Ramba: High-performance Distributed Arrays in Python

Babu Pillai, Todd A. Anderson, Tim Mattson
Overview

• First, a selective review of the history of numerical processing in Python along with computational trends that continue to drive innovation today.

• Second, a comparison of where Python stands versus the standard performance solution of C/MPI.

• Third, an introduction to our new distributed array system for Python called Ramba.
History of Python Numerical Processing

• Python created in 1991, with support for collections of numbers.
• Since then, data set sizes have grown and multicore/cloud revolutions have accelerated the desire for improved multithread/multi-node performance.
• Different approaches to performance have been tried:
  • Library-based API with C implementation
    • Numeric package introduced in 1995 trying to improve performance of Python numerical processing.
    • NumPy package introduced in 2006 that supersedes Numeric.
  • JIT compilation
  • Multiprocessing and distributed execution
    • Multiprocessing package.
    • Mpi4py introduced 2009.
    • Dask introduced 2015.
    • HeAT introduced 2020.
Python Numerics

• Python has interesting standard numeric support:
  • Arbitrarily large integers
  • Floating point is similar to IEEE
  • Standard arbitrary precision library

• Very convenient, powerful, but not a perfect match to underlying processor datatypes

• Numerical vectors, arrays, matrices based on List type
  • Internally an array implementation, but not typed – each element can be an arbitrary Python object

• Loop iteration over lists (or arrays) is quite slow
NumPy – Efficient numerical arrays

• Standard package
• Adds a typed array, internally represented like C or Fortran arrays
• Standard integer, floating point, complex data types
• De facto standard way of representing numerical arrays in Python – everyone uses / builds on this (e.g., PyTorch, OpenCV, etc.)
NumPy key features

• Vector-style operations on whole arrays, map operations, reductions
  • Avoid Python iteration loops

• Powerful indexing, slicing, view generation
  • Efficiently perform operations on parts of arrays, e.g. conditional on value, etc., while writing in vector style rather than explicit loop iteration

• Efficient internal implementation of arithmetic, matrix operations
  • Written using C or based on external libraries (e.g. Intel MKL as in the Intel Python distribution)
NumPy vs Python numerics

• Take a simple example:
  • Pure Python:
    ```python
    for i in range(len(A)):
        A[i] += B[i] + s*C[i]
    ```
  • NumPy vector-style:
    ```python
    A += B + s*C
    ```

• Some speed comparisons (64-bit floats, time in seconds)

<table>
<thead>
<tr>
<th></th>
<th>1M</th>
<th>10M</th>
<th>100M</th>
<th>1B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>0.178</td>
<td>1.78</td>
<td>17.9</td>
<td>Don’t bother</td>
</tr>
<tr>
<td>NumPy</td>
<td>0.003</td>
<td>0.061</td>
<td>0.618</td>
<td>7.0</td>
</tr>
</tbody>
</table>
NumPy limitations

- Most operations are still single-threaded (few exceptions like `matmul`)
- Mapping functions still runs slow Python code
- Vector style avoids explicit loop iteration, but can hurt performance:
  - Each vector operation completes before starting next – multiple cache-inefficient traversals of large arrays
  - Large temporary arrays are materialized
- Fancy indexing, operations on views can greatly increase overheads
Just-in-Time Compilation with Numba

• Numba = NumPy + Mamba (one of the fastest snakes in the world)
• Most of Python not very amenable to compilation
  • Main problems: weak typing of function arguments, untyped container classes, arbitrary introspection and changing of classes, methods on the fly
  • However, most uses of NumPy have consistent typing, don’t typically deal with Python objects

• Key idea: Selectively apply JIT compilation techniques to programmer-selected functions; Reduce to LLVM compiler, then generate and run native binary code
Numba example

• Same example as before:
```python
@numba.njit
def my_func(A, B, C, s):
    for i in range(len(A)):
        A[i] += B[i] + s*C[i]
```

• We just need to add the decorator (@numba.njit) to mark functions that should be compiled

• Note: since this will be compiled, we don’t need to worry about slow iteration in Python; we can write explicit loops, though vector-style code will work as well
Performance

- Function is transformed into “Dispatcher” object
- First call to will be very slow – dispatcher will compile the function based on argument types provided
- Subsequent calls will (with same types) will run cached binary

<table>
<thead>
<tr>
<th></th>
<th>1M</th>
<th>10M</th>
<th>100M</th>
<th>1B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>0.178</td>
<td>1.78</td>
<td>17.9</td>
<td>Don’t bother</td>
</tr>
<tr>
<td>NumPy</td>
<td>0.003</td>
<td>0.061</td>
<td>0.618</td>
<td>7.0</td>
</tr>
<tr>
<td>Numba</td>
<td>0.0013</td>
<td>0.022</td>
<td>0.211</td>
<td>2.15</td>
</tr>
</tbody>
</table>
Exploiting multicore CPUs

• Up to now, still used just a single core
• Numba has Parallel Accelerator component that parallelizes execution across cores
  • Parallel vector-style operations
  • Explicit parallel-for construct
• Parallel example:
  ```python
  @numba.njit(parallel=True)
  def my_func(A, B, C, s):
      for i in numba.prange(len(A)):
          A[i] += B[i] + s*C[i]
  ```
Parallel Execution on Multiple Cores

• Gains from parallel execution:

<table>
<thead>
<tr>
<th></th>
<th>1M</th>
<th>10M</th>
<th>100M</th>
<th>1B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>0.178</td>
<td>1.78</td>
<td>17.9</td>
<td>Don’t bother</td>
</tr>
<tr>
<td>NumPy</td>
<td>0.003</td>
<td>0.061</td>
<td>0.618</td>
<td>7.0</td>
</tr>
<tr>
<td>Numba</td>
<td>0.0013</td>
<td>0.022</td>
<td>0.211</td>
<td>2.15</td>
</tr>
<tr>
<td>Numba-parallel</td>
<td>0.0003</td>
<td>0.006</td>
<td>0.046</td>
<td>0.43</td>
</tr>
</tbody>
</table>

• Only 4x improvement on a 20 core/40 thread machine?
  ➔ This simple example is memory bandwidth bound
Numba limitations

• Works well for NumPy arrays, but does not work with Python objects

• No per-thread control / coordination
  • Only have an implicit barrier at end of parallel for sections
  • No thread-to-thread signaling primitives; can't make your own (no “volatile” variables)
  • No NUMA-awareness for multi-socket machines

• Tricky to differentiate compile-time and dynamic values
  • E.g., array dimension length, list length are dynamic; number of dimension, tuple size are compile-time values

• Limited to single node (can scale up, but not scale out)
Dask

• DaskArray implements the NumPy API on top of the Dask distributed (or multiprocessing) tasking system.
• Arrays are broken up into chunks.
• Operations are represented on these chunks in the task graph.
• Uses NumPy internally for operations on each chunk.
• Scales to large clusters
• https://examples.dask.org/array.html
HeAT

• Distributed NumPy-like arrays with MPI.
• SPMD programming model.
• Local arrays implemented with PyTorch tensors for CPU or GPU execution.
• https://github.com/helmholtz-analytics/heat
Overview

• First, a selective review of the history of numerical processing in Python along with computational trends that continue to drive innovation today.

• Second, a comparison of where Python stands versus the standard performance solution of C+MPI.

• Third, an introduction to our new distributed array system for Python called Ramba.
Python compared to C / MPI

• As seen in this graph, the performance gap between C/MPI and various Python alternatives is still quite large.
• This gap gets larger as we go from single-node (as in the graph) to multi-node systems.
• Programmer productivity in some of the Python distributed systems not that much better than C/MPI.
Why such a large gap?

• Some systems fail to efficiently utilize multiple cores or multiple nodes.
• Some systems still use slow Python code internally.
• Some systems fail to fuse consecutive operations, causing applications to become memory bound.
• Some systems divide work into chunks and then have large scheduling or data movement overheads.
Overview

• First, a selective review of the history of numerical processing in Python along with computational trends that continue to drive innovation today.

• Second, a comparison of where Python stands versus the standard performance solution of C+MPI.

• Third, an introduction to our new distributed array system for Python called Ramba.
Ramba Idea

• Can we combine good single node efficiency with a Python distributed systems package to get efficient scale-up and scale-out while largely maintaining the programmer productivity of the NumPy API?

• Idea:
  • Combine Numba for its best-in-class Python single-node efficiency...
  • ...with Ray or MPI4Py for distribution.
Ramba Programming Methodology

• Introduce a distributed array data structure
  • Looks like a NumPy array, but is sharded across a set of nodes
• Preserve NumPy-like operations, API:
  • Basic per-element arithmetic operations
  • Simple reductions
  • Array slicing / views [but limited fancy indexing or strided view support]

• Now, can write NumPy vector-style code, and have it execute in a distributed context
• Also, provide “skeletons” that represent common computation and communication patterns (e.g., map, reduce, cumulative)
How Ramba works

• Ramba starts a set of actors on each node – these are the Ramba remote workers

• Ramba array class (“ndarray”) provides a set of methods that mimic the NumPy API
  • Ndarray construction triggers an array shard to be constructed on each remote worker; shards are NumPy arrays
  • Ndarray operations, e.g., __add__(), trigger corresponding local NumPy operations on the remote worker shards
  • Indexing/slicing an Ndarray triggers construction of a new Ndarray object, which refers to (portions of) the original shards; Thus, provide in-place views like in NumPy

• Operations on remote workers use Numba-JIT functions where possible
Ramba examples

• $A = \text{ramba.zeros((100,100))}$

• $B = \text{ramba.ones(100)}$

• $B[20:60] += 4$
Lazy Evaluation

- Ndarray operations do not immediately trigger execution on remote workers.
- Instead, the operations are added to a graph of pending operations (i.e., a DAG).
- Main thread continues to additional operations which may also be added to the DAG.
- Accessing individual element of an array or I/O operations cause Ramba to determine which operations in the DAG must be run to generate the needed output.
- Subsets of those operations which do the same amount of work per neighbor and have no data dependence conflicts are fused together, for cache efficiency, and Numba-JIT-compiled for native code performance.
- Ramba does pattern matching on the operations in the DAG to replace a series of inefficient operations with a more efficient one.
Lazy Evaluation Example

• A = B + s*C

• Ramba lazy ops:
  • All are fused into single loop
  • Temp arrays not materialized
  • Single call to workers, single traversal of arrays
  • Generated Code:

```python
@numba.njit(parallel=True)
    for index in numba.pndindex(ramba_tmp_var_00002.shape):
        ramba_tmp_var_00000 = ramba_tmp_var_00001 * ramba_tmp_var_00002[index]
        ramba_tmp_var_00003 = ramba_tmp_var_00004[index] + ramba_tmp_var_00000
        ramba_tmp_var_00005[index] += ramba_tmp_var_00003
```

• Standard Controller-Worker:
  • Do s*C, store in tmp1
  • Do B+tmp1, store in tmp2
  • Do A+=tmp2
  • 3 separate remote worker fan outs
  • Extra storage for temporaries
Lazy Partitioning

• Part of lazy evaluation is that partitioning (i.e., sharding) decisions are also delayed.

• Ramba has mechanism for operations to describe partitioning constraints on their input arrays that enable performant execution, for example:
  • Some input should not be partitioned along a certain axis.
  • Two or more input arrays should have the same partitioning along a given axis.

• At DAG execution time, Ramba determines the partitioning of the arrays to be created such that they satisfy these constraints, if possible.

• Ramba’s automatic partitioning can be overloaded with a programmer specified partitioning at array creation time.
Controller-Worker vs. SPMD model

• Ramba by default uses the Controller-Worker model
  • Main Python thread is the controller
  • Each ndarray operation adds to the DAG
  • Fusion during lazy evaluation reduces the Fan-out, Fan-in for remote execution of operations

• Single Program, Multiple Data (SPMD) model
  • Ramba will soon support SPMD natively.
  • Dominant model for HPC; Native model for MPI programs
  • Program binary executes on each node, running on its own shard of data
  • Each runs independently until explicit synchronization or communications
  • Can be much more efficient / performant than Controller-Worker
More Complex Example: stencil

- Apply stencil operation on large 2D arrays
- Value computed at $A_{i,j}$ depends on neighbor values
- Canonical distributed implementation:
  - Shard in two dimensions, share boundary data
  - Iterations of communication, computation phases

\[
B_{i,j} = A_{i-1,j} + A_{i+1,j} + A_{i,j-1} + A_{i,j+1} - A_{i,j}
\]
Stencil Example Code

• Write code like we have one big local array (NumPy vector-style)

\[
+ W[2,0] \times A[2:n-2,0:n-4] \\ \\
+ W[0,2] \times A[0:n-4,2:n-2] \\ \\
+ W[1,2] \times A[1:n-3,2:n-2] \\ \\
+ W[4,2] \times A[4:n-0,2:n-2]
\]

• All operations fuse into single function
• Shifted slices incur communications – all automated
Stencil Performance

Stencil Strong Scaling

Stencil Weak Scaling

nodes

nodes

MFlop/s

MFlop/s

numpy
rumba
dask array
C/mpi
heat

numpy
rumba
dask array
C/mpi
heat

10^6

10^5

10^4

10^3

10^6

10^5

10^4

10^3

1
2
4
8
16
32

1
2
4
8
16
32
Blackscholes Performance

Black Scholes strong scaling
1B options (500M for Dask)

Black Scholes weak scaling
1B options per node (500M per node for Dask)
Ramba Skeletons

- Ramba skeletons capture computation and communication patterns.
- Skeletons take one or more functions as input that are applied to points in an index space or define how to combine sub-results from different Ramba workers.
- The skeletons also take at least one array argument but up to any number of additional arrays or scalars.
- Supported patterns: map, reduce, stencil, cumulative, “spmd”.
- Function passed to spmd skeleton can request the portion of an input array that is resident on the current Ramba worker.
- Skeletons may also be part of the DAG of operations and the functions to execute the skeleton on each worker are Numba-compiled.
Ramba Groupby

• Ramba supports groupby functionality similar to xarray.
• A grouped array, named RambaGroupby, is formed by invoking the ndarray.groupby() method.
• This method takes the dimension to group on and a 1D array equal in length to the size of that dimension and whose contents map that point in that dimension to a group number.
• RambaGroupby supports mean, sum, prod, min, max, var, std as well as the usual numeric binary operations, add, sub, mul, etc.
• Implemented using map and reduce skeletons.
Challenges and Limitations

• NumPy API is very large – only a fraction covered now
• Optimal distribution varies with algorithm/operation; hard to guess in advance the best approach
• Some operations (e.g., reshape) not practical in distributed case
• Explicit looping through arrays will be horribly slow
• System only “sees” sequence of array operations, not actual source; hard to reason about intent, scope of variables, etc.
• Keeping overheads down is hard – communications, calculating which remotes need to exchange data, compiling deferred operations, etc. all add to overheads
Ramba Availability

• Available publicaly on Git Hub: 
  https://github.com/Python-for-HPC/ramba

• Open Source, BSD-style license

• Please try it and contribute!
References

• Ray – https://ray.io/
• Numba – https://numba.pydata.org/
• Ramba – https://github.com/Python-for-HPC/ramba
• Parallel Research Kernels – https://github.com/ParRes/Kernels
• Dask – https://dask.org/
• NumS – https://github.com/nums-project/nums