Paper 10780-2016

Increasing Efficiency by Parallel Processing

Shuhua Liang, Fagen Xie, Kaiser Permanente Southern California

ABSTRACT

Data volume is increasing exponentially across industries and working with big data is often time consuming and challenging. The primary goal in programming is to maximize throughputs while minimize the use of execution time and resources. By utilizing the Multiprocessing (MP) Connect method on a Symmetric Multiprocessing (SMP) computer, a programmer can divide a job into independent tasks and execute threads in parallel on multiple processors. This paper demonstrates the development and application of a parallel processing program on large sets of healthcare data in SAS®.

INTRODUCTION

A single-thread SAS program is a serial process. When extracting, joining and processing complex datasets, a single-thread process typically requires a long execution time and occupies a large portion of computer resources to complete a job. A manual approach to reduce running time is to divide a large dataset into smaller subsets and execute multiple jobs at the same time. On the other hand, a multi-tread process exploits a multiprocessor machine to perform multiple tasks simultaneously [Shamlin]. When handling different data structures, a program or partial code of the program can be revised into one or a combination of the following parallel processing models:

- Boss/Worker Model
- Peer Model
- Pipeline Model

The Boss/Worker Model is structured to allow the main thread to generate and distribute tasks based on the input for sub-threads to complete, the Peer Model is to create multiple independent threads that can perform various tasks on the same set of input concurrently, and the Pipeline Model performs its tasks stage by stage sequentially. The purpose of this paper is to apply these techniques in developing parallel processing SAS® programs, and utilize them to manage big data in a healthcare research oriented organization to achieve higher performances and retrench resources from various aspects.

ASSUMPTIONS OF INDEPENDENCY

When a program is implemented using threads in parallel, the following assumptions are made:

- Threads are disjoint units; each thread is an independent task that can be computed without dependency on another [Powers].
- A single thread does not generate additional threads [Powers].
- In a perfect world, the degree of speedup is equivalent to the amount of resources added to the process.
- No physical memory or processors are shared among threads.

DATA STRUCTURE AND ARCHITECTURE

We will consider splitting, joining and consolidating three sets of data using threads in a parallel structure. The structure of this program is to have a boss thread distribute inputs to worker threads, and then the worker threads work in a peer model structure to perform the same set of tasks in parallel. Within each worker thread will be a pipeline model, where a collection of tasks are executed sequentially. This relationship is illustrated in Figure 1.

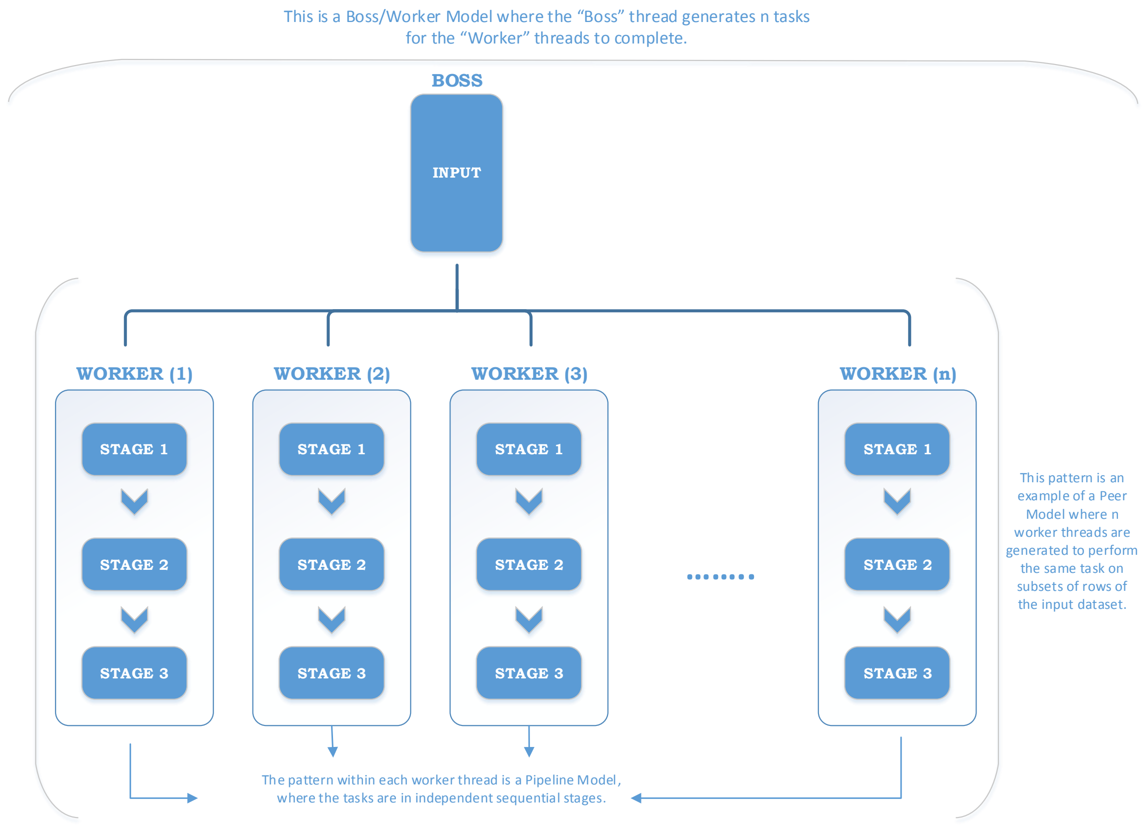

This is a Boss/Worker Model where the "Boss" thread generates n tasks for the "Worker" threads to complete.
BOSS
INPUT
WORKER (1)
WORKER (2)
WORKER (3)
WORKER (n)
STAGE 1
STAGE 2
STAGE 3
This pattern is an example of a Peer Model where n worker threads are generated to perform the same task on subsets of rows of the input dataset.
The pattern within each worker thread is a Pipeline Model, where the tasks are in independent sequential stages.

Figure 1. Overview of Program Structure

In the actual study, there are three sets of data that we need to join and analyze:

- Dataset A contains 26,912,194 observation, and it is the primary dataset (skeleton) that will be divided into n subsets.
- Dataset B contains 683,428 observations, and it will be inner joined with subsets of dataset A to create datasets AB_n.
- Datasets AB_n will be processed and analyzed in data and proc sort steps.
- Each AB_n will be inner joined with dataset C (contains 656,773 observations) for additional variables and further analyze; outputs will be datasets ABC_n.
- ABC_n sets will be merged into one final output, dataset D.

Figure 2 shows the relationship among datasets and tasks on the datasets:

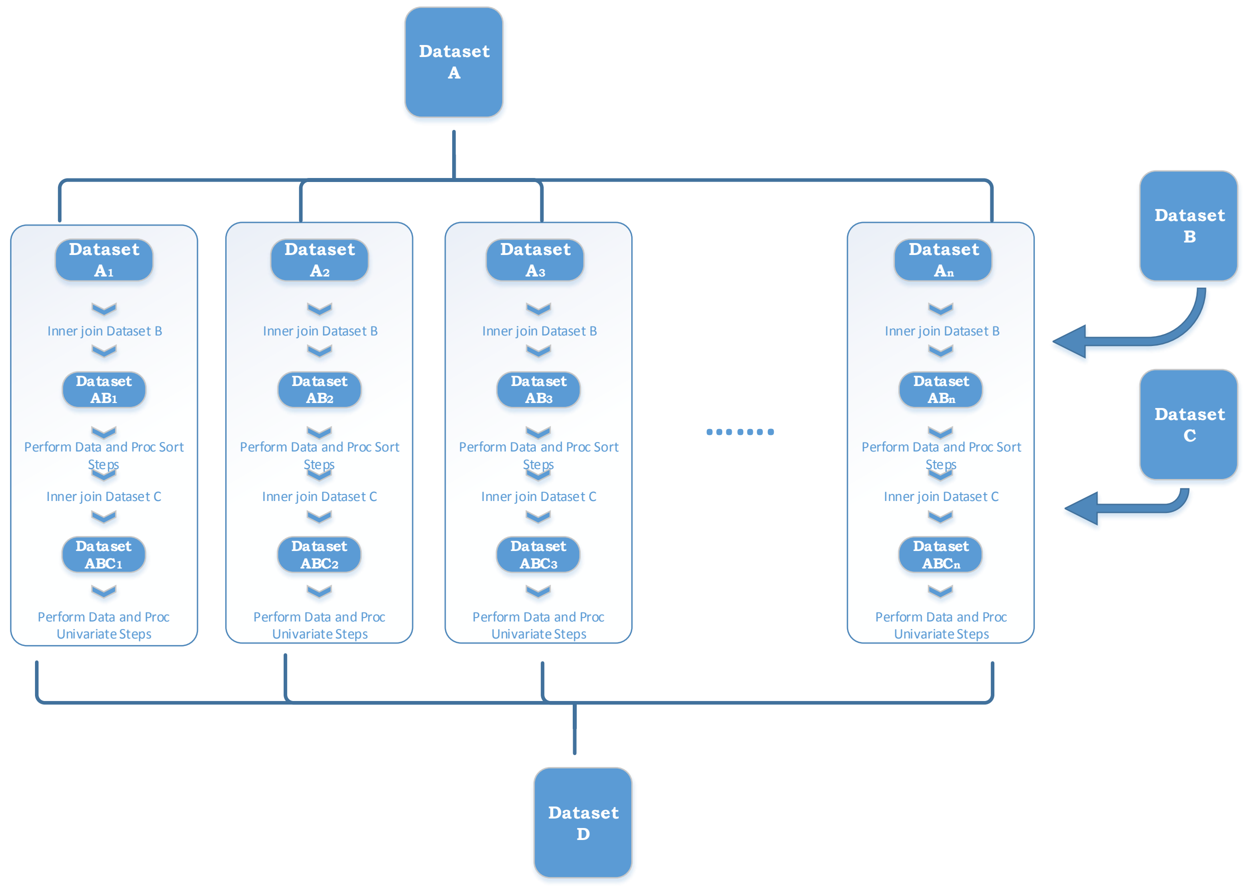


Figure 2. Overview of Data Structure

BOSS-WORKER MODEL

As shown in Figure 1, the Boss thread receives the input dataset (dataset A in our scenario), and it divides the input into n subsets for the n worker threads to process. One approach in dividing a large set of data is to assign each row of records a sequence number, and then divide the total number of records by n to get the number of records (X) in each subset. For each subset, we will count X records in sequential order from dataset A.

The following macro is an example of the Boss thread process:

```
libname ip '<directory where input datasets are saved>';
libname op '<directory where output datasets should be stored>';

%macro split_by_block(block_num);
data
  %do i = 1%to &block_num;
      op.A_&i
  %end;
;

set ip.<dataset A> nobs=nobs;

  SEQ = _N_;
  %do i = 1 %to &block_num;
     %if &i = 1 %then %do;
          if SEQ <= int(nobs/&block_num) then output op.A_&i;
     %end;

     %else %if i = &block_num %then %do;
          else if SEQ > int(nobs/&block_num)*(&block_num-1)
               then output op.A_&block_num;
     %end;

     %else %do;
          else if (int(nobs/&block_num)*(&i-1)) < SEQ and
               SEQ <= (int(nobs/&block_num)*&i)
               then output op.A_&i;
     %end;
  %end;
run;
%mend split_by_block;
```

Definition of terms and variables:

- &Block_Num = total number of subsets wanted
- &i = indication of iteration for the number of subsets
- A_&i = subset output of corresponding iteration

Since each subset of A, A_i, is n times smaller than A, directly joining datasets A and B, theoretically, would take n times longer than joining each A_i with B in parallel. Note: $i = \{1, \ldots, n\}$. In other words, an advantage of the Boss-Worker Model is its capability to reduce elapse time by n times, but the tradeoff is the need for more complicated code.

PEER MODEL

The Peer Model is also known as the Peer-to-Peer system which all threads send and receive data, and each thread contributes processing power. It allows us to decentralize the data to minimize inefficiencies, bottlenecks, and waste of resources [Fenech, Vella]. In our study, all n worker threads work in parallel and perform the four tasks listed in the overview of our data structure. Throughput of this structure scales with the n threads we implemented.

Elapsed time (T) in parallel processing segments is equivalent to the time used by the slowest thread. However, if these tasks were applied to the original dataset A instead of the subsets A_i simultaneously, total elapsed time would be comparable to the sum of times used by each thread (t_i), $T = \sum_{i=1}^{n} t_i$.

PIPELINE MODEL

The Pipeline Model demonstrates parallelism by processing inputs sequentially. It is comparable to an assembly line in a factory, where partially assembled products are passed down from one stage to the next for embellishment. This model is used when dependency on the processor's output is high, and it prevents the use of other parallel processing models. In our program, the Pipeline Model is within each worker thread, where each AB_i subset depends on its corresponding input data A_i, and each ABC_i set is derived from the corresponding AB_i output. Any analysis in a worker thread is dependent on the previous joined output. An advantage of this model is that it allows us to eliminate intermediate writing to the disk and, therefore, reduces disk space requirements [Kumbhakarna].

Below is the macro that performs the peer and pipeline tasks:

```
%macro parallel_join_analyses(block_num);
```

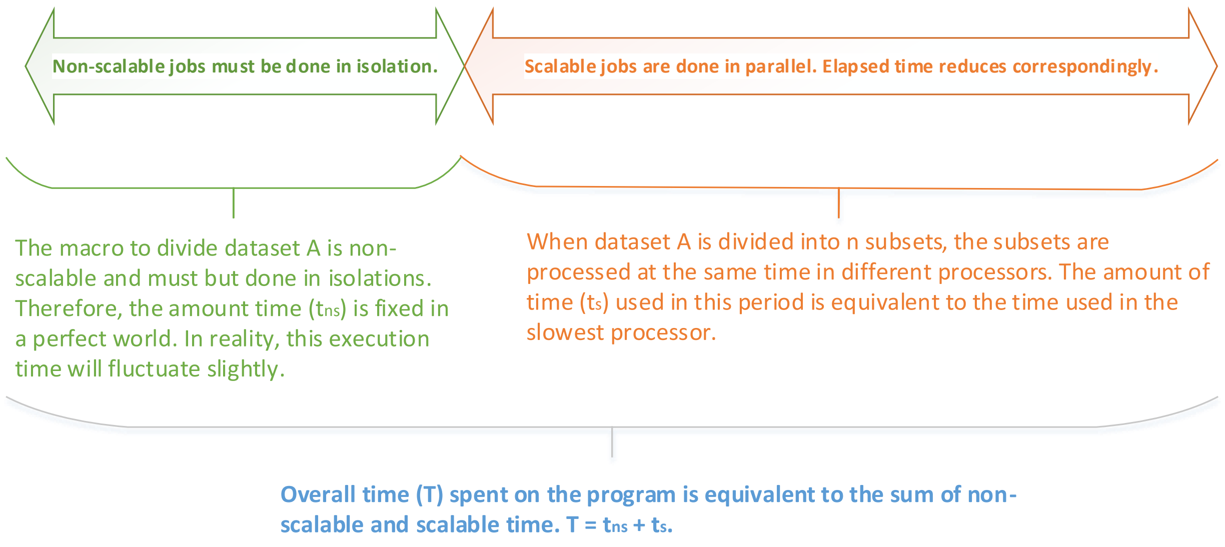
```
options symbolgen mprint fullstimer autosignon=yes sascmd="sas -nonews
-threads";
libname ip '<directory where input datasets are saved>';
libname op '<directory where output datasets should be stored>';

%do k = 1 %to &block_num;

%put k = &k;

  signon task&k. wait=yes;

  %syslput k = &k;

rsubmit process=task&k. wait=no sysrputsync=yes;
options symbolgen mprint fullstimer autosignon=yes sascmd="sas -nonews
-threads";
libname ip '<directory where input datasets are saved>';
libname op '<directory where output datasets should be stored>';

/*Join each subset A_i with dataset B to create subsets AB_i*/
  proc sql _method;
      create table mem.AB_&k. as
      select <variables needed>
      from op.A_&k. A, ip.<dataset B> B
      where <conditions>;
  quit;

/*Other analyses*/
  data op.bene_dates_&k.;
      set op.AB_&k.;

      <analytical statements>;

  run;

/*Join each subset AB_i with dataset C to create subsets ABC_i*/
  proc sql;
  create table op.ABC_sub_&k. as
  select <variables needed>
  from op.bene_dates_&k. A, ip.<dataset C> B
  where <conditions>;
  quit;

/*Other analyses*/
  data op.ABC_&k.;
    set op. ABC_sub_&k.;

      <analytical statements>;

  run;

  proc sort data= op.ABC_&k.;
```

```
    <by statement>;
  run;

  endrsubmit;
  %end;

 waitfor _all_ %do k = 1 %to (&block_num); task&k          %end; ;
               %do k = 1 %to (&block_num); rget task&k;     %end;
               %do k = 1 %to (&block_num); signoff task&k; %end;

/*Merge all ABC_i subsets to final dataset D*/
  data op.<dataset D>;
      set
      %do i = 1 %to (&block_num);
        op.ABC_&i.
      %end;
      ;

      <additional statements>;
   run;
%mend parallel_join_analyses;
```

EVALUATION

While utilizing multiple processors for the above jobs, elapsed execution time is reduced correspondingly, and SAS® was able to leverage additional resources. By increasing the number of CPUs on a SMP machine, processing subsets of the dataset in parallel becomes scalable.

Figure 3 illustrates elapsed time in scalable and non-scalable parts of the program:



Non-scalable jobs must be done in isolation.
Scalable jobs are done in parallel. Elapsed time reduces correspondingly.
The macro to divide dataset A is non-scalable and must but done in isolations. Therefore, the amount time (t_{ns}) is fixed in a perfect world. In reality, this execution time will fluctuate slightly.
When dataset A is divided into n subsets, the subsets are processed at the same time in different processors. The amount of time (t_s) used in this period is equivalent to the time used in the slowest processor.
Overall time (T) spent on the program is equivalent to the sum of non-scalable and scalable time. $T = t_{ns} + t_s$.

Figure 3. Non-Scalable vs. Scalable Jobs

To test and compare performances, dataset A is divided into n subsets, $n = \{1, 3, 5, 10, 15, 17, 20\}$, and then analyzed using the corresponding number of processors, respectively. Overall execution times of these trials are shown in Table 1 and Figure 4, and the percentages of improvement in real time compared to a serial operation are shown in Table 2. When no subsetting and parallel processing are applied to the dataset ($n = 1$), the program runs in a sequential manner without scalability, and, therefore, execution time is considerably high. However, execution time reduces significantly when the number of processors increases to three and five, where calculated improvement reaches 73%. A plateau occurs as the number of subsets exceeds ten, and total execution time starts to increase when the number of subsets reaches twenty.

Efficiency is not solely dependent on the total number of processors but also other factors like input (read) and output (write) responses, memory, and other concurring jobs. In our example, the tasks in each processor are required to join with dataset B, and system input responses can reach a speed limit as an increasing number of processors attempt to access dataset B. When the threshold is reached, efficiency will no longer improve; in contrast, it could slow down the process and weaken performances.

Number of Subsets	Real Time	User CPU Time	System CPU Time
1	58:22.99	45:32.85	17:08.15
3	21:48.42	2:37.19	1:41.59
5	15:44.99	2:39.61	1:37.52
10	14:22.76	2:45.46	1:38.14
15	14:27.27	2:59.11	1:44.07
17	14:16.37	2:57.17	1:38.11
20	15:39.25	3:05.80	1:39.84

Table 1. Overall Process Time

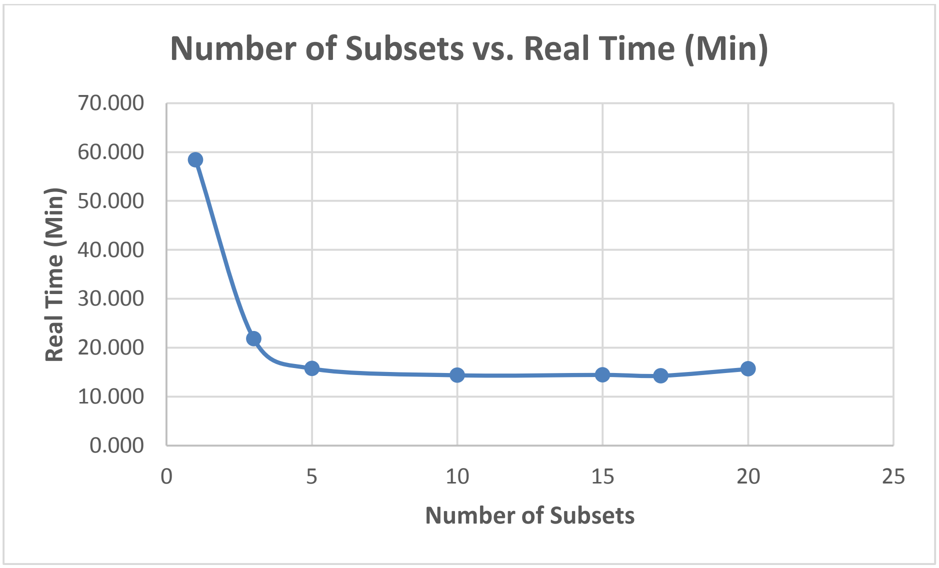


Figure 4. Real Time Comparison

Number of Subsets	% Improvement
1	Reference Baseline
3	63%
5	73%
10	75%
15	75%
17	76%
20	73%

Table 2. Percentage Improvements in Parallel Processing Versus Single Processing

LIMITATIONS

Various internal and external limitations prevent our program from performing at its full capacity. As a dataset is divided into more subsets, the improvement in efficiency plateaus and the time used increases. The most common limitation is shared resources such as disk space. As more users share the limited resources within a system, the jobs that each user submits would slow down correspondingly.

CONCLUSION

In conclusion, parallel processing is a powerful technique to increase efficiency by increasing throughput, reducing execution time, and taking advantage of SMP resources. However, the amount of memory required can be greater due to the increasing amount of data and replicated processes. Despite increase in efficiency, it is important to monitor overhead cost associated with parallelism setups. Performance can be weaken for small jobs because developing complicated code can comprise a significant portion of execution time.

REFERENCES

Shamlin, David. 2014. “Threads Unraveled: A Parallel Processing Primer.” The Twenty-Ninth Annual SAS® Users Group International Conference. May 12, 2004. Available at http://www2.sas.com/proceedings/sugi29/217-29.pdf.

Kumbhakarna, Viraj. 2013. “Parallel processing techniques for performance improvement for SAS® processes: Part II”. The Twenty-First Annual Southeast SAS® Users Group Conference. October 20, 2013. Available at http://analytics.ncsu.edu/sesug/2013/PA-08.pdf.

Fenech, Karl. Vella, Kevin. 2014. “Parallel Processing over a Peer-to-Peer Network”. WICT08. University of Malta. February 6, 2014. Available at https://www.um.edu.mt/__data/assets/pdf_file/0003/51744/wict08_submission_11.pdf.

Powers, David J. 2007 “Priming the Pump: Load Balancing Iterative Algorithms”. In Khaled Elleithy (Ed.), Advances in Innovations in Systems, Computing Sciences and Software Engineering (pp. 77-81). The Netherlands: Springer Netherlands.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Shuhua Liang
Kaiser Permanente Southern California
Shuhua.Liang@kp.org

Fagen Xie
Kaiser Permanente Southern California
Fagen.Xie@kp.org