Parallel Framework for Data-Intensive Computing with XSEDE

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

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

ABSTRACT

With the increase in data-driven analytics, the demand for high performing computing resources has risen. There are many high-performance computing centers providing cyberinfrastructure (CI) for academic research. However, there exists access barriers in bringing these resources to a broad range of users. Users who are new to data analytics field are not yet equipped to take advantage of the tools offered by CI. In this paper, we propose a framework to lower the access barriers that exist in bringing the high-performance computing resources to users that do not have the training to utilize the capability of CI. The framework uses divide-and-conquer (DC) paradigm for data-intensive computing tasks. It consists of three major components - user interface (UI), parallel scripts generator (PSG) and underlying cyberinfrastructure (CI). The goal of the framework is to provide a user-friendly method for parallelizing data-intensive computing tasks with minimal user intervention. Some of the key design goals are usability, scalability and reproducibility. The users can focus on their problem and leave the parallelization details to the framework. This paper will outline the rationale behind this framework, its detailed implementation and demonstrate its usage with practical use cases.

CCS CONCEPTS

• **Computing methodologies → Massively parallel algorithms**.

KEYWORDS

Cyberinfrastructure (CI), R , Divide-and-conquer methodologies

1 INTRODUCTION

A major driving force behind the increasing popularity of Data Science is the increasing need for data-driven analytics fueled by massive amounts of complex data produced by businesses, scientific applications, government agencies and social applications. However, one unique challenge in Data Science comes from the tight coupling between practical solutions and their computing resource requirements. As the volume of data grows bigger, the solution often becomes viable only with advanced CI. One major benefit of CI is parallel computing which is known to be effective in speeding up data-intensive computing tasks. Effective utilization of CI is vital in maximizing the benefits offered by them. This is achieved by combining the software and hardware tools to achieve data-intensive computing. Data-analytics process ingests time and resources. To speed up data-intensive computing tasks, we can break large dataset into smaller chunks, process them in parallel and finally combine the processed results using CI. This approach, also referred to as divide-and-conquer approach, fully utilizes the capability of high-performance computing resources to accelerate the knowledge discovery process.

DC methodology is best suited for data-intensive computing. This methodology works by recursively breaking down a problem into subproblems of similar type that can then be solved in parallel independently. The solutions to the subproblems are then combined to obtain a solution to the original program. DC methodology is a powerful tool in facilitating data-intensive analysis and is used to solve many data intensive problems like matrix eigenvalue problem [1], simulating quantum mechanical systems on quantum computers using Hamiltonian structure [2]. Exploiting parallelism offered by the DC paradigm, requires certain expertise in parallel programming and knowledge of parallel programming tools. In addition to that, users must know how to execute parallel implementations in CI. Users who have only recently started using large-scale computation as a research tool are not yet trained to take advantage of the data-intensive computing environment offered by CI. To overcome these issues, we propose a simple, user-friendly framework that enables parallel implementation of DC paradigm to perform data-intensive computing tasks using high performance computing resources. The backend of our framework can deploy code automatically to scale up the user script by running the analytics script over much larger datasets with XSEDE high performance computing resources thereby allowing users to focus on transforming theoretical models and methodologies to practical solutions.

2 BACKGROUND

Many legacy applications are inherently sequential and do not use parallel programming techniques. In addition to that, novice users do not have the skills needed to write programs that take advantage of parallelization. In sequential implementations, the programs work as a single unit of code. Data is processed one at a time using iterative style of programming. The algorithm for sequential program is as shown in Figure 1.

In this implementation, for-loops are used to iterate through each dataset. The first for-loop is used for user-defined computation and the second for-loop is used for aggregating intermediate results to produce the final output. For-loops causes significant overhead and is slow in processing. In order to convert this implementation to make use of parallel processing speeds offered by CI, significant code changes are required which can only be done by users who have advanced knowledge of parallel programming and its tools. This drawback has motivated us to adopt DC paradigm with is naturally suited for parallel processing.

2.1 Divide-and-Conquer (DC) paradigm

DC paradigm is extensively used in data science due to its versatility and computational benefits. It offers best performance in

,

```
Algorithm: Sequential_Implementation
Input:
D: Input data
ds:dataset
D:= Σds

Steps:
FOR each ds in D DO
   Read ds
   Perform User-defined computation on ds
   Write IntermediateResult to IntermediateFolder
END FOR

FOR each IntermediateResult in IntermediateFolder DO
    Result <- Aggregate
END FOR

Write Result to OutputFolder
```

Figure 1: Sequential Implementation

```
Algorithm: DC Paradigm – Divide
Input:
ds:dataset
f: filename of ds
i: Intermediate folder path

Divide (f,i)
   Read f
   Perform User-defined computation on f
   Write IntermediateResult to i
```

```
Algorithm: DC Paradigm – Conquer
Input:
i: Intermediate folder path
o: Output folder path

Conquer (i,o)
FOR each IntermediateResult in i DO
    Result <- Aggregate
END FOR
Write Result to o
```

Figure 2: DC Paradigm Implementation

multi-processor, shared-memory systems where the communication needed between processors is minimal because the distinct subproblems can be executed independently in different processors. DC algorithms work in two steps:

- **Divide** - In this step, the dataset is broken down into smaller chunks and each chunk is processed independently to produce intermediate results. This step involves the bulk of the processing with complex business logic.
- **Conquer** - This step involves light-weight processing like aggregation. In this step, the intermediate results from the previous step are aggregated to produce the final results.

DC algorithm can be implemented using sequential as well as parallel approaches. Despite the obvious tendency for parallelism, parallel implementation of DC algorithm requires programming expertise in order to achieve desired level of performance offered by CI. Sequential implementation, although easy to implement by a novice user, is infeasible due to poor processing time. However, we propose a sequential algorithm that can be automatically parallelized by our framework. In the sequential approach, both phases can be implemented as functions or can be implemented as two separate programs. Implementing them as two separate programs allows for script level parallelism depending on application-specific requirements and it does not require fixed computation steps. The proposed algorithm for divide and conquer phases is as shown in Figure 2. The user is responsible for both scripts. The divide phase takes two input parameters - filename which is an optional parameter and intermediate folder path. User-defined business logic is implemented in this script. The result of the processing is stored in an intermediate folder. The conquer script takes two input parameters - path to intermediate folder and path to output folder. This phase loops through the intermediate results, aggregates them and writes the final output to the output folder.

The volume of data and processing speeds have increased exponentially in the past few years, but the former has risen at a much higher rate than the later. The rapid growth in data volumes and the need for higher processing speeds for data intensive tasks has led to the rise in parallel processing techniques. Processing data-intensive computing tasks require significant computing power which can only be achieved by using supercomputers. The type of parallel processing model depends on the task at hand. For data analytics, data parallelism approach is extensively used. It involves partitioning large dataset among multiple nodes and processing them in parallel before combining the partial results [4]. Several parallel and distributed programming models have emerged to tackle data-intensive problems. MapReduce is a programming model that has become a standard for large scale data-intensive applications [14]. It provides a powerful interface for large-scale computations enabled by automatic parallelization and distribution. It works by splitting the data and processing it in parallel while hiding the details of load balancing, synchronization and fault-tolerance. We have adopted some features of MapReduce paradigm in our framework such as automatic parallelization of large-scale computations, data locality and batch processing model.

Although MapReduce paradigm offers great benefits, the implementation is at the functional level which requires advanced programming skills and good understanding of the underlying system architecture. Computation steps are fixed, and it must follow the map-shuffle-sort-reduce sequence. Not every computation can be expressed efficiently using this programming model.

In [5], the authors have explored the advantages of using divide-and-conquer approach for parallel computing and studied the performance gains for data-intensive problems like sorting and matrix multiplication. Another important application of DC strategy has been in the medical imaging field [8]. Image enhancement has been used in several applications such as biomedicine, video surveillance, remote sensing to name a few.

The performance of different types of DC implementations are well studied for large scale simulations and it has been outlined in [6]. The purpose of the study was to evaluate scalability and

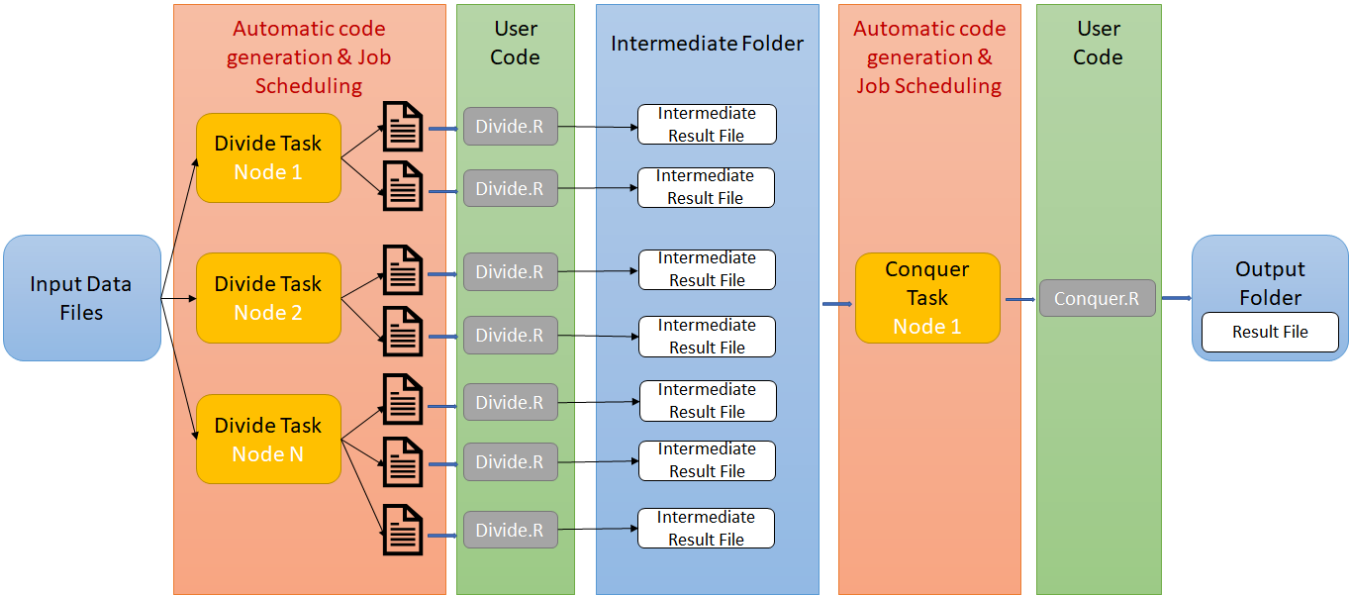


Figure 3: Process Flow

performance of parallel and serial implementations of DC tasks using high performance computing resources. The variables used for this study are batch size (***B***), number of intermediate files (***I***) and the dimensions of the file (***D***). The number of nodes used for each execution was set to ***B*** to enable parallel execution and set to 1 for serial implementation. For each implementation, ***B*** was varied from 100 to 2000, ***I*** was set to 10 and ***D*** was set to 100x100. The implementation also included a 45 second wait to mimic the time needed to perform application-specific tasks. The study found that using parallel model as opposed to serial model reduced the execution time by over 98%. The time decreased further as the size of the dataset increased. It was also found that as the size of the data increased, the serial mode of execution failed due to memory issues.

Although parallelizing both divide and conquer phases would increase the performance further, we chose to parallelize only the divide task in our framework. This design decision was made based on three important factors:

- Parallelizing serial implementation of user code is straightforward for divide task and requires no programming effort from the user. Parallelizing conquer task would require users to make significant changes to their code to avoid synchronization issues.
- The conquer phase is not as data-intensive as the divide phase
- It eliminates the need for the subproblems to be associative.

3 BASIC DESIGN

The framework enables DC paradigm for efficient processing of data-intensive computing tasks and provides a user-friendly access to advanced CI while requiring minimal programming effort and user intervention. Users are required to provide divide and conquer scripts.

3.1 Process Flow

The framework is designed for users who do not necessarily have knowledge of parallel computing. In our design, we use a programming paradigm that hides the underlying parallel computing details and allows the users to focus on sequential processing leaving the parallelization details to the framework. It automatically generates code and schedules batch processing jobs as depicted in Figure 3. The divide task automatically divides the input data among different computing nodes. Each file is then processed individually by the user-defined Divide program. The results of this step are written to an intermediate folder. The conquer task passes the files from the intermediate folder to the user defined Conquer program which is responsible for aggregating the intermediate results and producing the final output.

3.2 Cyberinfrastructure (CI) model

3.2.1 Job Scheduler. Job schedulers are a key component of scalable computing infrastructures. They are responsible for all of the work executed on the computing infrastructure and have a direct impact on the effectiveness of the system. Simple Linux Utility for Resource Management (**SLURM**) is the most popular scheduler used in many of the world's supercomputers. **SLURM** is an open source system used on large and small Linux clusters. It performs four important functions:

- Resource management
- Job lifecycle management
- Job scheduling
- Job execution

Users are required to specify the number of CPUs needed at the time of job submission. Jobs are submitted in batch to **SLURM** queue using a job script where the scheduler decides how to prioritize and allocate resources to jobs for execution.

3.2.2 Job Script. A job script is an executable file that is submitted to the scheduler to run on a collection of nodes. It contains a list of **SLURM** directives (or commands) that tells the scheduler what to do. ***SBATCH*** command is used to submit a batch script to **SLURM**. Similar jobs can be submitted using a single script and the jobs are run independently on different nodes. Each job is capable of obtaining the necessary compute nodes from the cluster. Job flags are used with ***SBATCH*** command and the syntax for the directive in a script is ***#SBATCH <flag>***. The flags we have used in our framework are N, –ntasks-per-node, –job-name, –output, –error, –partition and –time. The description and type of arguments accepted by each of the flags can be found at https://ubccr.freshdesk.com/support/solutions/articles/5000688140-submitting-a-slurm-job-script. Figure 4 shows a sample job script. ***ibrun*** is a TACC-specific MPI launcher to launch an MPI application.

```
#!/bin/bash
#SBATCH --job-name=Parallel DC #job_name
#SBATCH --output=slurmdivide.out #output_file
#SBATCH --error=slurmdivide.err #error_file
#SBATCH --partition=hadoop #queue_name
#SBATCH  -N = 10 #total_nodes
#SBATCH --ntasks-per-node=1 #tasks_per_node
#SBATCH --time=01:00:00 #hh:mm:ss

ibrun myjob.sh divide_script.R /path/to/intermediate/folder
```

Figure 4: SLURM Job Script

3.2.3 Access to CI. Although CI can speed up computations considerably, the skills needed to use them poses a major challenge. With the advent of data driven analytics, many tools have been developed to provide access to high performance computing resources and CI. Accessing these resources requires the users to know its application, languages and libraries. However, not all users are equipped with the knowledge to take advantage of these resources. Many CI providers have models to enable access to the underlying computing resources. Some of these models include batch job submission using secure shell connection [9], remote software session [10], web portal and gateway [11] to name a few. All these models require the users to know how to submit batch processing jobs and uploading and downloading data to and from CI. In order to overcome this issue, we present a simple user-friendly web user interface that enables non-expert users to access underlying CI.

3.3 Advantages

Some of the advantages of using this framework are:

- Flexibility - Our framework can handle different types of problems because it does not require the input to be in a particular format. The user can implement any business logic to suit their needs and can tailor the intermediate output and final output to be in the format that is specific to the problem at hand.
- Granularity - Our framework is designed to treat each file as one entity. This reduces communication and synchronization overhead and enables processing of heterogenous data.
- File size - Since the processing takes place at the file-level, the file size does not cause inefficiency. The framework can handle files of any size.
- Ease of use - Users can debug their sequential implementations of divide and conquer scripts locally and upload the code and input data using a web user interface. It provides better interactivity.
- Languages - The framework is designed to support R and python programming languages, two of the most widely used languages in Data Science. For the purpose of demonstration, we have used only R programming language.

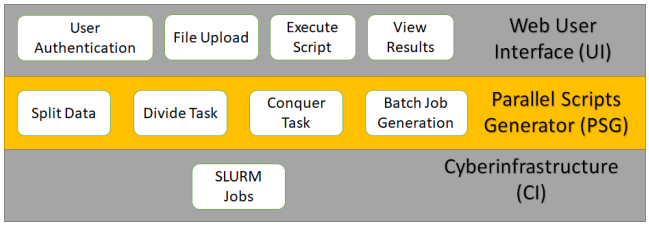


Figure 5: Framework Layers

4 IMPLEMENTATION DETAILS

The framework takes user defined scripts with custom business logic and automatically generates code for parallelization. We have used files to move data between the divide and conquer phases. The framework is developed using R programming language. The proposed framework consists to 3 layers - User interface (UI), Parallel scripts generator (PSG) and cyberinfrastructure (CI) as depicted in Figure 5.

4.1 User Interface (UI)

The framework by itself offers a command line interface to PSG. To support intuitive access and interactive analysis capability, we have integrated PSG into a web UI developed by Weija Xu et al. at Texas Advance Computing Center, The University of Texas [17]. The web UI is designed to enable the use of pre-defined tasks using a configuration file (workflow) which is reproducible and reusable. We have designed a workflow which enables the use of our framework. Users must login to the web application to use the workflow. The workflow consists for a set of predefined tasks - File upload, Execute script and View results.

4.1.1 File Upload. The first task is the file upload task, using which the user can upload the divide and conquer scripts and the input data as shown in Figures 6. The input files must be uploaded to the *Input* folder and the user scripts must be uploaded to the *User_Scripts* folder.

4.1.2 Execute Script. This task takes four input parameters from the user as shown in Figure 7:

(1) Application name
(2) Divide script name
(3) Conquer script name
(4) Number of nodes (N) needed for parallelization.

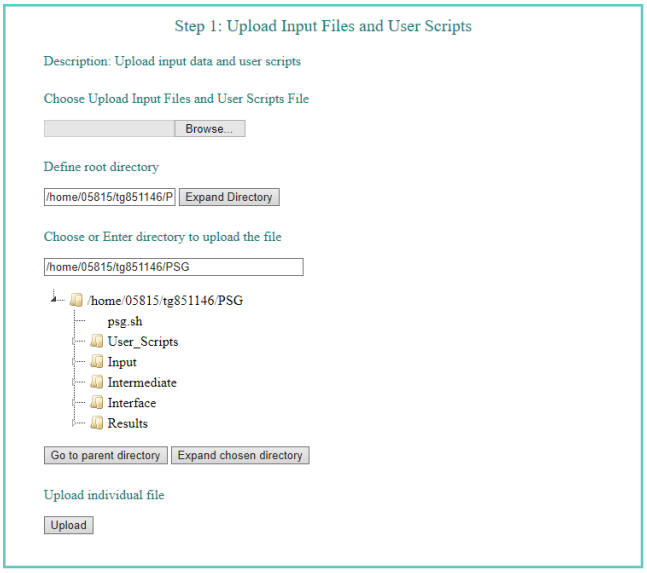


Figure 6: Step 1 - Upload Input Files and User Scripts

The location to the script to be executed is automatically populated. The script is run on remote computing resources. Each of user-specified parameters are used by the tasks performed by PSG as described in the next section.

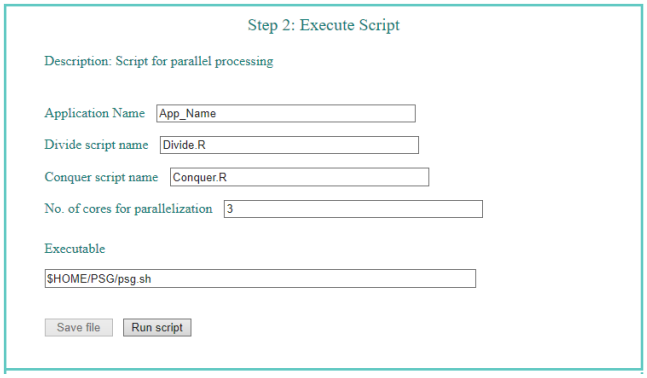


Figure 7: Step 2 - Execute Script

4.1.3 View Results. This task is used to view/download the output. The user has the option to view the first 10 rows of the text file or download the full output file from the remote resource as shown in Figure 8.

4.2 Parallel scripts generator (PSG)

PSG is responsible for automatic code generation to facilitate parallel computing tasks using DC paradigm and to hide the parallel computing details. PSG automatically generates code to enable the parallel divide-and-conquer paradigm without any user intervention. It uses batch processing model to access the underlying CI resources. PSG performs four tasks.

Figure 8: Step 3 - View Results

4.2.1 Folder Structure. - PSG has a predefined folder structure as depicted in Figure 9. The folder *Interface* contains the implementation of PSG. Users input data is uploaded to the *Input* folder and user-defined scripts to the *User_Scripts* folder. The results of the divide task are stored in the *Intermediate* folder and the results of the conquer task are stored in *Results* folder. PSG uses the application name specified by the user to create application-specific folder in *Input* and *User_Scripts* folder to store the uploaded data.

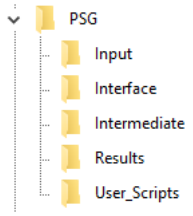


Figure 9: Folder Structure

4.2.2 Split Data. - In this task, PSG takes input data files from users and splits the input files equally among N nodes in order to process the files in parallel. Each set of file paths is written to a file using the notation files-x.txt where x is 1,2...,N. N files are created, one for each node. This task is performed only for problems that require input data. For problems that do not require input data, PSG automatically goes to the next task.

Figure 10 illustrates the code automatically generated by this task. In line 1, *input* points to the data location containing the input dataset from user. In line 4, *numNodes* variable holds the number of nodes specified by the user. Lines 6-8, read the number of files in the input folder, obtains the full path for each file by recursively loping through all the subfolders and splits them into a number of chunks which is equal to *numNodes*. In lines 11-18, each chunk is then written to a file. In our example, the path to input files are split into 3 chunks and written to files, files-1.txt, files-2.txt and files-3.txt.

,

```
1.#User specified dataset - path to input files
2.input <- "home/PSG/Input/App_Name"
3.#User specified # of nodes required for parallelization
4.numNodes <- 3
5.#List of full paths to all input files
6.pathlist <- dir(input,".*",recursive=TRUE,full.names=TRUE)
7.numfiles <- 1: length(pathlist)
8.#Split the paths into 3 chunks
9.chunk <- split(numfiles,sort(numfiles%%numNodes))
10.#Each chunk is written to a file - In this case 3 files.
   files-1.txt, files-2.txt, files-3.txt
11.for (x in 1:numNodes)
12.{
13.  for (c in 1:length(chunk[[x]]))
14.   {
15.     index <- chunk[[x]][c]
16.     write.table(pathlist[index],paste0("files-",x,".txt"),
        row.names = FALSE, col.names = FALSE,append =TRUE)
17.  }
18.}
```

Figure 10: Code generated by Split Data task. Code fragments in red are user specified parameters

Suppose there are 10 input files called NumText1.txt, NumText2.txt and so on. Then, this task would split the files as shown in Figure 11.

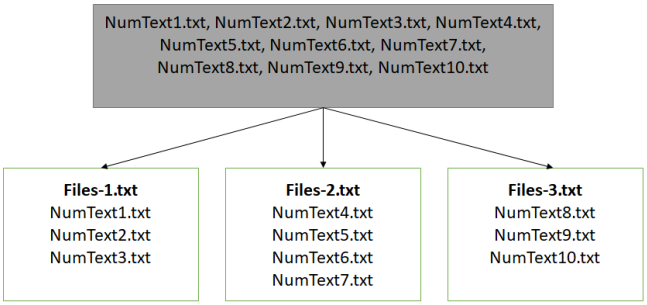


Figure 11: Split Data task

4.2.3 Divide Task. - The divide script provided by the user is designed to process one file at a time. The PSG divide task takes files-x.txt generated in the previous step, loops over the file paths and executes the user defined divide script until all the files are processed. The divide task is executed in each node. Node 1 reads files-1.txt, node 2 reads files-2.txt and so on. In this manner, all input files are processed in parallel. The results of this task are written to an intermediate folder.

The code generated by this task is depicted in Figure 12. In line 2, *node* variable stores the node#. Since there are 3 nodes as specified in the previous step, the nodes will be numbered from 0 to 2. Lines 3-7 takes input parameters for user-defined divide script name and intermediate folder path to store the output of this step. In line 9, depending on the node#, *filename* variable stores the name of the file auto-generated in task 1. This file contains the list of input file paths to be processed by each node. For each path in this file, user-defined divide script is executed once. All the nodes execute the divide script simultaneously until all input files have been processed.

4.2.4 Conquer Task. - In this task, PSG takes intermediate results from divide task as input and passes it to the user defined conquer script to aggregate the results and to produce the final result which is written to the *Results* folder. The results can then be downloaded

```
1.#Environment variable used to obtain the node #. The nodes
   are numbered from 0 to (numNodes-1)
2.node <- as.integer(Sys.getenv()["PMI_RANK"]) + 1
3.args <- commandArgs(trailingOnly = TRUE)
4.#User defined script name for divide task
5.divide_script <- "home/PSG/User_Scripts/App_Name/Divide.R"
6.#Path to intermediate output folder
7.IntermediateFolder <- "home/PSG/Intermediate/App_Name"
8.#Name of auto-genegrated file from task 1
9.filename <- paste0("files-",node,".txt")
10.filepaths <- scan(filename,what="")
11.for (fp in filepaths)
12.{
        system(paste("Rscript ",divide_script,fp,
        IntermediateFolder ))
13.}
```

Figure 12: Code generated by Divide task. Code fragments in red are user specified parameters

by the user using the web interface. This task is executed in serial. It takes three input parameters - user-defined conquer script, intermediate folder and output folder path. Each of the files in the intermediate folder is processed one at a time by the conquer script. This task only starts execution upon successful completion of the divide task. The code generated by this task is depicted in Figure 13.

```
Rscript "home/PSG/User_Scripts/App_Name/Conquer.R"
"home/PSG/Intermediate/App_Name" "home/PSG/Results/App_Name"
```

Figure 13: Code generated by Conquer task. Code fragments in red are user specified parameters

4.2.5 Batch Job Generation. - PSG automatically generates batch processing scripts for divide and conquer tasks. This task is responsible for scheduling batch jobs, distributing the tasks among compute nodes and tracking the job status. The conquer task will not be executed if the divide task fails thereby saving time and compute resources.

4.3 Cyberinfrastructure (CI)

All backend computing tasks are performed on remote resources using batch processing model. The jobs are submitted to Linux cluster job scheduler **SLURM**, using a batch script. All the tasks performed by the framework are executed on Wrangler cluster at The University of Texas at Austin's Texas Advanced Computing Center (TACC). Wrangler is the most powerful data analysis system available in XSEDE. The system is designed for large-scale analytics, data transfer and provides support for a wide range of workflows. It is highly scalable for growth in the number of users and data applications.

5 USE CASES

In this section, we demonstrate how the framework can be used to facilitate data analytics tasks. We have used two popular examples, wordcount and Pi estimation using Monte Carlo simulation. Wordcount program is data-intensive whereas Pi estimation is not.

```r
args <- commandArgs(trailingOnly = TRUE)
filename <- args[1]
outputfolder <-args[2]
sentences<-scan(filename,"character",sep="\n");
#Replace full stop and comma
sentences<-gsub("\\.","",sentences)
sentences<-gsub("\\,","",sentences)
#Split sentence
words<-strsplit(sentences," ")
#Calculate word frequencies
words.freq<-table(unlist(words))
output <- cbind.data.frame(names(words.freq),as.integer(words.freq))
outputfile <- paste0(outputfolder,"/",gsub("/","-",filename))
#Write output to intermediate folder
write.table(output, file=outputfile,row.names = FALSE, col.names = FALSE)
```

(a) Divide.R

```r
args <- commandArgs(trailingOnly = TRUE)
inputfolder <- args[1]
outputfolder <- args[2]
allFiles <- dir(inputfolder)
rfile <- list()
for (i in allFiles)
{
     contents <- read.table(paste0(inputfolder,"/",i))
     rfile <- rbind(rfile,contents)
}
#Sort
rfile <- rfile[order(rfile$V1),]
#Add the frequencies
names(rfile) <- c("words","freq")
final <- aggregate(freq ~ words, data=rfile, sum)
#Write output to Results folder
write.table(final, file=paste0(outputfolder,"/final.txt"),row.names = FALSE, col.names = FALSE)
```

(b) Conquer.R

Figure 14: Wordcount User Code

5.1 Wordcount

Wordcount, widely used to demonstrate the advantage of parallel computing, is the problem of counting the number of occurrences of each word in a collection of documents. We have used Wordcount problem to demonstrate how our framework handles data-intensive problems. Figure 14 represents the user code for divide and conquer phases. Each of the program takes two input parameters in accordance with the proposed DC algorithm.

The split data task splits the input data files equally among different compute nodes. The divide task iterates through the files created by the previous task and executes the user-defined divide script once for each file. The divide script reads the contents of the file and computes the number of occurrences of each word in the file. The output of this task is written to the *Intermediate* folder.The conquer task iterates through each of the intermediate files, aggregates the count for each word and writes the output to the *Results* folder.

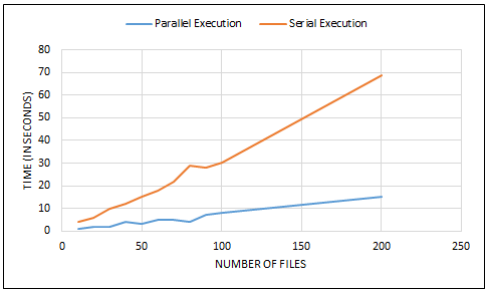
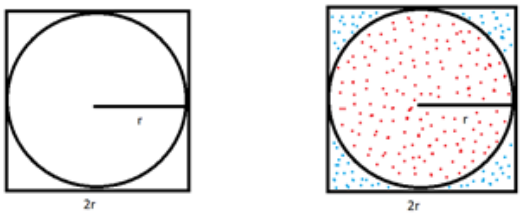


Figure 15: Execution time for wordcount program

For this study, we have varied the number of input files from 10 to 200 in increments of 10. Each file is approximately 6Kb in size. The number of nodes used for parallel processing is set to 10, which means that 10 files will be processed simultaneously until all the files have been processed. The goal of this study is to analyze the performance gain using serial execution and parallel execution facilitated by the framework. For serial execution analysis, the number of nodes is set to 1. For both the parallel and the serial programming models, we have recorded the time it took to complete both divide and conquer tasks. As seen in Figure 15, the framework offers significant improvement in execution time by parallelizing the data processing tasks. The execution time increases significantly as the number of files increases. Using the framework proves to be beneficial in data-intensive tasks since the load is divided among several nodes and processed in parallel.

5.2 Pi Estimation

Pi is a mathematical constant and it is calculated using the ratio of the circumference of a circle to its diameter. One common approach of estimating Pi is by using the Monte Carlo method. The Monte Carlo method uses statistics to solve problems. It uses random numbers to simulate various statistical outcomes. With this method, we cannot get the exact value but only an estimate by repeating a random process. The higher the number of repetitions, the closer we get to the exact value.



(a) **(b)**

Figure 16: Pi Estimation Using Monte Carlo Simulation

To estimate the value of Pi using Monte Carlo method, consider a circle inside a square as shown in Figure 16 (a). The radius of the circle is denoted by r and the length of each side of the square is 2r. The area of the circle is Pi * r * r and the area of the square is (2r) * (2r) which is equal to $4r^2$. Hence the ratio of the area of the circle to the area of the square is Pi/4. Suppose we generate N uniformly distributed random points inside the square as illustrated in Figure 16 (b). The total number of points inside the circle can be calculated using the distance formula. The points with distance less than 1 represents the points inside the circle and it is denoted by R. The value of Pi is then estimated using the formula Pi = R * 4/N, where N is the sum of red and blue points. The performance is studied by varying the number of random points generated. We have used Pi estimation problem to study the effectiveness of our framework for problems that are not data-intensive.

,

```
1. args <- commandArgs(trailingOnly = TRUE)
2. outputfolder = args[1]
3. rfile = "red.txt"
4. tfile = "total.txt"
5. x=runif(10000)
6. y=runif(10000)
7. z=sqrt(x^2+y^2)
8. red = length(which(z<=1))
9. total = length(z)
10. ifelse(!file.exists(paste0(outputfolder,"/",rfile)),file.create(paste0(
    outputfolder,"/",rfile)),FALSE)
11. ifelse(!file.exists(paste0(outputfolder,"/",tfile)),file.create(paste0(
    outputfolder,"/",tfile)),FALSE)
12. #Write output to intermediate folder
13. write.table(red,paste0(outputfolder,"/",rfile),append = TRUE,row.names
    = FALSE,col.names = FALSE)
14. write.table(total,paste0(outputfolder,"/",tfile),append =
    TRUE,row.names = FALSE,col.names = FALSE)
```

(a) Divide.R

```
1. args <- commandArgs(trailingOnly = TRUE)
2. inputfolder = args[1]
3. outputfolder = args[2]
4. rfile = "red.txt"
5. tfile = "total.txt"
6. final = "final.txt"
7. ifelse(!file.exists(paste0(outputfolder,"/",final)),file.create(paste0(
   outputfolder,"/",final)),FALSE)
8. rcount = sum(read.table(paste0(inputfolder,"/",rfile)))
9. tcount = sum(read.table(paste0(inputfolder,"/",tfile)))
10. PI = rcount *4/tcount
11. #Write output to Results folder
12. write.table(PI,paste0(outputfolder,"/",final),append = TRUE,row.names =
    FALSE,col.names = FALSE)
```

(b) Conquer.R

Figure 17: Pi Estimation User Code

The user code for divide and conquer phases are as depicted in Figure 17. Since this use case does not require input data, PSG skips the split data task. The divide script generates 10000 random points. The points within the circle are written to red.txt and the total number of points within the square are written to total.txt. Both of these files are stored in the *Intermediate* folder. The conquer task loops through the files in the *Intermediate* folder, estimates Pi value using the given formula and writes it to the *Results* folder.

To analyze the performance gains for Pi estimation, we have varied the number of iterations from 1000 to 100000. We varied the number of nodes used depending on the number of iterations and recorded the time it took to complete both divide and conquer tasks. As seen in Figure 18, the framework does not offer significant time gain. This is due to the fact that the program is not data-intensive,

Figure 18: Execution time for Pi estimation program

and parallelization causes some overhead thereby affecting the performance gain. However, the execution is still faster with the use of the framework and it also offers a way to easily scale the program for larger computations.

6 CONCLUSION

This paper proposes a framework that uses divide-and-conquer paradigm to automatically parallelize user scripts while hiding details of parallelization. The framework enhances accessibility to advanced CI and automatically generates code for parallel computing of data-intensive tasks. It is easy to use even for programmers who have little to no knowledge of parallel systems and parallel programming languages. In addition to that, a large variety of problems can be easily expressed as divide-and-conquer computations.

ACKNOWLEDGMENTS

This work was supported by NSF award #1726532. The workflow and web-based UI is developed in collaboration with TACC and tested on Wrangler cluster.

REFERENCES

[1] 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.

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

[3] M. D. A. Praveena and B. Bharathi, "A survey paper on big data analytics," 2017 International Conference on Information Communication and Embedded Systems (ICICES), Chennai, 2017, pp. 1-9. doi: 10.1109/ICICES.2017.8070723

[4] Parhami B. (2018) Parallel Processing with Big Data. In: Sakr S., Zomaya A. (eds) Encyclopedia of Big Data Technologies. Springer, Cham

[5] Horowitz and Zorat, "Divide-and-Conquer for Parallel Processing," in IEEE Transactions on Computers, vol. C-32, no. 6, pp. 582-585, June 1983. doi: 10.1109/TC.1983.1676280

[6] Hui Zhang, Yiwen Zhong, Juan Lin, "Divide-and-Conquer Strategies for Large-scale Simulations in R", 2017 IEEE International Conference on Big Data (BIG-DATA)

[7] Subramanian, Ranjini Zhang, Hui. (2018). Performance Analysis of Divide-and-Conquer strategies for Large scale Simulations in R. 4261-4267. 0.1109/Big-Data.2018.8622068.

[8] Zhuang, Peixian Fu, Xueyang Huang, Yue Ding, Xinghao. (2017). Image Enhancement Using Divide-and- Conquer Strategy. Journal of Visual Communication and Image Representation. 45. 10.1016/j.jvcir.2017.02.018.

[9] Yoo, M. Jette, and M. Grondona, "Slurm: Simple Linux Utility for Resource Management ," Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 2862, pp. 44-60, 2003

[10] Rstuido Team. R Studio. [Online]. https://www.rstudio.com/

[11] Merchant, Nirav, et al., "The iPlant Collaborative: Cyberinfrastructure for Enabling Data to Discovery for the Life Sciences," PLOS Biology, 2016.

[12] Vera, Gonzalo, Ritsert C. Jansen and Remo Suppi. âĂIJR/parallel âĂŞ speeding up bioinformatics analysis with R.âĂİ BMC Bioinformatics 9 (2008): 390 - 390.

[13] Ruan, Guangchen Zhang, Hui Wernert, Eric Plale, Beth. (2014). TextRWeb: Large-Scale Text Analytics with R on the Web. ACM International Conference Proceeding Series. 10.1145/2616498.2616557.

[14] Ji Kang, Sol Yeon Lee, Sang Lee, Keon Myung. (2015). Performance Comparison of OpenMP, MPI, and MapReduce in Practical Problems. Advances in Multimedia. 2015. 1-9. 10.1155/2015/575687.

[15] Dean, Jeffrey Ghemawat, Sanjay. (2004). MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM. 51. 137-150. 10.1145/1327452.1327492.

[16] V. Kalavri and V. Vlassov, "MapReduce: Limitations, Optimizations and Open Issues," 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, Melbourne, VIC, 2013, pp. 1031-1038. doi: 10.1109/TrustCom.2013.126

[17] Xu, Weijia Huang, Ruizhu Wang, Yige. (2018). Enabling User Driven Web Applications on Remote Computing Resource. 10.1109/SERVICES.2018.00038.

[18] I. Gorton, P. Greenfield, A. Szalay and R. Williams, Data-Intensive Computing in the 21st Century, in Computer, vol 41, no. 4, pp. 30-32.doi :10.1109/MC.2008.122 (2008)