Big Integer Operations with Dask
Specific Application

**Problem to solve:** Library for abstract algebra operations (e.g. matrix multiplication, addition) on VERY big integers (up to $2^{10000}$) with python

**Type of Work:**
- Partitioning lists/arrays of large integers
- Data parallel work
- Reducing lists in uncommon ways

**Other uses for big integers:**
- Cosmology
- Hash tables
- Random numbers/probability simulations
- Exact precision
- Exploring large math sequences
Dask...

- Makes the program scalable for a large range of computers
- Makes already complicated abstract algebra easier to understand, prototype, and modify
- Can be “hidden” from user of larger applications/libraries
- Complex sequence of operations dealt with well by Dask task graph abilities
Architectures

- **Summit Supercomputer**
  - Per Node:
    - Two 22-core IBM POWER9 processors (4 threads per core)
    - Six NVIDIA Volta V100 accelerators
  - Tested on one, two, four nodes

- **Raptor**
  - Single node of Summit
  - Unrestricted by permissions and security firewalls
  - Tops out at 24 threads
Experiences with Dask Primitives

● **Delayed**
  ○ Good for work that has to be ‘computed’/seen often
  ○ Good for unique/fine-tuned operations
  ○ Large overhead when the data is big and must be “delayed twice”
  ○ You can’t “chunk” lists of delayed objects
  ○ Occasionally confuses scheduler if wrapped around outside library objects

● **Bags**
  ○ Good for preprocessing
  ○ Good at holding python objects (even different types)
  ○ Doing “custom” mappings and reductions
  ○ Do not preserve order, cannot be called by index
Experiences with Dask Primitives

- Arrays
  - Good for performing simple operations in parallel, comparative usability to parallel for loops
  - Figuring out best chunking and distribution takes some testing
  - Do not hold python objects (i.e. ints that don’t fit into int64) well
  - Combining/zipping arrays is limited to the blockwise() method, limited reduction ability, methods like random do not work for large numbers

- Compute/persist
  - Combined with automatic task graphs, provides excellent barrier method and ability to order and track operations
Dask Array Parallelization

Algorithm: calculation of deltas, then array multiplications, then large sum

```python
b0 = [delayed(self.rgen.random_element)(-2**self.alpha, 2**self.alpha) for i in range(self.tau)]
b10 = [delayed(self.rgen.random_element)(-2**self.alphaI, 2**self.alphaI) for i in range(self.l)]

b1 = dask.compute(*b0)
b1 = dask.compute(*b10)

b = da.from_array(b1, chunks=self.chunks)
b1 = da.from_array(b1, chunks=self.chunks)

x = da.blockwise(operator.sub, 'i', (da.from_array(self.rgen.make_prin(self.x0, self.tau, self.x_seed), \\
   chunks=self.chunks)), 'i', self.x_deltas, 'i', dtype=object)
x1 = da.blockwise(operator.sub, 'i', (da.from_array(self.rgen.make_prin(self.x0, self.l, self.xi_seed), \\
   chunks=self.chunks)), 'i', self.xi_deltas, 'i', dtype=object)
i1 = da.blockwise(operator.sub, 'i', (da.from_array(self.rgen.make_prin(self.x0, self.l, self.ii_seed), \\
   chunks=self.chunks)), 'i', self.ii_deltas, 'i', dtype=object)

m_xi = da.blockwise(operator.mul, 'i', (da.from_array(m, chunks=self.chunks)), 'i', xi, 'i', dtype=object)
b_i1 = da.blockwise(operator.mul, 'i', bi, 'i', ii, 'i', dtype=object)
b_x = da.blockwise(operator.mul, 'i', b, 'i', x, 'i', dtype=object)

big = da.sum(da.concatenate([m_xi, bi_i1, b_x]))

final = modNear(big.compute(), self.x0)
```
Efficient Task Graph
Dask Delayed Parallelization

Algorithm: rounding and truncation of large rational number into binary array representation, part of a “noise reduction” process of larger matrix multiplication

```
@dask.delayed
def frac1(u,k,c):
    return mpq(u,2**k)*c

@dask.delayed
def frac2(y):
    return mpz((y%2)*32)

@dask.delayed
def round3(z):
    return c_div(z,mpz(2))

@dask.delayed
def toBinary(x,l):
    if (x==32):
        return np.array([0]*l)
    return np.array(digits(x+2**l)[::-1])

def digits(x):
    le = list('{0:0b}'.format(x))
    le.reverse()
    return le
```

```python
y = [frac1(ui,self.kap,c) for ui in u]
z = [frac2(yi) for yi in y]
z1 = [round3(zi) for zi in z]
zbin = [toBinary(zi,self.n+1) for zi in z1]
z_comp = dask.compute(*zbin)
```
“Inefficient” Task Graph
Compensating with Dask Delayed

```python
li = [arraymult(ski, cei) for ski, cei in zip(o_comp, z_comp)]

Q_adds = [0 for i in range(self.n+1)]

for t in range(self.Theta):
    Q_adds = sumBinary(Q_adds, li[t])

rounded = Q_adds[-1] + Q_adds[-2]  #"round"

final = rounded + (c & 1)

return final.compute()
```

Algorithm: summation of many binary representations in a “schoolbook” method, with carries, etc; part of the “noise reduction” process of larger matrix multiplication

```python
@dask.delayed
def sumBinary(a, b):
    c = [a[0] + b[0]]
    carry = a[0] * b[0]

    for i in range(1, len(a) - 1):
        carry2 = (a[i] + b[i]) * carry + a[i] * b[i]
        c.append(a[i] + b[i] + carry)
        carry = carry2
    c.append(a[-1] + b[-1] + carry)

    return c

@dask.delayed
def arraymult(c, a):
    return [c * int(xi) for xi in a]
```
Experiences with Dask Scheduling

- **Single Machine Schedulers**
  - Easy to get running anywhere (except, apparently, Summit)
  - Multiple options (threads, debugging option, etc)
  - Limited use

- **Distributed Scheduler**
  - Great speed-ups from vanilla python
  - Much more complicated to get running, Summit limits certain functionalities (job queue, etc.)
  - Errors are hard to debug: things that work fine on single machine schedulers sometimes malfunction due to strange package dependencies, unreplicatable network/worker errors, etc.

- Different schedulers can be used for different parts of the code, if you have a computer that runs both

- Dashboard will not run from Summit (port blocked)
Dask Primitive Wishlist

- Fine tuning of Array.reduction method
- Expanded object handling for Arrays

or

- Ordered Bags
- Ability to ask for specific indices from bags

or

- Data Structure between those two things
- Better python object representation and operations
<table>
<thead>
<tr>
<th></th>
<th>Python/Dask</th>
<th>C++/OpenMP</th>
<th>Julia</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Overall Performance</strong></td>
<td>~10x slower than C++</td>
<td>Excellent</td>
<td>Surprisingly comparable to C++</td>
</tr>
<tr>
<td><strong>Speedup/Scalability</strong></td>
<td>Good speed up from vanilla python, though some operations do better than others</td>
<td>Requires a lot of fine work, but there ends up being much less overhead; scales the best</td>
<td>Good for less work; somewhat unpredictable due to garbage collector</td>
</tr>
<tr>
<td><strong>Programmability</strong></td>
<td>Everyone (inside and outside CS) basically already knows it</td>
<td>More complicated for non-CS people; invisible memory errors/race conditions easier to miss</td>
<td>Straightforward, much like Python; parallel programming is “built in”; but newish language</td>
</tr>
<tr>
<td><strong>Portability</strong></td>
<td>Simple conversion between sequential, single machine parallelism, and distributed machine parallelism</td>
<td>Requires MPI for distribution, otherwise easy conversion between sequential and parallel</td>
<td>Requires code changes and MPI (for now) for distributed memory on Summit; installation challenges</td>
</tr>
<tr>
<td><strong>Runs on Summit</strong></td>
<td>Mostly</td>
<td>Yes</td>
<td>Mostly</td>
</tr>
</tbody>
</table>
Benchmarks for specific multiplication operation