Optimizing Python code on BGQ

Steven Ufkes
(Dated: February 22, 2018)

I. MPI

To use MPI in Python, we use the Python module mpi4py. This module includes methods for passing generic Python objects, which are named using all lower-case letters (e.g. `send`, `recv`), and "buffer-like" objects, such as NumPy arrays, which are named with upper-case letters (e.g. `Send`, `Recv`). The upper-case versions are said to be faster. In my experience on the BGQ, the upper-case versions are significantly faster, even when the data being passed is small (e.g. sending a boolean value between two processes using `send` was slower than sending a NumPy array containing 0 or 1 using `Send`).

II. AUTOMATIC MULTI-THREADING

The installation of Python on the BGQ will automatically multi-thread NumPy calculations. I don't know exactly which modules must be loaded for automatic multi-threading to work, but an example of a BGQ job script which gets Python to automatically multi-thread NumPy calculations can be found at `https://github.com/sufkes/scintillometry/blob/master/toeplitz_decomp/jobscript_bgq_large.sh` ; a similar example for a BGQ debug session can be found at `https://github.com/sufkes/scintillometry/blob/master/toeplitz_decomp/jobscript_bgq_debugjob.sh`

Automatic multi-threading doesn't require any modification to the Python script you are running.

III. PROFILING

To time/profile Python code on the BGQ, I used the Python module `cProfile`. With this, a single process can save an output file which specifies how long that process spent in each routine/subroutine. This requires adding a few lines of code to save the output file, and writing a small script to interpret the output file. As far as I know, cProfile only works within a single process—in a parallel code, you can force each process to save a unique profile, and read the profiles separately; but you can't generate a single profile for all of the processes in a parallel program. There are likely more powerful profiling options, but I'm not aware of any installed on BGQ that work with Python.

In a parallel code, one typically wants to time/optimize the rate-limiting process in a given section of code. This can be difficult when using cProfile, as the rate-limiting process in a given section of code can change from iteration to iteration. In practice, I found it more useful to manually time sections of the code, synchronizing the processes before starting the timer, and after the calculation completed, as needed. For example, I used blocks of code like:

```
comm.Barrier() # synchronize all processes
<start a timer>
<do some parallel computation>
comm.Barrier()
<stop the timer>
```

I found that cProfile behaved strangely in some cases. For example, it would not detect calls to `scipy.linalg.blas` functions, and would give nonsense timing results for fast routines.

IV. MATRIX OPERATIONS

The most significant optimizations of our deconvolution routine were related to matrix operations. In general, I found that BLAS/LAPACK functions, accessed through the modules `scipy.linalg.blas` and `scipy.linalg.lapack`, performed faster than the same computations performed using NumPy array arithmetic. In some cases, the speedup was negligible; in others, a ~20 times speedup was achieved.

BLAS and LAPACK functions are optimized for column-major order arrays, whereas NumPy arrays are stored in row-major order by default. Significant speedups using BLAS/LAPACK functions were only achieved when the input arrays were column-major contiguous. This can be achieved in two ways: (1) force NumPy to store arrays in column-major order using the `order='F'` option; or (2) store NumPy arrays in row-major order, and use the transposes of these arrays as input to the BLAS/LAPACK functions. This is usually easy to do. For example, if you have row-major contiguous arrays, and need to do the matrix operation:

$$A \leftarrow BC,$$

You can perform the equivalent operation

$$A^T \leftarrow C^T B^T,$$

in which case the input arrays are column-major contiguous. Transposing NumPy arrays is negligibly fast, while converting a given array from row-major order to column-major order (without transposing) is relatively slow. I believe that the better performance for column-major contiguous arrays is related to how the CPU loads chunks of data into the cache.

For cases in which matrix operations could be performed using column-major contiguous arrays and BLAS/LAPACK functions, we typically achieved speeds close to the advertised 204.8 GFlops/node. For example, I tested complex matrix multiplication: multiplying an $n \times m$ complex matrix with an $m \times p$ complex matrix requires $8nmp$ FLOPs.

A general tip for matrix operations is to avoid explicit computation of matrix inverses, as they are very rarely needed, the calculation is numerically unstable, and you can often save some computation time by avoiding it. For example, we had sections of code like:

```
A_inverse = np.linalg.inv(A)
X = A_inverse.dot(B),
```

which were sped up and stabilized by changing to:

```
<solve AX=B for X using a BLAS function>
```