Performance and portability of abstract algebra operations in C++, Python, and Julia

Jess Woods, Ada Sedova, Oscar Hernandez

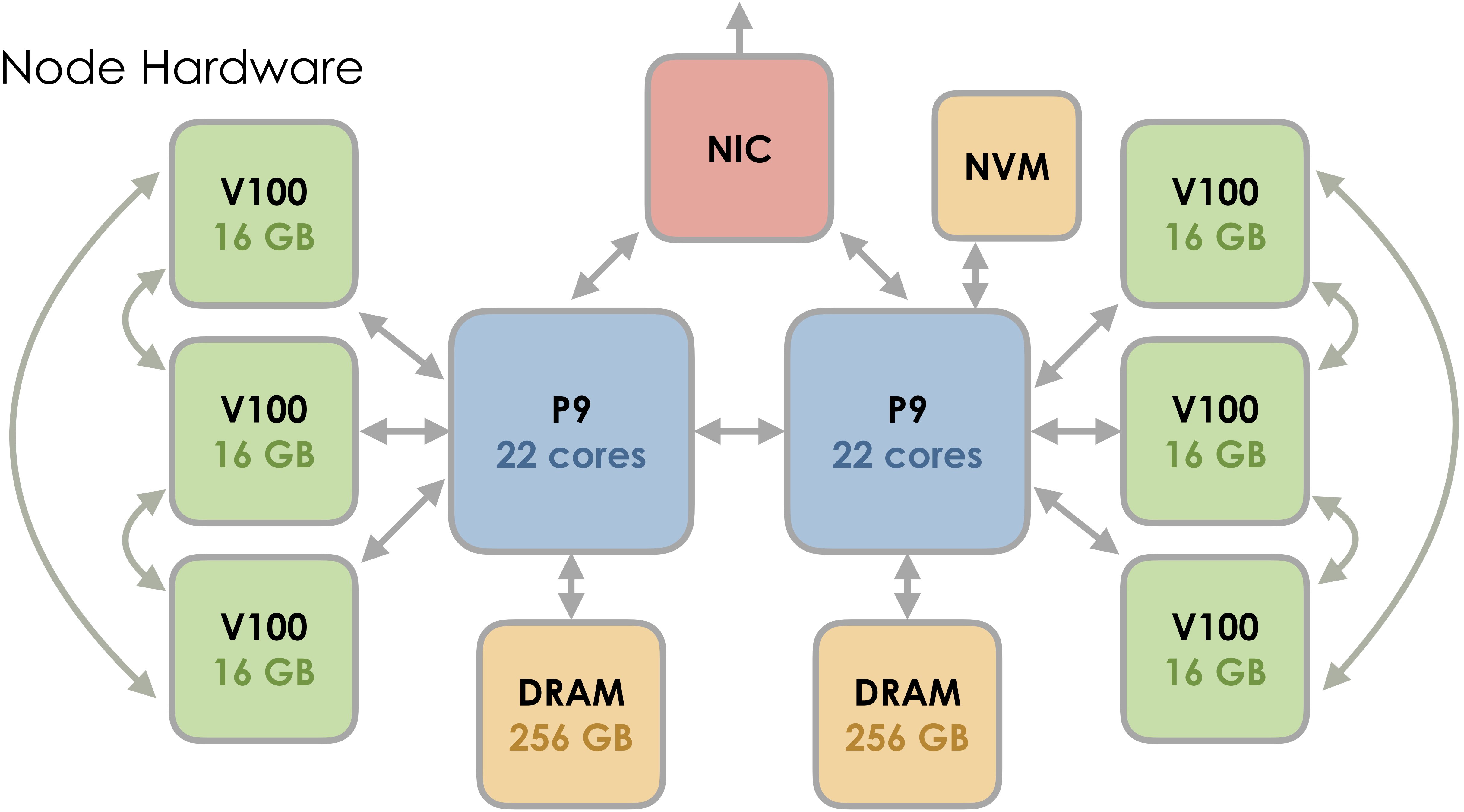
Post-Bachelors Intern

Computer Science Research Group
Computer Science and Mathematics Division

Oak Ridge, Tennessee
August 26, 2020

What is the best way to program a supercomputer?

Node Hardware



OAK RIDGE National Laboratory

Our Use Case

Library for abstract algebra operations (e.g. matrix multiplication, addition) on very big integers (up to 2^10000)

Type of Work

- Partitioning arrays of big integers
- Data parallel work
- Reducing lists in uncommon ways

Big Integer Applications

- Cosmology
- Hash tables
- Random numbers/probability simulations
- Exact precision

Our Implementations

python™

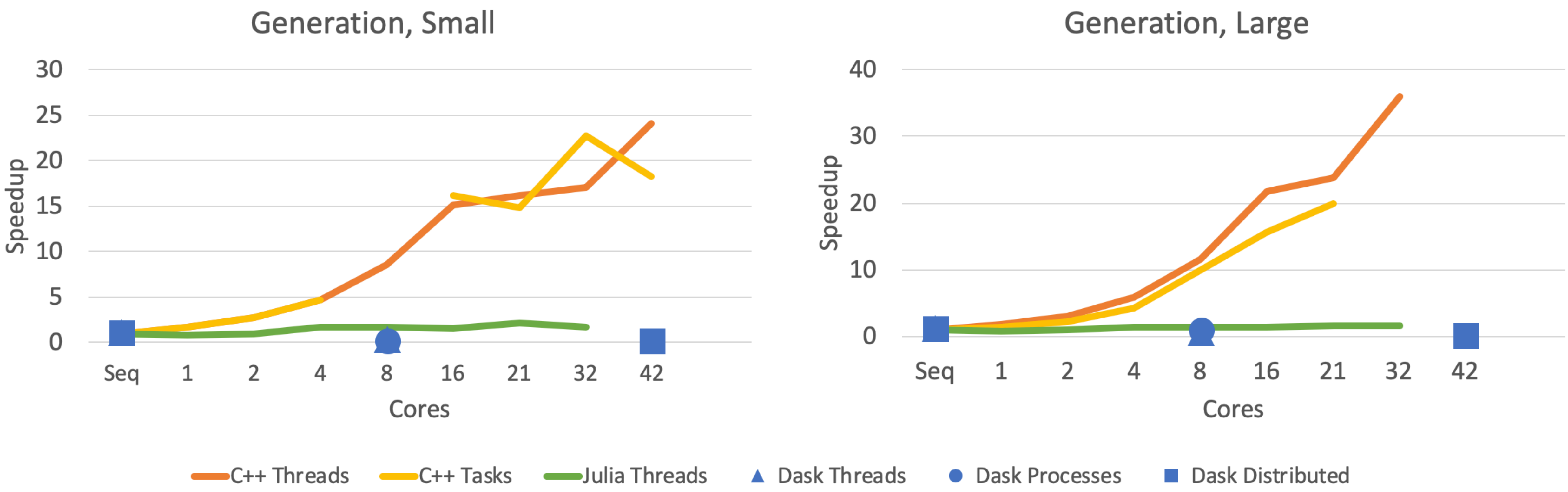
DASK

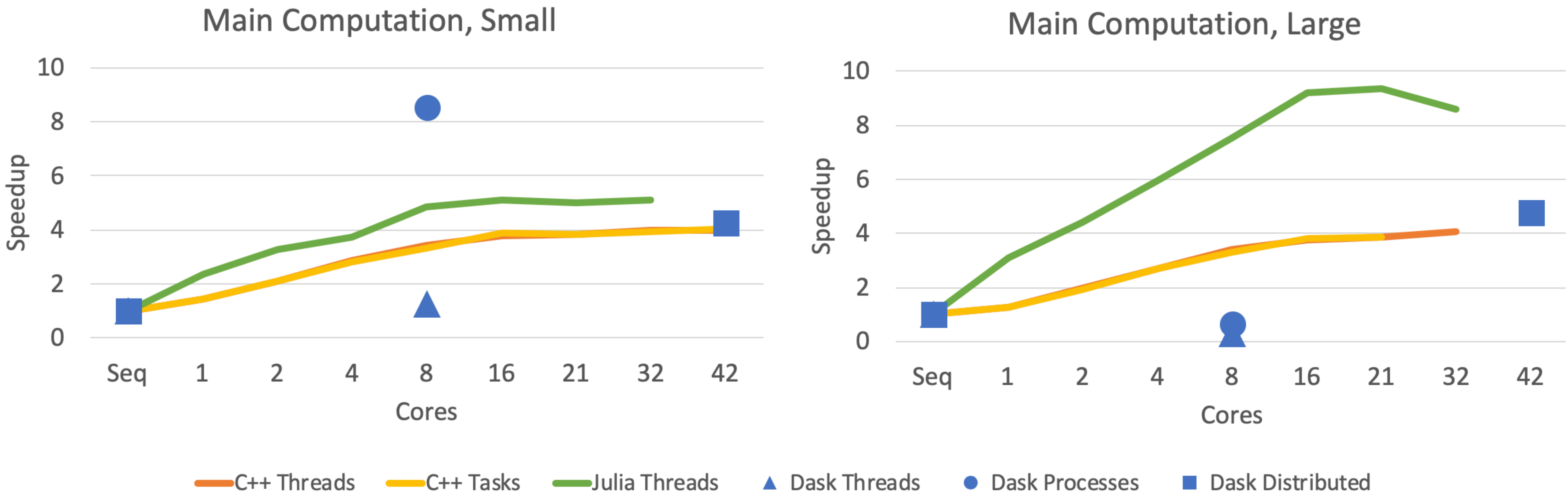
C++

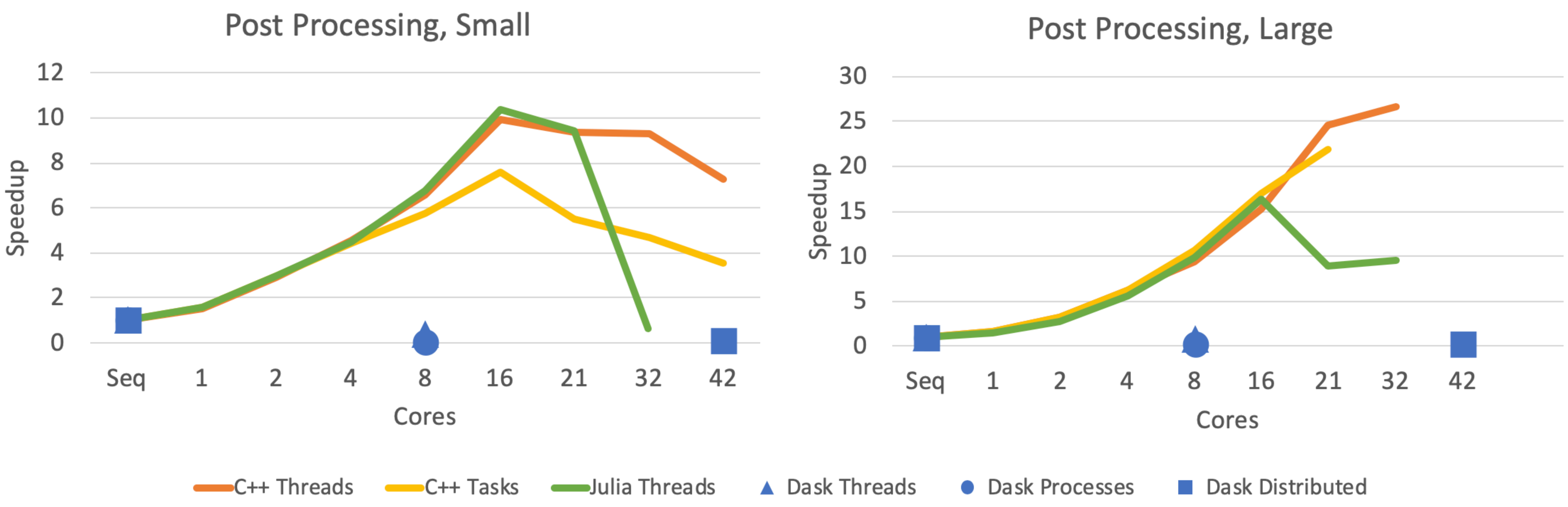
OpenMP®

julia

Performance


Generation, Small
Speedup
30
25
20
15
10
5
0
Seq
1
2
4
8
16
21
32
42
Cores
Generation, Large
Speedup
40
30
20
10
0
Seq
1
2
4
8
16
21
32
42
Cores
C++ Threads
C++ Tasks
Julia Threads
Dask Threads
Dask Processes
Dask Distributed


Main Computation, Small
Main Computation, Large
Speedup
Cores
Seq
1
2
4
8
16
21
32
42
0
2
4
6
8
10
C++ Threads
C++ Tasks
Julia Threads
Dask Threads
Dask Processes
Dask Distributed



Post Processing, Small
Speedup
12
10
8
6
4
2
0
Seq
1
2
4
8
16
21
32
42
Cores
Post Processing, Large
Speedup
30
25
20
15
10
5
0
Seq
1
2
4
8
16
21
32
42
Cores
C++ Threads
C++ Tasks
Julia Threads
Dask Threads
Dask Processes
Dask Distributed

Portability

Portability

```python
import dask
dask.config.set(scheduler='threads')
```

Python

```python
from dask.distributed import Client
if __name__ == '__main__':
    file = os.getenv('MEMBERWORK') + '/gen010/my-scheduler.json'
    client = Client(scheduler_file=file)
```

- Simple/drop-in changes for GPUs

C++

```cpp
#ifdef _OPENMP
    #include "omp.h"
#else
    #define omp_get_max_threads() 0
#endif
```

Portability Challenges

- POWER9 processors
- Unique supercomputer security, architecture
- Julia building and distribution issues
- Dask setup and troubleshooting issues

```
#!/bin/bash

# This script allows for installing the Julia programming language runtime on
# OLCF systems.
#
# Authors: Jess Woods, Matt Belhorn
#
#===============================================================================
# Notes for testing an installation:
# To use MPI/CUDA, put 'using mpi'/'using cuda' at the top of test julia script.
# Enter interactive job or batch script and use a jsrun command like:
#   ```
#   jsrun -n1 -g1 --smpiargs="-gpu" julia myprogram.jl
#   ```
#
# TODO:
# - Allow version to be set on command line, only fallback to default
#   version if specific version is not specified.
# - Capture build logs for production installations.
# - Use verbose makefiles/builds.
# - Suitesparse build does not automatically find CUDA/CUBLAS.
# - Allow for updating an existing install if the MPI and CUDA packages
#   must be rebuilt.
#===============================================================================

# ---------------------
# Set user-modifiable parameters.

TARGET_HOST="${1:-test}"
VERSION="1.4.2"

# ---------------------
# Set fixed installation parameters.

# If a host was passed at the command line, use that to construct the prefix.
JULIA_ROOT="${1:+/sw/${TARGET_HOST}/julia}"
MODULE_ROOT="${1:+/sw/${TARGET_HOST}/modulefiles/core}"

# If above root strings are null, install in an ephemeral test prefix.
JULIA_ROOT="${JULIA_ROOT:-/tmp/${USER}/opt/julia}"
MODULE_ROOT="${MODULE_ROOT:-${JULIA_ROOT}/modulefiles/core}"

PREFIX="${JULIA_ROOT}/${VERSION}"
MODULE_NAME="julia/${VERSION}"
MODULE_FILE="${MODULE_ROOT}/${MODULE_NAME}.lua"

BUILD_DIR="/tmp/${USER}/build.julia-${VERSION}.${TARGET_HOST}"
SRC_DIR="${BUILD_DIR}/julia"

# ---------------------
# Verify parameters are correct before continuing.

# Abort the install if the prefix parent dir does not already exist when not
# building a test deployment.
if [[ "${TARGET_HOST}" != "test" ]] && [ ! -d "${JULIA_ROOT%/julia}" ]; then
  echo "!=> ERROR: Directory '${JULIA_ROOT%/julia}' does not exist."
  echo "           Is the target host correct?"
  [[ "$0" = "$BASH_SOURCE" ]] && exit 1 || return 1
fi

echo "==> Installing Julia"
echo "    Target Host:     ${TARGET_HOST}"
echo "    Version:         ${VERSION}"
echo "    Prefix:          ${PREFIX}"
echo "    Module Dir:      ${MODULE_ROOT}"
echo "    Module Name:     ${MODULE_NAME}"
echo "    Modulefile:      ${MODULE_FILE}"
echo "    Source dir:      ${SRC_DIR}"
echo ""
echo "    WARNING: DO NOT RUN THIS SCRIPT UNATTENDED"
echo "             This script requires interactive input"
echo ""

read -p "Are the above values correct? (y/[n]) " -n 1 -r
echo    # (optional) move to a new line
if [[ ! $REPLY =~ ^[Yy]$ ]]; then
  # handle exits from shell or function but don't exit interactive shell
  [[ "$0" = "$BASH_SOURCE" ]] && exit 1 || return 1
fi

# ---------------------
# Perform the build

# Bailout on first error
set -e

# Setup the build environment. Use the default GCC module.
module purge
module load gcc cmake git spectrum-mpi cuda

# Capture the specific gcc module used to set as a hard dependency in the modulefile.
GCC_DEPENDS="$(module -t list gcc)"

# Setup the build directory and sources.
# MUST be built in tmp - building in home or proj causes issues
mkdir -p "${BUILD_DIR}"
BUILD_DATE="$(\date --iso-8601=minutes)"
LOG_FILE="${BUILD_DIR}/build.${BUILD_DATE}.log"

echo "=> Beginning build of Julia v${VERSION} at ${BUILD_DATE}" | tee "${LOG_FILE}"
echo "=> Build environment:" | tee -a "${LOG_FILE}"
module --redirect -t list | tee -a "${LOG_FILE}"

if [ ! -f "${PREFIX}/bin/julia" ]; then
  # Julia binary does not exist. Build and install it.
  echo "=> Fetching sources" | tee -a "${LOG_FILE}"
  if [ ! -d ${SRC_DIR} ]; then
    git clone https://github.com/JuliaLang/julia \
              --single-branch \
              -b "v${VERSION}" \
              ${SRC_DIR}

cat <<EOF > ${SRC_DIR}/Make.user
USE_BINARYBUILDER=0
GCCPATH=${OLCF_GCC_ROOT}/lib64
LDFLAGS +=  -L${OLCF_GCC_ROOT}/lib64 -Wl,-rpath,${OLCF_GCC_ROOT}/lib64
EOF

    echo "=> ...Done!" | tee -a "${LOG_FILE}"
  else
    echo "=> Sources already exist at '${SRC_DIR}'." | tee -a "${LOG_FILE}"
  fi

  # Build and install Julia
  # TODO: Build appears to use relative RPATHs by default as well as hard RPATHs
  # to the build directory. Might consider adding RPATHs to the GCC runtime libs
  # so the module does not need the same build-time GCC module loaded at runtime.
  cd "${SRC_DIR}"
  echo "=> Starting build stage." | tee -a "${LOG_FILE}"
  make VERBOSE=1 prefix=${PREFIX} -j4 | tee -a "${LOG_FILE}"

  echo "=> Starting install stage." | tee -a "${LOG_FILE}"
  make VERBOSE=1 prefix=${PREFIX} install | tee -a "${LOG_FILE}"

  # Generate the modulefile
  # FIXME - Block/prompt user if modulefile already exists before overwritting.
  # eval `make print-JULIA_VERSION`
  echo "=> Generating modulefile" | tee -a "${LOG_FILE}"
  mkdir -p "${MODULE_FILE%/*}"


cat <<EOF > "${MODULE_FILE}"
whatis("Name : julia v${VERSION}")
whatis("Short description : The Julia programming language.")
help([[The Julia programming language.]])
depends_on("${GCC_DEPENDS}")
always_load("cmake")
add_property("state","experimental")
prepend_path("PATH","${PREFIX}/bin")
EOF

  # Install base extensions.
  echo "=> Installing base extensions" | tee -a "${LOG_FILE}"
  cd "${PREFIX}/bin"
  ./julia -e 'using Pkg; Pkg.API.precompile(); Pkg.add("MPI"); Pkg.add("CUDA")' | tee -a "${LOG_FILE}"

else # Install julia if binary not in prefix
  # Julia is already installed, update base extensions.
  echo "=> Updating base extensions" | tee -a "${LOG_FILE}"
  cd "${PREFIX}/bin"
  ./julia -e 'using Pkg; Pkg.API.precompile(); Pkg.add("MPI"); Pkg.add("CUDA")' | tee -a "${LOG_FILE}"
fi

echo "==> Build finished successfully" | tee -a "${LOG_FILE}"
cp "${LOG_FILE}" "${PREFIX}/build.${BUILD_DATE}.log"
```

Julia Build

OAK RIDGE
National Laboratory

Programmability

Programmability

Python

- Everyone inside/outside CS already knows it
- High-productivity
- Requires outside libraries (Dask, sympy, gmpy2)
- Dask requires experimentation

C++

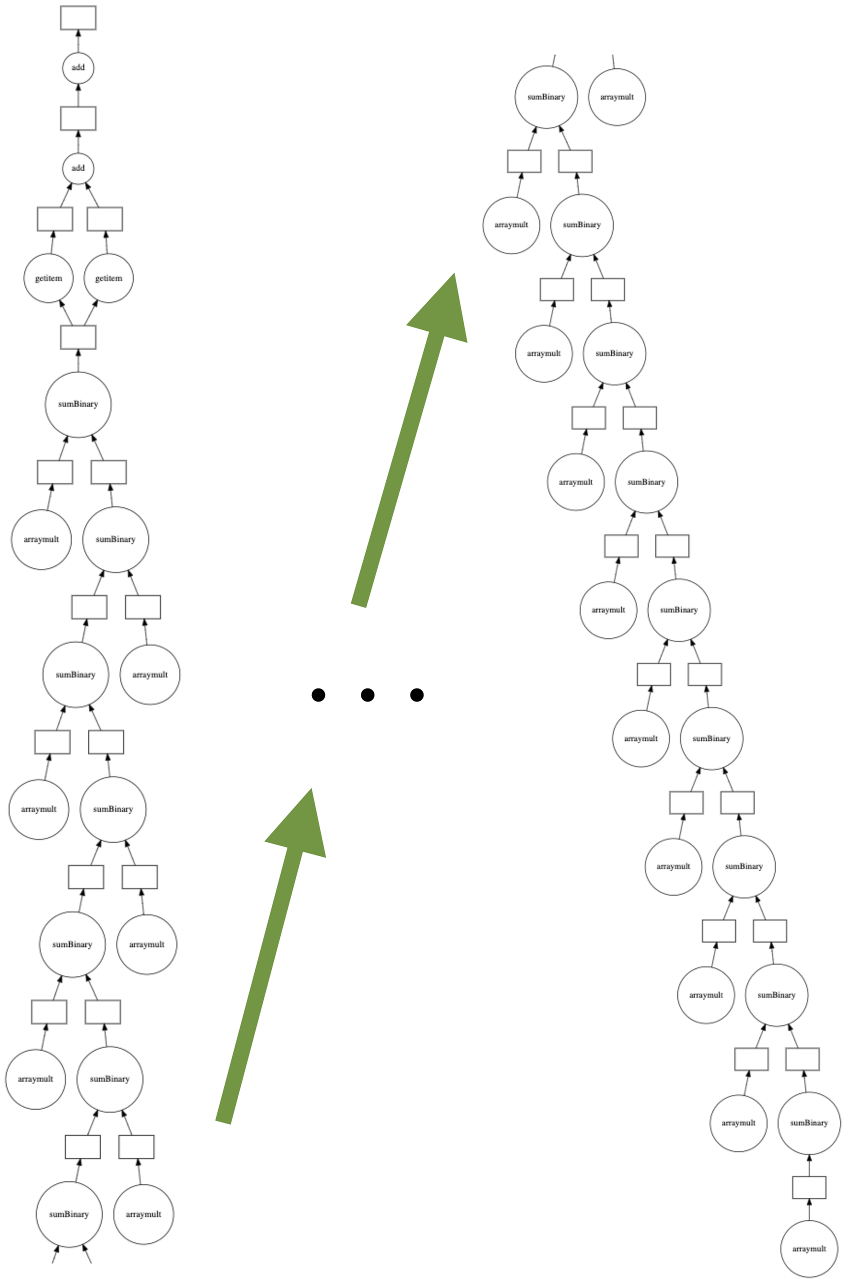
- Compiles to efficient C
- Requires CS knowledge
- Time consuming fine-tuning
- Race conditions and big number stack size issues

Julia

- New, unknown
- High-productivity
- Python like syntax
- Built-in constructs for parallelism, distribution, big number handling, and more!

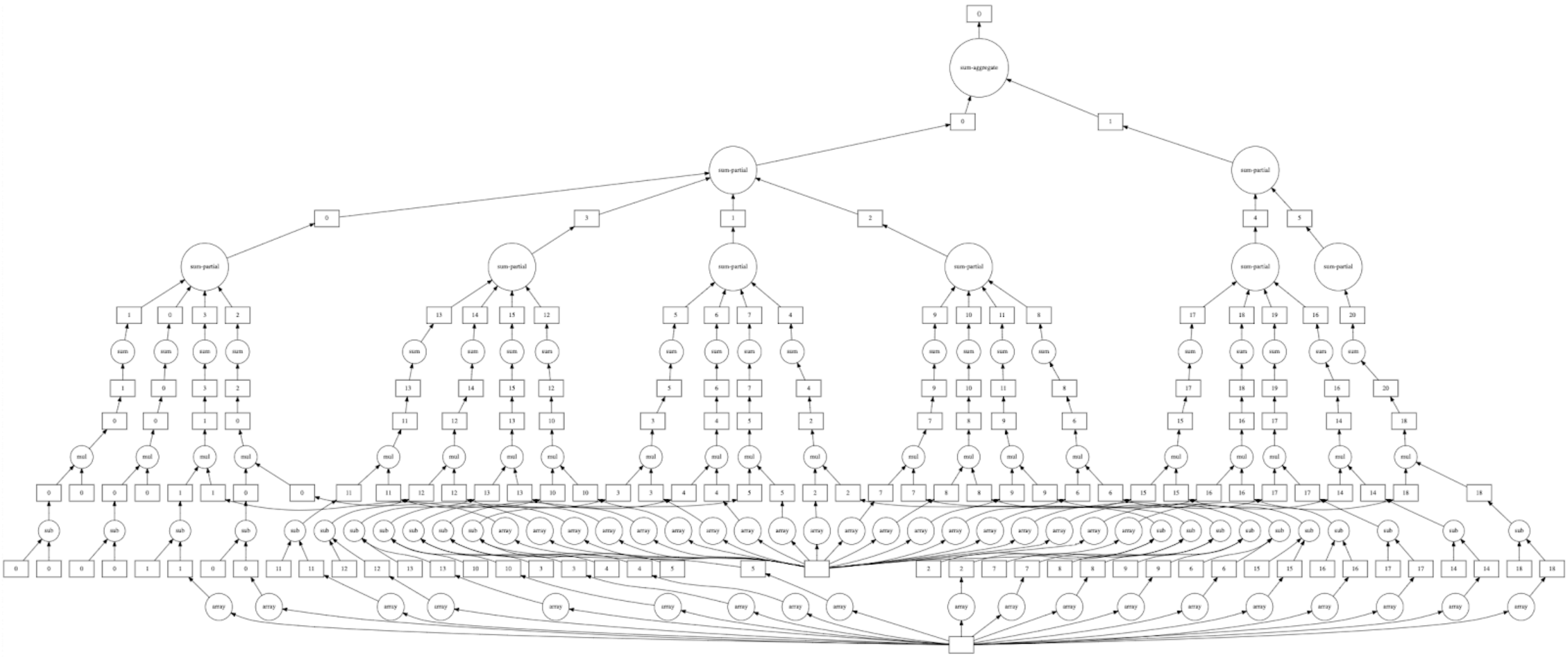
Programmability Challenges

- Holding and processing big integers
  - Outside libraries vs native structures
- How to schedule "tasks"



Inefficient Dask Task Graph

Efficient Dask Task Graph



Code Comparison

Python

```python
m_xi = [mj*xij for mj,xij in zip(m,xi)]
bi_ii = [bij*iij for bij,iij in zip(bi,ii)]
b_x = [bj*xj for bj,xj in zip(b,x)]

big_sum = sum(m_xi) + sum(bi_ii) + sum(b_x)
c = modNear(big_sum,self.x0)
return c
```

Julia

```julia
Threads.@threads for i = 1:l
    m_xi[i] = (xi_Chi[i] - xi_deltas[i])*m[i]
    bi_ii[i] = (ii_Chi[i] - ii_deltas[i])*bi[i]
end
Threads.@threads for i = 1:tau
    b_x[i] = (x_Chi[i] - x_deltas[i])*b[i]
end

big_sum::BigInt = reduce(+,m_xi) + reduce(+,b_x) + reduce(+,bi_ii)
return mod_near(big_sum,x0)
```

C++

```cpp
#pragma omp parallel
{
#pragma omp for nowait
    for (int i = 0; i < p_l; i++)
    {
        //m*xi
        m_xi[i] = m[i]*xi[i];
        //bi*ii
        mpz_class lb = power(-2,p_alphai);
        mpz_class ub = power(2,p_alphai);
        mpz_class bi = p_class_state.get_z_range(ub-lb);
        bi = bi + lb;
        bi_ii[i] = bi*ii[i];
    }

//b*x
#pragma omp for
    for (int i = 0; i < p_tau; i++)
    {
        mpz_class lb = power(-2,p_alpha);
        mpz_class ub = power(2,p_alpha);
        mpz_class b = p_class_state.get_z_range(ub-lb);
        b = b + lb;
        b_x[i] = b*x[i];
    }
} // end omp region

//summation
mpz_class big_sum = sum_array(m_xi) + sum_array(bi_ii) + sum_array(b_x);
mpz_class c = modNear(big_sum, p_x0);
return c;
```

Useful and Fun Julia Constructs

- Dynamic, high-level syntax
- JIT compilation
- Optional typing, type inference
- Simple core, easy to learn, free and open-source
- Function closures
- C and Fortran calling
- Metaprogramming
- Array broadcasting
- Built-in parallelism, distributed computing

Julia Example

```julia
function generate(array::Array{Int64,1})
  m = array .+ 1

  Multiply = function(x)
    return x .* m
  end

  Add = function(x)
    return x .+ m
  end

  return Multiply, Add

end
```

```
> array = [0,1,2]
3-element Array{Int64,1}:
 0
 1
 2

> M,A = generate(array)
(var"#5#7"{Array{Int64,1}}([1, 2, 3]), var"#6#8"
{Array{Int64,1}}([1, 2, 3]))

> M(2)
3-element Array{Int64,1}:
 2
 4
 6

> A(7)
3-element Array{Int64,1}:
  8
  9
 10

> A(M(0))
3-element Array{Int64,1}:
 1
 2
 3
```

Summary

	Python	C++	Julia
Performance	Overhead causes ~10x slow down	Excellent	Comparable to C++
Scalability	Good, Variable on different operations	Excellent, Requires fine-tuning	Excellent, Unpredictable garbage collector
Portability	One-line scheduler conversion	One-line, Requires MPI for distribution	Simple, Distributed memory requires code changes
Runs on Summit	Mostly	Yes	Yes, with comprises
Programmability	Excellent	More complicated for non-CS people	Straightforward, but new

Conclusion

- First parallel and fastest implementation
- First to incorporate both theoretical improvements
- Implementations available on github.com/jkwoods

- Python is workable
- C++ is classic
- Julia is very cool and overlooked