CS 267 Final Report
Reproducible Parallel Matrix-Vector Multiply

Willow Ahrens - 23367285

April 30, 2024

1 Introduction

Parallel code can be difficult to verify due to inherently non-reproducible execution models. When debugging or writing tests, users could benefit from getting the same result on different runs of the simulation. This is the goal that the ReproBLAS project (Nguyen et al.) intends to achieve. ReproBLAS [3] has so far introduced a reproducible floating point type (indexed float) and associated algorithms underlying BLAS1 serial and parallel routines. A long-term goal of ReproBLAS is, for example, the implementation of a fully-featured reproducible PBLAS [6].

ReproBLAS has yet to define a formal interface for its underlying algorithms, and it is as yet unknown whether the existing low-level routines can be assembled together to form reasonably efficient higher-level routines. Here, we implement a reproducible matrix-vector multiply in order to gauge the feasibility of and identify challenges in building a more complex reproducible linear algebra library.

1.1 Definitions

Given a set E, a **binary operation** on E is a function $\cdot$ mapping $E \times E$ to E. For example, the operation of addition is a binary operation mapping any pair of real numbers to a real number, their sum. A binary operation $\cdot$ is **associative** if for any inputs $a, b, c \in E$, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.

Because most sources of non-reproducibility in simple programs tend to arise from the non-associativity of the floating-point addition defined by the IEEE standard for Floating-Point Arithmetic (IEEE-754)[2], the definition of reproducibility used throughout the paper will center only around addition.

Given a function f and a procedure p that implements that function, p is defined to be **completely reproducible** if, for any two inputs (equivalent up to permutation) to f that produce equivalent output given only the associativity of addition, p produces equivalent output.

A parallel procedure p is said to be **block reproducible** if it produces the same output given the same input on each processor.

Note that the criterion for complete reproducibility is much stronger than block reproducibility, and that complete reproducibility not only implies block reproducibility, but implies the stronger property that the output of the procedure is the same given any data partitioning across processors.

1.2 Linear Algebra Software

If the reader is already familiar with the BLAS, BLACS, and PBLAS libraries, they are advised to jump to section 2.

1.2.1 BLAS

An acronym for Basic Linear Algebra Subroutines, this is the widely accepted interface for core linear algebra operations. The BLAS are at the core of LAPACK, a standard linear algebra library that defines higher-level linear algebra operations. [4]. BLAS defines three sets of operations:

- The BLAS1 deal with $O(n)$ operations on $O(n)$ data, and include operations for summation of vectors, norm of vectors, etc.
- The BLAS2 deal with $O(n^2)$ operations on $O(n^2)$ data, and include operations for matrix-vector multiplication and solving $Ax = b$, where A is a matrix and x and b are vectors.
- The BLAS3 deal with $O(n^3)$ operations on $O(n^3)$ data, and include operations for matrix-matrix multiplication and solving $AX = B$, where A, B, and X are matrices.

1.2.2 BLACS

An acronym for Basic Linear Algebra Communication Subroutines, this is a linear-algebra oriented communication layer built on top of existing distributed memory communication interfaces. The BLACS are used as the underlying communication layer for the PBLAS and ScaLAPACK. The BLACS define methods to send and receive vectors and matrices of real or complex single or double precision floating point values. Notably, the BLACS also define a sum reduction method. This is the only source of non-reproducibility in the BLACS. [5].

1.2.3 PBLAS

The PBLAS are the distributed memory parallel versions of the BLAS, and form the core of ScaLAPACK, the standard parallel linear algebra library. The PBLAS rely on both the BLACS and the BLAS. [6].

2 Context

Before starting work on this project, ReproBLAS defined the indexed floating point type. The indexed float represents a real number and is associative under addition. In short, this is achieved by breaking up each floating point number into components belonging to predefined sections of the exponent range. Within these bins, numbers may be added as integers, associatively. The indexed type represents k consecutive bins along the exponent range, and the underlying representation of an indexed floating point type is an array of several floating point numbers. When two indexed types with bins of differing magnitude are added, the higher magnitude bins are always used, leading to the same result regardless of the ordering of the input data. For a rigorous explanation of the indexed summation algorithm, see [3].

It should be said here that a reproducible library does not mean that the code using it will be reproducible. For example, if a user reproducibly sums each third of a vector separately, then reproducibly sums the sums of thirds, the final result will not be reproducible if the input vector is permuted. The grouping of elements in each third will be different, and the output (non-reproducible) types will be different.

ReproBLAS also previously defined BLAS1 methods for reproducible summation, norms, and dot products. These routines were heavily optimized and used both SSE and AVX vector intrinsics. However, the ReproBLAS library did not have a standardized interface across the library. As a part of this project a standard interface was defined for the indexed types and sequential routines, as such an interface would be necessary when building higher-order routines. The interface redesign will not be mentioned here to maintain focus on the performance aspects of the project.

The goals of this project are to determine what work must be done in order to build a completely reproducible sequential BLAS, a block reproducible PBLAS, and a completely reproducible PBLAS, all without sacrificing too much performance. Matrix-vector multiply was chosen as a test problem

because it is relatively easy to understand and implement, yet it involves $O(n^2)$ data and operations, including a possible reduction, which makes it representative of the higher-level BLAS routines.

3 Matrix-Vector Multiply

3.1 Problem Statement

Matrix-Vector Multiply is a routine that computes the product of a matrix and a vector. Given an M by N matrix A (M rows, N columns, where $A_{i,j}$ is the element in the i^{th} row and j^{th} column), and a vector X of length N, the problem is to compute the vector Y where $Y_i = \sum_{j=0}^{N} X_j * A_{i,j}$.

3.2 Checking Reproducibility

If the additions were performed associatively, then we would in theory be able to permute the input columns of A together with entries of X and achieve the same Y. Tests were implemented to perform this check for both parallel and sequential matrix vector products. To be more specific, under any permutation σ (a bijection from $\{1, ..., N\}$ to $\{1, ..., N\}$),

$$\sum_{j=0}^{N} X_j * A_{i,j} = \sum_{j=0}^{N} X_{\sigma(j)} * A_{i,\sigma(j)}$$

3.3 Implementation

Matrix-Vector Multiply was implemented for double precision numbers.

Matrix-Vector Multiply can be created naively from dot products, but the problem with this approach is that it is not memory aware, and caching effects slow down the final product. The solution for the reproducible routine is to block the dot products. Because the intermediate values of Y must be stored reproducibly, Y is computed first as an array of reproducible types, then converted into normal floating-point numbers. Tiles of the input matrix are processed sequentially.

To parallelize Matrix-Vector Multiply, one can break up processors to process rows or columns of A, or both. **Because processing rows of A in parallel can be accomplished trivially (all rows are independent**

and output elements can be computed independently), the implementation discussed here processes only columns of A in parallel to better explore the reduction of floating point numbers. It was known beforehand that this would negatively impact performance. This was done to save time, as the complexity of code is increased without the use of a communication library. If the Matrix-Vector Multiply were implemented in parallel as library code, a communication library for indexed types would be built and used. In practice, the computation would be made parallel across rows and columns.

4 Results

Throughout this section, it should be noted that the reproducible dot and matrix vector product require 7 floating-point additions, 1 floating point multiplication, and 3 bitwise-or operations per element in the input vector and matrix respectively. All results have been obtained on NERSC's Hopper computer [1]. This computer uses a twelve-core AMD 'MagnyCours' 2.1-GHz cpu, supporting Intel's SSE instruction set, which allows it to do two double-precision operations with one instruction. It was found that this machine cannot do bitwise-or operations in parallel with additions or multiplications, so they are counted as additional flops. It should also be noted that in the reproducible algorithm, additions cannot be fused with multiplications, so they are counted separately. Therefore, peak performance is calculated by multiplying the cpu instructions per second by 2 (accounting for SSE), and all 3 of the previously mentioned floating point operations are assumed to be run separately.

To give a basic understanding of the underlying speed of the reproducible routines, figure 1 is a graph of the performance of the completely reproducible dot product and the vendor dot product on different vector sizes. Because the indexed routine needs to perform 7 floating point operations and 3 bitwise-or operations per addition, it is clearly more computationally intensive than the vendor dot product, and runs slower. However, the gap between the two is only a constant factor, and figure 2 shows that the indexed routine is running close to peak.

It should be noted from figure 2 that the reproducible dot product performs most efficiently on longer vector sizes. This effect propagates to the reproducible matrix-vector product, as shown in figure 3, which compares

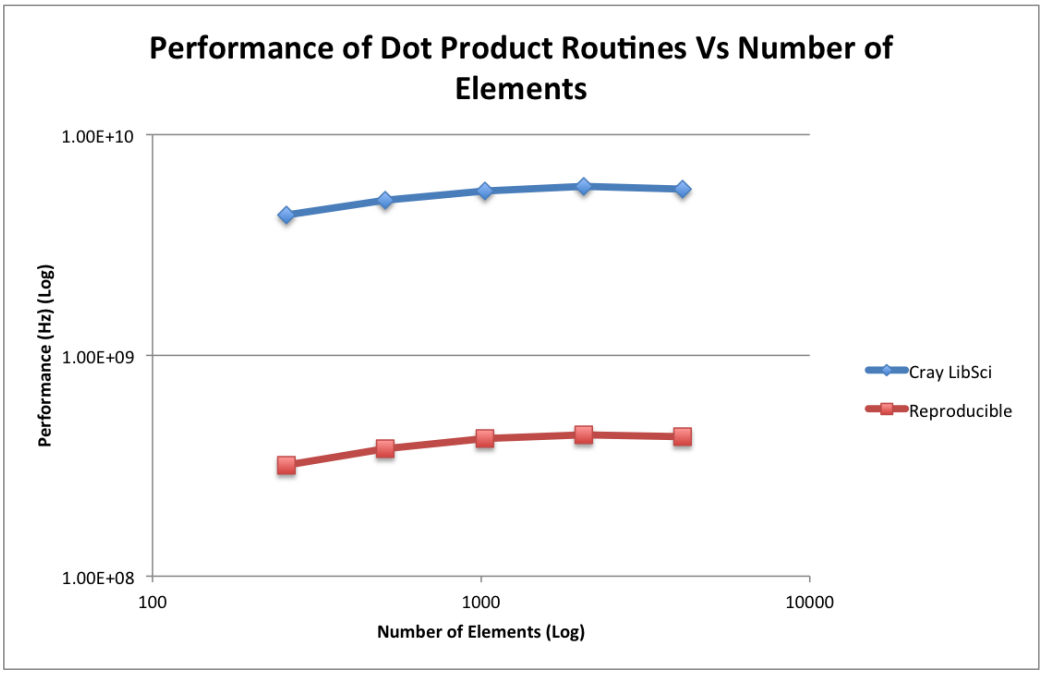

Figure 1:

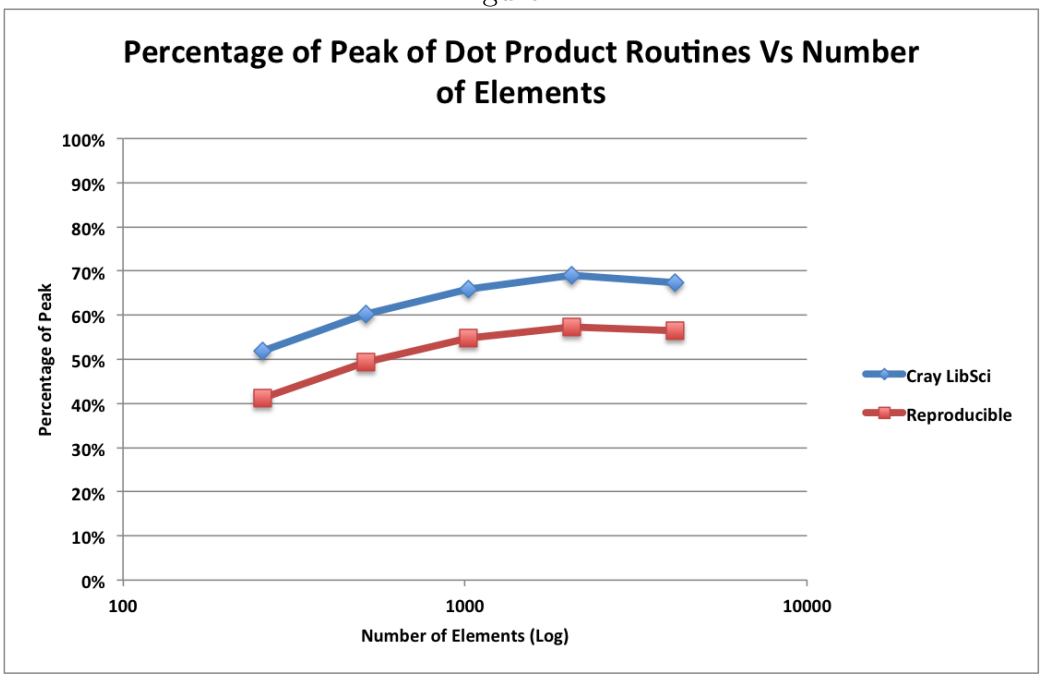

Figure 2:

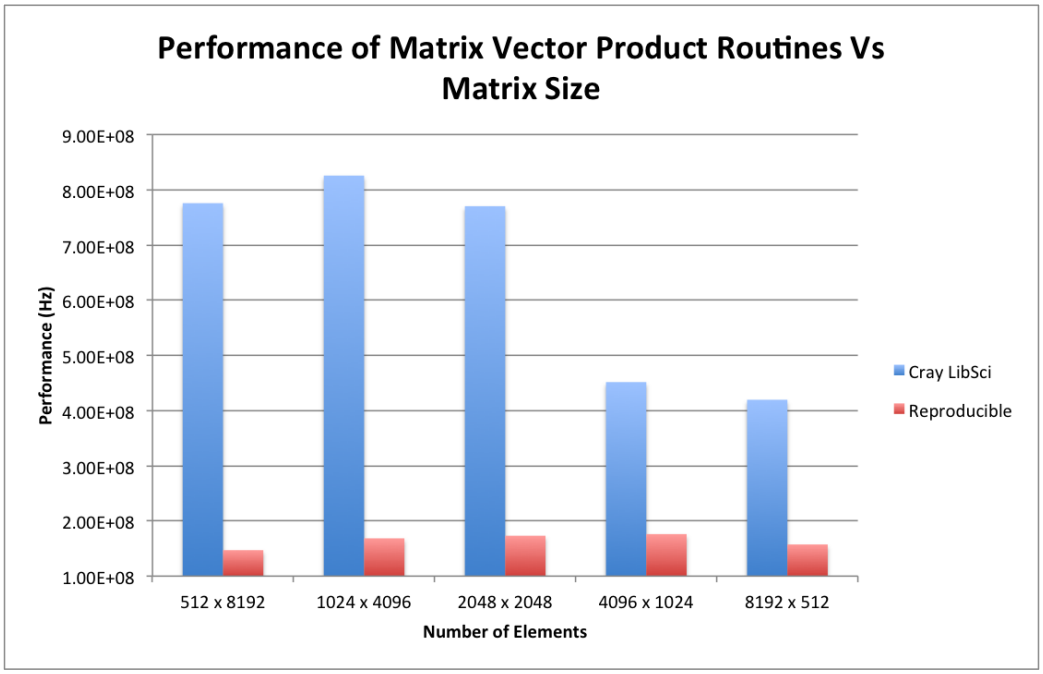

Figure 3:

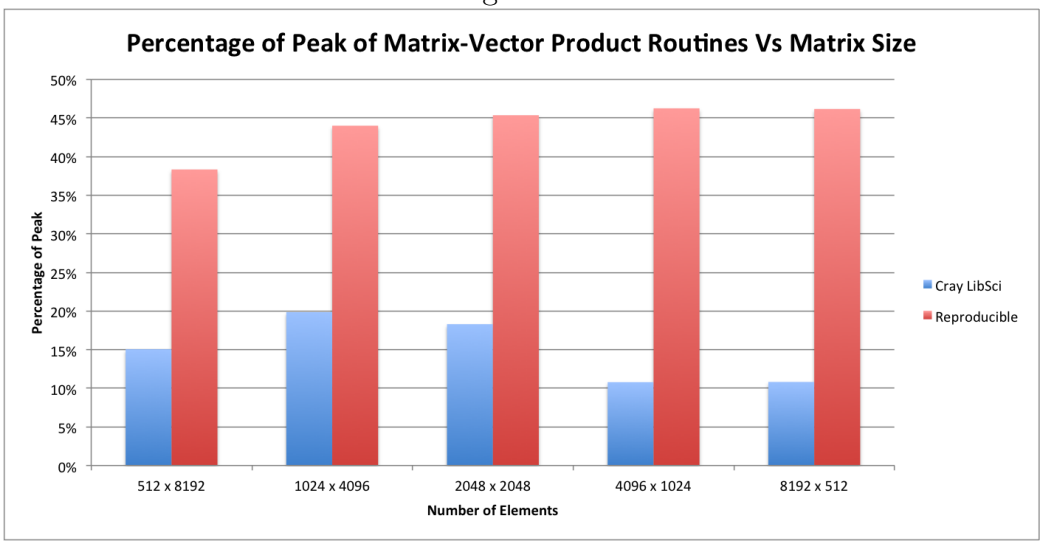

Figure 4:

the performance as a percentage of peak of the reproducible matrix-vector product on matrices with the same number of elements but different ratios between N and M. The performance (shown in figure 3), again, only differs from the vendor routine by a constant factor of 4. These results give some evidence that the reproducible BLAS1 routines can be combined to achieve efficient performance on high level sequential BLAS routines. Because the reproducible dot product is so computationally intensive, it can be used in combination with cache blocking to produce a reproducible sequential matrix-vector product with good performance. The performance as a function of peak is shown in figure 4.

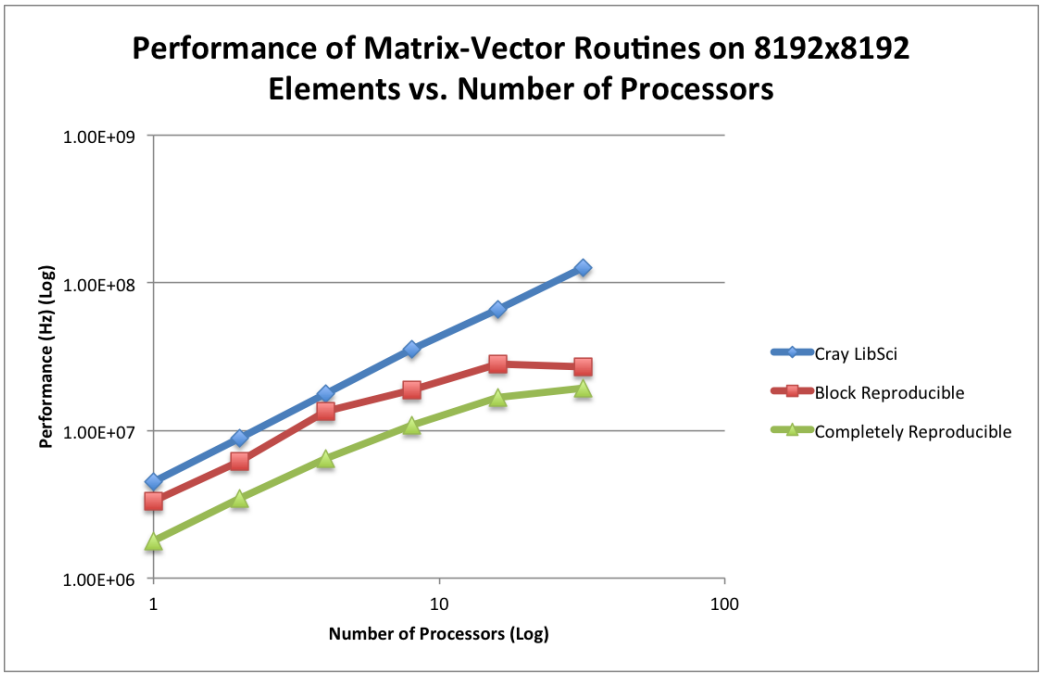

Figure 5:

Next, we consider parallel performance results. Again, we stress that the matrix-vector multiply routines were made parallel by column blocking in order to test the speed of the reduction, as row-blocking is trivially parallel. To get a good control, the data layout and processor grid is the same for the vendor matrix-vector multiply. The performance of the block and completely reproducible routines are compared to the performance of the vendor matrix-

vector multiply for different numbers of processors on a fixed matrix size in figure 5. It can be seen that although all routines initially scale well, performance issues are encountered around 32 Processors. More notably, the benefit of using the vendor routine for the block-reproducible routine becomes negligible as the number of processors increases. This hints that the reduction might be what's slowing down the routine, so we look at the reduction in detail. Figure 6 shows the cost of a reproducible reduction of 256 independent

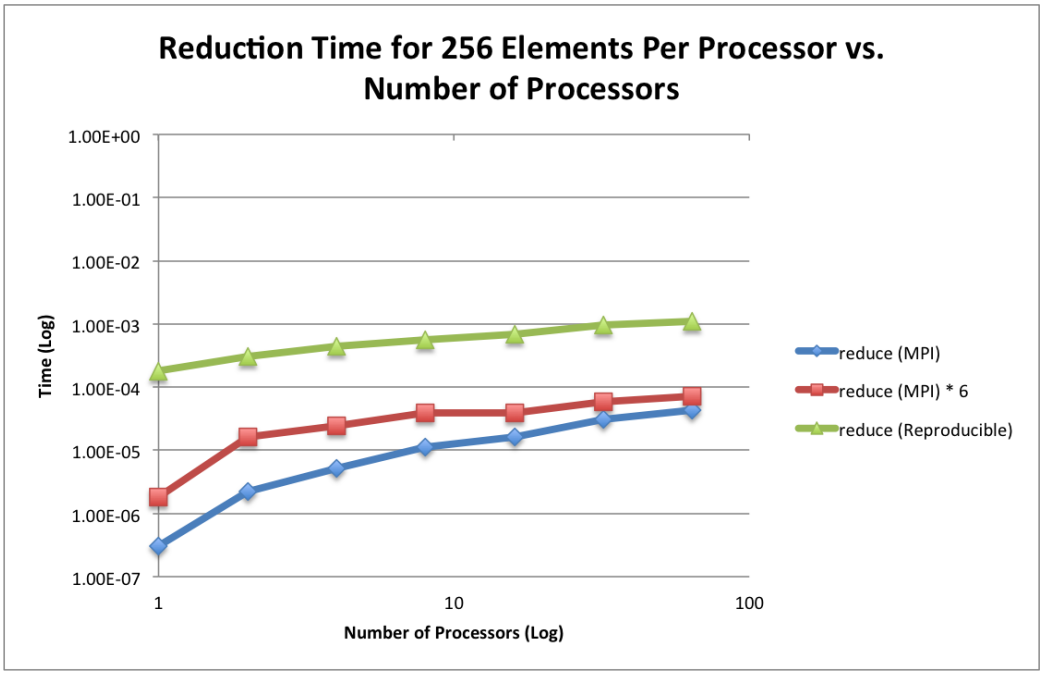

Figure 6:

elements per processor for various numbers of processors. Because indexed types are 6 times bigger than normal floats, we compare this with the cost of reducing 256 * 6 elements and the cost of reducing 256 elements. This chart shows that if the reproducible BLAS is to be competitive, the computation in the reduction routine must be accelerated. It may not be clear at this point to the reader why the reduction of indexed types is more expensive than the reproducible matrix vector product. This is because the reduction of indexed types involves adding two indexed types, while the reproducible matrix vector product involves adding a normal floating point type to an indexed type. The

second routine has been heavily optimized in the ReproBLAS library, the first has not been optimized at all.

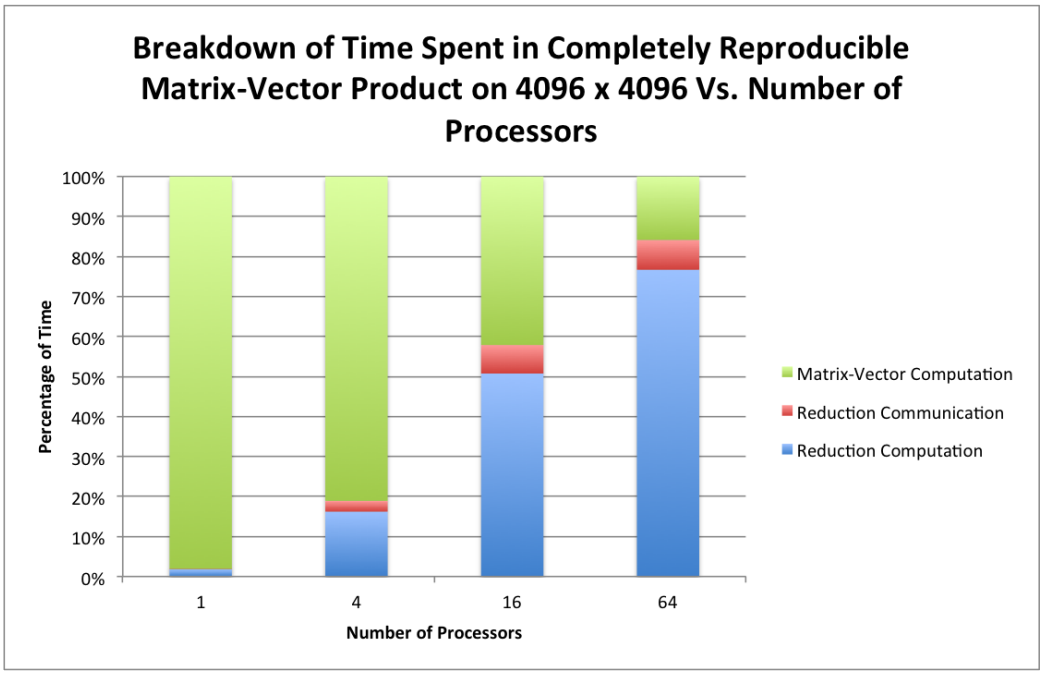

Figure 7:

To understand the cost of the reproducible reduction in context, figure 7 breaks down the time spent in the completely reproducible routine on the same matrix size for different numbers of processors. The area of color in each bar represents the proportion of time spent in that part of the routine. It is clear that the reduction computation takes a majority of the time, not the communication, which gives hope that the reduction can be optimized in the future.

5 Conclusions and Future Work

5.1 Completely Reproducible BLAS

The results of this project are most positive in this category. The time spent in improving performance of the reproducible BLAS1 routines (and

the redesign of the low-level indexed interface) have made it possible to easily piece together reproducible BLAS2 and BLAS3 routines. They are computationally-heavy and performance optimized, so minimal attention to memory layout is necessary to create higher-level BLAS routines that are reasonably optimized.

5.2 Block Reproducible PBLAS

Since most PBLAS routines are not block-reproducible due only to the BLACS reduction call, it looks as if the majority of the work in creating a Block Reproducible PBLAS lies in speeding up the existing reduction call and writing the necessary tests to verify the block reproducibility. To ensure that the PBLAS run efficiently, it may be necessary to enforce new blocking strategies that avoid larger reductions.

5.3 Completely Reproducible PBLAS

Making a completely reproducible PBLAS would depend heavily on good software engineering at the previous two levels. Because writing completely reproducible routines means that the intermediate steps must be performed using indexed types, the entire PBLAS would have to be rewritten using the indexed types and their corresponding BLAS routines. Additionally, it will be necessary to define a communication library for the indexed types (similar to the BLACS). Such a library would not be difficult to create because the underlying representation of indexed types is simply an array of floating point types, so all but the reduction operations could be written as wrappers on the original BLACS. Almost all PBLAS routines, however, would have to be rewritten in order to use indexed intermediate representations and reproducible local computations.

References

[1] https://www.nersc.gov/users/computational-systems/hopper/configuration/compute-nodes

[2] http://en.wikipedia.org/wiki/IEEE_floating_point

[3] http://bebop.cs.berkeley.edu/reproblas/index.php

[4] http://www.netlib.org/blas/

[5] http://www.netlib.org/blacs/

[6] http://www.netlib.org/pblas/