Performance of MPI Codes
Written in Python with NumPy
and mpi4py

Presented by
Ross Smith, Ph.D.
14 November 2016
Outline

• Rationale / Background
• Methods
• Results
• Discussion
• Conclusion
Rationale

• Common Knowledge that Python runs slower than compiled codes
  — Anecdotes
  — Websites
  — Very little in the way of citable references

• Test Usability/Performance of NumPy/mpi4py
• Become more familiar with NumPy/mpi4py stack
• Test out new Intel Python distribution
Methods

Overview: Find and test non-matrix multiply numerical parallel algorithms in traditional compiled languages and Python. Compare the results.

- Identify software stacks
- Identify candidate algorithms
- Implementation
- Optimization (Python only)
- Testing

<table>
<thead>
<tr>
<th>Test Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>GCC + SGI MPT + OpenBLAS</td>
</tr>
<tr>
<td>GCC Cpython 3 + SGI MPT + OpenBLAS</td>
</tr>
<tr>
<td>Intel Python + IntelMPI + MKL</td>
</tr>
</tbody>
</table>
Hardware

- **Workstation - used for development and profiling**
  - Dual Socket E5-2620
  - 32 GB RAM
  - RHEL 7

- **HPC System – used for testing**
  - thunder.afrl.hpc.mil
  - SGI ICE X
  - Dual Socket E5-2699 Nodes
  - 128 GB per Node
  - FDR Infiniband LX Hypercube
Software Stacks

- **Compiled code**
  - System provided gcc/g++ (4.8.4)
  - SGI MPT 2.14 on HPC system, OpenMPI 1.10.10 on workstation

- **“Open” Python stack**
  - CPython 3.5.2 built with system provided gcc
  - NumPy 1.11.1 built against OpenBLAS 0.2.18
  - mpi4py 2.0.0 built against system provided SGI MPT (OpenMPI on workstation)

- **“Intel” Python stack**
  - Intel Python 3.5.1 built with gcc 4.8 (June 2, 2016)
  - NumPy 1.11.0 built against MKL rt-2017.0.1b1-intel_2
  - mpi4py 2.0.0 using system provided IntelMPI 5.0.3.048
Algorithm 1 – Graph 500 Benchmark 1

- [www.graph500.org](http://www.graph500.org)

- **Measure performance of:**
  - Edge list generation time
  - *Graph construction time*
  - *Distributed breadth first search*
  - Validation of BFS

- **Data-centric metric**

```
<table>
<thead>
<tr>
<th>Vertex 1</th>
<th>Vertex 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>53</td>
</tr>
<tr>
<td>28</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>17</td>
</tr>
<tr>
<td>84</td>
<td>70</td>
</tr>
<tr>
<td>62</td>
<td>23</td>
</tr>
<tr>
<td>42</td>
<td>80</td>
</tr>
<tr>
<td>16</td>
<td>35</td>
</tr>
<tr>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>36</td>
<td>74</td>
</tr>
<tr>
<td>22</td>
<td>9</td>
</tr>
<tr>
<td>53</td>
<td>44</td>
</tr>
<tr>
<td>7</td>
<td>69</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```
Algorithm 2 – Parallel Fast Correlation-Base Filter (FCBF)

- Algorithm for identifying high-value features in a large feature set
- Based on entropy
- Supervised algorithm
- Use case: High-Throughput, High-Content Cellular Analysis (Poster Session Tomorrow Evening)
- Using HDF5 for data import

Input:
- \(S(f_1, f_2, \ldots, f_N)\) // Training set
- \(C\) // Class label for each element
- \(\text{th} \) // threshold for inclusion

Output:
- \(I\) // Included features

Distribute \(S\) among ranks, each rank \(r\) receives subset \(T_r(g_{r_1}, g_{r_2}, \ldots, g_{r_M})\) such that each \(f_i\) is represented by one \(g_{r_j}\)

1. \(I = \text{empty}\)
2. \(\text{Pool}_r = \text{empty}\)
3. for each \(g_{r_j}\) in \(T_r:\)
4. \(SU_{rjc} = \text{calculate}_SU(g_{r_j}, C)\)
5. if \(SU_{rjc} > \text{th}:\)
6. Append(\(\text{Pool}_r, g_{r_j}\))
7. sort \(\text{Pool}_r\) descending by \(SU_{rjc}\)
8. \(\text{features}_\text{left} = \text{Reduce}(\text{size(\(\text{Pool}_r\))), \sum})\)
9. while \(\text{features}_\text{left} > 0:\)
10. if \(\text{size(\(\text{Pool}_r\)))} > 0:\)
11. \(g_{rq} = \text{first}(\text{Pool}_r)\)
12. \(SU_r = SU_{rqc}\)
13. else:
14. \(SU_r = 0\)
15. \(\text{hot_rank} = \text{Reduce}(SU_r, \text{index_of_max})\)
16. \(f_b = \text{Broadcast}(g_{rq}, \text{root=} \text{hot_rank})\)
17. Append(\(I, f_b\))
18. if \(r == \text{hot_rank}:\)
19. Remove(\(\text{Pool}_r, g_{rq}\))
20. for each \(g_{rq}\) in \(\text{Pool}_r:\)
21. if \(\text{calculate}_SU(g_{rj}, g_{rq}) > SU_{rjc}:\)
22. Remove(\(\text{Pool}_r, g_{rj}\))
23. \(\text{features}_\text{left} = \text{Reduce}(\text{size(\(\text{Pool}_r\))), \sum})\)
Implementations

- Use pre-existing compiled code implementations for reference
- Use NumPy for any data to be used extensively or moved via MPI
- **Graph500**
  - No option for reading in edge list from file
  - Utilized NumPy.randint() for random number generator
- **Parallel FCBF**
  - Read HDF5 file in bulk (compiled reads 1 feature at a time)
- All executables and non-system libraries resided in a subdirectory of $HOME on Lustre file system
Graph 500 Run Parameters

- Ran on 16 Nodes of Thunder
- 36 cores available per node, 32 used (Graph500 uses power of 2 ranks)
- Scale = 22, Edge Factor = 16
- Used “mpi_simple” from Reference 2.1.4 source tree
- Changed CHUNKSIZE to 13 (from 23)
Graph500 Results

Graph 500 Performance Metrics
Normalized to Compiled Results

- Open Python
- Intel Python

- Edge List Generation Time [/0.1231 s]
- Graph Construction Time [/0.279 s]
- Validation Time [/10.4 s]
- TEPS Harmonic Mean [/4.01 x 108]

Lower is better
Higher is better
FCBF Run Parameters

- $2^n$ ranks, $n = \text{range}(8)$
- Up to 4 Thunder nodes in use
  - Scatter placement
- Used sample plate from cellular analysis project
  - 11,019 features
  - 39,183 elements (cells)
  - 11,470 positive controls, 27,713 negative controls
- For Intel Python, hdf5 library and h5py were built using icc
FCBF Results: HDF5 Read Time

FCBF Data Import From HDF5 File

- Open Python
- Intel Python
- Compiled

Time [s]

MPI Ranks

1 2 4 8 16 32 64 128
FCBF Results: Binning and Sorting Time

![Bar chart showing FCBF binning and sorting time for different MPI ranks and Python versions. The chart includes bars for Open Python, Intel Python, and Compiled versions, with time measured in seconds.]
FCBF Results: Filtering Time

FCBF Filtering Time

- Open Python
- Intel Python
- Compiled

Time [s]

MPI Ranks

1 2 3 4 5 6 7 8
Discussion – Performance – Graph500

- Original Compiled run vs Modified CHUNKSIZE
  - Computational overlap

- Compiled Edge List Generation 500x faster
  - Using NumPy.random.randint()
  - Make $2^{(\text{Edge Factor} + \text{SCALE})}$ calls to RNG

- Validation closest comparison at 3.75x faster

<table>
<thead>
<tr>
<th></th>
<th>Compiled, CHUNKSIZE = $2^{23}$</th>
<th>Compiled, CHUNKSIZE = $2^{13}$</th>
<th>Open Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge List Generation Time</td>
<td>5.08 s</td>
<td>0.1231 s</td>
<td>61.5 s</td>
</tr>
<tr>
<td>Graph Construction Time</td>
<td>1.12 s</td>
<td>0.279 s</td>
<td>6.64 s</td>
</tr>
<tr>
<td>TEPS Harmonic Mean ± Harmonic Std. Dev.</td>
<td>$3.59 \times 10^8 \pm 3 \times 10^6$</td>
<td>$4.01 \times 10^8 \pm 3 \times 10^6$</td>
<td>$5.7 \times 10^6 \pm 2 \times 10^5$</td>
</tr>
<tr>
<td>Validation time ± Std. Dev.</td>
<td>215.5 ± 0.8 s</td>
<td>10.4 ± 0.5 s</td>
<td>39 ± 13 s</td>
</tr>
</tbody>
</table>
Discussion - Optimizations

- **python3** –m cProfile $MAIN $ARGS
  - Use to identify subroutines

- **kernprof** –v –l $MAIN $ARGS
  - Requires line_profiler module
  - Use to identify specific commands

- **FCBF: Entropy Calculation**
  - Class counts – Map, convert to array
  - \( P = \frac{\text{counts}}{n} \)
  - Entropy = \(-P \times \log_2(P)\)

- **Graph500: RNG**
  - Use NumPy RNG
Optimization - Inlining

- $n = 2^{18}$
- $n_{\text{trials}} = 32$
- $p_{\text{time}}()$
  - $0.045 \pm 0.006$ s
  - $\sim 17$ $\mu$s per loop iteration
- $\text{pass}_{\text{time}}()$
  - $0.0104 \pm 0.0012$ s
  - $\sim 4$ $\mu$s per loop iteration

```python
def p():
    pass

def p_time(n):
    t1 = MPI.Wtime()
    for I in range(n):
        p()
    t2 = MPI.Wtime()
    return (t2-t1)

def pass_time(n):
    t1 = MPI.Wtime()
    for i in range(n):
        pass
    t2 = MPI.Wtime()
    return (t2-t1)
```
Discussion – Lines of Code

- Python used roughly half as many lines of code
  - This occurred even with manual inlining of functions
  - Header files contribute significantly to FCBF
  - Graph500 code has lots of “unused” code, tried to not count as much as possible
    - RNG lines significantly reduced due to use of numpy.random

<table>
<thead>
<tr>
<th></th>
<th>Python</th>
<th>Compiled</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCBF</td>
<td>~520</td>
<td>~1,400</td>
</tr>
<tr>
<td>Graph500</td>
<td>~1,100</td>
<td>&gt;2,300</td>
</tr>
</tbody>
</table>
Aside – Bit Reverse

- **Used in C version of Graph500 RNG**
  - 0b11001010 => 0b01010011

- **Python results**
  - tested on $2^{18}$ dataset
  - repeat 32 times, find mean and std. deviation

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Mean time [s]</th>
<th>Std. Deviation [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reverse string</td>
<td>1.268</td>
<td>0.005</td>
</tr>
<tr>
<td>Byte swap, table lookup</td>
<td>4.427</td>
<td>0.019</td>
</tr>
<tr>
<td>Byte swap, bit swapping</td>
<td>0.915</td>
<td>0.013</td>
</tr>
<tr>
<td>Loop on bit</td>
<td>8.119</td>
<td>0.018</td>
</tr>
<tr>
<td>Loop on byte, lookup</td>
<td>2.596</td>
<td>0.005</td>
</tr>
</tbody>
</table>
Tips and Tricks

- `arr2 = np.asarray(arr1).view(datatype)`
  - Union of arr1 and arr2

- `MPI.Alltoallv(send_data, recv_data)`
  - `send_data = (send_buff, send_counts, displacements, datatype)`
  - `recv_data = (recv_buff, recv_counts, displacements, datatype)`

- `[A()] *n vs. [A() for x in range(n)]`
Conclusion

- Compiled code is faster, lots faster
- Python requires fewer lines
- H5PY does not scale well
- MPI in Python appears to scale well
- Intel Python not faster than the open stack in these tests