Paper BI07-2013

Beating Gridlock: Parallel Programming with SAS® Grid Computing and SAS/CONNECT®

Jack Fuller, Experis, Portage, MI

ABSTRACT

Long running SAS jobs often include sets of independent subtasks that can be split up and distributed across a SAS Grid. When these subtasks are then run in parallel the total run time will usually increase; however, total elapsed time can often be made to decrease.

This paper will present an introduction to parallel processing using SAS Grid and SAS Connect with an emphasis on the following: when to use parallel processing, how to use parallel processing and points to consider when implementing parallel processing.

INTRODUCTION

I was recently asked to assist a statistician with decreasing the time needed to run a SAS program which performed a series of simulations. Fortunately, we had access to an installed SAS Grid and SAS/Connect which allowed us to decrease the time needed to run the analysis from approximately 5 hours using PC SAS to 30 minutes using a SAS Grid with parallelized code. This paper will share some of the lessons I learned: when to use parallel processing, how to use parallel processing and points to consider when implementing parallel processing.

WHEN TO USE PARALLEL PROCESSING

Long running SAS programs which contain multiple independent subtasks[1] can be spawned into multiple grid jobs which can then be executed in parallel. This ability to split a job into independently running subtasks is called *concurrency*.

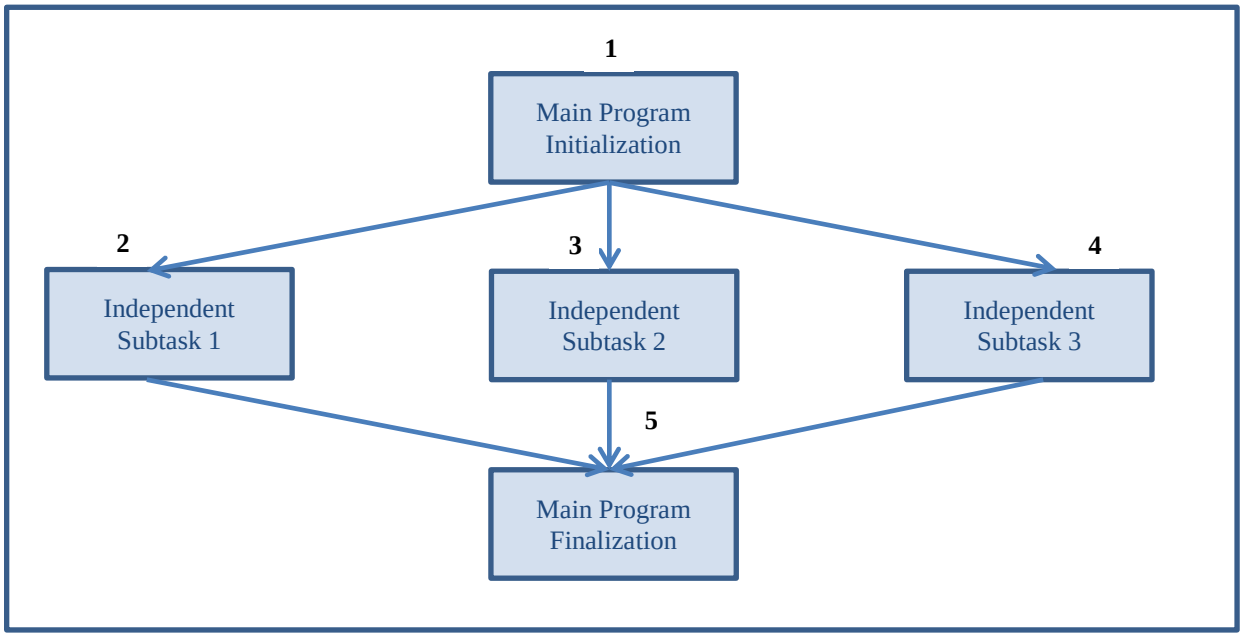

Figure 1
Simplified Diagram of Independent Subtasks Running in Parallel

[1] *Grid Computing in SAS 9.3, Second Edition* refers to ***tasks*** (i.e. servers) and ***subtasks*** (i.e. clients). *SAS/CONNECT 9.3 User's Guide* refers to **servers** (i.e. tasks) and **clients** (i.e. subtasks).

FIGURE 1 shows a simplified diagram of a parallelized program where:

1. The **Main Program** performs any initialization
2. The **Main Program** spawns **Independent Subtask 1**
3. The **Main Program** spawns **Independent Subtask 2**
4. The **Main Program** spawns **Independent Subtask 3**
5. The **Main Program** waits for all of the independent subtasks to complete execution and then performs any finalization

TYPES OF CONCURRENCY

It can be difficult when attempting to parallelize a program to determine how to decompose as much of the program as possible into concurrent subtasks which can then be run in parallel. Concurrent subtasks generally fall into one of the following forms:

- *Data parallelism* occurs when each subtask works on a different piece of the data. For example, each subtask could load different observations from a data set into a database:
    - *Subtask 1* loads observations 1-10,000
    - *Subtask 2* loads observations 10,001-20,000
    - …
- *Task parallelism* occurs when each subtask performs a different function. For example, each subtask could calculate different analyses:
    - *Subtask 1* processes demographics data
    - *Subtask 2* processes vital signs data
    - …

If a program does not have any concurrency then it cannot be parallelized.

AMDAHL'S LAW

The optimal increase in speed gained from converting a serial process to a parallel process is given by *Amdahl's Law*. Amdahl's Law assumes an ideal situation where there is no overhead involved with creating or managing the different processes. This implies that Amdahl's Law will overstate any potential gains. One of the major insights that this formula provides is that as the number of processors (P) increases, the increase in speed is constrained by the fraction of work that is not parallelized (α):

$$\lim_{P \to \infty} \frac{1}{\frac{1-\alpha}{P} + \alpha} = \frac{1}{\alpha}$$

Figure 2
***Amdahl's Law* as the Number of Processors Approaches Infinity**

This means that at some point there is a law of diminishing returns and adding more subtasks (i.e. processors) will not increase the speed of the program. In fact, the program may actually run more slowly in practice since there is a cost (or friction) involved with spawning and managing subtasks.

If a program contains a large section of code which cannot be parallelized then it may not be a candidate for parallelization.

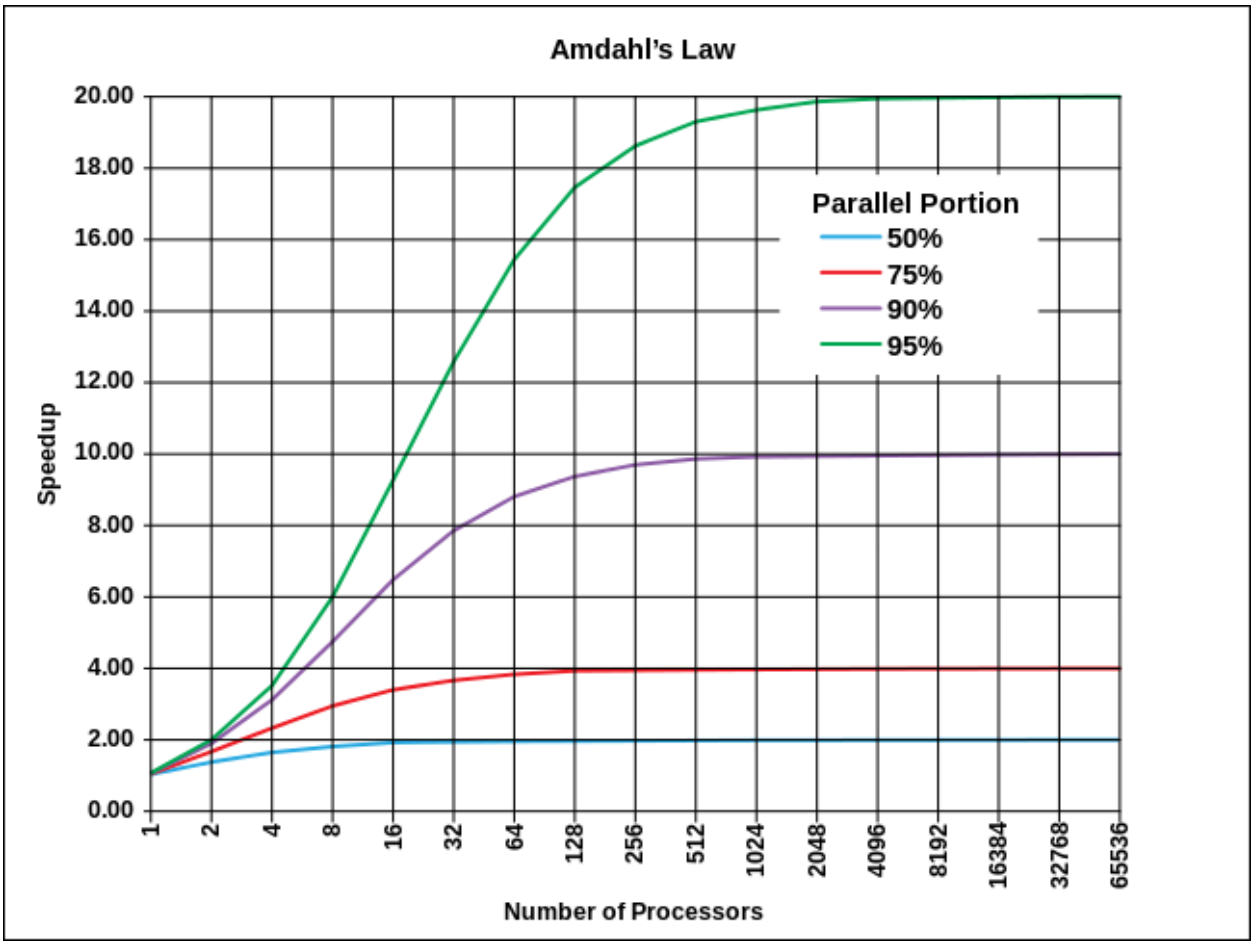

Figure 3
Graphical Representation of Amdahl's Law (File:AmdahlsLaw.svg)

HOW TO USE PARALLEL PROCESSING

Here is an example of code which is used to generate test data that exhibits concurrency and can be parallelized. One option would be to decompose the program using data parallelization (e.g. 10 subtasks which each process 100 patients). Another option would be to decompose the program using task parallelization (e.g. 1 subtask for each data domain).

For this example, we will decompose the following program into subtasks by data domain (demographics, labs, vital signs).

```
data test;
   do pt=1 to 10000;
      if uniform(0) > .45 then do;
         gender = 'M';
         /* Generate test data */
      end;
      else do;
         gender = 'F';
         /* Generate test data */
      end;
      output;
   end;
run;
```

PROGRAM INITIALIZATION

The first step is to perform the program initialization to set up the environment for parallel processing and to create the initial test data set named PT which will contain the variables PT and GENDER.

```
%let numPt = 1000;

%*-------------------------------------------------------------------
 | The SEED is defined once at the beginning of the main program     |
 |    and then passed to each subtask as a macro variable            |
 *-------------------------------------------------------------------;

%let seed = 0;

%*-------------------------------------------------------------------
 | Enable the grid service for this program                          |
 *-------------------------------------------------------------------;

%let rc=%sysfunc(grdsvc_enable(_all_, resource=SASApp));

%*-------------------------------------------------------------------
 | Specify AUTOSIGNON so our program can spawn subtasks without      |
 | user interaction                                                  |
 *-------------------------------------------------------------------;

options autosignon;

%*-------------------------------------------------------------------
 | Point a LIBREF to the main programs work directory so that the    |
 | different subtasks can access both their own WORK directory and   |
 | the main programs WORK directory                                  |
 *-------------------------------------------------------------------;

libname shared "%sysfunc(pathname(work))";

%*-------------------------------------------------------------------
 | Create the initial test data                                      |
 *-------------------------------------------------------------------;

data pt;
   do pt=1 to &numPt;
      gender = ifc(uniform(&seed) > .45, 'M', 'F');
      output;
   end;
run;
```

DEMOGRAPHICS SUBTASK

The *demographics* subtask will create the demographics data. It will use the PT data set created in the initialization phase as input and then create the data set TASK1 with the variables PT, HT and WT as output. Both the input data set and the output data set will reside in the main program's WORK library.

```
%*-------------------------------------------------------------------
 | Create a macro variable named SEED in the TASK1 subtask and       |
 | initialize it to the value of SEED from the main program          |
 |                                                                   |
 | NOTE: Use %SYSRPUT in a subtask to update a macro variable        |
 | in the main program.                                              |
 *-------------------------------------------------------------------;

%syslput seed=&seed / remote=task1;
```

```
%*------------------------------------------------------------------
 | Bracket the code which will run in the subtask with RSUBMIT and   |
 | and ENDRSUBMIT:                                                   |
 | + Name the subtask (TASK1)                                        |
 | + Instruct the subtask to run asynchronously (WAIT=NO)            |
 | + Provide access to the library SHARED (INHERITLIB=(SHARED))      |
 |                                                                   |
 | NOTE: Task names cannot exceed 8 characters in length             |
 *------------------------------------------------------------------;

rsubmit task1 wait=no inheritlib=(shared);
   data shared.task1(keep=pt ht wt);
      set shared.pt;
      if gender = 'M' then do;
         /* Compute statistics */
      end;
      else do;
         /* Compute statistics */
      end;
   run;
endrsubmit;
```

LABS SUBTASK

The *labs* subtask is set up similarly to the demographics subtask. It will create a data set named TASK2 in the main program's WORK library as output.

```
%syslput seed=&seed / remote=task2;
rsubmit task2 wait=no inheritlib=(shared);
   data shared.task2(keep=pt wbc rbc);
      set shared.pt;
      /* Compute statistics */
   run;
endrsubmit;
```

VITAL SIGNS SUBTASK

The *vital signs* subtask is set up similarly to the demographics subtask. It will create a data set named TASK3 in the main program's WORK library as output.

```
%syslput seed=&seed / remote=task3;
rsubmit task3 wait=no inheritlib=(shared);
   data shared.task3(keep=pt syst_bp diast_bp);
      set shared.pt;
      if gender = 'M' then do;
         /* Compute statistics */
      end;
      else do;
         /* Compute statistics */
      end;
   run;
endrsubmit;
```

TASK FINALIZATION

The main program's finalization will bring together all of the data and perform any cleanup.

```
%*------------------------------------------------------------------
 | Wait for all of the subtasks to complete execution               |
 *------------------------------------------------------------------;
```

```
waitfor _all_ task1 task2 task3;

%*-------------------------------------------------------------------
 | Bring together the data created by the subtasks and write it      |
 | to disk                                                           |
 *-------------------------------------------------------------------;

libname final "$HOME";

data final.myData;
   merge task:;
      by pt;
run;

%*-------------------------------------------------------------------
 | Close all of the subtasks                                         |
 *-------------------------------------------------------------------;

signoff _all_;

%*-------------------------------------------------------------------
 | Clean up                                                          |
 *-------------------------------------------------------------------;

proc datasets library=work nolist;
   delete task:;
quit
```

POINTS TO CONSIDER WHEN IMPLEMENTING PARALLEL PROCESSING

HOW MANY SUBTASKS SHOULD I HAVE?

I have seen applications try spawning one thousand subtasks. According to Amdahl's Law, this is almost never a good idea. In addition to the constraints implied by Amdahl's Law, the number of grid nodes available to your SAS Grid will also be a limiting factor. I like to begin by benchmarking my applications with 5, 10 and 15 subtasks and then adjusting the number of subtasks to arrive at an optimal number. If the code is subsequently refactored to change the amount of parallelization, then the question of how many subtasks should be used will need to be revisited.

HOW MUCH WORK SHOULD OCCUR IN EACH SUBTASK?

I have seen applications which created very small tasks designed to run in under 10 seconds. This is rarely a good idea because in the real world there is a cost associated with spawning and managing subtasks. Additionally, even though AUTOSIGNON is specified, there is a login process which occurs when using RSUBMIT. This login does not occur when calling the SAS Grid directly. In my experience, it can take anywhere from 5 to 10 seconds. This means that the amount of work in performed in each subtask should be long enough such that the time spent during AUTOSIGNON is negligible.

SHOULD MY LIBRARIES BE SHARED OR SHOULD EACH SUBTASK CREATE ITS OWN LIBRARIES?

I instinctively like to use the INHERITLIB option to share libraries because it keeps the library definition in one place as opposed to maintaining separate copies of the library definition in every subtask. However, there are instances when having every subtask access data using the same LIBREF can create a bottleneck as the subtasks queue up waiting for the data. In particular, this can happen when sharing a LIBREF which accesses a database. In this case, it is better to define a separate LIBREF inside of each subtask so that each subtask can have its own dedicated pipeline.

CONCLUSION

If you have a long running program and access to a SAS Grid, you may able to leverage the power of parallel programming by spawning concurrent tasks to other programs which can be run simultaneously. The most important

thing is to put as much of the work as possible in the spawned programs. Once the program is running and everything is working correctly, the last thing to be done is to benchmark your program and begin the fine tuning.

Good luck!

REFERENCES

- "File:AmdahlsLaw.svg - Wikipedia, the free encyclopedia." *Wikipedia, The Free Encyclopedia*. N.p., n.d. Web. 11 Aug. 2013. <http://en.wikipedia.org/wiki/File:AmdahlsLaw.svg>

- *Grid Computing in SAS 9.3, Second Edition*. 2nd ed. Cary, N.C.: SAS Institute Inc., 2012. Print.

- "Parallel computing." *Wikipedia, The Free Encyclopedia*. N.p., n.d. Web. 11 Aug. 2013. <http://en.wikipedia.org/wiki/Parallel_computing>

- "Parallel programming model." *Wikipedia, The Free Encyclopedia*. N.p., n.d. Web. 11 Aug. 2013. <http://en.wikipedia.org/wiki/Parallel_programming_model>

- *SAS/CONNECT 9.3 User's Guide*. Cary, N.C.: SAS Institute, 2011. Print.

CONTACT INFORMATION

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

Name: Jack Fuller
Enterprise: Experis Manpower Group
Address: 5220 Lovers Lane
City, State ZIP: Portage, MI 49002
Work Phone: 269.553.5126
E-mail: jack.fuller@experis.com
Web: www.experis.com