PMAPREDUCE: A CONFIGURABLE AND INTUITIVE TOOL FOR PARALLELIZING REDUCE IN MAPREDUCE

BRYAN PYO* AND JUNG SOO CHU†

Abstract. MapReduce is a programming model that facilitates concurrent processing and generation of large data using parallel processing of multiple chunks of the data through a map function and a reduce function. In this paper, we introduce the PMapReduce function, which integrates the default parallelized map function from julia in the form of Distributed.pmap, as well as our own implementation of a parallelized reduce, which takes advantage of multiple tiers of reduce layers in order to allow multiple workers to reduce at the same time. We tested our implementation on three different problems: PageRank, Sorting, and TF-IDF. We found that our PMapReduce generally performs faster than the non-parallelized counterparts at the cost of more memory being needed. If the cost of the overhead of using PMapReduce was less than the benefit from the additional parallelization, then there was an overall increase in performance. We found that this was more likely to occur with larger input sizes. Finally, we made an empirical observation that for the sorting problem, the optimal number of parallelized workers stayed fairly consistent regardless of the size of the input sequence to be sorted.

Key words. MapReduce, Parallelization, PageRank, Sorting, TF-IDF, Julia

Code repository: https://github.mit.edu/jschu99/18.337-Final-Project

1. Introduction.

We are currently in a very digital era, where data is becoming more and more important, as well as our ability to properly process and collect this data. With the abundance of data, parallelization techniques such as MapReduce have also grown in importance. MapReduce is a programming model that facilitates concurrent processing and generation of large data by allowing the parallel processing of multiple chunks of the data. Figure 1 shows the structure of MapReduce. The first of the two main components of a MapReduce problem is a map procedure, which filters and sorts the input data into an easier format for the next step, the reduce method. The next component, the reduce method, then takes these outputs and performs an operation to combine them in a meaningful way into a single output. The benefit of this architecture is that the map operations and the reduce operations can all be done in parallel. Thus, instead of operating on one element at a time, using MapReduce allows multiple elements to be processed at the same time through multiple workers.

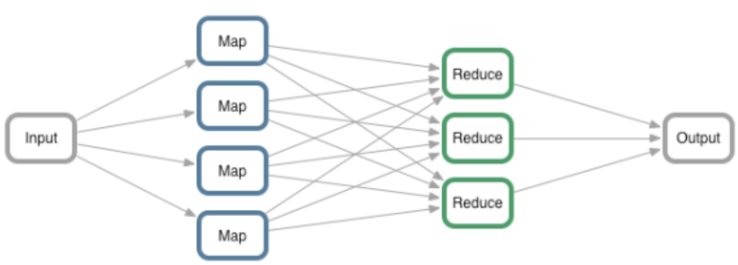

Fig. 1. *MapReduce structure*

Julia already has a default, parallelized version of a map function called pmap from the Distributed package. However, the focus of our project is to implement a

*MIT CSAIL (bpyo@mit.edu).
†MIT CSAIL (jshu99@mit.edu).

parallelized version of the reduce function that uses multiple tiers of reduce operations in order to perform multiple reduce operations at the same time. We combine the Dsitrbuted.pmap and our parallelized reduce function into a single framework, the PMapReduce.

There are five main classes of MapReduce problems. Metapatterns, organization patterns, summarization patterns, join patterns, and filtering patterns. We test our PMapReduce implementation on problems from three of these classes: metapatterns, organization patterns, and summarization patterns. Metapattern problems aims to find patterns within patterns within the data. We focus on the PageRank problem, which tries to measure how important a web page is by counting the number and quality of links to a page and is introduced in [7]. The central idea of PageRank is that the more important web pages on the internet are more likely to receive more links from other websites. For our implementation of PageRank, we use a simple, iterative algorithm that updates weights for each website in the input by analyzing the surrounding nodes. Organization pattern problems restructure the input data into a more relevant or easy to use structure. The problem we focus on is a standard sorting problem. The sorting algorithm we chose divides the array into multiple smaller arrays so that the reduce workers can individually sort each subarray in parallel. The final reduce step combines the smaller sorted arrays into the final sorted array. Summarization pattern problems group similar data to discover new information about the input data, such as word count. In particular, we focus on the TF-IDF problem, which measures how important each word in a document corpus is to that corpus, as described in [8]. This is done by finding the term frequency, which calculates how many times a word appears in a document and multiplying it with the inverse document frequency, which measures how many documents a word appears in.

We will first go over several related works, especially to our task of integrating parallelized reduce to Julia. We will then go over details regarding our implementation and overall design. We will begin by discussing our implementation of PMapReduce and then discuss the three problems we are testing our implementation on: PageRank, Sorting, and TF-IDF. We will finally conclude with the results of our experimentation on the run time and memory allocation of PMapReduce and our overall conclusions.

2. Related Works. The idea of MapReduce was introduced by Dean and Ghemawat as a way to process large amounts of data, inspired by the map and reduce functions often used in functional programming [1]. In this paper, they discuss how the map function can be used to process key-value pairs to generate a set of intermediate set of key-value pairs. A reduce function can then be used to combine this intermediate set of key-value pairs into a final relevant key-value pair. They discuss how their implementation can take any problem written in this format and automatically parallelize the problem and execute it across a set of machines. We expand upon these ideas by allowing inputs that are not necessary dictionaries and using multiple layers of reduce functions in order to allow for a higher level of parallelization spread across multiple workers on the same computer instead of across different computers.

Since Dean and Ghemawat published their paper on MapReduce, there have been many papers that explored the use of the MapReduce model to solve a variety of different problems. Li et al. used the MapReduce model in order to process and manage large-scale datasets in a distributed cluster [6]. They review how it can be used to generate search indices, perform document clustering, access log analysis, and perform a variety of other data analytics. Verma et al. used the MapReduce model in the field of biology in order to process genetic algorithms [9]. They were able to use

Hadoop, an open source implementation of MapReduce, to obtain stable results on genetic algorithm problems with up to 100000 variable problems. Finally, Ene et al. used the model to process large data in order to perform several different types of clustering, specifically k-center and k-median [2]. They were able to discover that their MapReduce performance performed equally or better than non-parallel implementations and better than other parallel implementations when using a sufficiently large dataset.

There have also been a few papers that explore implementing MapReduce models in Julia. Kavi discusses how they use Julia to implement several fast MapReduce algorithms to count word frequencies across a large number of documents [3]. Their first implementation was done on the CPU using two processes with MPI and their second implementation uses a GPU on Julia's CUDA library. Although, we did not use a GPU in our implementation, their implementation of finding the word frequencies was helpful in implementing our simpler algorithm for finding word frequencies. Another paper by Kourzanov uses a MapReduce model in Julia in order to perform simulations [4]. In particular, they use it to speed up a Digital Signal Processing (DSP) Intellectual Property (IP) model simulation for a Wireless LAN product. They found that with 120 workers, the MapReduce model was able to achieve speedups of around 40x and that with 480 workers, it was able to achieve speedups of around 260x. These results were very promising for our own implementation given that they also discuss how it was a fairly straightforward implementation of MapReduce for their simulation.

There has also been some work on parallelizing calculations in Julia with works from people such as Lee et al. that use parallelization to efficiently solve matrix calculations [5]. Although certain matrix calcuations can be formatted as MapReduce problems, they did not use the model for their matrix calculations. We will be taking that extra step of transforming our problems of interest into MapReduce problems and then further parallelizing the reduce function in Julia for these problems.

3. Design and Implementation. In this section, we discuss the implementation details for PMapReduce and its three application examples, PageRank, Sort, and TF-IDF.

3.1. PMapReduce. Julia already has an implementation for parallelized map in the form of Distributed.pmap. It spawns workers that can handle the given map task in parallel by splitting the input collection into batches. Its main advantage is that it provides an intuitive and easy to use tool for parallel computing, abstracting away details such as spawning the workers, distributing tasks, and combining them into the single output variable. It offers a multitude of features, such as configurable batch size and error handling. The PMapReduce that we implemented is an extension of this function, with parallelized reduce integrated as well. The goal is to provide an intuitive and easy to use function like Distributed.pmap while integrating parallelized reduce.

One way to achieve parallelized reduce is by having tiered reduce functions. Figure 2 shows the structure of PMapReduce. Suppose the map function generates n items in the output. Traditionally, this can be processed by a single non-parallelized reduce function that takes in an input collection and returns a single output. However, to parallelize this, we split the n map output items into m_1 batches ($n \geq m_1$) and spawn m_1 workers, each of which is running a reduce function on the batch. There will be m_1 outputs from this process. The aforementioned reduce workers form the first tier of reduce functions. With this design we can add multiple reduce tiers, where we let the ith tier spawn m_i workers. Then, with k total reduce tiers, the number of outputs after each reduce tier would be $m_1, \ldots, m_k$, a non increasing sequence. Note that we want the final output to be a single output, so m_k must be 1. In terms of performance, this design has benefits and drawbacks. Its main benefit is that all of the reduce tiers are parallelized, except for the last one. With a large input and suitable hardware, this can result in faster performance. However, the drawback is that compared to using a single reduce function, the total number of operations and memory allocation increase. So, the performance comparison results between PMapReduce and the traditional approach of combining Distributed.pmap with reduce depend heavily on the scenario.

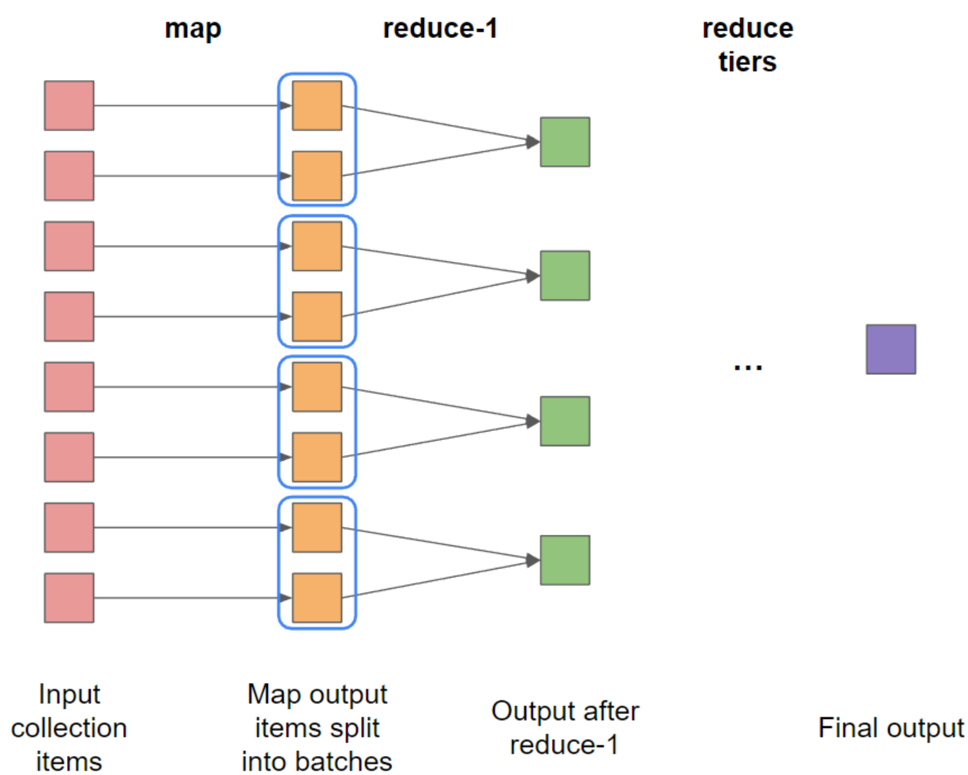

FIG. 2. *The structure of PMapReduce with tiered reduce functions*

Our implementation of PMapReduce takes in five inputs. The function inputs were designed so that as much of the implementation needed to run PMapReduce would be abstracted away inside the function while still offering a high degree of configurability. The five inputs are:

1. input: This stores the input collection for PMapReduce.
2. map_function: This is the map function that will be used by the Distributed. pmap part of PMapReduce.
3. reduce_functions: This is the collection of reduce functions to be used in the parallelized reduce part. Each reduce function is used in a reduce tier.
4. inter_results: This is a collection of preallocated collections for storing intermediate results after each reduce tier. The use of preallocated collections has

a performance advantage. This has to be passed in as an input, as opposed to automatically being defined inside PMapReduce, because the element type depends on the output type of each reduce function.

5. reduce_layer_sizes: This is a collection of integers for defining the size of the outputs after each reduce tier. This is equivalent to the number of workers each reduce tier should spawn. Since the last reduce tier should have only one worker, so that the final output is size one, reduce_layer_sizes should have exactly one less element than reduce_functions and inter_results. For robustness, if reduce_layer_sizes is larger than expected, only the first appropriate number of elements are used. Conversely, if it is smaller than expected, the missing elements are filled in with ones.

3.2. PageRank with PMapReduce. PageRank has many versions and different ways to implement using the MapReduce framework. Since our purpose is to compare PMapReduce and the traditional Distributed.pmap and reduce, not to implement PageRank most efficiently, we use the simple version of it. In this version, the weights for each webpage need to converge according to the following formula:

$$\begin{aligned} PR(A) =& \frac{1-d}{N} + d\left(\frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \dots\right) \\ d =& \text{Damping factor} \\ N =& \text{Total number of web pages} \\ PR(X) =& \text{The PageRank of web page } X \\ L(X) =& \text{The outdegree of web page } X \end{aligned}$$

Here, damping factor is the probability of the search engine user clicking one of the links in the current webpage, as opposed to going to a completely random webpage on the internet. We use the standard value of 0.85 for it. According to the above expression for PageRank, we can view the weight of a webpage as the sum of weight contributions from the webpages that link to it and a constant. Hence, we need to compute this efficiently in each iteration. Considering this, the iterative algorithm for computing PageRank is shown in Algorithm 3.1

Algorithm 3.1 PageRank iterative algorithm

Define $W :=$ weights for the websites
Initialize $W = \{1/N, \dots, 1/N\}$
while $|W - prev\_W| > t$ **do**
  $C :=$ N by N matrix storing weight contributions from web pages to others
  $C_{ij} = \mathbb{1}_{ij} d W_i / o_i$, where $\mathbb{1}_{ij}$ is 1 if web page i has a link to web page j, 0 otherwise. d is the damping factor, W_i is the weight of web page i, and o_i is the outdegree of web page i.
  $new\_W = \{(1-d)/N + \Sigma_i C_{i1}, \dots, (1-d)/N + \Sigma_i C_{iN}\}$
  $prev\_W = new\_W$
  $W = new\_W$
end while
return W

The iterative algorithm for computing PageRank in Algorithm 3.1 was implemented using PMapReduce. Specifically, the computation inside the while loop is compatible with PMapReduce. Figure 3 shows the implementation in PMapReduce.

The computation of C_{ij} is done using the Distributed.pmap of PMapReduce. Then, the reduce tiers compute $\Sigma_i C_{ij}$. While many reduce tiers could have been used for this, we implemented with just two reduce tiers for the sake of simplicity. The first tier computes the sum of a subset of the C_{ij} rows. For example, if it has m workers, each worker would compute one of $\Sigma_{1\le i\le N/m} C_{ij}, \dots, \Sigma_{(m-1)N/m\le i\le N} C_{ij}$. Then, the second tier adds up the m outputs from the first tier. With this implementation, it is easy to add additional reduce tiers and change the number of workers, which are useful when the input size becomes particularly large.

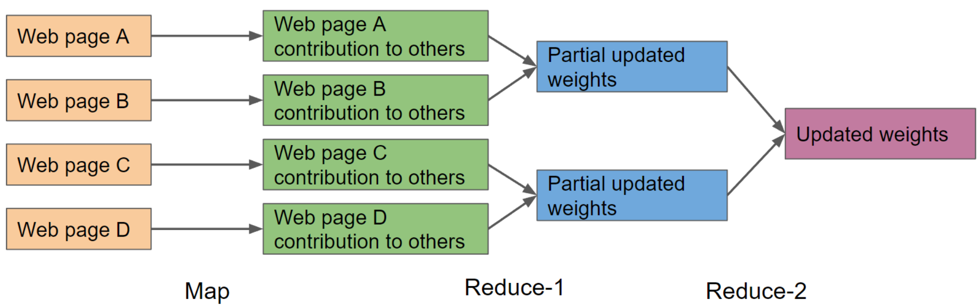

FIG. 3. *PageRank implementation using PMapReduce*

3.3. Sort with PMapReduce. While there are many sorting algorithms, only some of them can be parallelized without a significant change in the algorithm. We chose the sorting algorithm that works with the MapReduce framework, and it is shown in Algorithm 3.2. This algorithm can be implemented using PMapReduce with two reduce tiers. The Distributed.pmap component splits the input array into bins. Then, the first reduce tier spawns multiple workers, each sorting a subset of bins. For example, if there are 15 bins, $A_1, \dots, A_{15}$, and 3 workers, worker 1 sorts the bins $A_1, \dots, A_5$, worker 2 sorts $A_6, \dots, A_{10}$, and worker 3 sorts $A_{11}, \dots, A_{15}$. After that, the second reduce tier concatenates these sorted bins and produces the final output. Figure 4 shows a small example of the sort algorithm implemented with PMapReduce.

Algorithm 3.2 Sorting algorithm

Split the input array A into n bins: $\{A_1, \dots, A_n\}$ such that $\forall i, j \in [1, n]$ such that $i < j$, $\max A_i \le \min A_j$
Sort each bin, $A_1, \dots, A_n$
$A' = concat(A_1, \dots, A_n)$
return A'

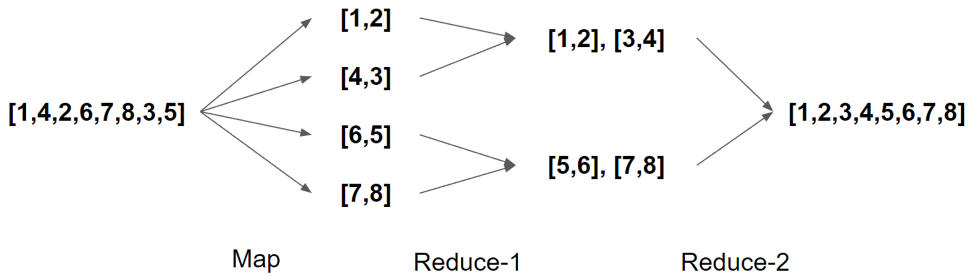

FIG. 4. *Sort implementation using PMapReduce*

3.4. TF-IDF with PMapReduce. TF-IDF stands for term frequency - inverse document frequency, and it is a statistical measure that evaluates how relevant a particular word in a document is for a corpus of documents. It is useful in many different fields, but especially in automated text analysis and as a method of numerically scoring words for natural language processing. It is the result from multiplying the term frequency term with the inverse document frequency term.

The term frequency measures how often a particular word appears in a particular document. Thus, even if the same word appears in the multiple documents, it will have its own term frequency value for each document. The term frequency can also be calculated in a variety of different ways as long as within the same document, the value gets larger with more occurrences. We chose the logarithmically scaled frequency method which can be calculated with:

$$TF(term, document) = log(1 + f_{term,document})$$

where $f_{term,document}$ is simply the raw number of times the *term* appears within the *document*. With this logarithmically scaled calculation for term frequency, the value increases with more frequent occurrences of the word within the document, but it also levels off at some point. This is to show the diminishing returns of importance from unique word that appears too many times within a document without being particularly useful in measuring relevance, such as certain proper nouns.

The inverse document frequency term measures the frequency of a word across a set of documents. In other words, it is a way to measure how rare or common a word is within a corpus of documents. The closer this term is to 0, the more common the word is to the document. This can also be calculated in many ways but the most common method way to calculate the term is as follows:

$$IDF(term, document) = log(1 + \frac{N}{|d \in D : t \in d|})$$

where N is the total number of documents within a corpus and $|d \in D : t \in d|$ is the number of documents within a corpus in which the term t appears. This term is helpful because even if a word appears many times within a single document, it is not very relevant within the corpus if it appears in many different documents. For example, words such as “the”, “a”, and “is” are likely to appear many times within a single document. However, since these terms also appear across many documents in a corpus, it is not very significant relative to a corpus.

Our implementation for calculating TF-IDF consists of several layers of mapping and reduction. There is first an initial mapping of the corpus to calculate the TF values. Once the TF values are obtained, they are then mapped again to represent a boolean flag that determines whether or not the word appears in the document. These initial boolean values are then reduced through two layers, across multiple workers, in order to get a final IDF dictionary for the corpus. Finally, the IDF dictionary as well as the previously calculated TF values are combined in order to get the final TF-IDF values for all the terms in the corpus. The algorithm is also summarized in Algorithm 3.3.

Algorithm 3.3 TF-IDF algorithm

Map input document corpus $\longrightarrow$ TF values
Map TF values $\longrightarrow$ boolean flags for term existance
Parallized reduce boolean flags $\longrightarrow$ IDF dicionaries
Final reduction to combine IDF dictionaries
$TF - IDF_values$ = combine IDF dictionary and TF values
return $TF - IDF_values$

4. Results. In order to show the efficacy of PMapReduce, we compared the PMapReduce implementations for the three aforementioned problems, PageRank, sort, and TF-IDF, with their single worker reduce counterparts. Since we expected that the results would heavily depend on the input data size, we experimented with varying input size, starting from a very small number until we saw an inflection point, where the run time ordering between various implementations changed. Since there could be run to run variance, for each datapoint, we ran the experiment 10-20 times and used the average. We also collected data on memory allocation, as using memory allocation is a major downside for PMapReduce that we anticipated, and it could be used to explain its performance behavior.

For PageRank, three implementations were compared for their run time and memory allocation. The first is an implementation using a single worker map and single worker reduce, to provide a baseline implementation. The second is the multiple worker map and a single worker reduce. Note that these two do not use PMapReduce, as they could be implemented with Julia’s default map, reduce, and Distributed.pmap. On the other hand, the last implementation had a multiple worker map and a multiple worker reduce implemented with PMapReduce. Figure 5 shows the resulting run time. Here, the input size represents number of web pages in the input. Each link from a web page to the other was put into the input randomly with a 0.3 chance. We see that at lower input size, all three implementations had a very similar run time, but with larger input sizes, the multi map, multi reduce implementation was the fastest, followed by multi map, single reduce and single map, single reduce, in that order. This is likely because the three implementations use similar amount of memory allocation, as shown in Figure 6, so the parallelization results in a performance benefit.

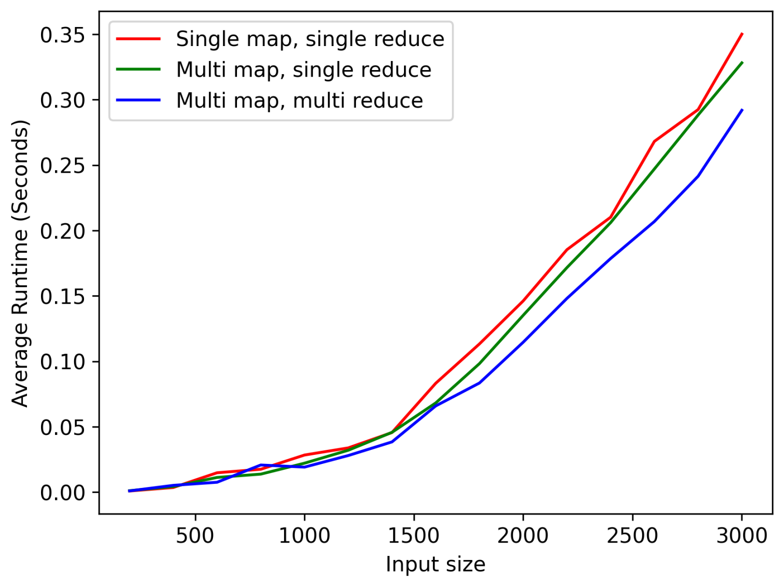

Fig. 5. *PageRank run time results*

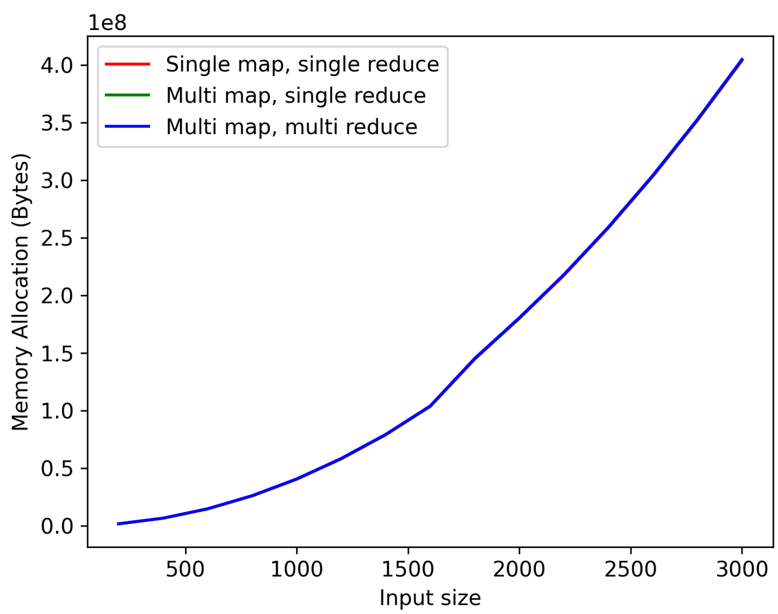

Fig. 6. *PageRank memory allocation results*

The next problem we experimented on was sorting. For this problem, the input size represents the number of randomly generated floats in the input array. We used them to compare three implementations. The first two were single worker map and single worker reduce; and multi worker map and single worker reduce, same as PageRank implementations. However, for the third implementation that uses PMapReduce, we opted to pair single worker map with the parallelized reduce. This is because, as shown in Figure 8, multi worker reduce uses too much memory allocation compared to its single worker counterpart. As a result, as shown in Figure 7, the multi map single reduce implementation is by far the slowest. So, we have a single map multi reduce implementation that we can compare to the single map single reduce implementation. According to Figure 7, the two use a very similar amount of memory allocation. So, due to the performance benefit with parallelization, single map multi reduce outperforms single map single reduce as expected. In addition to the run time and memory allocation analysis, we also experimented with changing the number of workers in the reduce tier one of the PMapReduce sort implementation. As shown in Figure 9, we experimented with varying input sizes from 10000 to 100000 and number of workers from 10 to 150. We observe several interesting trends in Figure 9. First, the number of workers can drastically change the average run time. We notice that the runs with low number of workers tend to be slow. However, having the highest number of workers (150) was never the fastest run. Also, the optimal number of workers remained fairly consistent at around 125 workers regardless of the input size. While this result is hard to generalize, since the data was collected only on a single implementation of a problem, it is an interesting empirical result showing the relation between the input size and the number of workers, and the optimal number of workers in general. Finding the optimal number of workers is a useful step for any implementation using PMapReduce.

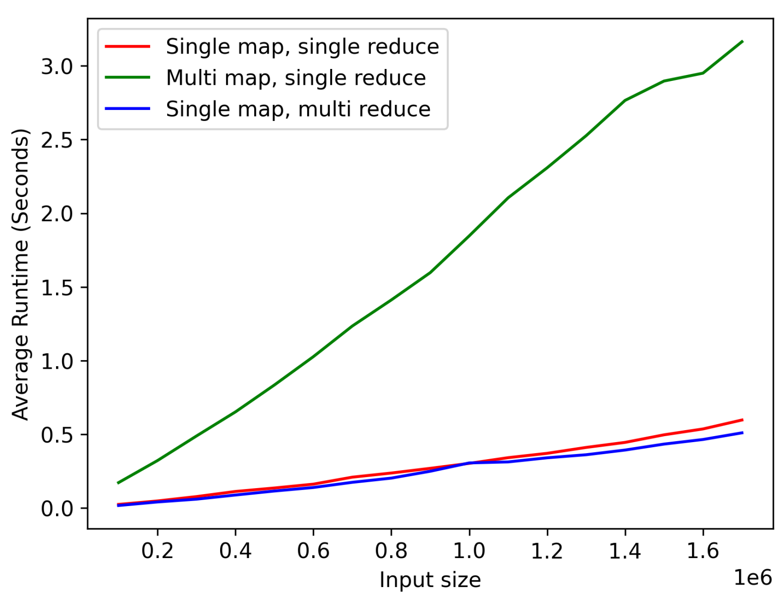

Fig. 7. *Sort run time results*

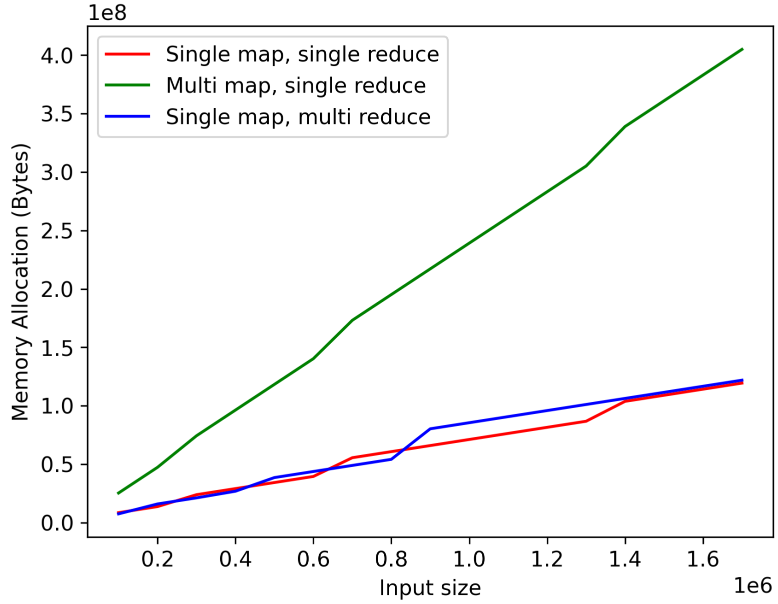

Fig. 8. *Sort memory allocation results*

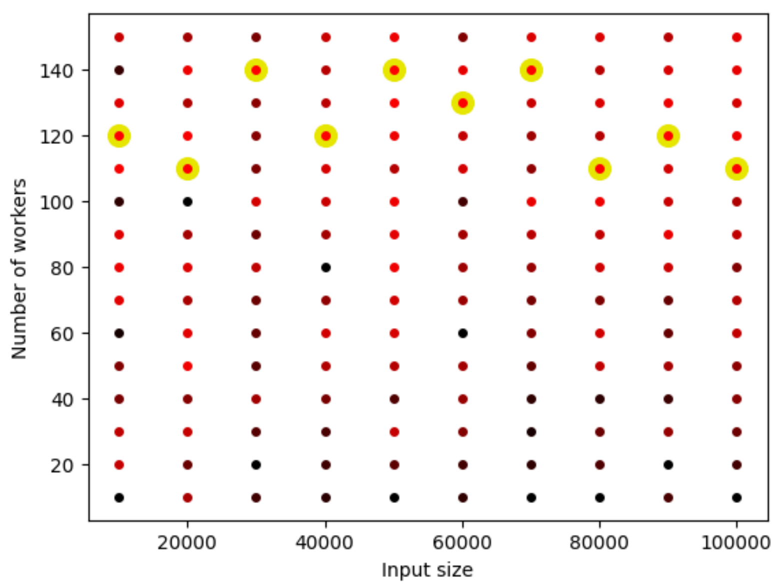

Fig. 9. *Sort optimal number of workers. Red represents faster speed, while black is for slower speed. The fastest run for each input size is highlighted with yellow.*

The last problem we examined was TF-IDF. Here, the input size represents the number of documents. Standard three implementations were used for this problem: single worker map and single worker reduce; multi worker map and single worker reduce; and multi worker map and multi worker reduce. Only the last implementation used PMapReduce. Also, it is important to note that the TF-IDF implementation had three map operations in total, and the multi map single reduce implementation used Distributed.pmap for all three, but the multi worker multi reduce implementation used Distributed.pmap for only two, and regular map for the other one. This is because that one map operation used too much memory allocation when parallelized with Distributed.pmap, to a point where it hindered the performance. This is shown in Figure 11, where the multi map single reduce uses substantially more memory allocation than the other two. As a result, as shown in Figure 10, it had the slowest run time. Comparing the single map single reduce and multi map multi reduce, the memory allocation of the two are close, although the latter uses slightly more. This is as expected, since parallelization and spawning workers involves more operations and memory usage. However, the multi map multi reduce ended up being faster than the single map single reduce in Figure 10, as the performance benefits from the parallelization likely outweighed the slight cost increase in memory allocation.

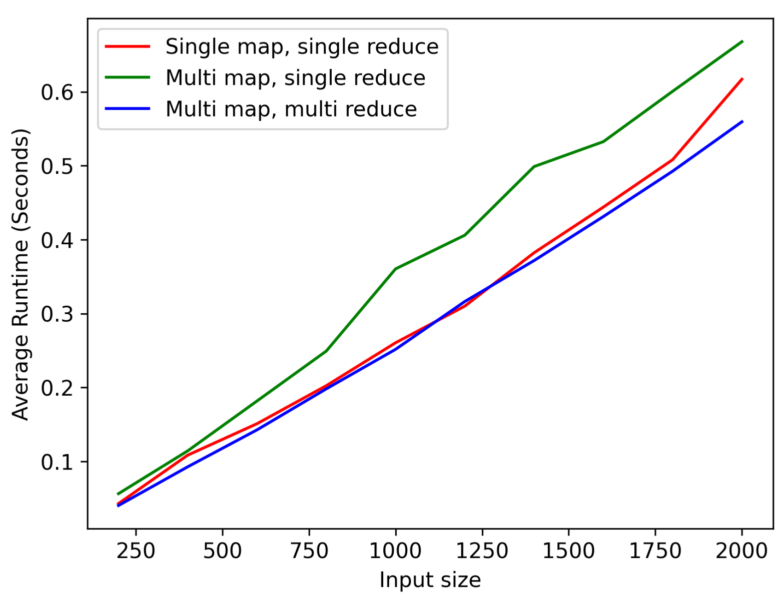

Fig. 10. *TF-IDF run time results*

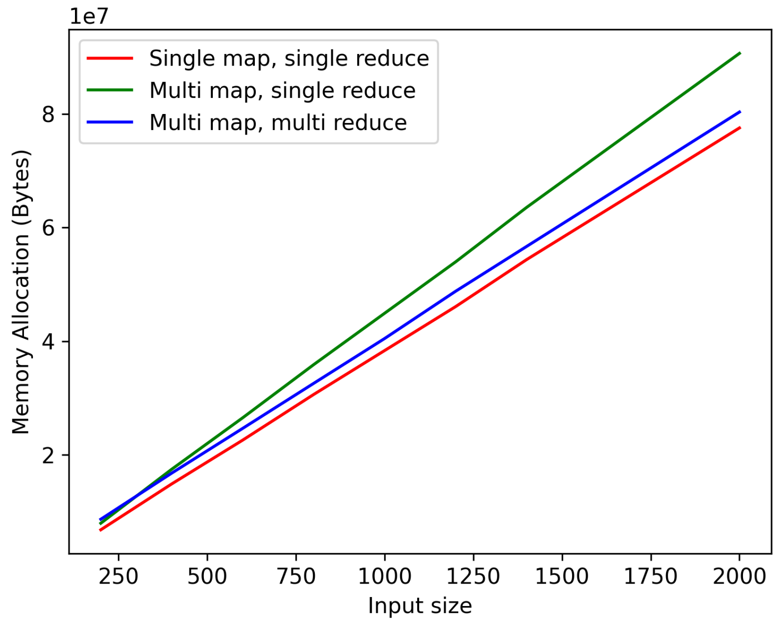

Fig. 11. *TF-IDF memory allocation results*

5. Conclusion. MapReduce is a common programming model that allows for the processing and generation of large data and consists of a mapping function and a reduce function. Although Julia already has a parallelized map function in the form of Distributed.pmap, in this paper we introduce PMapReduce which can use Distributed.pmap as well as our implementation of a parallelized reduce function. Our parallelized reduce function uses multiple tiers of reduce functions in order to allow multiple workers to work on reducing a set of outputs from the mapping step. Every tier except for the last tier of reduce functions can be done in parallel, with only the last tier being done sequentially in order to ensure that one correct output is generated.

Through our experiments, we discovered that in general the parallelized reduce method led to faster performance. However, there was also a cost from the overhead for implementing our PMapReduce function, which made this less clear. This additional cost came from various factors such as creating new workers, deleting workers, and setting up the correct input and output formats. These cost were observed through the increased memory allocation. When the cost of using the PMapReduce method was less than the benefit from parallelizing the reduce method, there were significant decreases in performance. This disparity was better seen with larger data inputs due to parallelization being more impactful than the less scalable cost of setting up the problem.

We also made an empirical observation that for sorting, regardless of the size of the input sequence, the optimal number of threads stayed fairly consistent. While this is not a generalizable result, since this data was collected on only one implementation for one problem, it was an interesting result. This is likely due to the balance between the cost of spawning and assigning tasks to more workers and the benefit from additional workers remains stable when the input size is sufficiently large and the hardware does not change.

REFERENCES

[1] J. Dean and S. Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, in OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, 2004, pp. 137–150.

[2] A. Ene, S. Im, and B. Moseley, *Fast clustering using MapReduce*, Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining, (2011), pp. 681–689, https://doi.org/10.1145/2020408.2020515, https://dl.acm.org/doi/10.1145/2020408.2020515 (accessed 2023-05-16). Conference Name: KDD '11: The 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining ISBN: 9781450308137 Place: San Diego California USA Publisher: ACM.

[3] N. Kavi, *MapReduce for Counting Word Frequencies with MPI and GPUs*, (2022), https://doi.org/10.48550/ARXIV.2206.05269, https://arxiv.org/abs/2206.05269 (accessed 2023-05-16). Publisher: arXiv Version Number: 1.

[4] P. Kourzanov, *Parallel evaluation of a DSP algorithm using julia*, Proceedings of the 3rd International Workshop on Software Engineering for Parallel Systems, (2016), pp. 20–24, https://doi.org/10.1145/3002125.3002126, https://dl.acm.org/doi/10.1145/3002125.3002126 (accessed 2023-05-16). Conference Name: SPLASH '16: Conference on Systems, Programming, Languages, and Applications: Software for Humanity ISBN: 9781450346412 Place: Amsterdam Netherlands Publisher: ACM.

[5] J. H. Lee, Y. Kim, Y. Ryu, W. Sodsong, H. Jeon, J. Park, B. Burgstaller, and B. Scholz, *Julia Cloud Matrix Machine: Dynamic Matrix Language Acceleration on Multicore Clusters in the Cloud*, Proceedings of the 14th International Workshop on Programming Models and Applications for Multicores and Manycores, (2023), pp. 1–10, https://doi.org/10.1145/3582514.3582518, https://dl.acm.org/doi/10.1145/3582514.3582518 (accessed 2023-05-16). Conference Name: PMAM'23: 14th International Workshop on Programming Models and Applications for Multicores and Manycores ISBN: 9798400701153 Place: Montreal QC Canada Publisher: ACM.

[6] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu, *Distributed data management using MapReduce*, ACM Computing Surveys, 46 (2014), pp. 1–42, https://doi.org/10.1145/2503009, https://dl.acm.org/doi/10.1145/2503009 (accessed 2023-05-16).

[7] L. Page, S. Brin, R. Motwani, and T. Winograd, *The PageRank Citation Ranking : Bringing Order to the Web*, Nov. 1999, https://www.semanticscholar.org/paper/The-PageRank-Citation-Ranking-%3A-Bringing-Order-to-Page-Brin/eb82d3035849cd23578096462ba419b53198a556 (accessed 2023-05-16).

[8] J. E. Ramos, *Using TF-IDF to Determine Word Relevance in Document Queries*, 2003, https://www.semanticscholar.org/paper/Using-TF-IDF-to-Determine-Word-Relevance-in-Queries-Ramos/b3bf6373ff41a115197cb5b30e57830c16130c2c (accessed 2023-05-16).

[9] A. Verma, X. Llorà, D. E. Goldberg, and R. H. Campbell, *Scaling Genetic Algorithms Using MapReduce*, 2009 Ninth International Conference on Intelligent Systems Design and Applications, (2009), pp. 13–18, https://doi.org/10.1109/ISDA.2009.181, http://ieeexplore.ieee.org/document/5362925/ (accessed 2023-05-16). Conference Name: 2009 Ninth International Conference on Intelligent Systems Design and Applications ISBN: 9781424447350 Place: Pisa, Italy Publisher: IEEE.