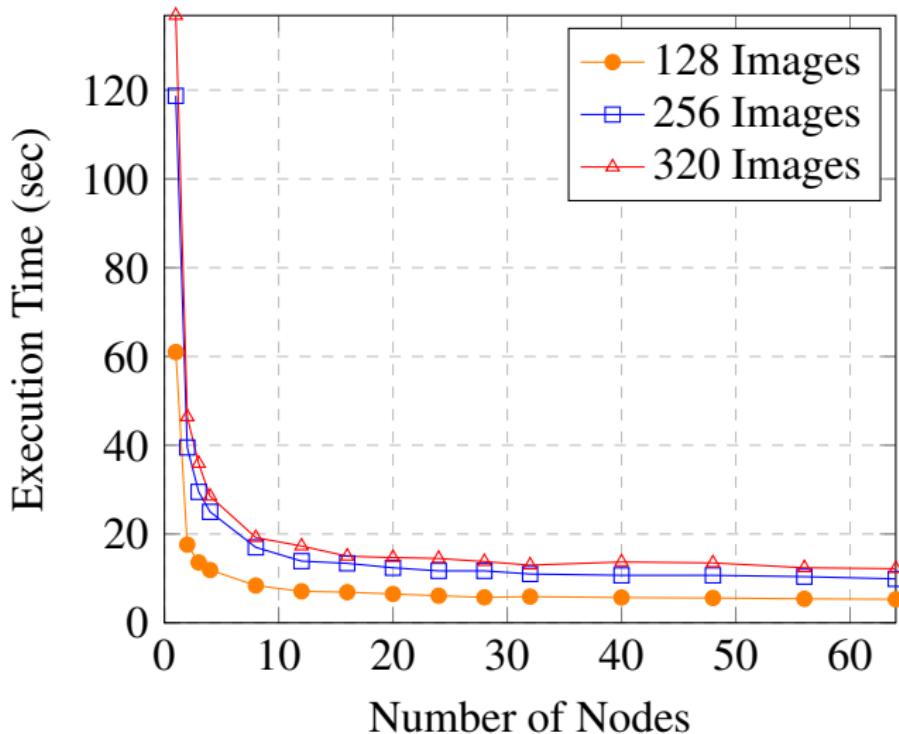
    fl = Pool(4).map(detect_faces, sd)

    labels, samples = break_lists(fl)

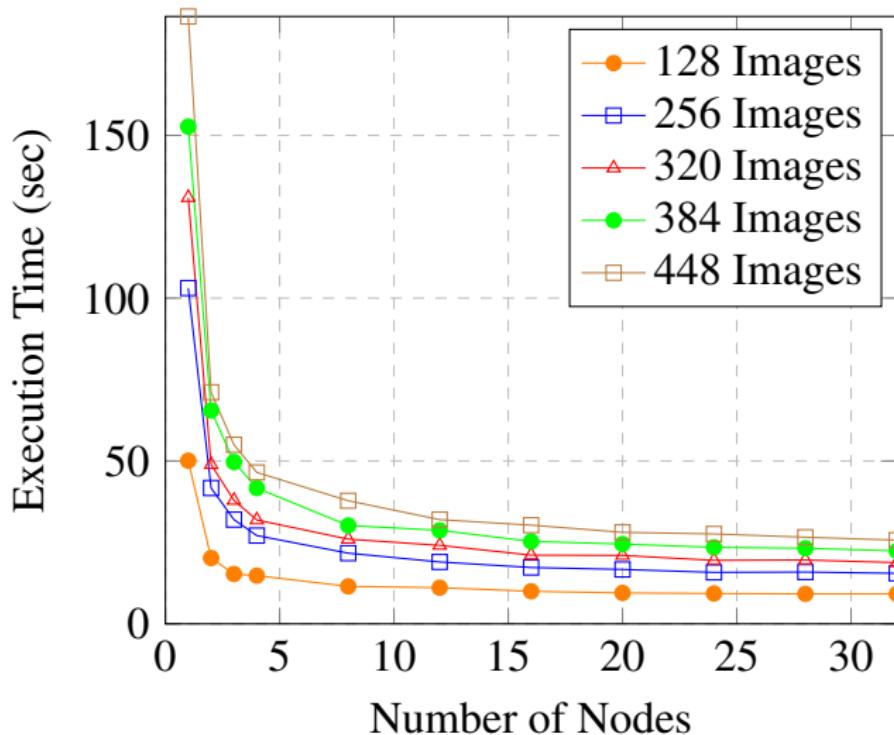
    s_data = comm.gather(list(samples), root=0)
    l_data = comm.gather(list(labels), root=0)

kill_process(rank)
```

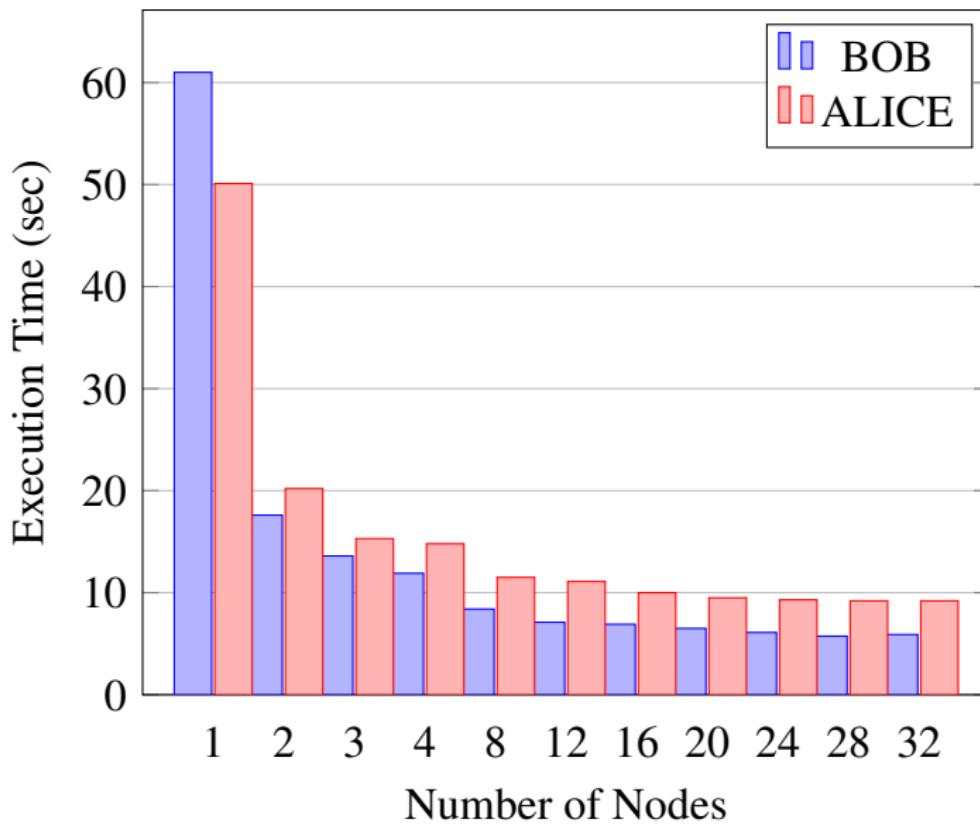
Facial Prediction- BOB Results



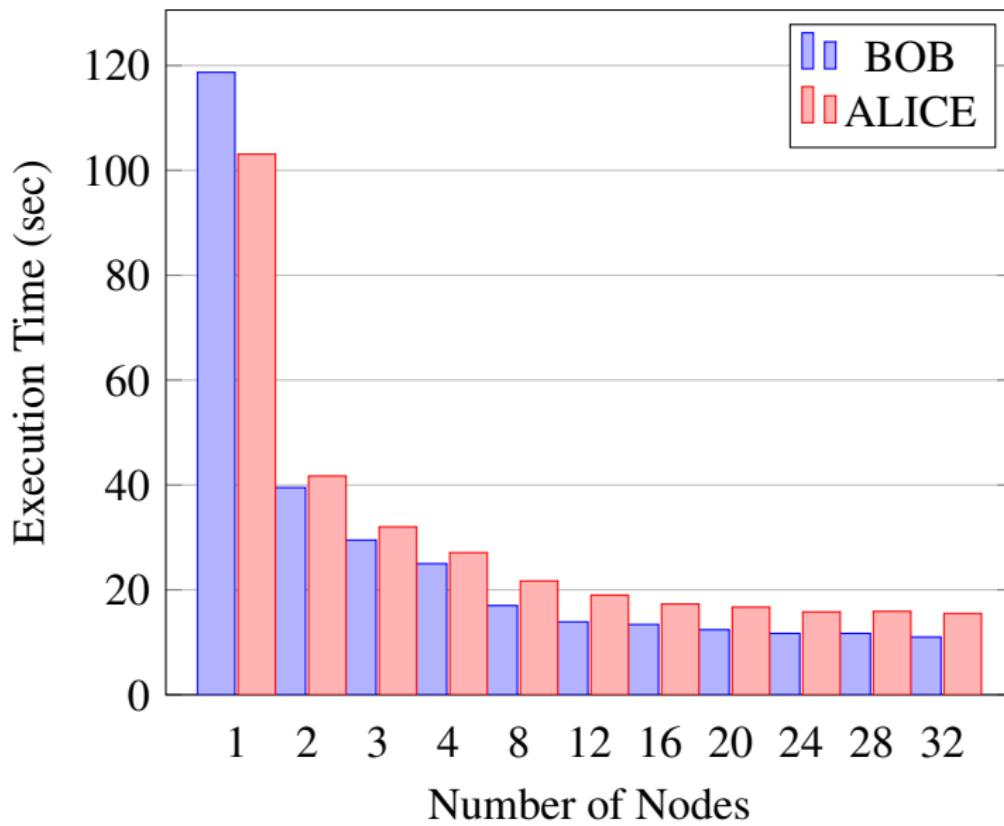
Facial Prediction- ALICE Results



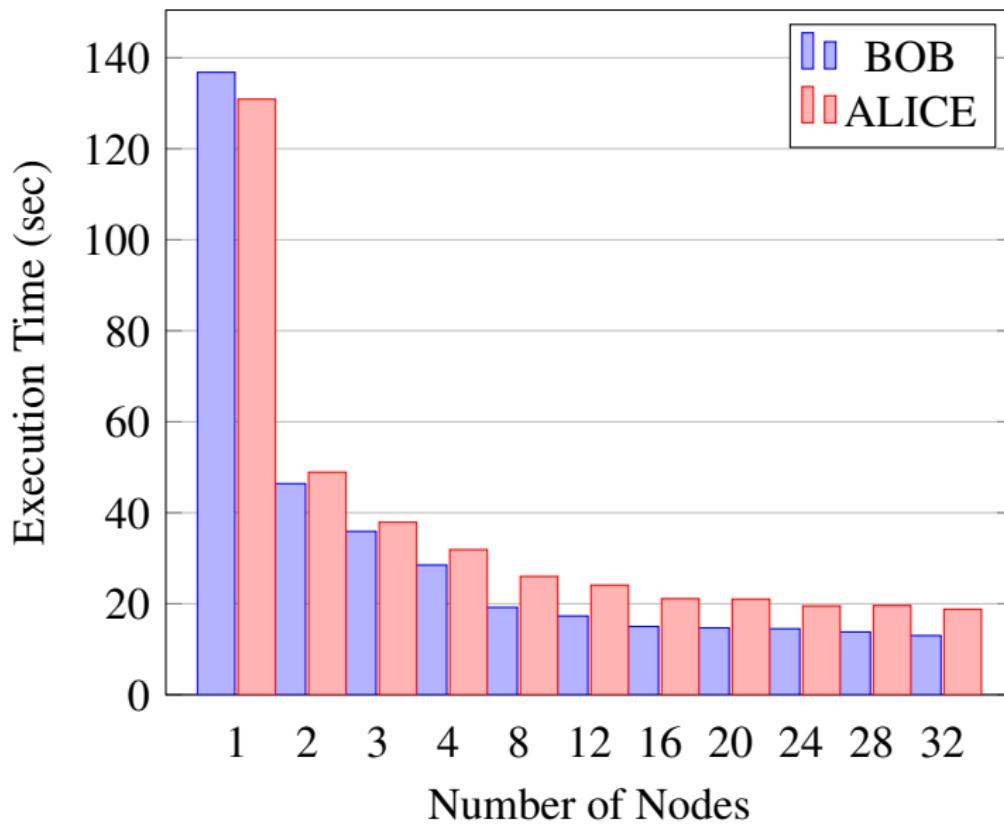
BOB vs ALICE- 128 Images



BOB vs ALICE- 256 Images



BOB vs ALICE- 320 Images



Facial Prediction- Conclusions

BOB

- Speedup across 64 nodes:
 - 128 images: 11.5x
 - 256 images: 12x
 - 320 images: 11.2x

ALICE

- Speedup across 32 nodes:
 - 128 images: 5.4x
 - 256 images: 6.7x
 - 320 images: 7x
 - 384 images: 6.8x
 - 448 images: 7.3x

FDS - Overview

Fire Dynamic Simulator is a large-eddy simulation (LES) code for low-speed flows, with an emphasis on smoke and heat transport from fires.

Takes advantage of parallel processing by dividing models up into a series of meshes

Each mesh interacts only with the meshes immediately spatially adjacent to it, monitoring things like air flow and heat transfer

It is best if one mesh is assigned to one processor, although it is possible to assign multiple meshes to a processor

FDS - Scaling Results

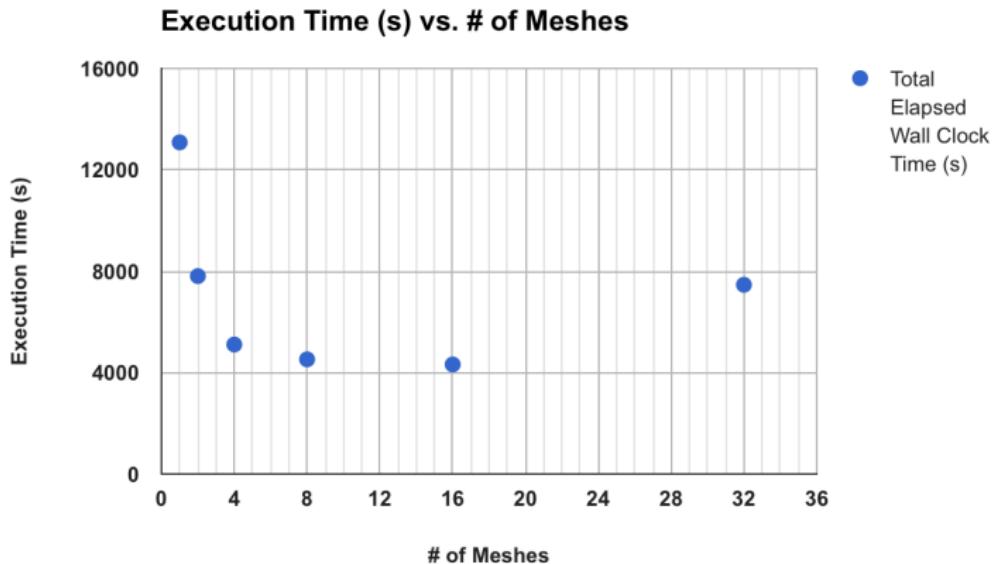


Figure 14: FDS results on BOB

FDS - Scaling Results

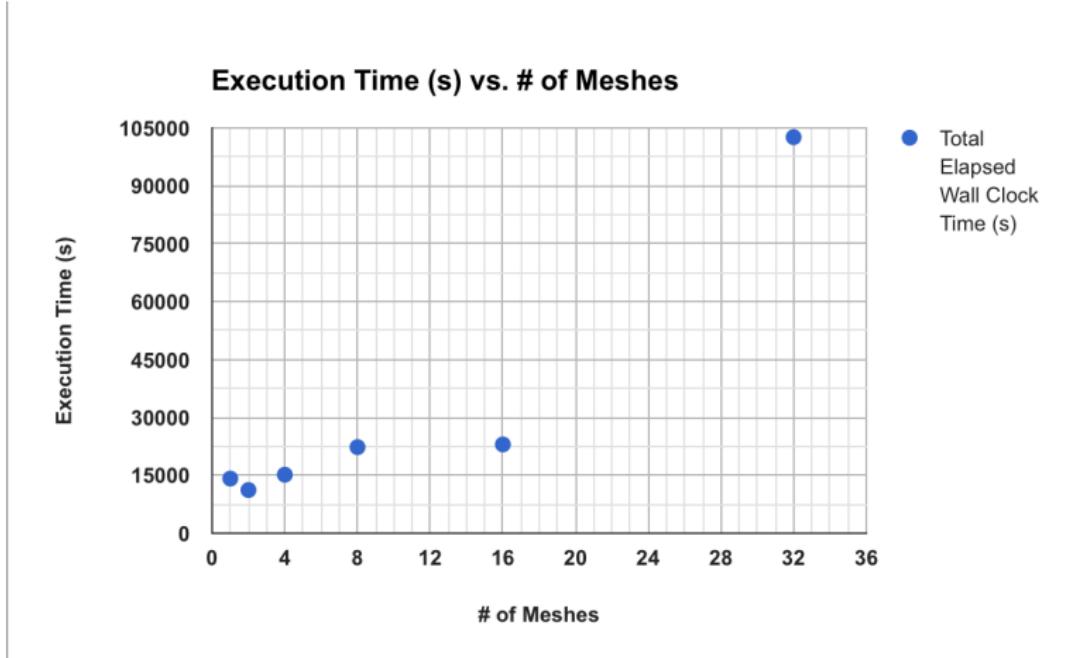


Figure 15: FDS results on ALICE

WRF - Overview

Atmospheric modeling system for weather prediction

National Center for Atmospheric Research, the National Oceanic and Atmospheric Administration, Air Force Weather Agency, and many more

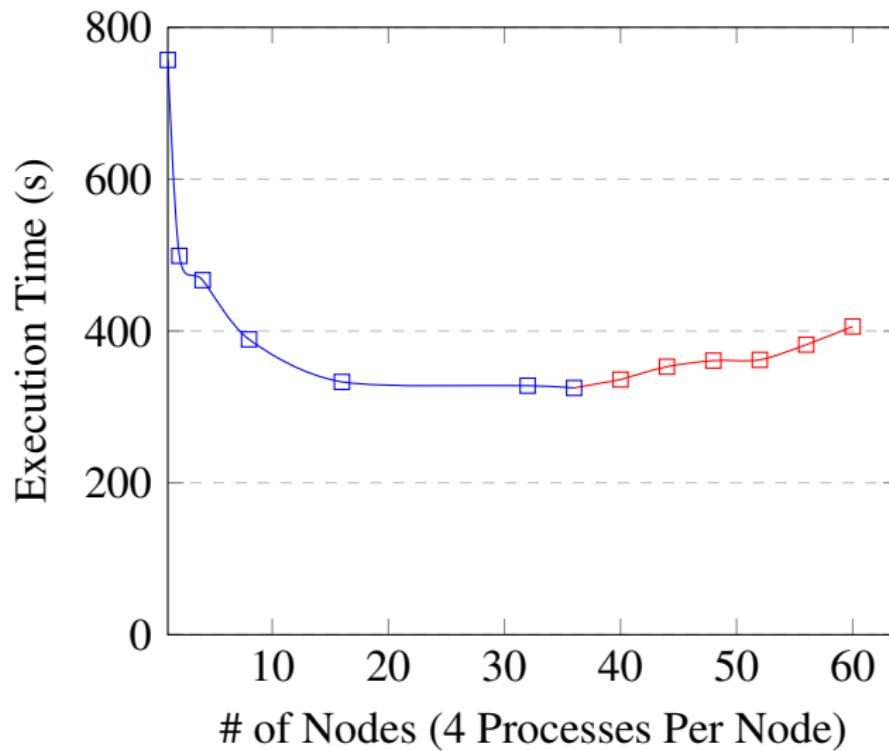
Domain is divided into grid, grid cells updated concurrently by each worker node

Primary challenges:

- compiling WRF and dependencies on BOB
- Network bottleneck when utilizing large number of worker nodes

Test case - Hurricane Katrina

WRF - Scaling Results



Gillespie Algorithm

Broadly used in chemistry, biology, and economics for stochastic simulations

Utilizes master equations and associated rates to create a time-based model of reactions in a system

Implemented with 2 Monte Carlo steps, thus is an expansion of the previous Monte Carlo application on BOB

Single simulation designated to single core

- No division of computation or parallelism
- Limited to 64 computations at any given time on BOB

Simulations can run indefinitely

Gillespie Algorithm - Steps

1. Initialization: concentration of each species and rates of reaction equations
2. Propensities: calculate the likeliness of each reaction to occur
 - A propensity is calculated by $\mu_i = C_s \cdot r_i$ for each reactant species (on left side of the arrow)
 - If the sum of propensities ever reaches 0, no more reactions can occur in the system
3. Monte Carlo - Time: time step is chosen using $\Delta t = \frac{1}{\text{total}} \ln k$ where $k \in (0; 1)$ and total is the sum of the propensities
4. Monte Carlo - Reaction: a reaction is chosen randomly, weighted by propensities
5. Update: concentrations are updated based on which reaction is chosen in previous step

Gillespie Example 1

Species

E	S	ES	P
10	11	12	5

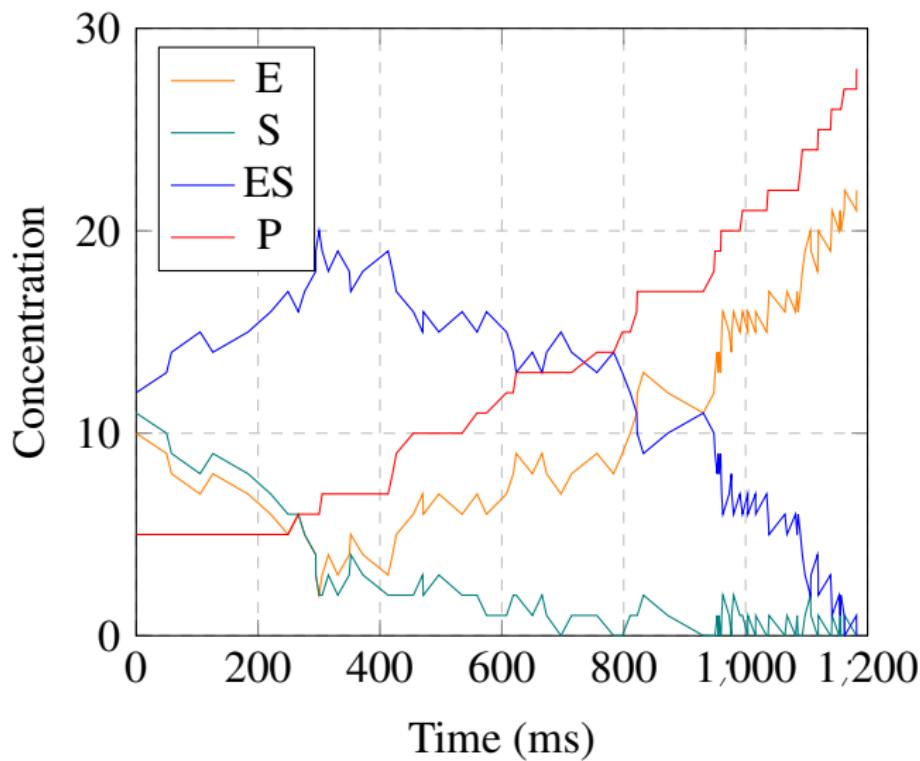
Equations

E+S !	ES	0.4
ES !	E+S	0.35
ES !	E+P	0.25

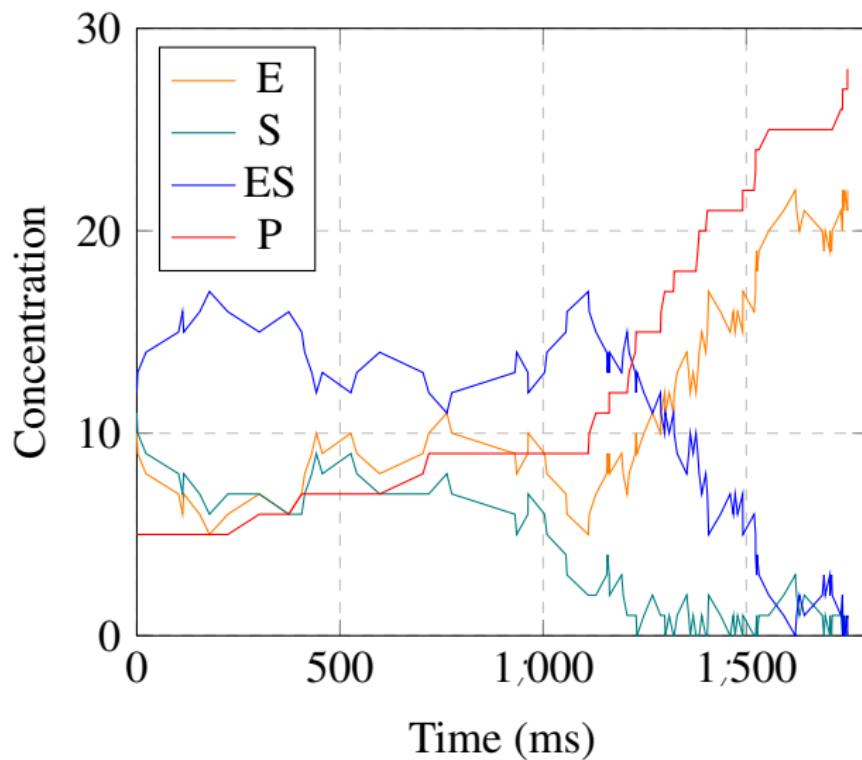
Initial Propensities

E+S /	ES	$(10 \quad 0.4) + (11 \quad 0.4) = 8:4$
ES /	E+S	$12 \quad 0.35 = 4.2$
ES /	E+P	$12 \quad 0.25 = 3$

Gillespie Example 1 Results



Gillespie Example 2 Results



Gillespie Example 3

Species

A	B	C
10	0	0

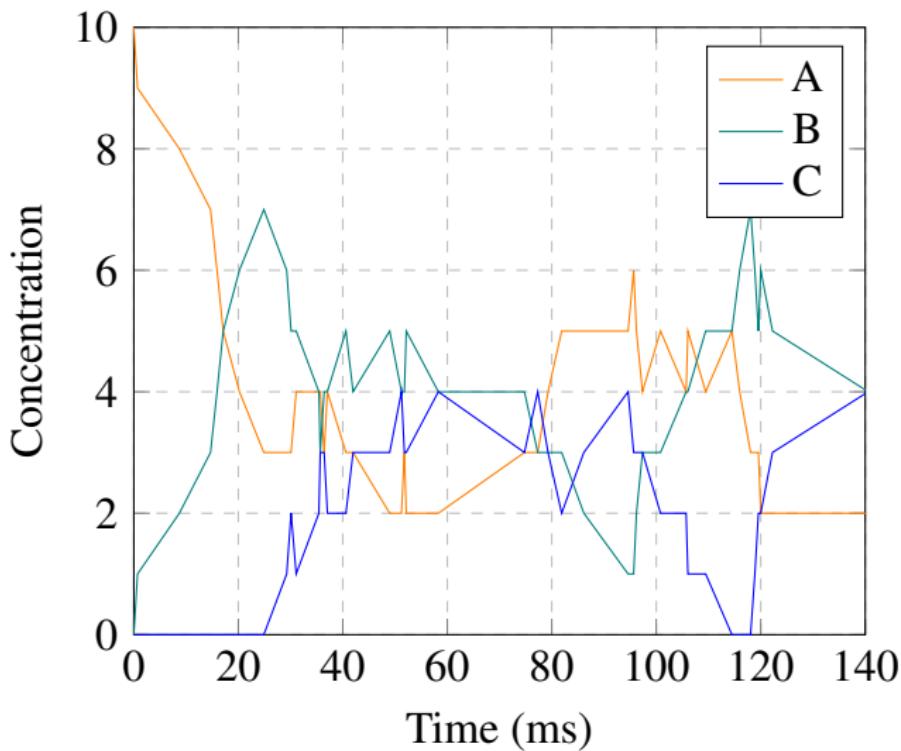
Equations

A ! B	0.33
B ! C	0.33
C ! A	0.34

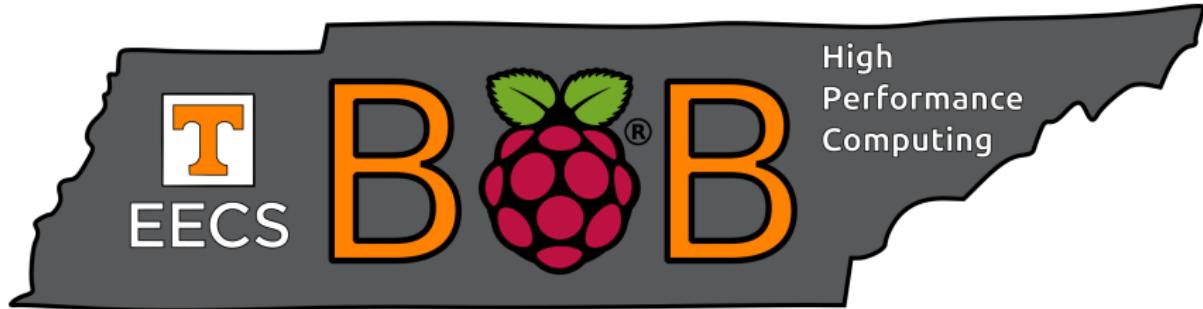
Initial Propensities

A ! B	10	0.33 = 3.3
B ! C	0	0.33 = 0
C ! A	0	0.34 = 0

Gillespie Example 3 Results



Big Orange Bramble+ BOB and ALICE



December 2, 2016