6.338 Project Report: Implementing PetaBricks in Julia

Kathleen Alexander

December 14, 2013

Contents

Abstract

A lower bound for the performance improvement that can be attained by incorporating the PetaBricks program in Julia was investigated. Performance was compared by considering both Julia and PetaBricks alone as well as a Julia implementation that ran PetaBricks executables. It was found that for sorting up to vectors of size 1,000,000 the performance of PetaBricks and Julia was comparable. For matrix multiply with square matrices of size 4096, the superior performance of PetaBricks compared to Julia was demonstrated. Additionally, it was found that Julia was twice as slow on a 64-core machine with AMD Opteron 6376 Abu Dhabi CPUs compared with an 8-core machine with intel i7 3370 CPUs while the tuned PetaBricks algorithm took approximately the same time on these two machines. It was found that by implementing PetaBricks from within Julia, it was possible to achieve PetaBricks performance without passing any additional burden onto the programer.

1 Introduction

As the variety of machines that are used in computing has expanded, traditional programming paradigms often result in programs that are far from optimal when they are applied across the available computing spectrum. PetaBricks was designed as a tool to help programmers write efficient code that could be dynamically optimized for a given system architecture thus extending the idea of code portability to performance. Similarly, the Julia programming language was developed to provide a high performance dynamic language in the scientific computing space which could simplify the parallel computing experience for the user, again, with the goal of improving utilization of modern multi-core architectures.

Since a fundamental motivation for the development of both Julia and PetaBricks was to help programmers write code that could more efficiently utilize modern computing architectures, the question as to whether there is any potential synergy between these approaches must be asked. This report is intended to outline a performance comparison of Julia and PetaBricks for solving a symmetric eigensystem, completing a sorting process, and performing a matrix multiplication problem on several different machines. In addition to this comparison, a lower bound of the potential performance gain that could be achieved by implementing a PetaBricks framework within Julia is discussed. Finally recommendations are made regarding the implications of this study with regard to incorporating an autotuner or autotuned program with Julia.

2 Background

2.1 PetaBricks

It has been shown that allowing for algorithmic flexibility and algorithmic choice can have great impact on program performance [1]. PetaBricks provides a framework to implement algorithmic choice wherein autotuning is utilized in the compiler to optimize the control flow of a program on a given architecture. Thus, a set of algorithm options are provided to the compiler, and the compiler utilizes autotuning techniques to generate a high performance executable [2].

PetaBricks is written around the constructs of transforms and rules [2]. Transforms are the wrappers to the set of functions that the compiler can utilize to accomplish the desired task, and rules are the specific functions available within the transform. After defining a transform, the user can access the transform either from another transform, another programming language, or directly from the command line, and the compiler will determine the best implementation of the transform, given the transform parameters and the machine architecture.

PetaBricks operates by first compiling a program written in PetaBricks to C++; the program is then analyzed in an autotuning step which ultimately creates a configuration file that can be either further optimized or used to generate a static binary executable [2].

2.2 Julia

Julia is a just-in-time (JIT) compiled language and its libraries and packages have largely been written in Julia, with some of the source code also written in C++. Julia utilizes the LLVM JIT infrastructure. It is capable of performing C++ calls and accessing shared libraries. Additionally, external programs can be run from within the Julia kernel.

3 Technical Approach

3.1 Implementation

There are three possible approaches that can be taken in an attempt to combine aspects of PetaBricks with Julia. One option would be to write the PetaBricks transforms in Julia and thus build the PetaBricks executable from Julia code. However, this approach would require re-writing the PetaBricks compiler to interpret the Julia language, which is a nontrivial task and not ideal for a proof of concept iteration.

Another option would be to use PetaBricks from within Julia. Julia currently has the capacity to utilize functions written in shared object files in C and fortran. PetaBricks, which is written in both PetaBricks and C, is run from an executable and, if it has been tuned, a configuration file. Thus, the most feasible way to incorporate PetaBricks inside Julia in its current state, for a proof of concept project such as this, is to compile and tune a PetaBricks executable and run it from within Julia using a shell command. This is a very naive approach, but it meets the goals of this project, and, for this reason, is the approach used here.

A final option would be to utilize the PetaBricks framework within Julia without using the PetaBricks programming language. The developers of PetaBricks have recently come out with a new compiling suite called OpenTuner [3] which performs similar tuning procedures to the PetaBricks compiler and can be implemented to work with any programming language. A notable difference between this option and an option that directly utilizes the PetaBricks language is that the functionality of algorithmic choice would be lost here as Julia does not have a built-in transform paradigm, and OpenTuner does not provide the means to incorporate this. Regardless, pursuing the use of OpenTuner with Julia may be a worthwhile future investigation.

3.2 Ensuring a Fair Comparison

Once it was decided that the most reasonable approach would be to compare the use of PetaBricks accessed from a shell command inside Julia to each of Julia and PetaBricks alone, it was necessary to determine test methods that would ensure a fair comparison between these methods. Firstly, it WAs noteD that the PetaBricks transforms of interest must read and write ASCII data files corresponding to the input and output data, respectively. The author of PetaBricks notes in the documentation that the functionality to utilize binary data files will be present in future versions of the program, but in the version used here, this was not an option. Thus, any fair test would require Julia both read and write ASCII files of the input and output data as well.

Another consideration relates to the fact that Julia can both be run from within an interpreter and from a script. To make the comparison to PetaBricks most straightforward (and because this is the style of usage that a computational scientist would be most likely to use) in the performance tests here, Julia was run from a script rather than within the interpreter. However, since Julia is a JIT compiled language, there is a startup time associated with running any Julia script. Since any of the tasks completed in these tests would, in practice, be part of much longer programs, this start-up time was subtracted from the Julia performance tests to make a fair comparison between the approaches. Thus, a scaled wall-clock time was compared to the PetaBricks timing. For each machine tests were performed on, a simple Hello World! script was run in Julia, and the time requried for this was taken to be the local Julia start-up time.

4 Results and Discussion

4.1 Tuning PetaBricks

As a first test of the PetaBricks framework, the performance of PetaBricks computing a matrix multiply before and after tuning was considered for a number of square matrices. The results of this test are presented in Figure 1.

When PetaBricks has not been tuned, the first algorithmic option presented in the transform is used for all cases. In the case of matrix multiply, this is pretty much the textbook matrix multiplication procedure with no recursion. From Figure 1, it can be seen that the autotuned algorithm has a greatly improved time complexity over the untuned algorithm. Thus, we see that PetaBricks tuning is effective in this implementation. However, it should be noted that tuning PetaBricks matrix multiply for this 4096×4096 matrix took several days on a machine with 4 3rd generation i7 intel processors (i.e. 8 cores).

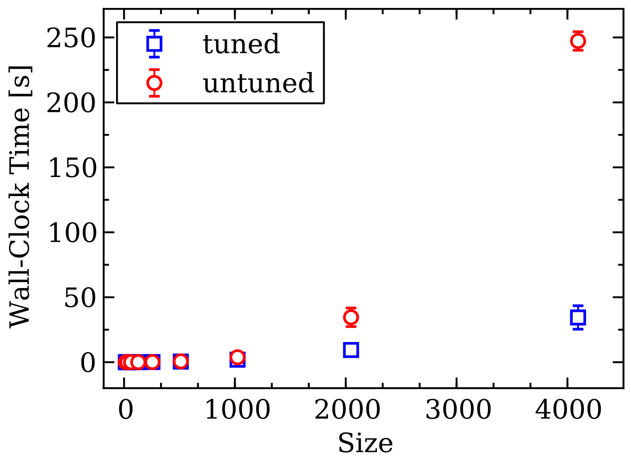


Figure 1: Matrix Multiply for both tuned and untuned PetaBricks on square matrices of varying size. Error bars that are not visible are smaller than the size of the data point.

4.2 EigenSolve

After establishing that the PetaBricks tuning mechanism was effective for matrix multiply. The time complexity of solving a dense symmetric eigenproblem was compared between PetaBricks and Julia. The results of this test for matrices with sizes between 16 and 1024 are shown in Figure 2.

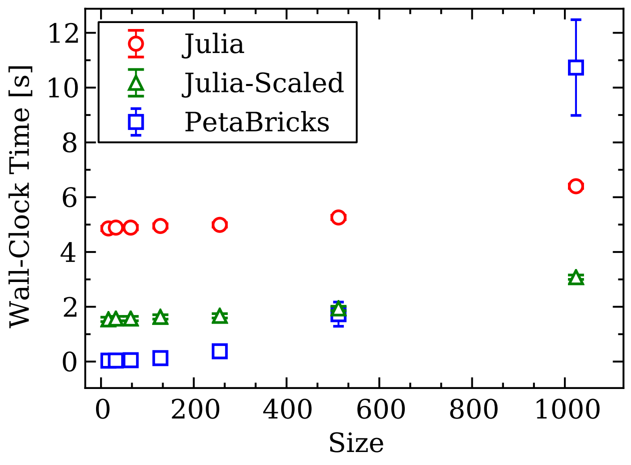


Figure 2: Eigenproblem for scaled and unscaled Julia and PetaBricks. Error bars that are not visible are smaller than the size of the data point.

Upon inspection of Figure 2, it seems that Julia (even unscaled) outperforms PetaBricks in terms of time complexity for the largest matrices, while PetaBricks is faster than Julia for smaller matrices. However, when the results of these spectral decompositions were viewed, it was found that the eigenvalues and vectors calculated for the same matrix were quite different between PetaBricks and Julia. In particular, it was found that the first approximately 10 of the eigenvalues calculated in Julia for this symmetric matrix were negative (the largest magnitude negative eigenvalue being around -10), while the smallest eigenvalue calculated with PetaBricks was about -0.88. In light of this result, it was clear that the accuracy threshhold for these two methods was significantly different, so comparing the time complexity of these programs is not particularly meaningful. Thus, the eigenproblem was not used for further analysis of PetaBricks-Julia performance testing. That being said, it should be noted that an accuracy threshhold is one of the tuning parameters that can be used in both PetaBricks and OpenTuner that is not currently a built-in option in Julia.

4.3 Sort

Sorting of vectors with lengths ranging from 16 to 1,000,000 was tested both in Julia and PetaBricks. The results of these tests are shown in Figure 3

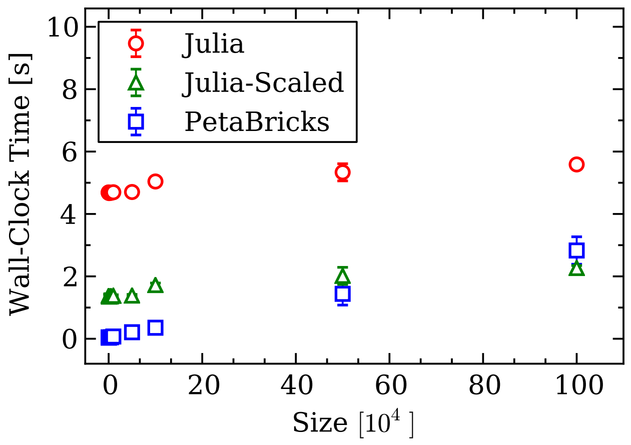
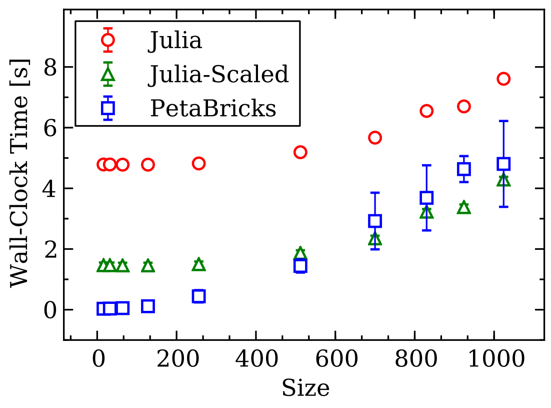


Figure 3: Sorting comparison between scaled and unscaled Julia and PetaBricks. Error bars that are not visible are smaller than the size of the data point. Tests were performed on a intel i5-3339 CPU (4 cores).

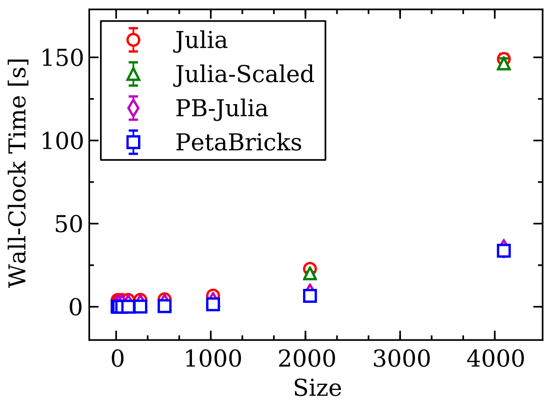
In Figure 3, it can be seen that PetaBricks performs faster than Julia for shorter vectors, but, for longer vectors, the wall-timings converge, and Julia seems to perform better on the longest vectors. Though, the errorbars for these large values currently overlap slightly, indicating that the superior performance of Julia in this test is not significantly relevant at a level of 95%. Here, Quicksort was the default algorithm used in Julia, though the user may specify an alternative sorting method.

4.4 Matrix Multiply

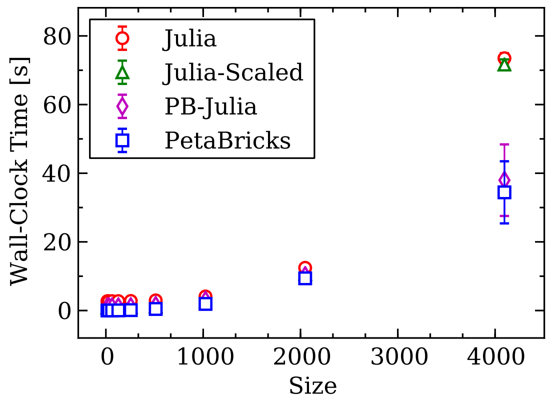
Matrix multiply was tested on 3 different machines for square matrices ranging in size from 16 x 16 to 4096 x 4096, and the results of these tests are presented in Figure 4.



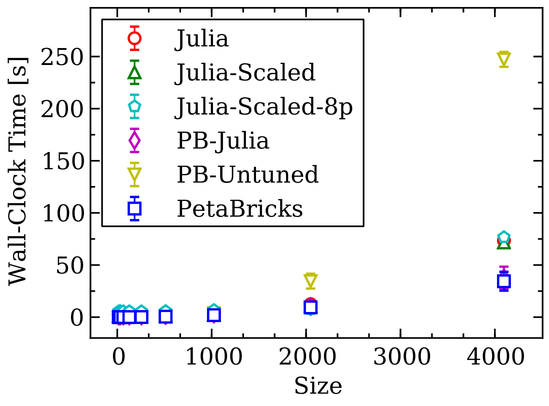
(a) 2 x i5-3339 (4 cores)



(b) 4 x AMD Opteron 6376 (64 cores)



(c) 4 x i7-3770 (8 cores)



(d) 4 x i7-3770 (8 cores)

Figure 4: Matrix multiply on 3 different machines for square matrices between 16 and 4096 in dimension. Error bars that are not visible are smaller than the size of the data point. (a) scaled and unscaled Julia and PetaBricks on 4 3rd generation intel i5 CPUs. (b) scaled and unscaled Julia, Julia executing PetaBricks, and PetaBricks on 64 AMD Opteron CPUs. (c) scaled and unscaled Julia, Julia executing PetaBricks, and PetaBricks on 8 3rd generation intel i7 CPUs. (d) scaled and unscaled Julia, Julia executing PetaBricks, PetaBricks, untuned PetaBricks, and Julia with an addprocs(7) command on 8 3rd generation intel i7 CPUs.

In Figure 4a, it is seen that for square matrices up to 1024 in dimension on this 4-core machine, PetaBricks and Julia perform about the same for the large matrices, but PetaBricks outperforms Julia for small matrices. In Figure 4b, it is seen that PetaBricks begins to outperform Julia for large matrices with a size of 2048, and at the size of 4096, PetaBricks is almost 5 times faster than Julia. This makes sense since PetaBricks is tuned to perform the multiplication in parallel while Julia is not using multiple cores. For this test, the 64-core machine was loaded at about 6.98 cores when running PetaBricks matrix multiplication and at about 0.88 when running the Julia tests. Thus, only a fraction of the computational power of the machine was utilzed here; however assumably, the use of more cores resulted in overhead that was not offset with the additional computational resources, so this was the optimized loading for a matrix of this size. Additionally, in Figure 4b, we note that, running PetaBricks within Julia achieves the same performance as PetaBricks alone, so, an argument for using PetaBricks within Julia could be made based on this data.

Figures 4c and 4d show the results of testing matrix multiply on an 8-core machine. In Figure 4c, it is again seen that for the 4096 size matrix, PetaBricks significantly outperforms Julia, in this case by about a factor of 2 and that implementing the PetaBricks executable within Julia allows for the advantages of PetaBricks to be directly utilized within Julia. In Figure 4d, the same data from Figure 4c is presented, with some additional tests superimposed. Firstly, the untuned PetaBricks data is included, demonstrating that, though Julia performs worse than PetaBricks in the larger matrix test, it is still quite well optimized when compared with the untuned PetaBricks algorithm. Lastly, to be completely fair, a test is included for which the addprocs(7) command is issued in Julia before performing the usual matrix multiply routine. This test is simply included to demonstrate that, an automatically parallelized matrix multiply that could take advantage of multiple processors, if they were made available, is not present in Julia and, as a result, this script offers no speedup over the normal Julia matrix multiply routine.

5 Conclusions and Recommendations

The lower bound of potential performance gains that could be achieved through combining PetaBricks and Julia was investigated by comparing the performance of Julia and PetaBricks alone with an implementation wherein a tuned PetaBricks executable was called from within Julia. In this study, it was found that for many cases Julia performed just as well as PetaBricks without requiring days of autotuning. Additionally, the usefulness of an autotuner that requires days to tune for a matrix multiplication of size 4096 must be brought into question, as often times the size of systems we may be interested in are orders of magnitude larger than this. Regardless, the superior performance of PetaBricks compared to Julia was made clear for matrix multiply on a square matrix with dimension 4096 on two machines. In particular it should be noted that Julia was twice as slow on the 64-core machine with AMD Opteron 6376 Abu Dhabi CPUs compared with an 8-core machine with intel i7 3370 CPUs while the tuned PetaBricks algorithm was equally fast between these machines. Thus, the need for machine-specific compiling as well as the successful autotuning via PetaBricks was well demonstrated on these systems.

It was demonstrated that by implementing PetaBricks from within Julia, it was possible to achieve PetaBricks performance without passing any additional burden onto the programer. Thus, for a hypothetical application for which the need for algorithmic choice was really paramount in some subroutines of a program, but for which it was convenient for most of the infrastructure of the program to be built in Julia, it would be possible to write and compile the transform for the subroutines of interest in PetaBricks and call those from Julia wherever necessary in the driver program. Between this type of implementation and the fact that Julia's performance was comparable to PetaBricks in many tests without the need of the autotuning overhead, the argument could be made that the combination of Julia alone and this naive-style implementation could be sufficient for most users. From this perspective, the most important next step in development would be to incorporate binary data-type read-write functionality, at the very least, into PetaBricks, as the data-passing between programs would be expected to be a significant bottleneck for large systems.

However, in terms of broader future directions, it is possible that implementation of the OpenTuner with Julia would allow for improved performance of Julia between machine architechtures, the weakness of which was demonstrated between Figures 4b and 4c. Implementing an infrastructure of machine-leared algorithmic choice within Julia would require a much greater overhaul of the compiliation system, and, though it has been demonstrated to be effective, seems to be an unlikely direction that the developers will take given the additional tuning time that would be required for this nature of program. Lastly, some argument could be made to improve the Julia start-up time; however, for most applications, this start-up time becomes negligible compared with the actual computation required, as is demonstrated for scaled and unscaled wall-clock times in Figure 4.

References

[1] Jason Ansel and Cy Chan. PetaBricks. *XRDS: Crossroads, The ACM Magazine for Students*, 17(1):32, September 2010.

[2] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[3] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Una-May O'Reilly, and Saman Amarasinghe. OpenTuner : An Extensible Framework for Program Autotuning. *MIT CSAIL Technical Report MIT-CSAIL-TR-2013-026*, 2013.