Automatic code parallelization for data-intensive computing in multicore systems

Automatic code parallelization for data-intensive computing in multicore systems

Ranjini Subramanian and Hui Zhang*

University of Louisville, Computer Science Department, Louisville, Kentucky, USA

* Corresponding author: h0zhan22@louisville.edu

Abstract. A major driving force behind the increasing popularity of data science is the increasing need for data-driven analytics fuelled by massive amounts of complex data. Increasingly, parallel processing has become a cost-effective method for computationally large and data-intensive problems. Many existing applications are sequential in nature and if such applications are ported to multi-processor systems for execution, they would make use of only one core and the optimal usage of all cores is not guaranteed. Knowledge of parallel programming is necessary to ensure the use of processing power offered by multi-processor systems in order to achieve better performance. However, many users do not possess the skills and knowledge required to convert existing sequential code to parallel code to achieve speedups and scalability. In this paper, we introduce a framework that automatically transforms existing sequential code to parallel code while ensuring functional correctness using divide-and-conquer paradigm, so that the benefits offered by multi-core systems can be maximized. The paper will outline the implementation of the framework and demonstrate its usage with practical use cases.

1. Introduction

Data analytics is a branch of science which involves applying algorithmic process to analyse datasets in order to extract useful information from it. It is widely used in healthcare, scientific applications, gaming etc. Data and computing power have grown rapidly in the last few decades, but the former has risen at a much higher pace than the latter making the use of high-performance computing (HPC) resources necessary to meet the computing resource requirements. Exponential growth in data has led to a two major challenges [1]:

- Managing and processing large volumes of complex data
- Reducing data analysis time to enable researchers to make timely decisions

The challenges have led to the emergence of data-intensive computing which is crucial in analysing massive datasets quickly in a highly scalable environment. Data-intensive computing uses data parallelism to process large volumes of data. This is can only achieved by using multi-core systems that are best suited for parallel processing and offer huge computing power.

We have used the divide-and-conquer (DC) paradigm to enable parallel processing. DC methodology works by recursively breaking down a problem into similar subproblems that can then be processed in parallel. The solutions to the subproblems are then aggregated to obtain solution to the original problem [2]. This methodology can be easily adapted to multicore systems. However, exploiting parallelism offered by DC methodology requires certain expertise in parallel programming and

knowledge of underlying HPCs. There exist access barriers in bringing these resources to a broad range of users. Users who are new to large-scale computing are not yet trained to take advantage of the tools offered by HPCs. In addition to that, most legacy applications are inherently sequential and do not make use of parallel programming.
Sequential implementation of programs works as a single unit of code, processing the dataset one at a time. Parallelizing a sequential program demands time and resources since it involves understanding the domain, data and control dependencies, underlying computing cluster architecture and rewriting the code that synchronizes various sections and executes in parallel.
We propose breaking down the sequential code into two individual programs – divide and conquer, each processing the dataset sequentially. This code conversion can be easily achieved by novice users since it does not require the knowledge of parallel programming tools and methodologies.
In this paper, we propose a framework that generates code automatically to enable parallel implementation of DC methodology using high performance computing cluster. The framework takes divide and conquer user scripts along with input data and enables parallelization of data processing without user intervention. Some of the key design goals are reusability, scalability and reproducibility. The framework is designed to support both R and Python programming languages which are two of the most widely used languages in data science.
The paper is organized as follows – section 2 addresses background, section 3 details the implementation of the framework, and section 4 demonstrates the usefulness of this framework by analysing its performance.

2. Background

Processing data-intensive computing tasks requires significant processing power offered only by HPCs. They use multicore, multithreaded processors as computing resources. Though HPCs offer great scalability and computing power, porting the existing sequential code to exploit the features of HPC is not straightforward. In this section, we explore the serial and parallel modes of implementation.

2.1. Data processing in serial mode

In the serial mode, programs are written in iterative style. The algorithm for serial mode is as illustrated in figure 1:

```
Algorithm: Serial_Mode
Input:
D: Input data
ds:dataset
D:= Ids

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. Data Processing in Serial Mode.

In this mode, the for-loop loops through each dataset, reads it, performs user-defined computation and then writes the intermediate results to an intermediate folder. The intermediate results are then processed one at a time and the aggregated results are written to the output folder.
For-loops are slow in execution and leads to substantial overhead for large data sets. This style of implementation cannot be executed in parallel without significant code changes which requires

advanced programming skills. This disadvantage has motivated us to adopt DC paradigm which is naturally suited for parallel processing.

2.2. Data processing using DC paradigm

DC paradigm is extensively used in data-intensive computing due to its performance gains and versatility. It has been used to solve many data-intensive problems like matrix eigenvalue problem [3], simulating quantum mechanical systems on quantum computers using Hamiltonian structure [4]. This methodology can be easily adapted in multi-processor, shared-memory systems like HPCs because the subproblems can be solved independently on different processors and the communication needed between processors is minimal. DC methodology works in two phases:

- ***Divide phase*** – In this phase, the input dataset is broken down into chunks and processed simultaneously independent of each other. This constitutes the bulk of the processing in a data-intensive problem. The result of parallel processing is stored in an intermediate folder.
- ***Conquer phase*** – This step involves aggregating the intermediate results of the previous step to produce the final output which is the original solution to the problem.

In [5], performance of different types of DC implementations [6] are outlined. The study was aimed at evaluating scalability and performance of serial and parallel implementations of DC methodology for large-scale simulations. The variables used for the study were batch size (B), number of intermediate files (I) and the dimension of the files (D). The number of processors used for each execution is set to 1 for serial implementation and B for parallel implementation. To evaluate performance, B was varied from 100 to 2000, I was set to 10, D was set to 100x100 and a 45 second wait time was added to mimic application-specific tasks. It was found that parallel mode reduced the execution by over 98% compared to serial implementation. The time increased linearly with data size. In addition to that, execution failed due to memory issues for increasing data sizes in serial 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. Data Processing using DC paradigm.

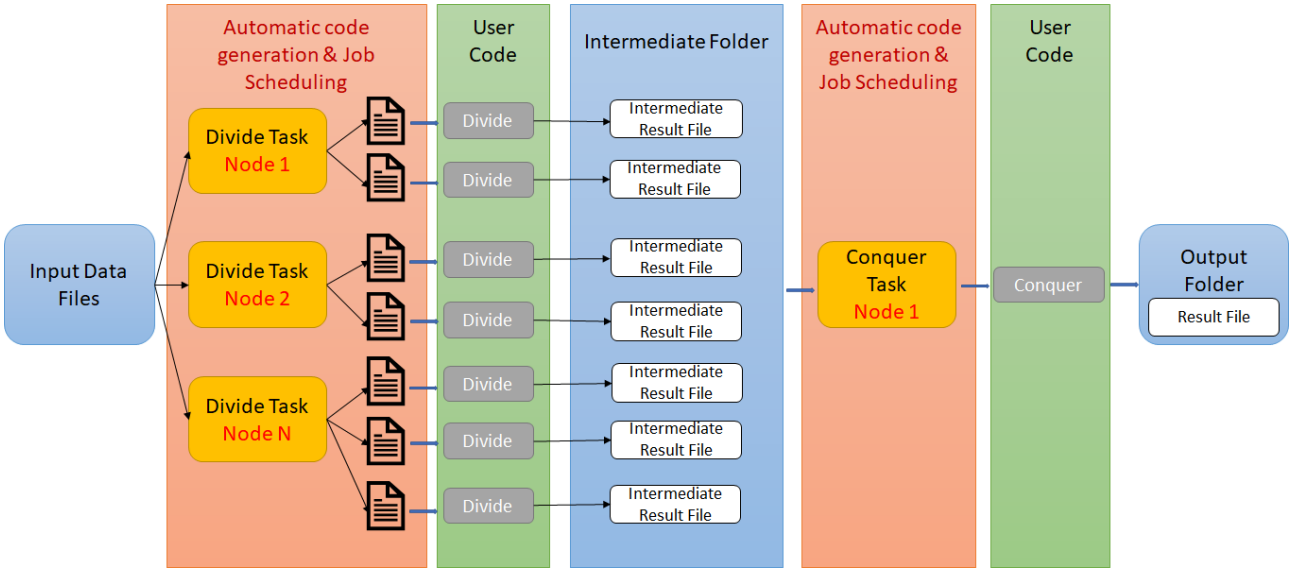

Figure 3. Process Flow.

Both phases can be implemented as functions in a single program or can be implemented as two separate programs. Implementing them as separate programs allows for script level parallelism. It doesn't require fixed computation steps as is needed in MapReduce model [7,8] which uses function-level parallelism. Separate implementations also allow for either both phases to be parallelized or just a single phase to be parallelized depending on application-specific requirements. We have chosen to parallelize only the divide phase because it involves the bulk of the processing and the conquer phase is not as data intensive. Also, parallelization of divide phase is straightforward whereas parallelizing conquer phase would require significant programming effort from the user. The proposed algorithm for divide and conquer phases is as shown in the Figure 2. User is responsible for both the programs and they are implemented for sequential execution.

The divide script takes two input parameters, filename of the dataset and intermediate folder path. It processes the dataset by performing user-specified business logic and stores the result in an intermediate folder. The conquer phase takes two input parameters, path to intermediate folder and path to output folder. It loops through all intermediate results, aggregates them and write the result to the output folder.

The design consideration to implement divide and conquer phases as individual scripts was done for the following reasons:

- It offered greater flexibility to implement user-defined business logic. The user could also tailor the intermediate and output result format to suit their individual needs
- File level processing reduces the communication between processors thereby reducing synchronization issues and overhead
- Users can easily debug their sequential code implementation locally and leave the parallelization details and scalability to the proposed framework.

3. Our framework

The proposed framework is aimed at providing the users with little to no knowledge of parallel computing, a way to focus on sequential implementation and leave the parallelization details to the framework [9]. The framework hides the parallel computing details and automatically generates code to process data in parallel using batch processing jobs. The process flow is depicted in Figure 3.

The divide task splits the input data among the compute nodes which in turn processes each file using the user-defined divide script. The results of this task are written to the intermediate folder. The conquer task passes each file from the intermediate folder to user-defined conquer script which aggregates the intermediate results and writes the final output to the output folder.

The framework is implemented using R programming language. It consists of three layers:

- User Interface
- Parallel Scripts Generator (PSG)
- Underlying compute cluster

We will demonstrate the functionality of the framework using wordcount program as an example. This program is written using R programming language.

3.1. User interface

Users can access the framework using a command line interface. It takes the following input parameters from the user:

- Application name
- User script names
- Number of cores required for parallelization

Application name is used as a default name for input, intermediate and output folders. In addition to the parameters, users are required to upload the scrips and input data when necessary to a predefined folder.

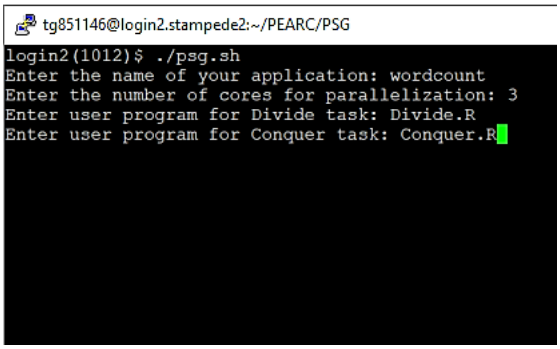

Figure 4. Input parameters for wordcount program.

In order to execute the wordcount program, we have used the following parameters are shown in Figure 4.

- Application name – wordcount
- Number of cores for parallelization – 3
- User program for divide task – Divide.R
- User program for conquer task – Conquer.R

3.2. Parallel Scripts Generator (PSG)

PSG has a predefined folder structure as depicted in Figure 5.

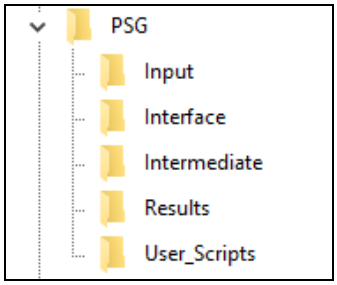

Figure 5. PSG folder structure.

The folder Interface contains the implementation of PSG. Users must upload the input data to Input folder and user-defined scripts to 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 both of which are automatically created using the application name specified by the user.
PSG is responsible for hiding parallel computing details and automatic code generation. It performs three important tasks – Split data, Divide task and Conquer task.

3.2.1. Split data. This task is responsible for splitting the input files equally among the different nodes for parallel processing. In our case, the flies are split into 3 set. Each set of file paths are written to a file with the notation files-x.txt where x is 1,2, or 3. One file is created for each node. This task is only performed where the application requires input data.

```
1.#User specified dataset – path to input files
2.input <- "home/PSG/Input/wordcount"
3.#User specified # of cores required for parallelization
4.numCores <- 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%%numCores))
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:numCores)
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 6. Code generated by Split Data task. Code fragments in red are user specified parameters.

Figure 6 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, *numCores* variable holds the number of cores specified by the user for parallelization. Lines 6-8, reads 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 several chunks which is equal to *numCores*. 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.
The wordcount application contains 10 input files. Figure 7 illustrates the files split into chunks to be processed by individual cores.

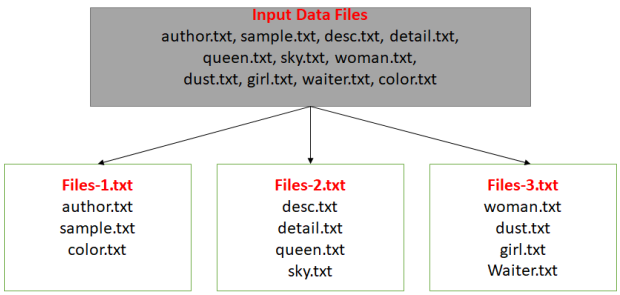

Figure 7. File chunks generated by split data task.

3.2.2. Divide task. The user-defined divide script is designed to process input files sequentially. This task is responsible for iterating through files-x.txt created in the previous step and passing the file paths to Divide.R script one at a time until all files are processed. Core 1 iterates through files-1.txt, core 2 iterates through files-2.txt and so on. The results of each execution of Divide.R are written to an intermediate folder.
The code autogenerated by this task is depicted in figure 8.

```
1.#Environment variable used to obtain the core #. The cores
   are numbered from 0 to (numCores-1)
2.core <- 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/wordcount/Divide.R"
6.#Path to intermediate output folder
7.IntermediateFolder <- "home/PSG/Intermediate/wordcount"
8.#Name of auto-genegrated file from task 1
9.filename <- paste0("files-",core,".txt")
10.filepaths <- scan(filename,what="")
11.for (fp in filepaths)
12.{
       system(paste("Rscript ",divide_script,fp,
       IntermediateFolder ))
13.}
```

Figure 8. Code generated by Divide task. Code fragments in red are user specified parameters.

In line 2, *core* variable stores the core# which is obtained from the environment variable. This variable is used to determine the files-x.txt file for this core. Lines 5-7 reads the user specified parameters. The intermediate folder is automatically created using the application name specified by the user. In line 9, depending on the core#, *filename* variable stores the name of the file autogenerated in task 1. f*ilename* contains list of file paths to be processed by the *core*. For each file path, Divide.R is executed once. This process is continued until all the files have been processed. All the nodes processes files simultaneously.

3.2.3. Conquer task. The conquer task is responsible for passing the intermediate folder path and the output folder path to Conquer.R. This task is executed sequentially and thus requires only one core. The conquer task begins execution only upon successful completion of divide task thus saving time and compute resources in case of failure. The code autogenerated by this task is depicted in Figure 9.

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

Figure 9. Code generated by Conquer task. Code fragments in red are user specified parameters.

3.3. Underlying compute cluster

All backend computations are performed using batch processing model in a research computing cluster. The jobs are submitted to Linux cluster job scheduler SLURM. SLURM is the most popular scheduler and it is responsible for resource management, job lifecycle management, job scheduling and execution. Jobs are submitted to SLURM using a job script. A job script is an executable file which contains a list of directives that tells the scheduler what to do. Similar jobs can be submitted using a single script and they are designed to run independently on different cores [10].

4. Results and performance analysis

Wordcount is the problem of counting the number of occurrences of each word in a collection of documents. Wordcount is a popular problem that has been widely used to demonstrate the advantage of parallel computing. We have used Wordcount problem to demonstrate how our framework handles problems that require input data or are data-intensive.

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. In the absence of the framework, this program would be execution in serial. 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 10, the framework offers significant improvement in execution time by parallelizing the data processing tasks.

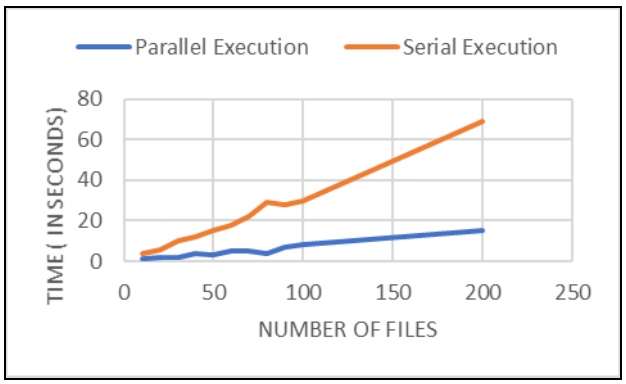

Figure 10. Execution time for wordcount program.

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. Conclusion

In this paper, we have proposed a framework which automatically parallelizes data processing jobs using DC paradigm. A key motivation for this framework is to enable parallelization of data-intensive computing tasks without user intervention. It is aimed at users from various domains who do not necessarily have the skill to parallelize their code and at legacy applications that are inherently sequential in nature and could benefit from parallel processing. Our ongoing work focusses on extending the usability of the framework by providing a better user interface aimed at users who are from non-computation background.

6. References

[1] 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)

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

[3] Y. Cui, J. Qu, W. Chen and A. Yang, Divide and conquer algorithm for computer simulation and application in the matrix eigenvalue problem, International Conference on Test and Measurement, Hong Kong, 2009, pp. 319-322.doi :10.1109/ICTM.2009.5412930 (2009)

[4] S. Hadfield and A. Papageorgiou, Divide and conquer approach to quantum Hamiltonian simulation, New Journal of Physics (2018)

[5] R. Subramanian and H. Zhang. Performance Analysis of Divide-and-Conquer strategies for Large scale Simulations in R. 4261-4267. 0.1109/BigData.2018.8622068 (2018)

[6] H. Zhang, Y. Zhong, J. Lin, Divide-and-Conquer Strategies for Large-scale Simulations in R", IEEE International Conference on Big Data (BIGDATA) (2017)

[7] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM. 51. 137-150. 10.1145/1327452.1327492 (2004)

[8] V. Kalavri and V. Vlassov, MapReduce : Limitations, Optimizations and Open Issues, 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 (2013)

[9] G. Ruan, H. Zhang, E. Wernert and B. Plale. TextRWeb : Large-Scale Text Analytics with R on the Web. ACM International Conference Proceeding Series. 10.1145/2616498.2616557 (2014)

[10] 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)

Acknowledgements
This work was supported in part by NSF DUE award #1726532.