DD2356 Methods in High Performance Computing Final Project Report

Group member: Weijiang Kong [weijiang@kth.se](mailto:weijiang@kth.se)

(your name here) (your email address here)

*Abstract*—Our group implemented a parallel method of map and reduce in MPI according to project specification. We validated and tested our program through multiple methods and concluded that we have reached a proper performance.

# Introduction

In this project, our purpose is to implement perform parallel computing in map and reduce. The project includes three files: backend.c, user\_case.h and toolbox.h. Backend implementation is included in backend.c, to provide a parallel method of map and reduce function and also specify communication between paralleled processes. User-case implementation is included in user\_case.h providing separation of map, reduce and hash function. toolbox.h contains functions required by other features, for example, sort. We include following optional features during our implementation: collective I/O for input, separation of back end and user case implementation, configurable block size and file path, support for non-multiple file size, master process sharing workload during map phase. Note that one of the benefit from collective I/O is that rank 0 which is the master process can share effort with other part of the program during map phase. But it also distributes the effort of reading input file to all the processes.

Both strong and week scaling evaluation are performed to our implementation, to verify whether the parallel method is scalable. Our result indicates…

# Implementation details

According to functionality various for different part of the code, the program was divided into different phases, making it easier to separate effort during cooperation and identify bugs during evaluation. While entering each phase, program will print messages indicating its executing status. Details for each phase is explained as below.

## Initialization phase

Configurable parameters are passed to program during this phase, including file path and user defined block size through --filename and --blocksize. Default block size is 64MB. Parameters of the program will be extracted to form parallelism, for example, number of ranks and rank for present process. A new MPI datatype is defined for transporting self-defined structure datatype through MPI protocols. To specify such datatype, a struct is created and its content and displacement are extracted. Also input file is opened and loop parameters are initialized for later computation.

## Input-Scatter phase and Map-shift phase

File input and map functions are executed through while loop iterations. If the file is not drain, a while loop will keep reading input file and apply map function toward char array acquired from the input. Note that we are using collective I/O so each process calculates its own offset for next iteration and fetch the data needed from the file. Our implementation will ignore tail of the file if it detects file will be ended during next iteration, so file size which is not multiple of block size can be supported. However, program will output the number of data skipped for validation purpose, for normal condition, this number should be 0. Also note that map functions used in this phase is implemented in user\_case.h, it’s designed to extract a string from a designated array each time and advance offset of that array due to implementation simplicity. So, a while loop is required to drain the array.

## All to all communication phase

A hash function is given by project specification to provide match between word and ranks. It’s also located at user\_case.h. With this function, we can split all world collected to different ranks with all same word in one rank. After computing how each word located, all words will be relocated with word to same rank adjacent with each other. A all to all communication is issued at this time, to distribute the number of data received for each rank. Displacement array will be generated through these data. After all those preparation, all to all(MPI\_Alltoallv) was issued to distribute key/word pairs. Note that there is a debug feature implemented before all to all to view all send and receive count and displacement arrays.

## Reduce phase

Reduce function implemented in user\_case.h will be called to compress output size by deleting words with same key but increase value. It’s using vector from STD to simplify the enumeration. Reduce function will take a key/word pair and search through vector storing its previous results looking for matches. If there is no match, the pair will be stored at the end of result vector, if there is match, the pair will not be stored, but value of the matched pair will be incremented. Note that different from map function, reduce function dealing with the whole container, so it doesn’t need to be put into a loop.

## Configuring output and DEBUG flags

Several debug flags are created during our debug effort and are preserved as features of the program, to provide easy access to several certain output.

SORT\_RESULT: enabling sort function before result output.

SHOW\_PROGRESS: enabling program to output the phase it’s executing on. Note that since there are multiple processes, only rank 0 which is the master process is allowed to print out its phase under the purpose of conciseness.

SHOW\_RESULT: printing out computation result. All key/value pairs will be printed out. The size of input for a 10 GB dataset is around 64 MB which is very large for a terminal output. Thus, we recommend printing those pairs into a file for clear view.

TIME\_REPORT: print out time report, it will give a report on how much time is consumed by each phase of execution.

# Validation Banckmarking and evaluation

Our efforts to evaluate whether our program is well structured and implemented consists of following several steps. The main ideal of such effort is first to validate locally that the program functions correctly, then using large data set in Beskow to measure performance and do evaluation.

First, the program was tested locally using a small data set. The word sorted out were printed into a file. We randomly chose keys and search it through the data set to verify if the corresponding value is correct. Note that for small data set and early debugging phase, the block size for each reading iteration is relative small, so it can be printed out for debugging analysis. This cause higher possibility that a word was broken and separated into two different blocks. We observed a few of such errors and its number decreases while we increase block size. So, we consider such error ignorable while using 64MB block size for later tests.

Then, the program was uploaded to Beskow and validated through smallest PUMA data file, to test compatibility of our implementation since local compilers and libraries may different from what’s in Beskow. Implementation is considered feasible if it finishes within an acceptable time and given output with reasonable form and size. Considering the large size of testing data set, we didn’t find a feasible way of implementing an automated and completed validation procedure to verify output correctness.

The last step for evaluation is testing the scalability of our implementation. Both strong scaling and week scaling are performed. All non-necessary optional features are disabled to reduce noise of result acquired, including sort, result print in terminal, result print in file and detailed time report.

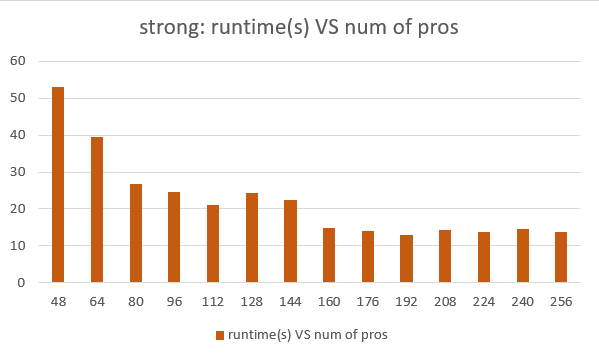
For strong scaling, we use test file size of 10 GB, since low file size provides maximum test efficiency and performance under large dataset can be evaluated during week scaling test. 16 nodes are allocated so maximum number of threads is 256 considering 2 openMP threads are assigned to each MPI process. Note that to further improve test efficiency without introducing too much interference, each number of processes are executed twice, among which runtime for second executing will be recorded if doesn’t have too much variance with first one. Acquired runtime from each number of processes is listed below.

Note that in order to observe the pattern runtime scales along with number of processes, we select tested process numbers linearly. This configuration may reduce efficiency of communication by breaking address space, however, it does give a clear view of how number of runtime changes along with processes.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Process | 48 | 64 | 80 | 96 | 112 |
| Runtime(s) | 53.02 | 39.45 | 26.78 | 24.53 | 21.00 |
| Process | 128 | 144 | 160 | 176 | 192 |
| Runtime(s) | 24.18 | 22.55 | 14.89 | 14.06 | 13.06 |
| Process | 208 | 224 | 240 | 256 |  |
| Runtime(s) | 14.24 | 13.82 | 14.66 | 13.84 |  |

We observe that when number of processes scales linearly, runtime doesn’t follow this linear pattern, instead, it is close to a step function with exponential pattern. This probably because the runtime for all to all communication does not impacted that much by change of number of processes, but instead, reduced exponentially as amount of data sent reduces.

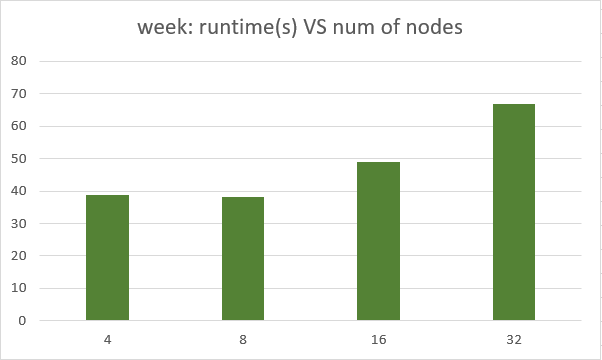
Previous table can be visualized to give a clearer view of the scale down progress, if we use a curve to fit the diagram, we can see an exponential decreasing pattern. However, this pattern glitches when number of threads equals to 128 and 144. Detailed time report indicates a vast increase of runtime in all to all phase , possibly caused by number of threads exceeds certain threshold.



For week scaling, we select data size(GB)/number of nodes equals to 1.25, which means 10 GB data requires 8 nodes, 20 GB data requires 16 nodes and so on.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Num of nodes | 4 | 8 | 16 | 32 |
| Size of file | 10 | 20 | 40 | 80 |
| runtime | 38.64 | 38.27 | 48.81 | 66.71 |

Result acquired is listed below. Note that Beskow seems very busy while we doing the testing, usually it takes around 5 minutes to receive grant for using 32 nodes, so for week scaling, we only acquire data for using 4, 8, 16 and 32 nodes. We noticed that runtime tends to stay stable while number of nodes is small and increases as number of nodes exceeds 8. We concluded this pattern caused by increase runtime of all to all communication as number of nodes increases but amount of communication per node stays the same.



# ConClusion and discussion

In this project, our group experienced the process of developing parallel program in large data size through online team work. Through all the evaluation efforts, we conclude that our implementation is functionally correct and timely efficient. Also, scalability test indicates a exponentially strong scalability for process number less than 256, although week scalability is not so obvious due to lack of data and features of the program. Acquired data implies an acceptable performance hence we considered reaches the purpose of this project.

One of the main pitfall of our implementation is its memory efficiency. We are using global variables, part of them are expected to occupy memory through the whole runtime and we didn’t find a good way to free those memories. Also, further optimization can be done to algorithms during both gather and map phase. We didn’t use std::map for storage during map phase since we are expecting another iteration loop to extract data. Also, memory management during gather phase is implemented in a very simple way by using a single vector. This makes program performs poor while running small number of threads in large data set. Our test indicates running program with 2 nodes in a 10 GB data set could possibly cause problems since program run out of memory.