Analyzing Parallel Programming Models for Magnetic Resonance Imaging

Forest Danford[1]**, Eric Welch[1]**, Julio Cárdenas-Ródriguez[2], and Michelle Mills Strout[1]

[1] University of Arizona,
Department of Computer Science
Tucson, Arizona,
{fdanford, welche, mstrout}@email.arizona.edu
[2] University of Arizona,
Department of Medical Imaging,
Tucson, Arizona,
cardenaj@email.arizona.edu

Abstract. The last several decades have been marked by dramatic increases in the use of diagnostic medical imaging and improvements in the modalities themselves. As such, more data is being generated at an ever increasing rate. However, in the case of Magnetic Resonance Imaging (MRI) analysis and reports remain semi-quantitative, despite reported advantages of quantitative analysis (QA), due to prohibitive execution times. We present a collaborator's QA algorithm for Dynamic Contrast-Enhanced (DCE) MRI data written in MATLAB as a case study for exploring parallel programming in MATLAB and Julia. Parallelization resulted in a 2.66x speedup in MATLAB and an almost 60x speedup in Julia. To the best of our knowledge, this comparison of Julia's performance in a parallel, application-level program is novel. On the basis of these results and our experiences while programming in each language, our collaborator now prototypes in MATLAB and then ports to Julia when performance is critical.

Keywords: MATLAB, Julia, Parallel Programming Languages, Parallel Applications, Medical Imaging, Dynamic Contrast-Enhanced MRI

1 Introduction

Over the last 30 years there has been a dramatic increase in the usage of Magnetic Resonance Imaging (MRI) and other imaging modalities in the research and medical communities [26, 27]. Simultaneously, there have been substantial improvements to the underlying technologies themselves that allow for images to be captured at significantly higher spatial and temporal resolutions. One of the consequences of these improvements has been a massive increase in the amount

** Authors contributed equally

of data being generated [16]. For scientists and clinicians, this has often translated to making the difficult decision to forgo a truly quantitative analysis (QA) and instead sub-sample the available data to perform analyses and generate reports within an acceptable time frame. One application that currently suffers this plight is Quantitative Analysis (QA) model-based Dynamic Contrast-Enhanced MRI (DCE MRI) [22]. DCE MRI is used to visualize and characterize cancerous tumors in animal models and humans. The results can be used to predict patient-specific response to anticancer drugs and provide insight for new drug discovery [10, 19]. DCE MRI is thus highly valuable to clinical oncologists and scientists. Dr. Julio Cárdenas-Ródriguez, a researcher at the University of Arizona, developed a serial implementation of DCE MRI analysis code written in MATLAB that had an unacceptable run time unless it used dramatically subsampled datasets ($\geq$ 50x). For example, evaluating the QA algorithm on the researcher's pancreaticand breastcancer datasets was estimated to take approximately six months, which is not practical.

To understand the volume of data being generated, it is valuable to briefly summarize the workflow of a DCE MRI experiment. Two separate regions, the volume containing the tumor and a volume containing normal vascular tissue (known as the reference region; in our case the leg), must be imaged continuously for several minutes. The MRI machine generates a 2-dimensional double array of intensities (at a specified spatial resolution) for every time point captured during the imaging time frame. These volumes must be imaged twice - first as a baseline to determine the tissue's innate relaxation properties (T_R volume), and then many times after the injection of a gadolinium-based contrast agent (CA). The CA's properties cause a change in the relaxation properties of the tissue, translating to different voxel intensities. These changes in signal intensity over time are used to calculate the CA's concentration, which is run through to a non-linear least squares computation to estimate the permeability of the tumor [19]. For more information, we refer readers to Cárdenas-Rodríguez et al., 2016 [5].

The computationally expensive portion of the QA DCE MRI algorithm lies in the non-linear fitting step, which must be performed for each voxel containing tissue in the image volume. Even in our heavily sub-sampled dataset, thousands of non-linear least square problems must be solved. Analysis of the original serial MATLAB code revealed that the majority of the workload could be computed in an embarrassingly parallel fashion, as each spatial point in the volume is independent of the others.

Given that the original code was already implemented in MATLAB and the popularity of the language in the biomedical engineering community, the straightforward approach to solving the performance problem was to implement an equivalent parallel implementation using the *Parallel Computing Toolbox* (PCT) in MATLAB [31]. However, this only improved performance on a by 2.66x using 8 cores at best. Therefore, we decided to do a case study where we implemented a version in a newer language called Julia [4]. With Julia, we were

able to achieve an approximately 57x performance improvement over the serial MATLAB implementation.
This paper makes the following contributions:

- We analyze the serial algorithm and identify areas where parallelization is expected to increase performance.
- We provide a more substantial benchmark for the Julia programming language (previous publications only include microbenchmark results) via quantitative performance data for the MATLAB and Julia versions.
- We provide a comparison of the process of writing parallel code in MATLAB and Julia based on our experiences in this case study.

Based on the significant increase in performance observed in the Julia implementations (18x speedup for the serial version and 57x speedup in the parallel version), Dr. Cárdenas-Ródriguez's research group has begun to translate all their performance critical code to Julia.

The remainder of the paper is organized as follows: Section 2 contains a thorough description of the serial code and the experimental methodology. Sections 3 and 4 discuss the MATLAB and Julia parallel implementations. Section 5.1 details our quantitative timing results. Section 5.3 provides an anecdotal comparison of the programming languages. Section 6 presents related work, and Section 7 concludes.

2 Analysis of the Serial MATLAB Code

The QA DCE MRI code consists of the following five tasks:

1. Loading the data
2. Segmenting the tissue from the background
3. Calculating T_1 time (seconds) on a per-voxel basis via non-linear least squares curve fitting to its phenomenological equation and evaluating the goodness-of-fit via the R^2 value
4. Calculating the change in R_1 as a function of time
5. Estimating the relative permeability between the tumor and muscle tissue ($R^{K_{trans}}$) using the linear reference region model and evaluating the goodness-of-fit via the R^2 value.

The original serial algorithm was revised to remove all unnecessary operations (e.g. those done solely to produce human-interpretable images), and the standards [32] recommended by The MathWorks were enforced. This resulted in about a 1.5x speedup (execution time of 56.49 ± 1.22 seconds vs. 83.03 ± 1.52 seconds). This simplified version, whose execution time is profiled in Figure 1, is used as the baseline for all comparisons presented in the paper. The remainder of this section analyzes the performance of the serial MATLAB implementation and describes each task in more detail.

Step	Time (seconds)	
	Tumor	Leg
Load data	0.141 ± 0.013	
Segmentation	0.018 ± .002 sec	0.019 ± 0.002
T_1 calculation	41.332 ± 0.930	13.383 ± 0.321
ΔR_1 calculation	0.028 ± 0.008	0.009 ± 0.0003
K^{trans} via RRM	1.555 ± 0.068	
Total run time	56.486 ± 1.222	

Fig. 1: Execution time for main tasks of serial MATLAB implementation.

2.1 Experimental Methodology

The execution times reported are the average of five runs ± their standard deviation measured after a warm up run whose time was not included. This allows just-in-time (JIT) compilation to complete in both languages to avoid comparing compiler speeds and efficiencies, and instead provide a comparison of the performance of the JIT-compiled computations themselves. Additionally, given that these are one-off costs that are amortized over many datasets, the contribution to the overall run time becomes negligible.

All execution times were obtained by submitting jobs to a PBS scheduling system on the University of Arizona's high-performance computing cluster consisting of compute nodes with Xeon Westmere-EX Dual 6-core or 8-core processors operating at 2.66 GHz. The dataset used for benchmarking the various implementations consisted of 250 time points, each composed of a 256 x 256 array of doubles. MATLAB 2015b was used.

Timing information in MATLAB was obtained using the built-in `tic/toc` construct. These values were validated against MATLAB's built-in code profiler and were not found to be significantly different.

2.2 Performance Bottleneck: T_1 calculation

As shown in Figure 1, the T_1 calculation accounts for approximately 97% of the total run time on average. The disparity between the run time of the tumor and leg volume arises due to the difference in the number of tissue voxels present post-segmentation (3199 voxels vs 1083 voxels). If we account for this difference, it appears that the execution time for this function scales linearly, so there is nothing intrinsically different about the tumor.

In this task, each voxel from the T_R volume is fit to the following phenomenological equation to solve for T_1:

$$S(t) = M_z(1 - e^{-T_R/T_1(t)}) \tag{1}$$

via MATLABs built-in non-linear least squares solver, `lsqcurvefit`. As previously mentioned, dependence analysis revealed that each spatial location within the volume was independent (i.e the voxel at (1, 1) at timepoint 1 is independent

of all other voxels at timepoint 1), and that the two T_R volumes are independent of each other as well. As such, calculations at each voxel can be computed in an embarrassingly parallel fashion. Parallelization efforts were focused on this function for the MATLAB and Julia implementations.

2.3 Other Tasks

The raw output from the MRI machine is pre-processed into an array of doubles that are stored in a MATLAB `.mat` file and a `csv` format for the Julia implementation. MATLAB's built-in command `load()` takes virtually no time as can be seen in Figure 1, and the same is true of Julia's `readdlm` function.

Segmentation is performed using MATLAB's built-in k-`means` clustering algorithm. A 1-D vector containing all voxels is input and the function categorizes all voxels as either tissue or background. MATLABs built-in implementation of Otsus method [21] to determine the gray-scale threshold for binarization was also explored, but k-means generated the best mask (results omitted). The k-means algorithm in the clustering package of JuliaStats generated identical masks. The run time of this task is also negligible.

After T_1 is calculated, two equations are combined to describe R_1 ($\propto$ CA concentration) as a function of time, T_1, and T_R:

$$R_1(t) = -\frac{1}{T_R} \cdot \ln(1 - S(t)e^{-T_R \cdot R_1(0)/S(0)}) \tag{2}$$

As this is a straightforward calculation that takes a negligible amount of time, we do not parallelize it. At this point, concentration as a function of time is known —that is, we have solved the non-linear relationship between voxel intensity and CA concentration for both tissue volumes. The reference region model (RRM) [5] is used to calculate the permeability of the CA, K^{trans}, and the R^2 value on a voxel-by-voxel basis for the tumor tissue via a linear least squares fitting with non-negativity constraints (`lsqnonneg`). Despite each voxel being independent, the function runs for a short enough time that overhead costs would likely outweigh parallelization gains.

2.4 Performance Analysis Summary

To summarize, the tumor and reference region voxels are entirely independent and are processed identically until the calculation of K^{trans}. Additionally, in all tasks that process the segmented tissue voxels, the calculations performed on each voxel are independent, and can therefore be computed in an embarrassingly parallel fashion. Analysis of the computations that occur within these tasks does not reveal anything inherent to the code that would result in one language handling the computation preferentially.

3 MATLAB Implementations

The DCE MRI algorithm was parallelized in MATLAB by a graduate student who had several years of experience working with MATLAB and the *Parallel Computing Toolbox* (PCT) in the biomedical domain. A 2.66x speedup over the serial code was achieved using the approaches outlined below to perform the T_1 calculation using 8-cores of a 12-core machine.

3.1 MATLAB Background

Because most scientists do not receive formal software engineering training [2], MATLAB's ease of use (e.g. weak and dynamic typing system, lack of need to declare dimensions, etc.) and trusted libraries have made it a popular language for scientific computing applications [9, 23]. Additionally, the core intentions of MATLAB's parallel programming model (the PCT) were to extend the aforementioned traditional strengths of MATLAB onto the cluster via first-class language constructs to deal with embarrassingly parallel problems [24]. This allows users to easily utilize multicore processors, GPUs and clusters with minimal modification of code or impact on readability.

However, MATLAB is not as performant as other programming languages, especially those used for parallel programming [4]. MATLAB worker threads that execute concurrent computation are heavyweight, and the PCT is proprietary and has limited scalability [14]. Additionally, the dynamic and complex typing system results in significant overhead [1, 8, 13]. Despite these drawbacks, in the realm of scientific computing, time to solution, readability, portability, and maintainability often trump pure performance [2], so MATLAB is utilized significantly [13].

As a bridge, continuous and significant work has been done on static analysis, ahead of time speculation, JIT compilation, and automatically porting existing code to more performant languages [1, 9, 12, 14, 15]. While automatically ported code makes sacrifices in terms of both the highly human-interpretable syntax and interactive nature of MATLAB, The MathWorks has adopted some of the other techniques, and as of release 2015b MATLAB is now entirely JIT compiled [29, 30]. Based on selected case studies [30] and comparing the benchmarks performed by the Julia language creators on MATLAB 2011a [4] to MATLAB 2015b (http://julialang.org/), it appears that this has had a generally positive impact on performance.

3.2 Parallelization using the Parallel Computing Toolbox

For the DCE MRI application, the T_1 calculation was rewritten so that every voxel was fit in parallel, while the tissue was processed serially (version 1). Because the code was already simplified and optimized, modifications were minimal —consisting of slicing variables [17] and changing the `for` to a `parfor` in the function shown in Figure 2. As nested parallelism is not supported in MATLAB, restructuring the serial code to process the tissues and voxels in parallel

required more modifications (version 2). This negatively impacted readability, but resulted in a small performance improvement.

Table 1 contains the execution time, speedup, and efficiency for the parallelized versions as a function of the number of cores provided to the MATLAB parallel pool. Unexpectedly, version 1 was the most performant with a 2.66x speedup at 33.24% efficiency using 8 cores. Version 2 was nearly as fast with a 2.65x speed up at 33.11% efficiency also using 8 cores. It is important to note that on average, it took 10.27 ± 1.48 seconds to initialize the parallel pool, which is nearly 47% of the total execution time of the fastest version. Amortizing this cost over the processing of multiple files would greatly increase MATLAB's speedup and efficiency.

Table 1: Summary of Parallel MATLAB implementations

Iteration	# cores	Execution time (seconds)	Speedup	Efficiency
Version 1	1	81.00 ± 15.58	0.73x	73.23%
	2	44.59 ± 0.15	1.33x	66.52%
	4	29.51 ± 0.24	2.01x	50.25%
	8	22.31 ± 0.21	2.66x	33.24%
	12	22.91 ± 0.41	2.59x	21.58%
Version 2	1	71.54 ± 0.11	0.83x	82.92%
	2	44.71 ± 3.87	1.33x	66.33%
	4	29.19 ± 0.42	2.03x	50.81%
	8	22.40 ± 0.25	2.65x	33.11%
	12	23.11 ± 0.86	2.57x	21.39%

4 Julia Implementations

The DCE MRI algorithm was converted from MATLAB to Julia by a graduate student who had never worked with MATLAB or Julia, had no experience with medical imaging, and had no prior experience with parallel computing. Thanks to several features of the two languages, this process was straightforward, as was the parallelization of the T_1 calculation, which ultimately resulted in a 57x speedup over the serial MATLAB code. Julia v0.4.3 was used and timing information was obtained using the `@time()` macro as per the recommended best practices [33].

4.1 Julia Background

Julia was designed specifically for numerical and scientific computing, and has a syntax similar to languages such as MATLAB and R, but has been shown to outperform such dynamically-typed languages on microbenchmarks, often achieving performance comparable to C and Fortran [4]. Julia's creators have indicated that the language's speed is accounted for by its robust type inference system and high-performance LLVM-based JIT compiler that generates optimized, on-the-fly native machine code directly [3].

Julia features built-in parallel capability [3], and although it is still in pre-release, its user base has created a significant number of native libraries for the language [25], as well as interfaces to commonly utilized libraries from other languages such as NLOpt.

4.2 Serial Julia Implementation

The run time of the T_1 calculation in the serial Julia implementation was 3.01 ± 0.04 seconds, roughly a 18x speedup over the serial MATLAB version. Because MATLAB and Julia were designed for programmability and are thoroughly documented, it was easy to investigate MATLAB functions and determine how to implement them in Julia. Indeed, Julia seems to liberally "borrow" features from many other languages, including MATLAB, and we found that many MATLAB functions, such as the matrix manipulation function `reshape`, have been implemented in Julia with nearly identical syntax and semantics (Figure 5)

When required MATLAB functions had no equivalent in the Julia standard library, it was easy to locate third-party, open-source libraries providing the needed functionality. This occurred in the segmentation task —Julia did not have a built-in k-means function. However, because Julia features a package manager, typing the command `Pkg.add("Clustering.jl")` at the Julia command prompt immediately downloaded the latest version of the library `Clustering.jl` from a github repository (https://github.com/JuliaStats/Clustering.jl), and provided the use of its `kmeans` function, which had the same syntax as MATLAB's implementation. For the task involving least-squares curve fitting, a function from the JuliaOpt package `LsqFit.jl` that appeared to mirror MATLAB's `lsqcurvefit`

MATLAB

```
% Normalize signal
Signal = masked_T1_vTR ./
  repmat(max(masked_T1_vTR, [], 2), 1,
  numTimePoints);

% Set curve fitting parameters
x0 = [1.2, 3.0];
lb = [1, 1];
ub = [2, 5];

% Define model function
T1vTRfunc = @(pars, xdata) pars(1) .*
  (1 - exp(-xdata./pars(2)));

% Do curve fitting on per voxel basis
for q = 1:numSignalPixels
    [beta, ~, resid, ~, ~, ~, J] =
      lsqcurvefit(T1vTRfunc, x0, TR,
      Signal(q, :)', lb, ub, options);
    T1vTR.map(q) = beta(2);
end
```

Julia

```
for q in 1:size(indices,1)
    Signal = T1vTR_Images[indices[q],:]
    Signal ./= maximum(Signal)

    # Create solver and set objective function
    model = Model(solver=NLoptSolver(
      algorithm=:LN_COBYLA))
    @defVar(model, beta[1:2])
    @setNLObjective(model, Min, sum{(Signal[i]-
      beta[1]*(1-exp(-TR[i]/beta[2])))^2, i=1:N})

    # add constraints and initial guess
    @addNLConstraint(model, beta[1] >= lb[1])
    @addNLConstraint(model, beta[1] <= ub[1])
    @addNLConstraint(model, beta[2] >= lb[2])
    @addNLConstraint(model, beta[2] <= ub[2])
    setValue(beta[1], x0[1])
    setValue(beta[2], x0[2])

    status = solve(model)
    T1vTR_Map[indices[q]] = getValue(beta[2])
end
```

Fig. 2: Curve-fitting function in MATLAB and Julia

was inadequate, as it did not allow for bounds on the solution. An interface allowing Julia to call the NLopt library was used instead (`NLopt.jl`) (Figure 2).

A few MATLAB functions involved in the permeability estimation step, such as `cumtrapz` and `nans`, were not present in Julia, but it was straightforward to simply implement them.

4.3 Parallelization of Julia Implementation

Julia allows for a variety of approaches to parallel programming [34], but for the embarrassingly parallel computations required in this application, a simple shared memory model was sufficient. We utilized a special Julia datatype designed for this purpose, called a SharedArray, which is accessible by multiple processors. The demonstration code for SharedArrays in the Julia documentation provided a clear blueprint for our implementation. Following this example, the key steps were to copy data from several arrays into SharedArrays, and create two kernels that: (1) determined which voxels to process on each processor, and (2) ran the appropriate subset of the least-squares curve-fitting calculations on each processor.

The first of these (Figure 3a), which was essentially copied from the Julia documentation [34], partitions a collection of indices into equal-sized groups for each processor. The second (Figure 3b) merely substitutes an iteration over all voxel indices for an iteration over the indices assigned to the worker running the kernel.

Index determination kernel

```
@everywhere function my_range(shared_indices)
    idx = indexpids(shared_indices)
    if idx == 0
        return 0:1
    end
    n_chunks = length(procs(shared_indices))
    splits = [round(Int, s) for s in linspace(0, size(shared_indices, 1), n_chunks+1)]
    return splits[idx]+1 : splits[idx+1]
```

(a) Kernel to divide indices in SharedArray to be processed on different processors

Serial curve-fitting in Julia

```
for q in 1:size(indices,1)
    Signal = T1vTR_Images[indices[q],:]
    Signal ./= maximum(Signal)
    T1vTR_Map[indices[q]] =
        calculate_T1(Signal, TR)
end
```

Parallel curve-fitting kernel run on each worker

```
for q in my_range
    Signal = T1vTR_Images[indices[q],:]
    Signal ./= maximum(Signal)
    T1vTR_Map[indices[q]] =
        calculate_T1(Signal, TR)
end
```

(b) Serial vs. Parallel curve-fitting function in Julia

Fig. 3: Functions involved in serial and parallel curve fitting in Julia

As can be seen in Table 2, the parallelization produces gains of up to 3.17x over the serial Julia implementation, with reasonable efficiency.

Table 2: Summary of Parallel Julia implementations

# Cores	Execution time (seconds)	Speedup relative to serial Julia	Total efficiency
1	3.53 ± 0.04	0.85x	85.47%
2	2.07 ± 0.01	1.46x	72.90%
4	1.39 ± 0.02	2.17x	54.31%
8	1.00 ± 0.01	3.02x	37.76%
12	0.95 ± 0.02	3.17x	26.38%

5 Results

In this section, we evaluate our implementations of the researcher's algorithm in terms of performance and reliability, and report on differences in programmability between MATLAB and Julia. We find that rewriting the algorithm in Julia resulted in performance gains exceeding those of using the MATLAB PCT.

5.1 Performance

Quantitative performance metrics were generated for the parallel MATLAB, serial Julia, and parallel Julia implementations. The most performant version of parallel MATLAB achieved a speedup of 2.66x with 33.11% efficiency. As can be seen in Figure 4, the serial version of Julia achieved a 18.15x speedup compared to the serial MATLAB version, a 57.45x speedup with parallelism, and a 23.3x speedup when comparing the fastest parallel Julia implementation with the fastest parallel MATLAB implementation. If we discard the cost of starting the parallel pool in MATLAB and only compare the running time of the algorithm itself, the fastest Julia implementation is 2.24x faster than the fastest parallel MATLAB implementation. Julia exhibits more efficient parallelism compared to MATLAB, although the differences are far less dramatic than the speedup (Tables 1 and 2).

5.2 Reliability

The results obtained with Julia were comparable to those obtained with MATLAB. The T_1 calculation results obtained from Julia had insignificant numerical differences compared to the original MATLAB implementation: within the 1,083 signal pixels in the leg, the root mean square error was 7.90e-4 and the maximum absolute difference was .0108; within the 3,199 signal pixels of the tumor, the root mean square error was 9.07e-4 and the maximum absolute difference was .0347. For the final $R^{K_{trans}}$ calculation, the root mean square error was 2.31e-3 and the maximum absolute difference was 0.0744.

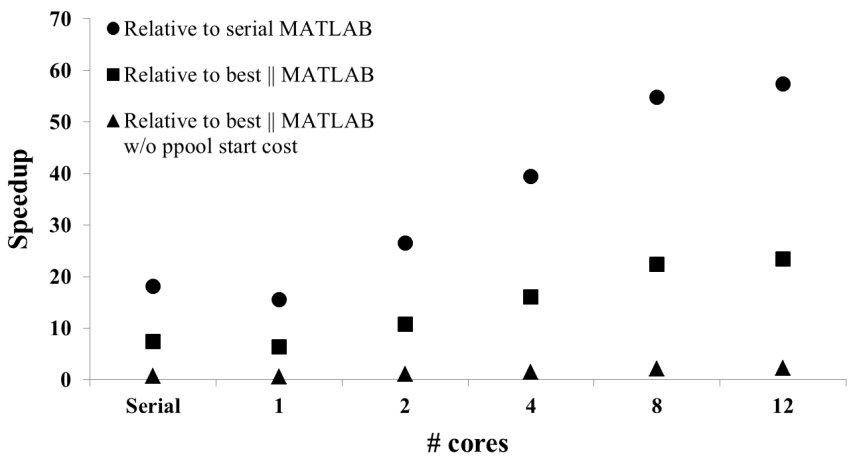

Fig. 4: Speedup of Julia implementations relative to MATLAB implementations

5.3 Programmability

MATLAB and Julia are similar syntactically (Figure 5) and also in terms of their expressivity. The serial MATLAB and Julia implementations contain 292 and 244 source lines of code (SLOC) respectively. The fastest parallel implementation in MATLAB contains 239 SLOC compared to Julia's 284 SLOC. Productivity was also comparable after familiarity was gained with Julia —in both languages it took approximately 1.5 hours to parallelize the algorithm. This was somewhat surprising given the differences in familiarity with the languages and parallelism. The graduate student who learned MATLAB and Julia in tandem did not find either language particularly more difficult to understand than the other —however, he was simply translating MATLAB code to Julia code, as opposed to prototyping purely in Julia.

One recurring difficulty that arose when writing code in Julia was that Julia's error messages can appear cryptic, often including information about types that the programmer did not explicitly specify, but were inferred during JIT compilation. In Figure 6, there is an error because `size(out)` returns a `Tuple`

Mask extraction code in MATLAB

```
[x,y,z] = size(data);
data_mtx = reshape(data, x*y, z);
k = 2
kmeans_group_index = kmeans(data_mtx, k);
group_index_noise = kmeans_group_index(1);
mask = reshape(kmeans_group_index ~=
group_index_noise, x, y);
```

Mask extraction code in Julia

```
(x,y,z) = size(data)
data_mtx = reshape(data, x*y, z)
k = 2
kmeans_group_index = kmeans(data_mtx', 2)
group_index_noise = kmeans_group_index.assignments[1]
mask = reshape([d == group_index_noise ? 0 : 1
  for d in kmeans_group_index.assignments], x, y)
```

Fig. 5: k-means mask extraction code in MATLAB and Julia

```
julia> function seq(a, b)
           out = zeros(b-a+1)
           for i in 1:size(out)
               out[i] = a+i-1
           end
           out
       end
seq (generic function with 1 method)

julia> seq(3,5)
ERROR: MethodError: `colon` has no method matching colon(::Int64, ::Tuple{Int64})
Closest candidates are:
  colon{T<:Real}(::T<:Real, ::Any, ::T<:Real)
  colon{A<:Real,C<:Real}(::A<:Real, ::Any, ::C<:Real)
  colon{T}(::T, ::Any, ::T)
  ...
 in seq at none:3
```

Fig. 6: A typical Julia error message

type, while the colon operator expects the right-hand operand to be an integer. In MATLAB, comparable code runs with no error, since the colon simply uses the first element of the 2×1 array returned by `size`. MATLAB was found to be typically more "forgiving" than Julia.

Moreover, MATLAB's Code Analyzer will infer and warn the programmer if, for instance, there is code that would distribute a large array to many parallel workers, while Julia does not provide warnings of this type. More broadly, the MATLAB IDE was felt to be more convenient and powerful than using a Julia the Juno IDE (http://junolab.org/) or a Jupyter Notebook (http://jupyter.org/).

Another difficulty that arose when implementing the algorithm in Julia was locating a library that would provide the same functionality as MATLAB's `lsqcurvefit` function. The fact that a Julia library function with the same name had been written, but which used a different algorithm that did not allow bounds constraints, was troubling.

In spite of these hurdles, our experience suggests that learning Julia is comparable to learning a dynamically-typed programming language.

6 Related Work

To the best of our knowledge this is the first comparison of the performance of two implementations of the linear reference region model (LRRM) for the analysis of DCE MRI data. Smith et al. [25] recently described DCEMRI.jl, a Julia implementation of the commonly used Tofts model for DCE MRI, and compared it briefly against implementations in IDL [20] and R language [28]. DCEMRI.jl was reported to be 24X faster than DCE@urLAB; both implementations used the Levenberg–Marquardt algorithm (LMA). Additionally, DCEMRI.jl was reported to be 10X faster than dcemriS4, but this is not a straight comparison because dcemriS4 uses a Bayesian hierarchical approach for curve fitting that is more demanding than the LMA.

Much research has been done on compiling and automatically parallelizing MATLAB [6,7,11,18]. We did not compare the Julia and MATLAB parallel implementations against what the most active MATLAB compiler project, McLab at McGill, can perform. This is future work in terms of understanding how much the type system in Julia helps over the dynamic typing in MATLAB. One important consideration in the selection of Julia was that it is a programming environment that is gaining significant community support. For the medical imaging research community to switch to a new programming platform, the platform will need to show signs of significant community support and longevity.

A review of the literature found no references comparing Julia's performance to that of other languages for the same algorithm in an authentic parallel programming application (i.e. beyond microbenchmarks).

7 Conclusion

Although the initial MATLAB code processed the benchmark data set on the order of minutes, a typical experiment for a medical imaging researcher often includes more and larger images, taken from dozens of patients, and may take weeks to run. The almost 60x speedup achieved using Julia would reduce weeks to hours, removing a significant constraint on researchers. Based on our experience while writing this paper, Julia appears to be a very attractive, emergent programming language for scientific computing.

As a result of our work on the DCE MRI algorithm, and his subsequent investigations into Julia, the research scientist has adopted the following model: (1) prototype and validate in MATLAB, (2) use MATLAB to identify bottlenecks, (3) port performance-critical portions of code to Julia. Taking into account the results of this case study, we feel that it would not be difficult for other research groups (who have already navigated the MATLAB learning curve) to similarly utilize Julia. Furthermore, since none of the performance gains found in this study resulted directly from the use of MRI data, it seems likely that other scientific computing applications could be prototyped rapidly in MATLAB, then efficiently ported to Julia for high-throughput applications.

Acknowledgment

An allocation of computer time from the UA Research Computing High Performance Computing (HPC) and High Throughput Computing (HTC) at the University of Arizona is gratefully acknowledged by the authors.

References

1. G. Almási and D. Padua. Majic: Compiling matlab for speed and responsiveness. In *ACM SIGPLAN Notices*, volume 37, pages 294–303. ACM, 2002.

2. V. R. Basili, J. C. Carver, D. Cruzes, L. M. Hochstein, J. K. Hollingsworth, F. Shull, and M. V. Zelkowitz. Understanding the high-performance-computing community: A software engineer's perspective. *IEEE software*, 25(4):29, 2008.
3. J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. *arXiv preprint arXiv:1411.1607*, 2014.
4. J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012.
5. J. Cárdenas-Rodríguez, X. Li, J. G. Whisenant, S. Barnes, R. Stollberger, J. C. Gore, and T. E. Yankeelov. The basic principles of dynamic contrast-enhanced magnetic resonance imaging. In R. Bammer, editor, *MR & CT Perfusion Imaging: Clinical Applications and Theoretical Principles*, chapter 31. Lippincott Williams & Wilkins, Philadelphia, PA, 2016.
6. A. Casey, J. Li, J. Doherty, M. Chevalier-Boisvert, T. Aslam, A. Dubrau, N. Lameed, A. Aslam, R. Garg, S. Radpour, O. S. Belanger, L. Hendren, and C. Verbrugge. Mclab: an extensible compiler toolkit for matlab and related languages. In *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, C3S2E '10, pages 114–117, New York, NY, USA, 2010. ACM.
7. A. Chauhan and K. Kennedy. Optimizing strategies for telescoping languages: Procedure strength reduction and procedure vectorization. In *Proceedings of the 15th ACM International Conference on Supercomputing*, pages 92–102, New York, 2001.
8. M. Chevalier-Boisvert, L. Hendren, and C. Verbrugge. Optimizing matlab through just-in-time specialization. In *International Conference on Compiler Construction*, pages 46–65. Springer, 2010.
9. L. De Rose and D. Padua. Techniques for the translation of matlab programs into fortran 90. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(2):286–323, 1999.
10. J. B. DeGrandchamp, J. G. Whisenant, L. R. Arlinghaus, V. G. Abramson, T. E. Yankeelov, and J. Cárdenas-Rodríguez. Predicting response before initiation of neoadjuvant chemotherapy in breast cancer using new methods for the analysis of dynamic contrast enhanced mri (dce mri) data. *International Society for Optics and Photonics*, SPIE Medical Imaging:978811–978811, Mar. 2016.
11. L. DeRose, K. Gallivan, E. Gallopoulous, B. Marsolf, and D. Padua. A matlab compiler and restructurer for the development of scientific libraries and applications. In *Preliminary Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, pages 18.1–18.18, May 1995.
12. J. Doherty, L. Hendren, and S. Radpour. Kind analysis for matlab. *ACM SIGPLAN Notices*, 46(10):99–118, 2011.
13. A. W. Dubrau and L. J. Hendren. *Taming matlab*, volume 47. ACM, 2012.
14. V. Kumar and L. Hendren. Compiling matlab for high performance computing via x10. *Sable Technical Report*, 2013(03), 2013.
15. X. Li and L. Hendren. Mc2for: A tool for automatically translating matlab to fortran 95. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 234–243. IEEE, 2014.
16. D. Markonis, R. Schaer, I. Eggel, H. Müller, and A. Depeursinge. Using mapreduce for large-scale medical image analysis. *arXiv preprint arXiv:1510.06937*, 2015.
17. MathWorks. Matlab parallel computing toolbox users guide. http://www.mathworks.com/help/pdf_doc/distcomp/distcomp.pdf, 2016. Accessed: 2016-05-20.

18. V. Menon and K. Pingali. A case for source-level transformations in MATLAB. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 53–66, Berkeley, CA, 3–5 1999. USENIX Association.
19. J. P. O'Connor, A. Jackson, G. J. Parker, and G. C. Jayson. Dce-mri biomarkers in the clinical evaluation of antiangiogenic and vascular disrupting agents. *British journal of cancer*, 96(2):189–195, Jan. 2007.
20. J. E. Ortuño, M. J. Ledesma-Carbayo, R. V. Simões, A. P. Candiota, C. Arús, and A. Santos. Dce@ urlab: a dynamic contrast-enhanced mri pharmacokinetic analysis tool for preclinical data. *BMC bioinformatics*, 14(1):1, 2013.
21. N. Otsu. A threshold selection method from gray-level histograms. *Automatica*, 11(285-286):23–27, 1975.
22. J. P. B. OConnor, P. S. Tofts, K. A. Miles, L. M. Parkes, G. Thompson, and A. Jackson. Dynamic contrast-enhanced imaging techniques: Ct and mri. *The British Journal of Radiology*, 84(Spec Iss 2), 2011.
23. S. Radpour, L. Hendren, and M. Schäfer. Refactoring matlab. In *International Conference on Compiler Construction*, pages 224–243. Springer, 2013.
24. G. Sharma and J. Martin. Matlab: a language for parallel computing. *International Journal of Parallel Programming*, 37(1):3–36, Feb. 2009.
25. D. S. Smith, X. Li, L. R. Arlinghaus, T. E. Yankeelov, and E. B. Welch. Dcemri.jl: A fast, validated, open source toolkit for dynamic contrast enhanced mri analysis. *PeerJ*, 3:e909, 2015.
26. R. Smith-Bindman, D. L. Miglioretti, E. Johnson, C. Lee, H. S. Feigelson, M. Flynn, R. T. Greenlee, R. L. Kruger, M. C. Hornbrook, D. Roblin, L. I. Solberg, N. Vanneman, S. Weinmann, and A. E. Williams. Use of diagnostic imaging studies and associated radiation exposure for patients enrolled in large integrated health care systems, 1996-2010. *JAMA*, 307(22):2400–2409, 2012.
27. R. Smith-Bindman, D. L. Miglioretti, and E. B. Larson. Rising use of diagnostic medical imaging in a large integrated health system. *Health Affairs*, 27(6):1491–1502, 2008.
28. B. Whitcher, V. J. Schmid, et al. Quantitative analysis of dynamic contrast-enhanced and diffusion-weighted magnetic resonance imaging for oncology in r. *Journal of Statistical Software*, 44(5):1–29, 2011.
29. Matlab execution engine. http://www.mathworks.com/products/matlab/matlab-execution-engine/. Accessed: 2016-08-26.
30. Loren on the art of matlab: Run code faster with the new matlab execution engine. http://blogs.mathworks.com/loren/2016/02/12/run-code-faster-with-the-new-matlab-execution-engine/. Accessed: 2016-08-26.
31. Parallel computing toolbox. http://www.mathworks.com/products/parallel-computing/, 2013.
32. Techniques to improve performance. http://www.mathworks.com/help/matlab/matlab_prog/techniques-for-improving-performance.html.
33. Julia timing practices. http://docs.julialang.org/en/release-0.4/manual/performance-tips/#measure-performance-with-time-and-pay-attention-to-memory-allocation. Accessed: 2016-08-26.
34. Parallel computing - julia language 0.4.7 predocumentation. http://docs.julialang.org/en/release-0.4/manual/parallel-computing/. Accessed: 2016-07-11.