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?
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
Performance
Portability
Portability

Python

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

```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
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”
Efficient Dask Task Graph
Code Comparison

Python

```python
m_xi = [mj*xij for mj,xij in zip(m,xi)]
b_i = [bij*iij for bij,iij in zip(bi,ii)]
b_x = [bj*xj for bj,xj in zip(b,xi)]
big_sum = sum(m_xi) + sum(b_i) + sum(b_x)
c = modNear(big_sum,self.x0)
return c
```

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];
    }
}
```

Julia

```julia
Threads.@threads for i = 1:l
    m_xi[i] = (xi_Ch[i] - xi_deltas[i])*m[i]
    bi[ii[i] = (ii_Ch[i] - ii_deltas[i])*bi[i]
end
Threads.@threads for i = 1:tau
    b_x[i] = (x_Ch[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)
```

//summaion
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

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