Julia Language Feature Analysis

Matthew Farrell
Department of Computer Science
Urbana, IL
mbfarre2@illinois.edu

David Taylor
Department of Computer Science
Urbana, IL
djtaylr2@illinois.edu

***Abstract*— Julia is a free, open source programming language used for high performance technical computing, and attempts to provide an interface that is familiar for those attempting to learn a new technical language [1]. While it is a relatively new language (work began in 2009 and 1.0 was released in 2012) the author's goals of an open source, efficient, and reduced-overhead language are quickly becoming reality. In this paper, we have collected a corpus of Julia programs which encompass nearly the entire set of publicly available Julia source code in the Git repository. We investigate the sizes of these programs, how features are being used/how often they are being used, and how the language is handling some of the bigger challenges of technical computing. Additionally, we hope to educate those who may be looking to perform their own feature usage studied by presenting risks and obstacles to account for when running such research.**

With the ability to investigate a young language at such an early point, we hope to provide the creators and others who may be analyzing programs in Julia, unparalleled insight into the near-complete body of work in Julia that has been made available to the public. These insights can be used to improve the language itself, as well as assist those looking to build tools which rely upon Julia.

***Index Terms*—software engineering, programming languages, Julia, feature usage**

I. Introduction

The demand for high performance and scientific computing is growing. The need for quick solutions to difficult questions persists. In response to this demand new programming languages have attempted to assist in the need to quickly compute complex mathematical problems. The Julia Programming language attempts to provide programmers a modern dynamically typed language, for technical computing while maintaining the performance of seasoned strictly typed languages.

While boasting to almost match or exceed the performance [1] of the C language, Julia offers modern programmers a quick and familiar syntax which aims to cut down the lines of code necessary for most technical computing. An open source language that was, from its creation, built in a distributed manner, offers the technical computing world an open source alternative to current standards.

To our knowledge we present the first usage study that answers how the software industry has adopted the language, and identify trends within its usage. Through this study we aim to answer several questions:

Question One: How large are Julia Programs?

Question Two: Which features and functions define the core of language? Are there any identifiable common functions that are either left out or always included in each program? (Parallel feature focused)

Question Three: What features are users experiencing issues with and what areas of technical computing go unaddressed that Julia can or does address?

Question Four: Are the parallel features in the Julia Programming language being used? And how often?

Question Five: Can we identify an adaptable process for feature usage on technical languages?

Our study will benefit the industry in many ways that will help engineers and researcher in deciding on whether the language solves their needs by presenting them with the most common, most under-utilized, and most requested technical features of current implementations.

First, our study will provide insight into the subset of problems that the programming language is capable of solving by examining current implementations, empowering the engineers with clarity on whether this free and open source solution is good enough to solve their technical problems.

Second, the paper will provide insight to developers in identifying shortcomings of the language, by showing how the industry has worked with the language currently, which features developers have avoided, and aggregating direct user feedback to show where higher-volume issues lie. Researchers and vendors will have the added clarity, and ability to focus on the features that are commonly used or requested in the language, providing a springboard into further implementation/research.

Third, the open source tools that we have created in an attempt to study the usage patterns of this technical language will be freely available to future researchers to benefit from and improve. We hope this will encourage deeper analysis of Julia and other technical languages, and have provided additional areas we think may be of great value to the research community.

Fourth, we contribute what we feel are helpful processes, benefits, and potential pitfalls of a feature usage analysis with thoughts on how the analysis could be improved. We attempt

to save future researchers time by listing issues and risks/problems encountered during the analysis process, with possible solutions, if available.

As mentioned previously, all tools created to collect the data in this report, are freely available [4].

II. Related Work

As of the publishing of this paper, we are not aware of any research done in regards to feature analysis and usage within the Julia language. We have studied a number of previous works, such as PHP Feature Usage [8] and Java Feature Usage [9] which provided us with additions to our own selected analyses in order to better help compare how the language is truly performing and how its usage relates to other languages.

These works utilize a number of third-party components and extracted relevant data. No tools were in development or available for this type of analysis, so we have created our own specific tools to pull relevant information from the corpus of programs we selected. This provides us with a much higher degree of control over the parsing of the code to ensure correct results, but as with any new software, may not be as robust as those with high utilization in the programming community.

Common statistics are used across all feature analyses covered, often including basic statistics such as file sizes, line counts, and feature grouping / distribution / coverage, which we have also included to provide closer comparisons amongst the languages. We attempt to identify any commonalities across similar languages, and how user patterns have developed. The language patterns are observed in some, but not all related works are analyzed. Often the base features of the language are discussed in conjunction with advanced features that are sometimes (but not always) unique to the language or perhaps rare amongst related languages. We analyze similar advanced or unique functionality to educate the reader on how such functions improve upon or create new functionality as opposed to its peers.

III. Corpus

A. Selection Process

The selection of projects for this analysis was chosen in the manner which provided the least bias - and that was by selecting every available Julia project that we could obtain via the primary repository for Julia programs, Github. Our tool [4] provided us with the ability to download and extract all source code from every non-trivial Julia project the site has publicly listed. We filtered out only those with very low download numbers and those which were clearly meant as non-functional tests by end users. In doing this setup, we were able to obtain over 70 valid programs with which to run our analysis. This is actually a large number considering the amount of proprietary research work that is done with technical languages, and the young age of the Julia language. Utilizing such a large subset of the available projects allows us to draw stronger conclusions about the usage of the language, and possible modifications to improve it should carry additional weight.

In addition to analyzing feature usage, we studied a number of user-submitted topics to various online groups [5] [6] [7]. These included development, user, and other community question/answer websites which provided us a measure of the recent work and possible trajectory the language may be taking as it matures. We utilized the most recent available ~3 months of data to determine these items. In not going farther back in time, we attempted to avoid counting issues or questions that have since been resolved in a fast-changing language such as Julia.

B. Program Size

Julia Program's Source Code Statistics

Mean	5142
Median	1134
Mode	105

Table 1

Julia programs varied greatly but overall tended to be small when compared to other modern languages [4]. The average Julia Program was 5142 lines of code. At first glance this does not seem very small, but upon a closer look we see that a majority of the programs are in fact smaller, but the average is bigger due to a couple extremely large programs. Looking at the median size of a program gives us a more accurate depiction of your Julia program, with the lines of code being 1134, and the mode of 105 [13].

IV. Research Method

We implemented three tools to assist in the data gathering, from which we could collect and store, in a quickly computable form, all the grammar used in the Julia Programming Language. We studied 72 of the most popular open source Julia Programming Language and analyzed 378,288 lines of code in order to determine how developers are adopting the language.

The first tool JGOGETTER will parse all the functions and data structures documented on the Julia programming languages website and stores it in a JSON file. This enables us to compare usage in programming languages and gather statistics on programming behavior. The JGOGETTER inspects the documentation of the programming language and takes a snapshot of the available features, and its associated data.

GITGET assisted in the collection of 70+ open source projects whose description listed them as Julia programs. This tool would collect, and unzip the programs into a local folder on a computer allowing further data gathering. GITGET allowed us to capture an image of how developers had implemented the language by collecting all the programs with one star or greater. This condition allows us to avoid trivial or test programs not created for actual projects.

Our final tool, SPAGHETTI, navigated the source folder and it's collection of Julia programming, collecting data on each program. The program parsed all files with a .jl notation which is an identifiable Julia Program and ignored all others. The SPAGHETTI collected data on each program, and

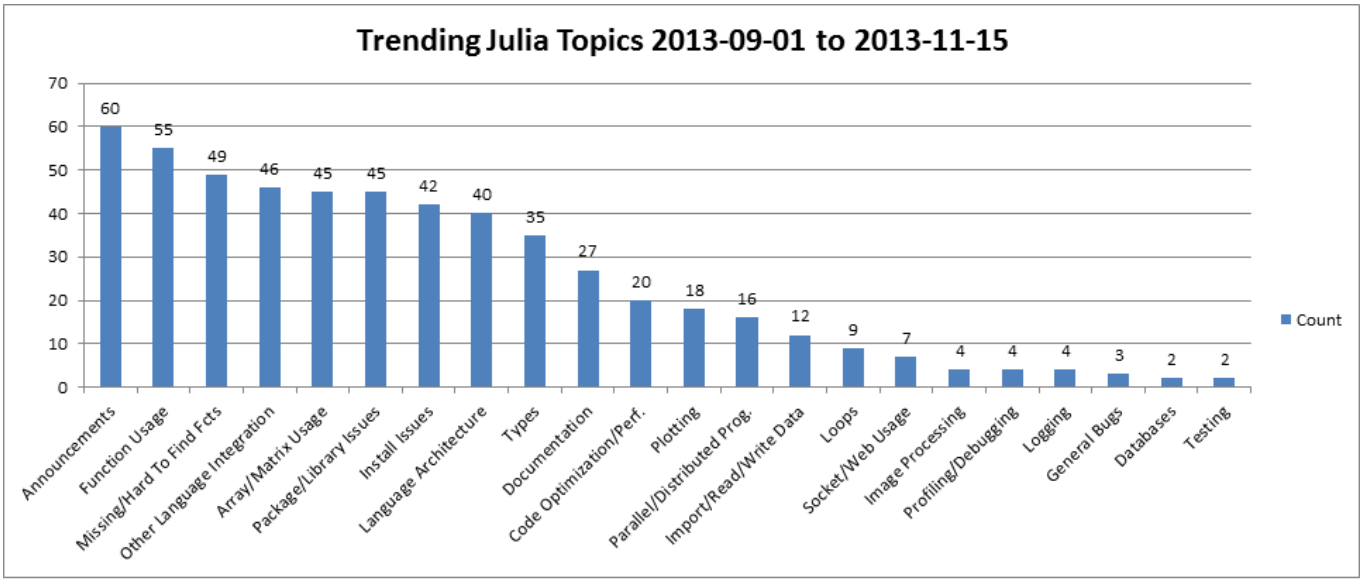

Figure 1

collective usage. The input to the program is a source folder containing Julia Programs, and the output is a website detailing usage data.

Four categories of usage data were collected for further analysis. Core language data was a usage count for each grammar item in the Julia Language throughout all 72 programs, as well as individually. Average Program Data is the mean, median, and mode of the Julia program's size compared to the whole corpus of data. Parallel Usage Data is the usage of the parallel features in each program, and the count of features used. Total Lines of Code Data is the total lines of code for each program, as well as the total corpus. Categorization was also assigned to each piece of the grammar utilized, based on Julia's own categories as found in the documentation [10].

In addition to source code analysis, our research also included reviewing three of the largest Julia groups available for technical assistance on the internet. By manually reviewing these groups, we were able to collect data on trending features/issues, user concerns, and other user-submitted items that may not traditionally be available to code analysis. An analysis of the core language data yielded a categorization (per the Julia documentation [10]) of all functions used across the corpus and attempted to show how trends across user's usage and user's comments (via forums / groups / questions / etc.) can be analyzed together and show where the greatest needs of the language lie.

V. Julia Analysis

A. User Issues/Feedback

In our attempt to better understand not only the direct usage of features, and the lack thereof, we collected information on various user/developer-submitted topics to a few of the highest volume Julia groups available online.

Examining Figure 1, the top three categories provide a great snapshot of the Julia language and how it is truly in a state of flux. The number of announcements, which includes new features, fixes, and new packages, outnumbers any other single category. With all this new functionality comes a wave of questions and unexplained features that are shown in the next two groups below Announcements. Feature usage and missing functionality account for a large number of requests as users look to adopt the newly added items, and build even more complex applications using them. We can cross-reference these requests with the number of closed issues in their core language Github repository to see how frequently these types of issues are addressed. It appears as though the answer is quite frequently, as bugs, features, and build repairs are the top 3 closed issue categories [16]. Documentation questions, while not at the top of the chart, were trending strongly upward in more recent postings, and in combination with the Function Usage and Missing Function categories, showed that developers need to put forth the same effort in explaining their new functions as they are putting forth to add in new ideas. Without the support of strong usability, the language will have a difficult time growing the user base it desires.

The Julia language has some strong quantitative results in Figure 1 to support the notion that the founders of Julia have made great strides in luring developers from other languages. This can be seen by the large number of questions being asked and announcements being made in packages which relate to integration with other technical computing languages. Another example can be seen in the contribution stats of the language itself, which signify a far larger investment of developer time and shows higher interest in the project [17]. The steady trend upward in contributors and commits validates the work the creators have been putting forth to make the language accessible. While languages such as R and Matlab are far higher in volume of usage [2], we see a strong support by users to gain the performance increases Julia offers [1] but also want familiar hooks to languages they are familiar with. Packages such as IJulia [3] and strong similarities in syntax with R/Matlab for some functions make for a language that most technical users can learn quickly, but also utilize the benefits of

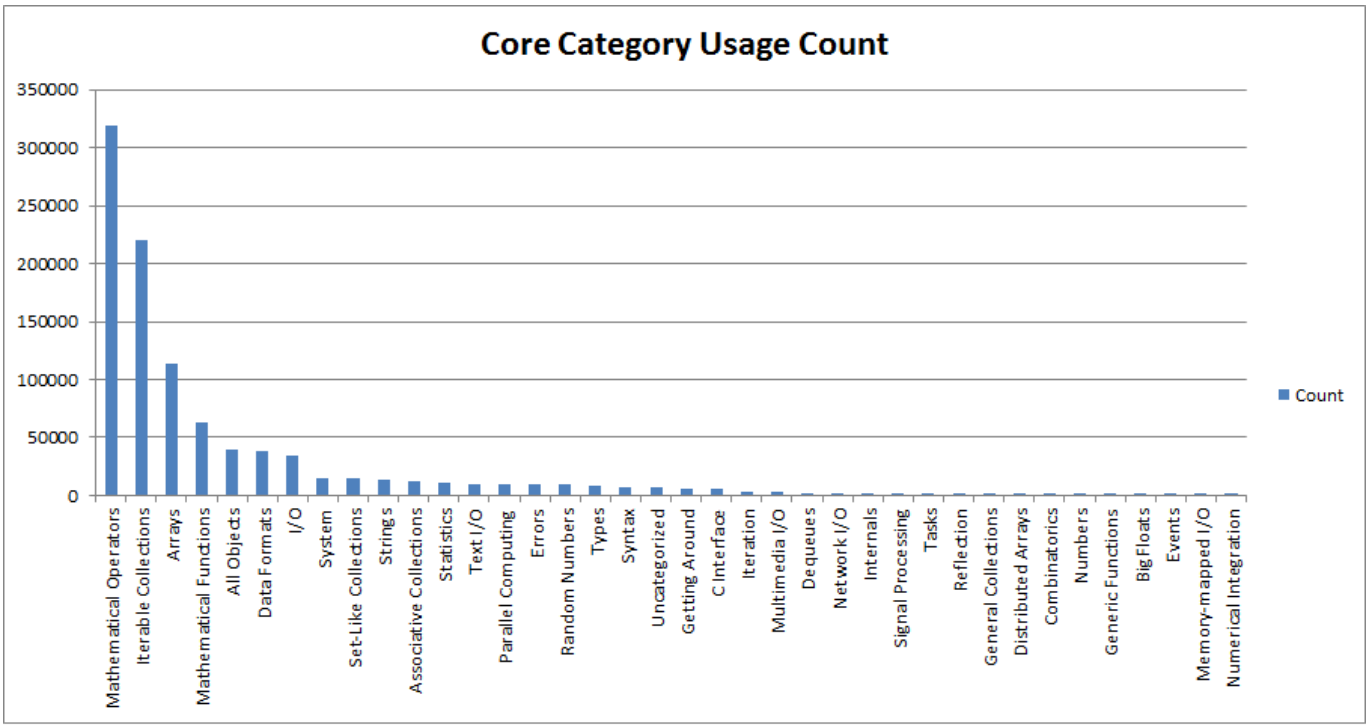

Figure 2

Julia in their previous work. Since Julia looks to improve upon these very popular languages, it would certainly make sense to make a concentrated effort by the developers of Julia to continue on this path - expanding the appeal of a language which can run faster, and integrate easily with existing code, thus reducing transition issues for projects.

B. Usage Trends/Patterns

1) Category Usage Counts

In our endeavor to glean some additional information on a more macro level, before delving into a function-level analysis, taking a look at the categorization of each feature used and summing the results on our full corpus reveals some interesting patterns (see Figure 2). For detailed information on what each category contains in terms of functions and their summary, please review our reference link for Julia documentation [10], as it is too large to be directly included with this report. The data provides strong data regarding high usage of iterable collections/arrays and perhaps some missed opportunities when working to integrate with other technical languages.

First, with even a shallow analysis of Figure 2, we can see that iterable collections and arrays (matrices) dominate the usage of Julia programs, ignoring for now the fundamentals like math operators. As with most technical languages, this area of focus is critical to the majority of the full corpus of programs we selected for analysis due to their highly mathematical and often, research-based focus. Despite it being a relatively common-sense conclusion, it is worth mentioning here that performance improvements in base array functionality, and the research into adding additional core functions to support more complex features in calculations involving arrays, should be a top priority for the Julia developers.

Second, we can see some items that are being utilized heavily that perhaps deserve some additional attention and resources in order to possibly improve efficiencies or provide more complex functionality as a built-in feature. One of the most notable areas includes the category of "C Interface" in Figure 2, which aligns with "Other Language Integration" of Figure 1. Despite the high number of requests by users to be able to program or re-use some scripts in languages such as C or R, the utilization in projects is rather low, coming in at 22 of the 38 total categories measured with 5553 calls across all programs. This would appear indicative of one of two possible scenarios: 1) The cross-language support is not strong enough to warrant usage for most users or 2). Other libraries (those not utilized in this study) are providing a superior structure to programmers. If the cause is the former, the solution is simply to investigate better (or additional) ways that cross-language scripting can be enhanced. Adding the ability to natively call some of the most popular R or C functions would likely be appreciated, as opposed to being forced into external script execution. If the issue is simply being addressed in a superior foreign library, the developers need to take under consideration the possibility of merging such features into the core language. While it could still be an optional installation, including this built-in option gives developers the flexibility they will likely need to transition into using Julia full time for their technical computing needs. Further research would be needed to determine what the cause of this discrepancy is, and what solution would be best for the developers.

Looking at a couple points mentioned above together yields that giving additional attention to array/matrix

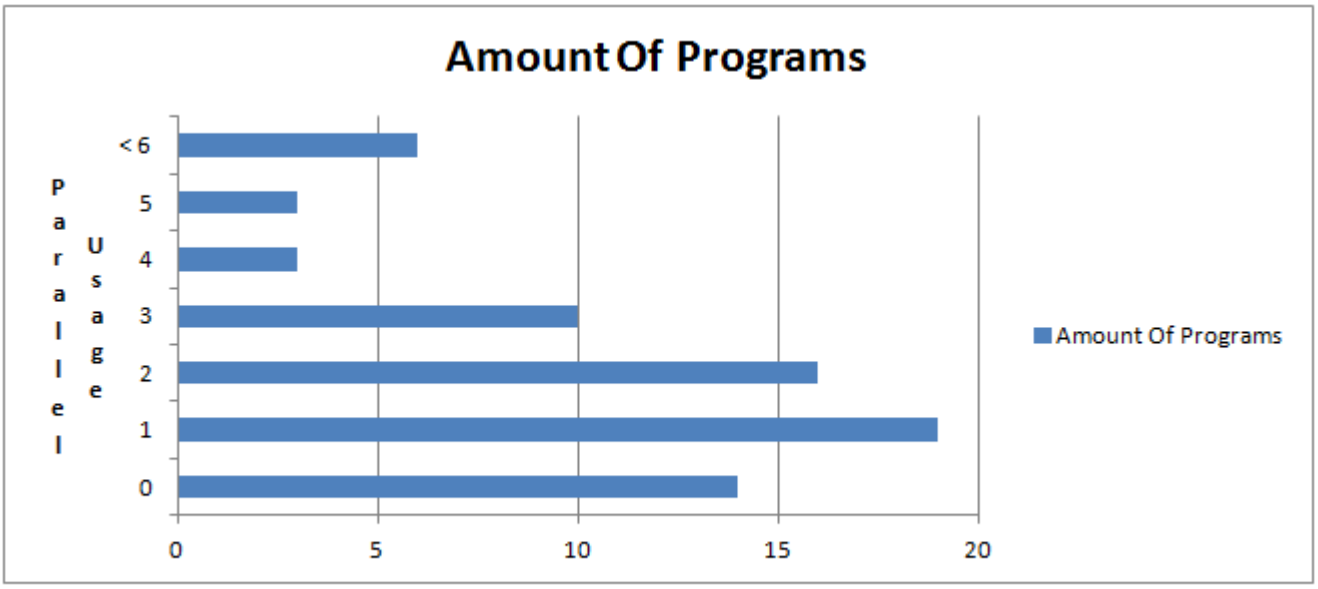

Figure 3

mathematics to improve their calculations to both out-perform and provide more features than similar technical languages, will increase exposure and lessen pressure on developers to interface with better known technical-languages over the long term. Since the syntax of Julia already attempts to mock current popular languages (for lower entry barriers) it would be wise to improve upon these existing prototypes, if possible, and make the transition as easy and the improvements users receive, as obvious as possible. Supporting other languages within a framework that a programmer wishes to surpass the performance and utility of similar languages can be counter-intuitive, but is a common practice for building a user base for a start-up.

2) Parallel Usage

The authors of the Julia programming language designed and use Julia for parallel computing [1] and boasts of a dynamic approach to parallel computing. With relative ease a programmer can identify a parallel section of code.

For example:

```
nheads = @parallel (+) for i=1:100000000
        int(randbool())
end
```

This simple for loop will execute in parallel, with little effort on the programmer's part. Due to this simple yet powerful paradigm, enthusiasts argue that Julia is a real contender in scientific computing. By analyzing our corpus of work we decided to see how the community was using the parallel features provided in the language. On average a Julia program makes 2.4 calls to a parallel feature where a parallel feature can be defined as any function in the documentation categorized as “Parallel Computing”, omitting @parallel calls [12].

Looking at 71 programs we found a Mean of 2.4, a Median of 2, and a Mode of 1 instance(s) of a parallel feature being used in a program. Figure 3 shows a layout with the amount of programs on the x axis and an instance of a parallel feature being used on the y axis. This graph shows that about 69% of the programs analyzed in our corpus of data used less than three calls to a parallel feature. Furthermore only six programs used 6 or more calls in a given program, with more than 10 calls being extremely rare. Only two programs showed instances of more than 10 instances of a feature usage, and 14 programs had no instances of a parallel feature in their program. In other words, 20% of our corpus had no instances of a parallel feature being called, and a large majority using a feature a 2-3 times.

Looking at the parallel features individually we noticed that a little over half of all the instances of parallel features are due to two features put and take. In total our corpus gives us 168 instances of a parallel feature being called and 85 of those instances are put or take. Out of the 24 parallel features in the documentation, only 20 features were used. The features rmprocs, isread, timedwait, and @fetchfrom are never used. Figure 4 shows the uneven distribution of calls, with relatively few getting more than 10 instances of use and a majority being used less than that. The functions that show the greatest usage tend to be features that are predominantly classic parallel features. In other words, these features are basic parallel features and are available in most parallel libraries in other languages.

Using this data we can start to see that although the language was built for distribution, the community is less likely to venture out past the basic @parallel call with a simple for loop. Looking at the feature calls specifically we notice that the bread-and-butter parallel features like wait, take, and put comprise a vast majority of the parallel usage. Compare that with the parallel usage per program we see that there are relatively few programmers venturing out past the basics of

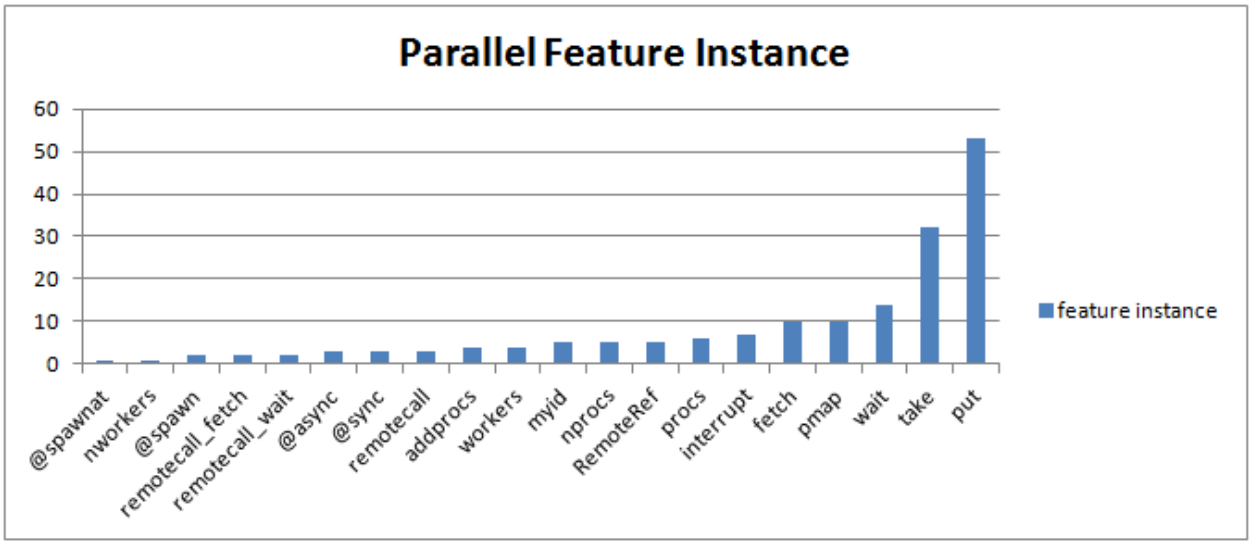

Figure 4

parallel computing, and are not necessarily doing it often. On the other hand 80% of our corpus uses parallel features; this is not including the basic @parallel call described earlier. It's apparent that Julia programmers do in fact use parallelism, but aren't necessarily diving deep into the standard library with their algorithms and programs. Combining these findings with our earlier items on user feedback and questions regarding parallel functionality along with performance concerns, shows that further investigation may be required in this area. Documentation of these advanced features along with the feature's effectiveness would be obvious starting points for follow-up research and reasons users are possibly not taking advantage of the full suite of parallel features in Julia.

VI. Recommendations

A. On Julia Implementation and Maintenance

Looking at the information we have gathered regarding usage by end-users, trends in feedback, documentation notes, and testing the language ourselves, we now look to provide a few recommendations for furthering the Julia cause and improving the language moving forward. These include providing a richer support structure for packages integrated with the primary source, attracting users to more advanced features which may benefit them and are underutilized (specifically, as related to our parallel feature analysis), and continued strong integration with other popular technical languages.

First, packages that have either been fully integrated into the main source or are simply widely used provide invaluable additions to Julia, that (based on forum feedback) a large percentage of users are utilizing. Adding an official support structure or perhaps a guide for contributors to support/document their add-ons will only serve to make the language that much easier to use. In this way, the developers and users of the language are able to quickly find and integrate packages which relate to and/or solve their technical programming requirements. As it stands now, far too many packages (even those considered integrated or "core") often have large numbers of questions on both installation and usage. Setting some standards for Julia package support would go a long way in serving a larger audience who would gain a lot from the high-performance of Julia, but are currently running into avoidable issues during its usage and installation. Careful attention needs to be given to this topic, as to avoid being onerous and creating too large a barrier for those looking to extend the language and perhaps looking to integrate with the core functionality. By offering some incentives such as official recognition of packages that are Julia-compliant would encourage such behaviors, setting a higher bar for add-ons, while giving new users and reliable repository of packages for their setup.

Second, the property most championed by the developers of Julia is performance, and our investigation into parallel features showed how little performance-improving parallel functions are actually used. Directing users with more complex code samples and integration of these features into some of the existing high-usage add-ons for Julia could go a long way in showing the true power of the Julia language. While the core of the language is inherently faster at many computations than C or R, advanced parallel features can sometimes increase this exponentially, and should be highlighted to even better showcase how users can continue to improve their program's performance. In addition, IDEs that may be available for Julia would be an ideal location to offer up recommended changes or auto-corrections to more efficient parallel features. This could help increase awareness at a minimum, and potentially lead to more widespread usage of more complex features developers are unaware of.

Thirdly, while Julia has already provided a number of components to help ease the entry into Julia syntax and compilation, the continued support and updating of such packages is important in growing the user base. Integration with languages such as C, R, Python, and others has undoubtedly led to a number of new developers contributing to

the language and adding invaluable new functionality to Julia. Adding support for the most popular languages, perhaps even consideration for non-technical languages such as JavaScript or Java may be prudent. These are not traditional languages for complex mathematical calculations, but by providing extension of the Julia features into this realm, you are able to appeal to a much wider audience, and perhaps addresses a sector of the scientific community who may not be as willing or able to obtain the (admittedly more difficult) skills of C or R. The high usage of the Python-Julia extensions speaks to the potential popularity of such packages, and we believe sets the needed precedent to expand into OO languages.

B. On the Feature Usage Process and Risk

In order to provide the maximum benefit from this research to readers, this section will briefly detail a few items which may help creators of follow-up or similar research in the area of feature usage. We noted that important risks exist in selecting your analysis tools, corpus, and even the selection of language for analysis. We discuss these risks below with potential pitfalls to try and avoid.

Analysis tool selection was critical in our project, and our initial investigations took us in the direction of meta-programming languages such as Rascal. This turned out to be a path which could have been useful in creating some interesting charts of language structure and relationships -- but a preliminary test run with the tool showed a relatively high barrier for entry for novices of meta-programming. We attribute this to simply our inexperience with the structure and implementation of the language. Combine this issue with a limited time frame for the research to be published; we were unable to continue our work with it for this particular piece of research. This example illustrates the risks that exist for the tools you may assume are simpler-to-use for creating the basic foundation of your feature usage analysis, and additional time should be budgeted for this critical activity. Without proper tools and assurance that your usage will provide reliable results, you may be undermining your research before it begins -- hence our decision to write a tool in languages suited to our team, and ensure accurate results were provided.

While we did not run into issues with the selection of a corpus of implementations to analyze, we felt it was important to note what potential issues we saw, and other issues we discovered when discussing feature analysis research with other teams doing similar work. First, we were able to utilize a selection of all non-trivial Github (the primary Julia repository) projects because of Julia's young age. Other projects will often not be as fortunate, and project selection can be difficult with a few potential solutions to this issue being offered by other researchers [14] [15]. Noting projects which will provide high value for your metrics and potential comparisons to existing software/technologies must be near the top of your priorities. In particular, we noted that some researchers run into problems when they are attempting to benchmark themselves against more widely accepted languages in their area of specialty. Even finding a handful of programs which are implemented (ideally) in your analyzed language as well as in the benchmark language would be best. This is often difficult, and many times will require additional time to perhaps rewrite heavily utilized functionality in your language to show how the conversion can be done and the improvements it provides. Narrowing down projects for potential rewrite (if necessary) can hopefully be done on the basis of robustness/popularity of the benchmark, as well as your preliminary results of LOC/feature usage that you are able to obtain; thereby making selection of programs with features you are focusing on a much more fruitful process.

Corpus selection shares a few of the same issues as language selection for analysis, but will briefly mention a couple notes here on this step. First, making sure related work does not completely encompass your suggested research is important. While reinforcing conclusions of other projects may be helpful, it will not be if you are seeing the same result for the 20th time. Your language selection for analysis will affect the amount of work found and influence how deeply you will need to look to find unique research to conduct. In our case, utilizing such a new language with a niche in performance improvements over already low-utilization technical languages offered a large set of projects to analyze and findings/patterns in the output of our tool were often notable. Larger/more popular languages for analysis will offer longer time frames to create the tools necessary for unique research.

C. On Further Work/Follow-Up Studies

The limitations of this study mostly involved time-based constraints due to the nature and environment of the research, but offers up a foundation on which further work could be derived. Our team was able to create a starting point with a robust code retrieval setup, a code parser for a few metrics, and significant user-related data in recent trends for Julia-related topics. These could certainly remove a significant portion of the start-up time of research on Julia usage, and the code to be extended in a few ways. First, more analysis on the context of features on a broader scale (perhaps on trending topics we identified in our user submission research) could yield findings that would assist in creating functionality which is more user-friendly and directly address the largest volume of user comments. We delved farther into the parallel setup concurrently with our user research for efficiency reasons, but if they had been run in a linear fashion, some of the top topics (which included parallel/optimization) that were found from users would have likely been included in the focus of our parsing tool. This is a logical step forward in the evolution of the tool, and would certainly yield concrete results on more topics for addressing in Julia.

Also, while the Julia repository offers a number of comparison metrics for performance, it would seem prudent to run a study on a large set of scenarios which are common for technical languages. This additional data would hopefully (for the Julia developers) provide a more robust argument when new users are looking for a technical language for their projects. Our work assists in finding areas of high user concern, and also identifying where Julia developers may have spent more of their time fleshing out advanced functionality for such concerns.

VII. Conclusion

The Julia Programming language is young, evolving, and disruptive, albeit in the smaller field of technical programming. The programming language stands to benefit a subset of the software engineering community who currently use related statistical programming languages like MatLab and R.

Currently, the Julia Programming language assists programmers who create small programs, using mainly the mathematical standard library. Julia appears to be focusing on a number of topics which are highly requested including parallelism and foreign language integration, while other issues such as documentation and more complex matrix mathematics still require some addressing.

The Julia programming language has done a good job assisting programmers in creating mathematical programs in a modern way but may be experiencing some growing pains in the area of parallel computing utilization. In Julia, basic for-loop parallelism is incredibly easy and fast. Although the programming language can run many algorithms in parallel easily, we have not seen any evidence that Julia programmers are diving deep into parallelism, which warrants further research as to user's opinions and usage. While parallel processing results were impressive, high memory usage in some areas dampened the enthusiasm for Julia as truly revolutionary. Yet, from their rate of change from our forum analysis, improvements may be imminent. A major advantage that this programming language has over other similar languages is that it's open source, while other are not. There is a subset of statistical programmers who could benefit from using this free and open source language, who need speed, simplicity, and basic parallelism. To the knowledgeable programmer, the open-source status is often invaluable, especially for highly-specialized projects that Julia would likely be considered for.

This paper explored a number of high-volume forums for trending issues and complaints, and found some positive and some negative items. The original development team appears to be heavily involved in user-requested items, but is not always able to address them in a timely matter due to sheer volume. While seemingly successful at getting major items in the core mathematical areas optimized [1], other, more complex issues, such as advanced matrix manipulation, and missing documentation seem to be becoming a large issue in the near future.

The tools developed for this research, provide users with an automated way of loading the latest grammar of a language, selecting a very large cross-section of the available source code from repositories such as Github, and finally parsing that code for the grammars initially loaded. This base information provides a number of opportunities for insight into whichever language is under analysis, and should save future researchers valuable setup time. The automated generation of base reports (very extensible) provides an added benefit for researchers who require formatted output for reference or documentation as well.

Lastly, our process in researching this feature usage yielded some helpful results in risk-aversion for such projects, and allowed us to share our experience with those doing similar work. Being fully prepared with meta-programming tools, or knowing you are developing tools from the first day can hasten work considerably. Our work with a smaller language such as Julia allowed us to remove a large amount of bias from our corpus selection, and provide more concrete insight.

Addressing a number of strengths and weaknesses in the overall development process of Julia and current trends noted will help the founders and other contributors to better focus their efforts in making Julia fill the greatest needs of technical computing users. Continued work with the open-source parsing tools and suggestions provided, has the potential to lay the foundation for more in-depth work outside just the parallel arena. Utilizing user-submitted concerns and considering them in the selection of processes for analysis in future work regarding Julia research will prove fruitful.

Acknowledgment

We would like to express our appreciation for the guidance and feedback during our research from Professor Tao Xie from the Department of Computer Science at UIUC.

References

[1] Julia Language Home Page, http://julialang.org/

[2] StackOverflow language tag rankings, http://stackoverflow.com/tags

[3] IJulia - https://github.com/JuliaLang/IJulia.jl

[4] Research Tool/Results - http://julia.lifehug.com

[5] StackOverflow Julia Questions, http://stackoverflow.com/questions/tagged/julia-lang?sort=votes&pageSize=15

[6] Google group for Julia dev. Support, https://groups.google.com/forum/#!forum/julia-dev

[7] Google group for Julia user support, https://groups.google.com/forum/#!forum/julia-users

[8] M. Hillis, P. Klint, J. Vinju, "An Empirical Study of PHP Feature Usage: A Static Analysis Perspective," ISSTA 2013.

[9] R. Dyer, H. Rajan, H. A. Nguyen, T. N. Nguyen, "A Large-scale Empirical Study of Java Language Feature Usage," June 2013.

[10] Julia Base Library, http://docs.julialang.org/en/release-0.2/stdlib/base/

[11] Enthusiast Blog, http://www.johnmyleswhite.com/notebook/2012/03/31/julia-i-love-you/

[12] Parallel Features, http://docs.julialang.org/en/release-0.2/stdlib/base/#parallel-computing

[13] Program Size http://julia.lifehug.com/gqm/program_size.html

[14] M. Nagappan, T. Zimmerman, C. Bird, "Diversity in Software Engineering Research," ESEC/FSE 2013. August 18-26, 2013.

[15] E. J. Weyuker, "Empirical Software Engineering Research – The Good, The Bad, The Ugly," ESEM 2011. Sept 22-23, 2011, pp. 1-9.

[16] Github issue tracker (Julia) - https://github.com/JuliaLang/julia/issues?page=1&state=closed

[17] Github commits/contributors graph - https://github.com/JuliaLang/julia/graphs/contributors