Performance Analysis of Divide-and-Conquer strategies for Large scale Simulations in R

Ranjini Subramanian
University of Louisville
Louisville, KY, USA
Email: r0subr05@louisville.edu

Hui Zhang
University of Louisville
Louisville, KY, USA
Email: h0zhan22@louisville.edu

***Abstract*— As the volume of data and technical complexity of large-scale analysis increases, many domain experts desire powerful computational and familiar analysis interface to fully participate in the analysis workflow by just focusing on individual datasets, leaving the large-scale computation to the system. Towards this goal, we investigate and benchmark a family of Divide-and-Conquer strategies that can help domain experts perform large-scale simulations by scaling up their analysis code written in R, the most popular data science and interactive analysis language. We implement the Divide-and-Conquer strategies that use R as the analysis (and computing) language, allowing advanced users to provide custom R scripts and variables to be fully embedded into the large-scale analysis workflow in R. The whole process will divide large-scale simulations tasks and conquer tasks with Slurm array jobs and R. Simulations and final aggregations are scheduled as array jobs in parallel means to accelerate the knowledge discovery process. The objective is to provide a new analytics workflow for performing similar large-scale analysis loops where expert users only need to focus on the Divide-and-Conquer tasks with the domain knowledge.**

Keywords—Divide-and-conquer, Slurm array jobs, Parallel processing, R programming language

I. INTRODUCTION

Simulation is a good means to evaluate various methodologies that can be used to study the accuracy and efficiency of the model. It allows the user to study the model's behavior under different conditions. In this paper, we consider a general framework that is useful in performing large-scale computational simulations, where each simulation generates its intermediate analysis outcome towards the final aggregated results to be analyzed. The question being asked in our study is how to divide large-scale simulation tasks for parallel execution and aggregate the intermediate simulation outcomes in the most efficient way in a high-performance computing environment.

We have designed and implemented the simulation methods in R programming language. The R statistical programming environment provides an ideal platform to conduct simulation studies. R incorporates features common in most programming languages such as loops, random number generators, conditional (if-then) logic, branching, and reading and writing of data, all of which facilitate the generation and analysis of data over many repetitions that is required for many simulation studies [1]. R is open source and can be run across a variety of operating systems.

R programming language is designed to make analysis and statistical investigation easier but it is not designed for high performance. Often times, we are required to conduct large-scale simulations studies using high volumes of data. It is easy to run small-scale simulation in R but it poses a challenge in terms of experimental setup and computational capability when it comes to processing and generating large datasets. In this paper, we discuss the procedures in designing and implementing Divide and Conquer strategies for performing large-scale simulations using R that can be used in a variety of simulation studies. The major contribution of this work are simple and powerful frameworks that enables parallelization of large-scale computation. We will illustrate possible performance gains from using parallel (divide-and-conquer) framework by comparing it to the base case.

Sections II and III address the background and motivation behind this work, section IV describes the basic structure. Section V explores the different programming models. Section VI has performance measurements of our implementation for different batch sizes.

II. BACKGROUND

In many academic domains, there are several opportunities for discovering new things due to the availability of high volumes of data. But it poses a challenge in terms of the computational power needed to analyze the data and extract information from it. This raises a question of if we should invest in new computing models or if the existing software and hardware tools can be combined to meet the computing needs. Although the shift in analytical models to meet the needs of Big Data analysis is inevitable, it is costly and could take a significant amount of time. To meet our immediate needs, it is possible to combine current software and hardware tools to scale up computations. In this paper, we focus on how current hardware technology can be coupled with R programming language to achieve massive parallelism.

R programming language is widely used in Big Data analysis due to its high extensibility and open source development. Several packages in R was developed to solve problems in various domains, such as clinical science [2], bioinformatics [3, 4], geoscience [5], social science [6], to name a few. Although R is a "high productivity" language, it lacks the control and structures to support highly efficient code. One approach is to use parallel packages in R. It is done by rewriting some basic functions or processing flow with the corresponding parallel version provided by the parallel packages. This requires extensive knowledge of R code and parallel mechanism supported by these packages. The second approach is to break large data sets into chunks and process each chunk in parallel. This approach makes use of the hardware capability instead of fully relying on R parallel packages. We use the second approach for our study [7].

To fully exploit the hardware capability, we use Slurm job array jobs on Stampede2. Job arrays offer a mechanism for submitting and managing collections of similar jobs quickly and easily. Job arrays with millions of tasks can be submitted in milliseconds. This is best suited for the Divide-and-Conquer strategy we use for our study.

III. MOTIVATION

Divide-and-Conquer is an algorithm design paradigm based on multi-branched recursion. A Divide-and-Conquer algorithm works by (recursively) breaking down a problem into many sub-problems of the same or related type, until these become (computationally) simple enough to be solved directly. Individual solutions to the sub-problems are then aggregated to give the final solution to the original problem [8]. The divide and conquer paradigm is often used to find the optimal solution of a problem. Its basic idea is to decompose a given problem into two or more similar, but simpler, subproblems, to solve them in turn, and to compose their solutions to solve the given problem. Problems of sufficient simplicity are solved directly. Divide and conquer is a powerful tool for solving conceptually difficult problems: all it requires is a way of breaking the problem into sub-problems, of solving the trivial cases and of combining sub-problems to the original problem. Divide and conquer algorithms are naturally adapted for execution in multi-processor machines, especially shared-memory systems where the communication of data between processors does not need to be planned in advance, because distinct sub-problems can be executed on different processors. The Divide-and-Conquer technique is the basis of efficient algorithms for many large-scale simulation problems. Yuhuan proposes using divide and conquer algorithm for simulation of matrix eigenvalue problem [9]. Harfield used divide and conquer approach for simulating quantum mechanical systems on quantum computers. They have obtained fast simulation algorithms using Hamiltonian structure [10].

Divide and conquer methodology is often used for large-scale simulations. This approach works in two phases – Divide phase and Conquer phase. In the Divide phase, an array of simulation jobs is run to produce a list of intermediate results. This is done by running the same computation to different values. In the Conquer phase, the intermediate results are combined or aggregated to produce final results.

While R is increasing in popularity over the last few years, many large-scale simulation studies are conducted in the R programming environment with parallel computation implemented. When implementing parallel R computation, the simulation program needs to start with setting up a cluster — a collection of "workers" that will be doing the job. This way of parallel R computing is designed aiming at executing long jobs in an enormous number of combine computing nodes offering computing and storage. Acquiring the "workers" becomes non-trivial for simulations that can be divided into thousands of short running jobs. The waiting and configuration time for the parallel resources can be much longer than the actual parallelized processing time, resulting in poor response time and inefficient performance with the traditional parallel R solution [11].

Our task in this paper is to show how one can fully exploit R and use Slurm array jobs to conquer large-scale simulations in R. We start with a simple R simulation that produces a number of intermediate CSV files and extend that for more repetitions performed by different "workers". The intermediate files are then aggregated to produce the final output files.

IV. BASIC STRUCTURE

In this section, we present three R scripts and their parameters to illustrate the Divide-and-Conquer phases in large-scale simulation.

All models have three main parameters:

- B - Batch size
- I - Number of intermediate files
- D - Dimension of intermediate file

The two scripts are:

- Simulation.R – The simulation script is coded to be executed as a Slurm task array job. The simulation R script takes two parameters, (*B,I*). *B* is the batch size and *I* is the number of intermediate output files. Simulation.R is executed *B* times and for each execution, the process is imitated I times. Each simulation generates *I* types of intermediate files. The total number of files produced by this step are *B*x*I*. The intermediate files are generated using random numbers.
- Aggregate.R – This script first checks to see if all B simulations have completed execution. It checks for the last intermediate file in each folder. The intermediate files generated by Simulation.R are then aggregated by the type of file. *I* number of files are generated by this phase – one for each type. These files are used for further analysis

The Divide-and-Conquer strategy performs well with small-to-medium sized outcome data. However, when a large number of simulations are involved, the size of the output data increases linearly. Processing large data sets could take too long or even fail since R keeps all objects in memory.

In the next section, we will present four different programming models to process and generate large data sets.

V. PROGRAMMING MODEL

For this study, we have implemented four divide-and-conquer strategies to process and generate large datasets. These strategies explore the effects of serial and parallel modes of processing.

A. Serial Framework (S_SA_S)

In this model, simulation and aggregation are executed sequentially. The simulation job performs multiple tasks sequentially. The simulation job is run *B* times and each run produces *I* number of intermediate files. The aggregation job then executes sequentially, each time merging a certain type of intermediate file. The aggregation task is executed *P* times, once for each type of file.

In the serial framework, an R script called Scheduler.R is used to execute Sim.R and Aggregate.R. The batch size, *B*, the number of intermediate files, *I*, are specified in the Scheduler.R file. The batch file executes the Scheduler.R file once. The scheduler then loops through Sim.R *B* times and each run of Sim.R creates a folder and produces *I* intermediate file in each folder. In this model, we use S_S to

refer to serial execution of simulation and A_S to refer to serial execution of aggregation.

This framework requires only one node to execute. It uses a single core on a single node. This model is the least efficient of all since it will take a long time to process and generate all the files. The execution might also fail because R keeps all the objects in memory.

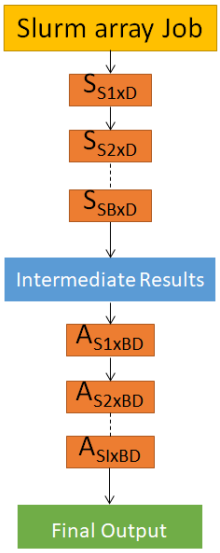

Fig. 1. Serial (S_SA_S) Framework

B. *Parallel - Serial Framework (S_PA_S)*

In this model, the simulation task is split into independent jobs that are executed by the Slurm array job in parallel. *N* nodes execute simulation job once. Each job produces *I* intermediate files. The batch size, *B*, is specified in the batch file. *B* is equivalent to *N*, the number of nodes executing simulation job in parallel.

The aggregation job then executes sequentially, each time merging a certain type of intermediate file. The aggregation task is executed *P* times, once for each type of file. In this model, we use S_P to refer to parallel execution of simulation and A_S to refer to serial execution of aggregation.

This framework requires *B* nodes for simulation and one node for aggregation. Two batch files are needed to execute the simulation and aggregation task separately. This framework exploits parallelism to some extent. This model performs better than the serial framework but the aggregation task could still take a long time to complete or it could result in the task failing to complete.

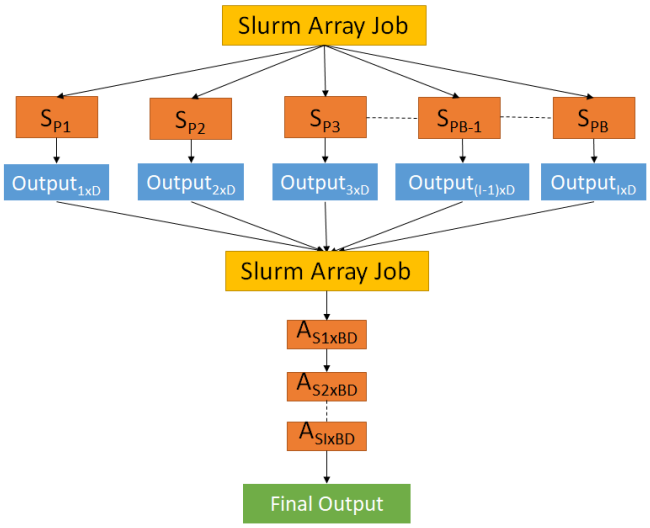

Fig. 2. Parallel - Serial (S_PA_S) Framework

C. *Parallel Framework*

In this framework, simulation and aggregation jobs are both executed in parallel. The simulation task is split into independent jobs that are executed by the Slurm array job in parallel. *B* nodes execute simulation job once. Each job produces *I* intermediate files.

The aggregation job is then split into independent jobs executed in parallel by another Slurm array job. *I* nodes execute the aggregation job once. Each job aggregates a certain type of intermediate file.

Two batch files are used to execute the simulation and aggregation tasks. In the batch file for simulation, batch size is specified to be equivalent to the number of nodes running Simulation.R in parallel. In the batch file for aggregation, we specify the number of intermediate files, *I*, which is also the number of nodes performing aggregation to produce final output.

This framework requires *B* nodes for simulation and *I* nodes for aggregation. In this model, we use S_P to refer to parallel execution of simulation and A_P to refer to parallel execution of aggregation

This framework exploits parallelism to a great degree. Since each job requests new compute resources from a separate process, there could be a delay if compute nodes are unavailable. This leads to another framework that combines the simulation and aggregation job and executes them using one Slurm array job.

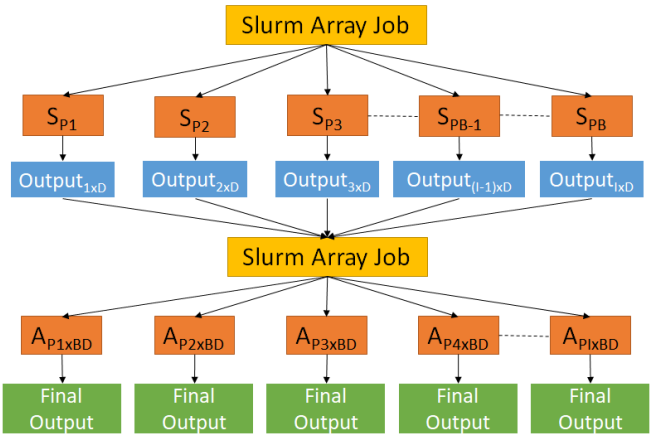

Fig. 3. Parallel (S_PA_P) Framework

D. *Improved Parallel Framework*

In this improved parallel framework, simulation and aggregation is integrated into one script. Both tasks are executed in parallel by a single Slurm array job submission by running on the same set of allocated compute nodes. This model exploits parallelism to the fullest. It accelerates the aggregation task by using the already available compute nodes instead of requesting new compute nodes from a separate process.

Since the simulation and aggregation logic are integrated into one script, they can communicate and synchronize with each other. This R script has two functions:

- The simulation (map) tasks is done in parallel by *B* nodes, where *B* is the batch size
- The aggregation task is then performed by the first *I* (number of intermediate files) nodes out of the *B* compute nodes already reserved for this task. The aggregation task begins execution when the number of simulation tasks remaining is less than or equal to *I*

This framework requires a maximum of *B* (batch size) nodes for the entire execution. Since both simulation and aggregation scripts are integrated into one script, we will refer to this as *SA*. This new structure utilizes the available resources to the maximum degree. The large number of compute nodes running simulations will continue (if necessary) to perform as Aggregators. Since the simulation and aggregation logic are now implemented within one R script, it is easy for the two tasks to communicate and coordinate with each other to complete the whole task using the Divide-and-Conquer paradigm. In this way we also minimize Slurm scheduling burden on HPC side [11].

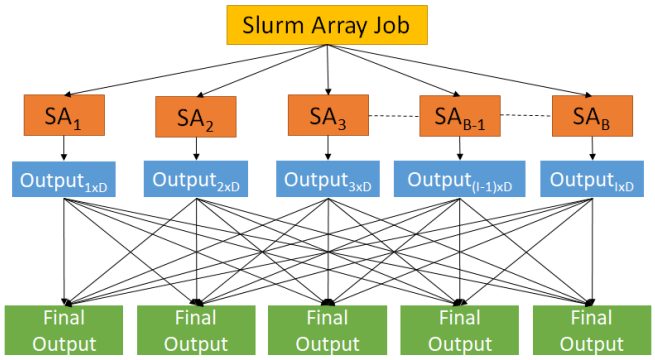

Fig. 4. Improved Parallel (SA) Framework

VI. CASE STUDY

To study the effects of various models, we have used a supercomputer, Stampede2. Stampede2, generously funded by the National Science Foundation (NSF) through award ACI-1134872, is the flagship supercomputer at the Texas Advanced Computing Center (TACC), University of Texas at Austin. It entered full production in the Fall 2017 as an 18-petaflop national resource that builds on the successes of the original Stampede system it replaces. Stampede2 hosts 4,200 KNL compute nodes. Each of Stampede2's Knights Landing (KNL) nodes includes 96GB of traditional DDR4 Random Access Memory (RAM). They also feature an additional 16GB of high bandwidth, on-package memory known as Multi-Channel Dynamic Random Access Memory (MCDRAM) that is up to four times faster than DDR4.

We connect to Stampede2 through a login node and run the jobs on compute nodes. The compute nodes are accessed by submitting a batch job using the *sbatch* command.

For this case study we have used three main parameters:

- *B* - Batch size
- *I* - Number of intermediate files
- *D* - Dimension of intermediate file

B is varied from 100 to 1000 in increments of 100 and 2000. *I* is set to 10 for all batch sizes and *D* is set to 100 x 100. In Simulation.R, we have included a 45 second wait to mimic the time needed to perform application-specific tasks.

For each programming model, we have recorded the time it took the model to complete simulation and aggregation tasks for different batch sizes.

a) Serial Framework - In this model, all tasks are executed sequentially. All simulation tasks must complete before aggregation can begin. Simulation.R is executed *B* times and Aggregation.R is executed *I* times. Execution time for each batch size is provided in Table I. The execution takes several hours to complete, and the time increases linearly with batch size. This method is highly inefficient since it only uses one core per each node thereby wasting significant processing power. As batch sizes increase, there is a possibility of incomplete or failed tasks due to memory issues and unavailability of compute nodes.

TABLE I
EXECUTION TIME FOR SERIAL FRAMEWORK

Batch Size	Time (*in hours*)
100	1.26
200	2.52
300	3.78
400	5.05
500	6.31
600	7.57
700	8.84
800	Failed
900	Failed
1000	Failed
2000	Failed

b) Parallel - Serial Framework - In this model, simulation is executed in parallel and aggregation tasks are executed sequentially. All simulation tasks must complete before aggregation can begin. Simulation.R is executed *B* times in parallel which is equivalent to running it just once and Aggregation.R is executed *I* times sequentially. Two separate processes are used to request the compute nodes needed for simulation and aggregation. This could cause a delay if the required amount of compute nodes is not available. Execution time for each batch size is provided in Table II. The execution takes few seconds to several minutes, greatly reducing the execution time compared to serial execution. The bulk of the time is due to the serial execution of aggregation task. This method exploits parallelism to some degree. This model may be useful for applications where bulk of the processing is done in the simulation phase. For aggregation-heavy tasks, this model will be inefficient.

TABLE II
EXECUTION TIME FOR PARALLEL - SERIAL FRAMEWORK

Batch Size	Time (*in seconds*)
100	73
200	104
300	130
400	162
500	185
600	223
700	252
800	271
900	304
1000	336
2000	725

c) Parallel Framework - In this model, simulation and aggregation tasks are executed in parallel but aggregation tasks cannot begin before the completion of all simulation tasks. Simulation.R is executed *B* times in parallel which is equivalent to running it just once and Aggregation.R is executed *I* times in parallel. Two separate processes are used to request the compute nodes needed for simulation and aggregation. Execution time for each batch size is provided in Table III. The execution takes a few seconds to complete, further reducing the execution time compared to the previous models. Although this method exploits parallelism to a great degree, there could be a delay if the compute nodes are unavailable. Also, this method does not reuse compute nodes that have finished execution thereby wasting valuable resources. Table IV shows the execution time including wait time to acquire compute nodes for aggregation.

TABLE III
EXECUTION TIME FOR PARALLEL FRAMEWORK

Batch Size	Time (*in seconds*)
100	53
200	56
300	60
400	63
500	65
600	69
700	71
800	74
900	77
1000	80
2000	111

TABLE IV
EXECUTION TIME FOR PARALLEL FRAMEWORK (Incl. Wait time)

Batch Size	Time (*in seconds*)
100	100
200	104
300	188
400	106
500	454
600	238
700	114
800	325
900	123
1000	174
2000	916

d) Improved Parallel Framework - In this model, simulation and aggregation tasks are executed in parallel but unlike previous models, aggregation tasks begin execution before the completion of all simulation tasks. Simulation.R is executed *B* times in parallel which is equivalent to running it just once and Aggregation.R is executed *I* times in parallel. The aggregation task begins execution once the number of simulation tasks remaining is less than or equal to *I*. Only one Slurm array job is used to request the compute nodes needed for simulation and aggregation, thereby reusing compute nodes and avoiding the delay caused by unavailability of new compute nodes for aggregation. Execution time for each batch size is provided in Table V. The execution takes a few seconds to complete and it is comparable to parallel framework if we don't take into the account the delay caused for requesting new set of compute nodes for aggregation. This method exploits parallelism to the maximum. It also saves time and processing power by reusing compute nodes.

TABLE V
EXECUTION TIME FOR IMPROVED PARALLEL FRAMEWORK

Batch Size	Time (*in seconds*)
100	53
200	58
300	62
400	66
500	69

600	74
700	78
800	82
900	87
1000	91
2000	132

A. *Comparison of different frameworks*

Serial execution takes considerably longer to complete execution compared to the other three models as seen in Fig 5. As seen in the figure, time increases linearly with batch size. The execution failed for batch size >= 800 due to memory issues. Hence it is evident that this method is not suitable for processing large CSV files. Hence, we will focus our discussion on the benefits and shortcomings of the other three models in the rest of the paper.

Fig. 5. Comparison of All Frameworks

B. *Parallel – serial framework vs Parallel framework*

As shown in Fig 6, there is a 19% decrease in execution time between the two models. This could mean a significant difference if either the batch size, dimension of file or application-specific task processing time increases. When the batch size is increased from 1000 to 2000, the time for parallel-serial framework more than doubles. Whereas, in the parallel framework there is only a 38% increase in execution time. This shows that parallel framework performs significantly better than parallel-serial framework.

Fig. 6. Parallel-Serial Vs Parallel Frameworks

C. *Parallel framework vs Improved Parallel framework*

The time recorded for parallel framework in Fig 7, does not include the extra time needed to acquire the new set of compute nodes for aggregation. The improved parallel framework takes a few extra seconds to complete compared to the parallel framework. This is due to the way R language handles memory usage. R programming language was designed to make analysis and statistical investigations easier, but it is not designed for high performance. The increasing data in memory and IO operations cause the execution to take a few extra seconds to complete. For batch size of 2000, there is an 18% increase in time. Even though it takes extra time, it is negligible compared to the amount of time it could take for new set of nodes to be available to perform aggregation.

In order to compare these two frameworks, we take into account the wait time for compute nodes to demonstrate the advantage of reusing already allocated compute nodes.

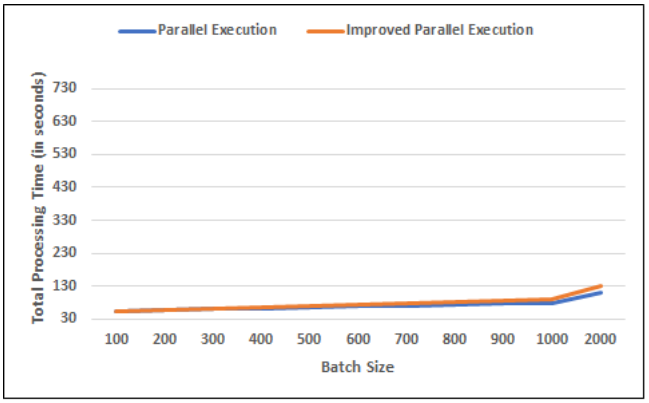

Fig. 7. Parallel Vs Improved Parallel Frameworks

D. *Parallel framework vs Improved Parallel framework (with wait time)*

The Fig 8 shows the execution time for parallel and improved parallel framework with the wait time for allocating new compute nodes for aggregation. Allocation of new compute nodes could take anywhere from a few seconds to few hours depending on the availability of compute nodes. When wait time is taken into account, parallel framework is significantly slow compared to improved parallel framework. In the parallel framework, there is a 500% increase in execution time for batch size of 2000.

Even though, improved parallel framework takes a few extra seconds to finish execution due to memory management and IO operations, it far outweighs the disadvantage of allocating new compute nodes. Hence, reusing existing nodes saves computational resources and time.

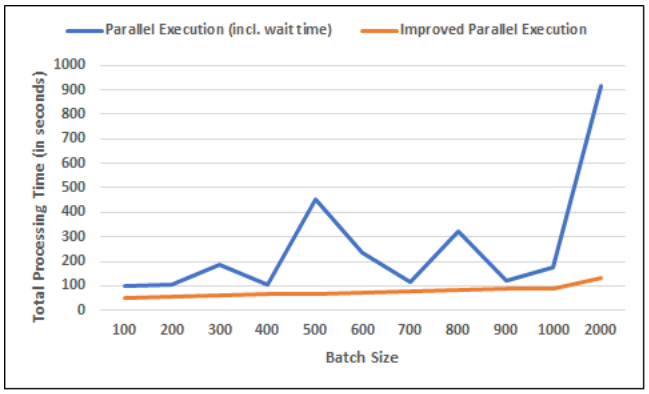

Fig. 8. Parallel Vs Improved Parallel Frameworks (*Incl. Wait Time*)

VII. ACKNOWLEDGMENT

This work was supported in part by National Science Foundation grant #1726532. Experiments were conducted using Stampede2 computing cluster at Texas Advanced Computing Center, via awarded SU CCR180029.

VIII. CONCLUSION

In this paper, we have discussed four different divide-and-conquer frameworks to process and generate large CSV files. We have learned several things from this work. First, our model makes it easy to parallelize computations. Second, compute nodes are expensive and take significant time for allocation. Third, our framework shows that the increase in execution time due to memory management can be offset by reusing compute nodes. In the improved parallel framework, our study shows the tradeoff between reusing compute nodes and memory management. We have provided a new analytics workflow for performing similar large-scale analysis loops where expert users only need to focus on the Divide-and-Conquer tasks with the domain knowledge. R programming language is extensively used for data analysis and this study provides a generalized framework that can be tailored to specific applications with little effort.

REFERENCES

[1] Kevin A Hallgren. “Conducting simulation studies in the r programming environment. Tutorials in quantitative methods for psychology”, 9(2):43, 2013.

[2] S. Pyne, X. Hu, K. Wang, E. Rossin, T.-I. Lin, L. M. Maier, C. Baecher-Allan, G. J. McLachlan, P. Tamayo, and D. A. Hafler, "Automated high-dimensional flow cytometric data analysis," Proceedings of the National Academy of Sciences, vol. 106, pp. 8519-8524, 2009

[3] C. Gondro, L. R. Porto-Neto, and S. H. Lee, "R for GenomeWide Association Studies," in Genome-Wide Association Studies and Genomic Prediction, ed: Springer, 2013, pp. 1-17.

[4] R. C. Gentleman, V. J. Carey, D. M. Bates, B. Bolstad, M. Dettling, S. Dudoit, B. Ellis, L. Gautier, Y. Ge, and J. Gentry, "Bioconductor: open software development for computational biology and bioinformatics," Genome biology, vol. 5, p. R80, 2004.

[5] E. Grunsky, "R: a data analysis and statistical programming environment–an emerging tool for the geosciences," Computers & Geosciences, vol. 28, pp. 1219-1222, 2002.

[6] J. FoX. (Aug. 18). “CRAN Task View: Statistics for the Social Sciences”.

[7] Yaakoub El-Khamra, Niall Gaffney, David Walling, Eric Wernert, Weijia Xu, Hui Zhang , “Performance Evaluation of R with Intel Xeon Phi Coprocessor”, IEEE International Conference on Big Data, 2013.

[8] Eric A Posner, Kathryn E Spier, and Adrian Vermeule. “Divide and conquer”. 2009.

[9] Yuhuan Cui, Jingguo Qu, Weili Chen and Aimin Yang, "Divide and conquer algorithm for computer simulation and application in the matrix eigenvalue problem," 2009 International Conference on Test and Measurement, Hong Kong, 2009, pp. 319-322.doi: 10.1109/ICTM.2009.5412930.

[10] Stuart Hadfield and Anargyros Papageorgiou, "Divide and conquer approach to quantum Hamiltonian simulation", 2018 New Journal of Physics.

[11] Hui Zhang, Yiwen Zhong, Juan Lin, “Divide-and-Conquer Strategies for Large-scale Simulations in R”, 2017 IEEE International Conference on Big Data (BIGDATA).