PhUSE 2016

Paper CS04

Reducing Build Time of Integrated Clinical Databases with SAS® Grid and SAS/OR®

Eric C. Brinsfield, Meridian Analytics, Virginia Beach, VA, USA
and d-Wise Technologies, Inc., Morrisville, NC, USA

ABSTRACT

Clinical trial organizations frequently build "integrated databases" that incorporate data from multiple studies by running build jobs that call other programs. If faster execution time is needed, build jobs can run programs in parallel (asynchronously) using solutions such as SAS Grid Manager®. Depending on the program number and the dependency complexity (e.g. Program 10 depends on programs 1 and 3), determining the schedule of programs that minimize execution time while honoring the dependency constraints can be challenging.

We developed methods to: 1) construct program dependency lists, 2) predict shortest parallel job times, 3) suggest optimal distribution of programs across nodes, 4) find required number of grid nodes for shortest job time, and 5) generate a new grid-enabled driver program. Applying these techniques to long running jobs, parallel processing consistently reduced run times to 25% of sequential. These techniques apply to any situations where you need to run a large set of individual programs that take a long time to complete when run sequentially.

INTRODUCTION

The Problem:
Sometimes you are faced with a problem that reminds you of something you learned at the university. I was recently presented with a challenge that not only took me back to my operations research education but also back to the often under-utilized capabilities of SAS/OR. Many project managers will recognize this story as a simple scheduling problem.

While building integrated safety databases, usually derived from CDISC[1] Study Data Tabulation Model (SDTM) or Analysis Data Model (ADaM), programmers need to combine data from multiple studies. If you are lucky, the studies are all similar and the merging is simple. There are cases, however, when the data is voluminous and/or the integration process requires significant rearrangement of data or calculation of additional values. In these situations, the run-time for a list of integration programs can be long (10-20 hours).

As a challenge, we were asked to reduce the execution time for a sequence of programs for multiple integration projects using parallel processing and specifically using the SAS Grid. Obviously, if you had unlimited number of SAS Grid nodes and each program was completely independent of the others, the shortest execution time would be the run-time of the longest running program if you submitted all programs at the same time in parallel. Unfortunately, you usually cannot get access to all the nodes you need at once and the programs are not independent.

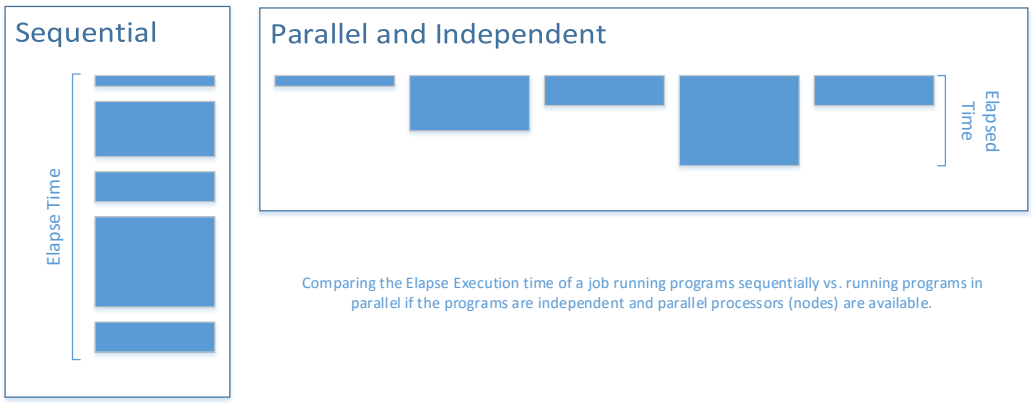


Figure 1: Simple illustration of sequential vs. parallel processing

By “dependency”, we are referring to the condition where one program depends on a data set(s) created by another program. As an objective, we want to find the shortest overall execution time to run all programs while still honoring any dependencies and using only the necessary and sufficient number of SAS Grid nodes.

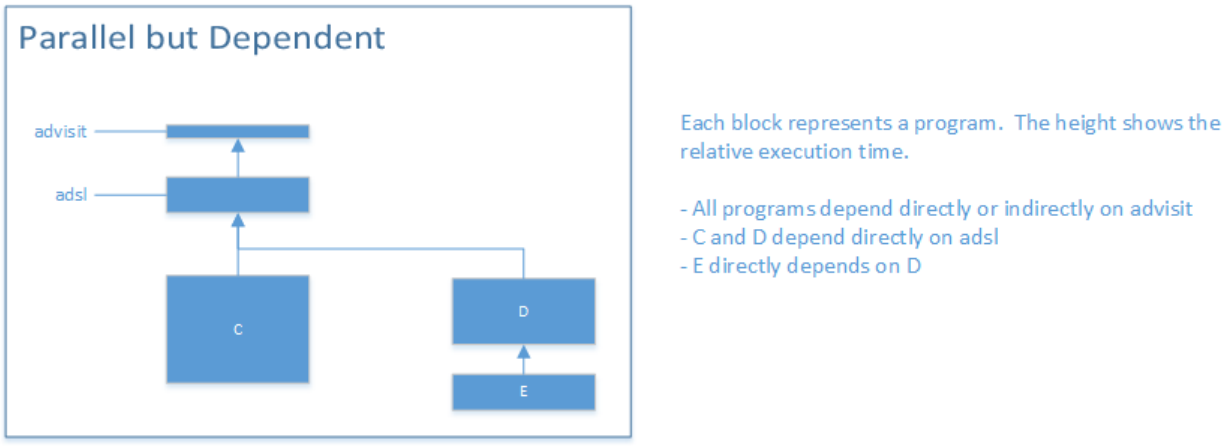


Figure 2 Simple illustration of parallel but dependent processing

Specific Use Case:
For our particular challenge, the programmers were running multiple SAS programs sequentially by “%including” them from a single “driver” program. In some cases, the dependencies were known and in others they were not. The users submit the driver programs to batch SAS or from SAS Display Manager. The submission method was not an issue.

The example diagram in Figure 3 shows a more complicated dependency network with varying program durations. If this set of programs runs in sequence, the total execution time is extremely long. If we can determine the program dependencies and run some programs in parallel, we can shorten the overall run-time of the job.

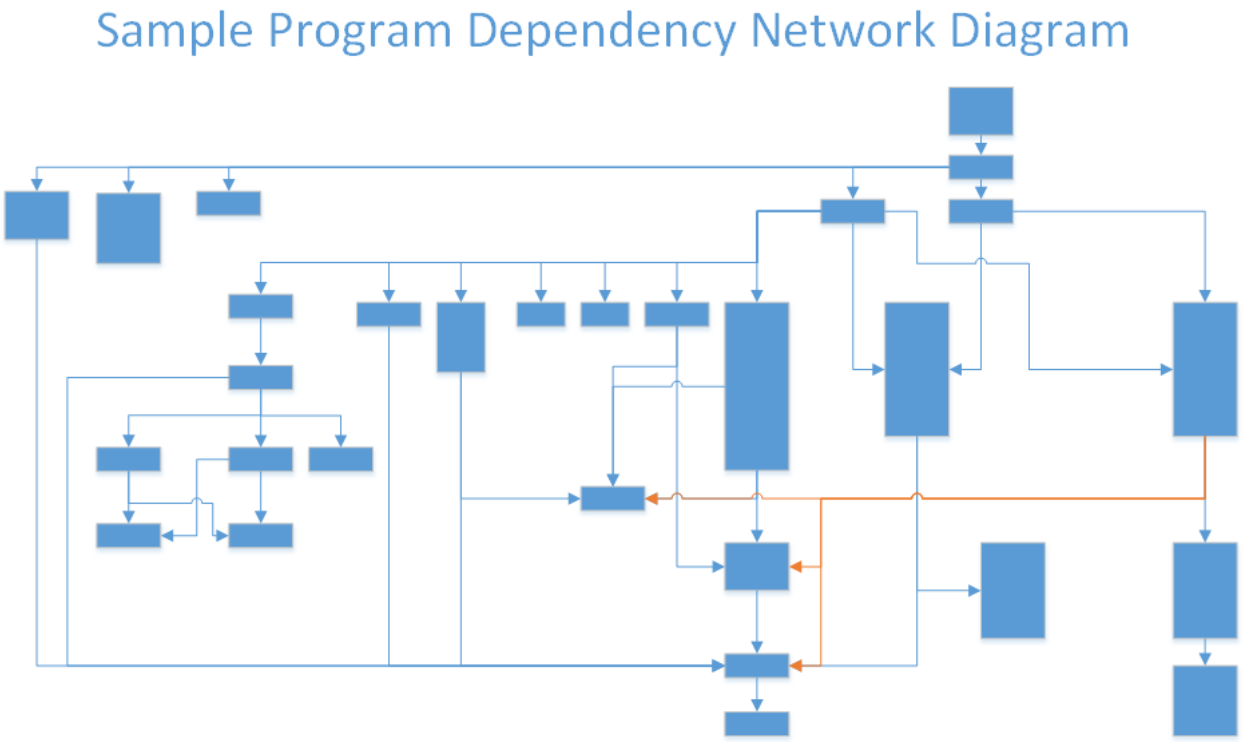


Figure 3 Sample of complex dependency flow

Variations:
The techniques described in this paper apply to any scheduling of jobs or programs whether using SAS Grid, LSF batch jobs, or some other parallel processing of programs.

Value:
The effort required to analyze and predict the optimal distribution of programs across a parallel grid must be weighed against the value of reducing the overall time of execution. In other words, if the programs run fast, you do not need this analysis. If you run the sequence only once a year, you probably do not need this. If you are running a full sequence of jobs on a regular basis or you are submitting batch jobs for a nightly run and they must finish before everyone starts work in the morning, you will find this approach worthwhile.

APPROACH

In order to tackle this problem, we took the following approach:

Analyze Sequential Timings:
Run all the programs as usual in sequence and save all of the SAS logs. In our case, the original set ran on a non-SAS Grid system, so we also ran the sequence again on the SAS Grid system to provide a true benchmark on a comparable hardware and software platform.

For analysis, we parsed the SAS logs in order to collect timing information, I/O count, and the names of data sets that were input and output to each step. See the Methods section below for a discussion of parsing the logs using the downloadable LOGPARSE macro from SAS.

Analyze Dependencies:
Using the list of input data and output data, subset the list to those data sets that are coming from permanent libraries or being written to permanent libraries. In particular, if you are building an integrated ADaM database, you want to find all the final ADaM data sets that each program output as well as the final ADaM data sets that each program input.

By connecting programs that output an ADaM data set to those that input an ADaM data set, you can create a reliable dependency list. See Methods section below.

Predict the Shortest Path:
Now that you know the normal duration (execution time) for each program individually and which programs depend on the completion of other programs, you can use the well-known Critical Path Method[2,3,6] to calculate the shortest path or the shortest job run-time. Basically, you want to find the longest running sequence of programs based on dependencies (the critical path). Next you want to schedule all other programs within that timeframe.

We used SAS's PROC CPM [4] to calculate the shortest elapsed time that we might expect with optimal distribution of programs across the SAS Grid while still honoring the dependencies. We could also determine the maximum number of SAS Grid nodes that we would need to achieve the shortest time.

All of this could be determined without running any build programs other than the original sequential run.

Optimize the Distribution of Programs across the SAS Grid:
Using the scheduling results from PROC CPM along with the graphical representation of the processing using the Gantt procedure, you can manually modify your driver program to submit individual or groups of programs to different SAS Grid nodes using remote submit.

Optionally Generate the New Driver Program Automatically:
We took one additional step in automation and attempted to generate new driver programs using the output from PROC CPM. This worked fine with one exception. PROC CPM cannot account for the fact that a program runs when submitted and not at a specified time. CPM schedules the programs so that they do not start until all predecessors complete, but it does not know that once a program is submitted to a node (queue), it will start running as soon as any other jobs on that node complete; thereby not ensuring the predecessors running on other nodes are done.

To solve this problem, we had to make some basic assumptions and generate WAITFOR[4] points in the generated driver program. With the WAITFOR statement, you can hold further processing in the SAS program until specified grid sessions has completed the last RSUBMIT.

Analyze New Results (refine as needed):
Once the new driver program runs, you can reanalyze the logs and timings to determine if you achieved the predicted shortest run-time. At this point, you can also gain some saving by looking at the programs in the critical path and decide if it is worth refining the logic within the program to run faster by using more efficient techniques. You can also consider submitting portions of an individual program to the grid for parallel processing.

For example, we found a few programs that were running the same macro logic 4 times on different subsets of a very large data set. By submitting those to run in parallel, we chopped the program execution time by more than half and thereby also reduced the critical path time.

Note: The execution time and CPU times for each individual program will remain statistically constant whether running in sequence or in parallel. The time savings comes from running programs in parallel and shortening the job time. SAS Grid is not changing the CPU time or processing required for any given program.

METHODS

Analyzing SAS Logs:
To parse the SAS logs, we used the macro provided by SAS called LOGPARSE, which can be downloaded from SAS at the following sample note: ***Sample 34301: Parse SAS® Logs to Extract Performance and Timing Information.***

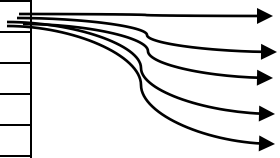
Before you run the SAS programs, set the SAS options FULLSTIMER NOTES SOURCE SOURC2 MPRINT. You want as much information to be printed in the logs as possible. FULLSTIMER will give you more complete timing information. LOGPARSE can be run on one program at a time or can parse all members in a directory. In either case, the macro stored statistics in a SAS data set that we post-processed and further analyzed.

We took the output of the parsing process and further parsed and organized those results. First we summarized the total times (execution time and CPU times) per program along with other statistics of interest. These are always saved as part of the benchmark data for each run.

Additionally, the output data set contains variables DS_IN and DS_OUT which contain a list of input data sets and output data sets respectively for each step. The list also includes the number of observations and variables, but those are already captured, so we ignore the numeric values. Using this post-processing, we created data sets IN_DATA and OUT_DATA. IN_DATA contains one row per program per input ADaM data set name and OUT_DATA contains one row per program per output ADaM data set. We use these data sets in the next step.

Determining Dependencies
We created a program that reads the OUT_DATA and IN_DATA data sets along with a master list of program names. Each program name (pgmname) was also assigned a sequence order (seq_order) based on prior sequential run order (not shown below). The master list was joined with each data set and the two data sets were joined by data set name (output connected to input).

DS_OUT	
PROGRAM	OUTPUT DATA SET
Pgm1	DS1
Pgm2	DS2
Pgm2	DS2B
Pgm3	DS3
Pgm4	DS4
Pgm5	DS5
And so on…	



DS_IN	
INPUT DATA SET	PROGRAM
DS1	Pgm2
DS1	Pgm3
DS1	Pgm4
DS1	Pgm5
DS1	Many others
DS2	Pgm3
DS2B	Pgm12
And so on…	

Figure 4 Illustration of joining programs by output data set and subsequent input data set

Preparing Data as Input to PROC CPM
This connected data set is transposed to create one row per program name with variables called DEP1—DEP*n* containing the name of each program that depends on that program. This is the structure that PROC CPM expects the input SAS data set to use in order to define the precedence relationship (or successor relationship).

PROGRAM	DEP1	DEP2	DEP3	DEP4	DEP5	Etc.
Pgm1	Pgm2	Pgm3	Pgm4	Pgm5	Others	
Pgm2	Pgm3	Pgm12				

Table 1 Sample of joined and transposed data set

We also created a similar "predecessor" list just because it helped to visualize the precedence relationships better, but it was not required for CPM. Finally, we needed to merge in the total execution times per program that were collected from the LOGPARSE process. That final data set was called PGMSUCCESSORS and was input to PROC CPM.

To control how Grid Nodes are assigned, we created a data set called NODES_AVAIL which becomes the RESOURCEIN data set required by CPM.

OBSTYPE	GRIDNAME	PERIOD	GRID1	GRID2	GRID3	GRID4	GRID5	GRID6
restype			1	1	1	1	1	1
resrcdur			0	0	0	0	0	0
altrate	grid1		1	1	1	1	1	1
altprty	grid1		1	2	3	4	5	6
reslevel		0:00:00	1	1	1	1	1	1

Table 2 RESOURCEIN control data set

The RESOURCEIN data set (NODES_AVAIL) instructs CPM to try to use grid1 for all work, but to take additional grid nodes if necessary to shorten the execution time. The OBSTYPE passes specific instructions to CPM, where

- restype = 1 indicates that all resources (grid nodes) are replenishable (reusable). Once a program is done, the grid node is available to be reused.
- resrcdur = 0 indicates that the resource uses a fixed duration. In other words, the specific grid node does not impact the duration of the program. This may not always be the case for all systems, but it was in our case.
- altrate = 1 indicates that if grid1 is not available, the Grid Manager select only 1 alternate node from the list.
- altprty = the priority of alternate selection. If grid1 is not available, pick grid2 before 3, 4 ,5 or 6.
- Reslevel = the number of resources available per grid node starting at time 0:00:00.

Running PROC CPM:
After some additional preparation of data, we call PROC CPM with the following input data sets and options:

```
proc cpm data=pgmsuccessors
       out=firstrun date='00:00:00't interval=second
       resin=nodes_avail rsched=node_schedule resout=nodes_used;
   activity pgmname;
   duration dur_seconds;
   id seq_order tot_realtime;
   successor dep1-dep&dim;
   resource grid1-grid&nodes_available  /
       obstype=obstype period=period resid=gridname
       esprofile rcprofile routinterval=MINUTE t_float;

            ←   project project /orderall addwbs sepcrit;  /* for second run */
run;
```

where

- OUT = FIRSTRUN, the output data set name

- DATE = 0 starts the timing of the process at time zero
- INTERVAL = “second” instructs CPM to interpret DURATION units as seconds
- RESIN= the resource data set called NODES_AVAIL
- RSCHED= an output data set that will contain schedules for each resource
- RESOUT = an output data set containing resource usage profiles
- ACTIVITY = the variable in PGMSUCCESSORS that is the target of the scheduling.
- DURATION = the variable in PGMSUCCESSORS that contains the length of time per program (pgmname)
- ID = a list of variables to carry along to the output
- SUCCESSOR = a list of the variables that contain the dependents.
- RESOURCE = the names of the resource variables. In this case the SAS Grid nodes.
- TFLOAT = a flag to control inclusion of float time in FIRSTRUN data set. It is useful in showing any time gaps between unconstrained and scheduled start times. This becomes a rather useful variable if a second round of refinement is needed.

The RESOURCE options OBSTYPE, PERIOD, and RESID simply name the variables in the NODES_AVAIL data set

Optional Use of PROJECT Statement:

Note at the end we show an inserted PROJECT statement. During the first run, we let CPM do its best regardless of any grouping or clustering you might see. For example, you typically need to finish the visit and adsl data sets first, then you can release another batch. In some cases, the AE-related data was all be grouped together or dependent, so we create a project data set to artificially place breakpoints and force certain programs into specific grid nodes. This worked well and we used it later to generate WAITFOR statements in the driver programs.

Viewing Results and Graphs:

In the Table 3, you will notice the “project” column. This labels a project or group. The blank project represents the total execution time or total project with all subprojects. Otherwise where the gridnum is missing, you have a subproject. You can see that subproject idb01 only used 2 grid nodes and ran one program per node. After they completed, subproject idb2 started and several more nodes are launched. CPM optimizes (assigns programs to Grid nodes) within the project or subproject.

CPM results showing programs, elapsed time, and start options
Based on project assignments and some targeted grid nodes

WBS_CODE	project	pgmname	gridnum	tot_realtime	dur_seconds	S_START	S_FINISH	T_FLOAT
0		idb	.	.	.	0:00:00.00	0:48:24.70	0.00
0.0	idb	idb01	.	.	.	0:00:00.00	0:00:19.16	0.00
0.0.0	idb01		1	0:00:19	19.17	0:00:00.00	0:00:19.16	0.00
0.0.1	idb01		2	0:00:05	4.53	0:00:00.00	0:00:04.52	14.64
0.1	idb		.	.	.	0:00:19.17	0:04:18.23	0.00
0.1.10	idb02		1	0:01:57	116.67	0:00:19.17	0:02:15.83	24.23
0.1.24	idb02		2	0:01:51	111.36	0:00:19.17	0:02:10.52	29.54
0.1.03	idb02		3	0:01:06	65.57	0:00:19.17	0:01:24.73	0.00
0.1.16	idb02		4	0:01:38	98.38	0:00:19.17	0:01:57.54	42.52
0.1.04	idb02		5	0:01:25	84.94	0:00:19.17	0:01:44.10	55.96
0.1.18	idb02		6	0:01:24	83.69	0:00:19.17	0:01:42.85	57.21
0.1.11	idb02		3	0:01:23	82.56	0:01:24.74	0:02:47.29	58.34
0.1.05	idb02		6	0:00:32	32.32	0:01:42.86	0:02:15.17	108.58
0.1.12	idb02		5	0:00:30	30.27	0:01:44.11	0:02:14.37	110.63
0.1.13	idb02		4	0:00:27	27.47	0:01:57.55	0:02:25.01	113.43
0.1.21	idb02		2	0:00:21	21.48	0:02:10.53	0:02:32.00	119.42
0.1.17	idb02		5	0:00:16	16.29	0:02:14.38	0:02:30.66	124.61
0.1.26	idb02		6	0:00:15	14.82	0:02:15.18	0:02:29.99	126.08
0.1.07	idb02		1	0:00:14	13.59	0:02:15.84	0:02:29.42	127.31

Table 3 Sample output data set from PROC CPM

The OUT= data set is useful for drilling into details, but Gantt charts seem to be the most informative view of the results. You can create the Gantts in multiple ways with or without the project information.

In the two Gantt charts in Figure 5, the one on the left is unconstrained by resources (grid nodes) but does honor dependencies. This should represent the fastest that you could complete the execution of all programs. In reality, it does not make much sense to submit each small program to a separate grid node, especially because you do encounter an initialization overhead or a queue wait-time with each new node.

The Gantt chart on the right includes project/subprojects and constrains the resources so that some programs will be stacked within the same grid node. With the priority set in the RESOURCEIN data set, CPM will only use what it needs. Also, in the right Gantt chart, the project and subproject lines are shown.

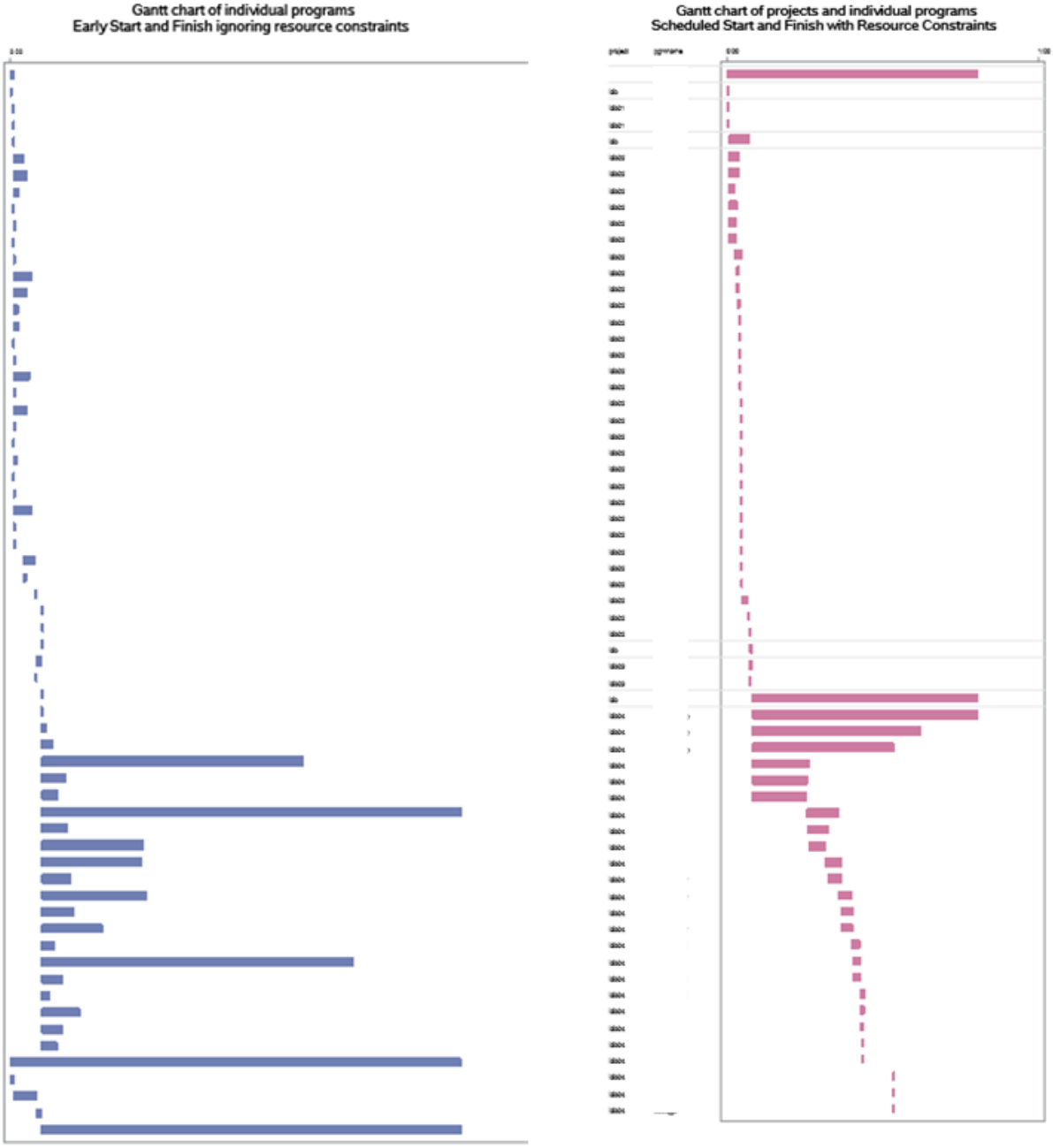


Figure 5 Useful Gantt charts

Alternatively, a nice way to view the use of each Grid node was to group or organize the bars by grid as shown in Figure 6. This really gives you a sense of how CPM schedules the programs within a Grid node.

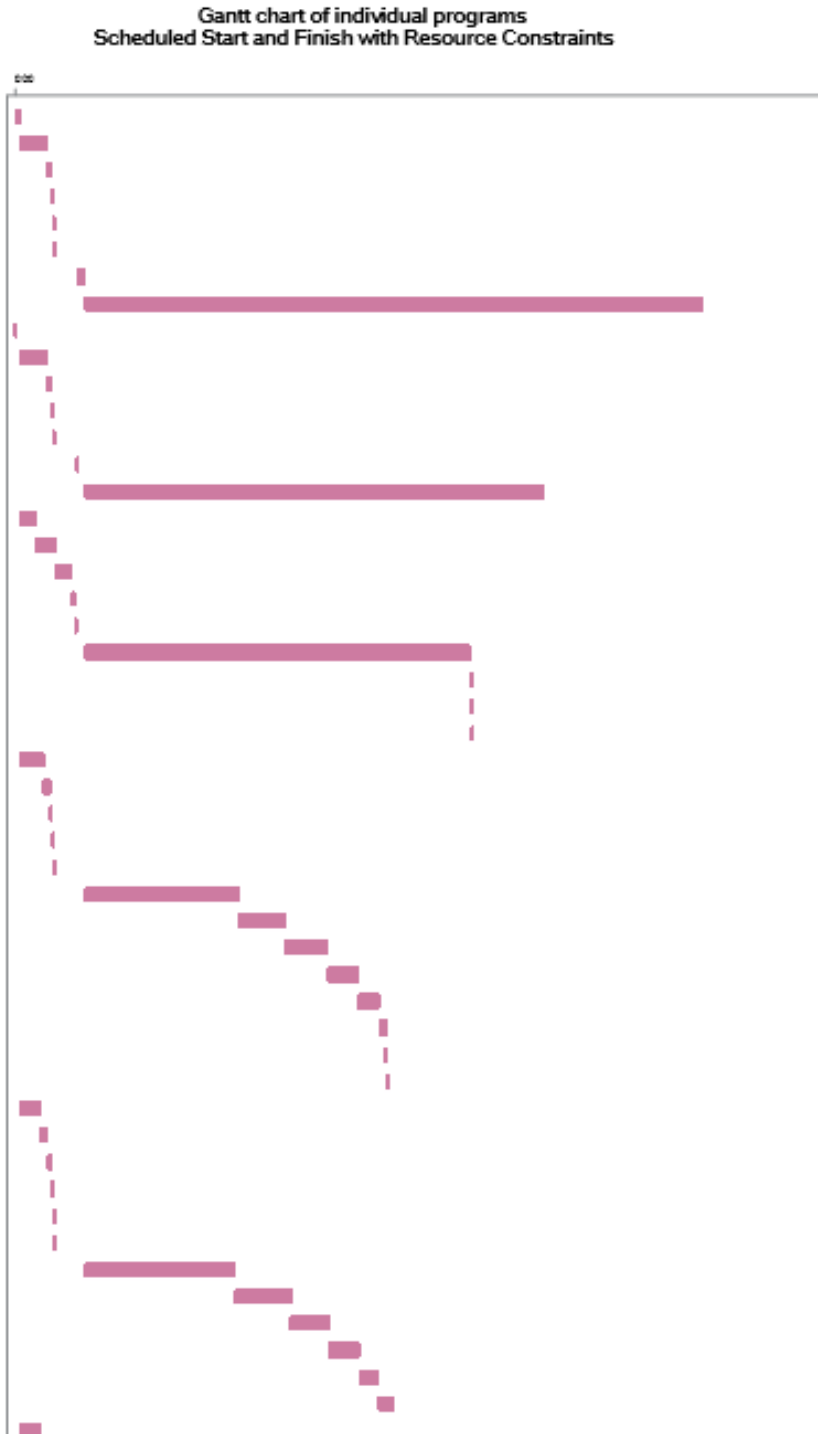


Figure 6 Per grid Gantt chart

Using these results, we take the next step to either manually modify the driver program or attempt to auto-generate the driver program.

Submitting Parallel Steps to the SAS Grid:

First, we created a few simple macros that initialized my Grid sessions. The following AUTOSIGN macro (in stripped down form) signs onto a specific SAS Grid session and passes local macro variables over to the remote session via the %SYSLPUT statement. In our situation, we pass the location of an autoexec.sas file and we used PRINTTO to route the log and output, so we passed the location for each.

```
%macro autosign(sessid);
       signon &sessid cmacvar=SignonStatus wait=no ;
       %syslput execloc=%bquote(&execloc);
       %syslput pgmversion=&pgmversion;
       %syslput logdir=&logdir;
       %syslput outdir=&outdir;
%mend autosign;
```

Note that we used WAIT=NO in the SIGNON statement so the program can continue processing during the SIGNON process.

The driver program (extracts shown below) was auto-generated in the post-processing section of the PROC CPM program.

```
%global execloc pgmversion logdir outdir snum waitlist;
%let snum=0;              /* Initialize the session number to 0 */
%let waitlist= ;          /* Initial waitfor sessions to blank  */
/*-------------------------------------------------------------*
* Set the pgmversion which points to the source and related
* subdirectories.
*-------------------------------------------------------------*/
%let pgmversion=xxxxxxx;

/*-------------------------------------------------------------*
* Set up directories
*-------------------------------------------------------------*/
%let execloc=directory_path/autoexec.sas;
%inc "&execloc";
%let logdir  =&projdir/logs/&pgmversion;
%let outdir  =&projdir/output/&pgmversion;

/*-------------------------------------------------------------*
* Get information about the Grid and send all RSUBMITS to Grid
*-------------------------------------------------------------*/
%let allnodes=%sysfunc(grdsvc_nnodes(resource=&SasAppSvr));
%put NOTE: Number of grid nodes available =&allnodes;
%put NOTE: Job name displayed in Grid Manager=SASGrid:&sysjobid;
%put;
%put NOTE: Enabling all rsubmits to run on the Grid.;
%let gerc=%sysfunc(grdsvc_enable(_all_,resource=&SasAppSvr));
%put NOTE: Grid enable request returned code=&gerc;

/*-------------------------------------------------------
* Sign On to grid session grid1 for project=idb01
*-------------------------------------------------------*/
%autosign(grid1);

/*-------------------------------------------------------
* Rsubmit programs to grid session grid1 project=idb01
* Note that CONNECTPERSIST instructs SAS to remain signed onto
* GRID1 session to wait for another RSUBMIT.
*-------------------------------------------------------*/
rsubmit grid1 cmacvar=rsubstatus wait=no connectpersist=YES;
  %put pgmversion=&pgmversion;
  %inc "&execloc";
  %inc "&projdir./programs/idb_grid_new/adam/pgm1.sas";
endrsubmit;
```

```
/*-------------------------------------------------------------
* Sign On to grid session grid2 for project=idb01
*-------------------------------------------------------------*/
%autosign(grid2);
/*-------------------------------------------------------------
* Rsubmit programs to grid session grid2 project=idb01
*-------------------------------------------------------------*/
rsubmit grid2 cmacvar=rsubstatus wait=no connectpersist=YES;
  %put pgmversion=&pgmversion;
  %inc "&execloc";
  %inc "&projdir./programs/idb_grid_new/adam/pgm2.sas";
endrsubmit;
/*-------------------------------------------------------------
* The next project level must wait for the last RSUBMITS
* to finish before starting.
*-------------------------------------------------------------*/
waitfor _all_ grid1 grid2;

/*-------------------------------------------------------------
* Rsubmit programs to grid session grid1 project=idb02
*-------------------------------------------------------------*/
rsubmit grid1 cmacvar=rsubstatus wait=no connectpersist=YES;
  %put pgmversion=&pgmversion;
  %inc "&execloc";
  %inc "&projdir./programs/idb_grid_new/adam/pgm3.sas";
  ---- include more programs here for this grid session ----
endrsubmit;
/*-------------------------------------------------------------
* Rsubmit programs to grid session grid2 project=idb02
*-------------------------------------------------------------*/
rsubmit grid2 cmacvar=rsubstatus wait=no connectpersist=YES;
  %put pgmversion=&pgmversion;
  %inc "&execloc";
  ---- include more programs here for this grid session ----
 endrsubmit;
/*-------------------------------------------------------------
* Sign On to grid session grid3 for project=idb02
*-------------------------------------------------------------*/
%autosign(grid3);
/*-------------------------------------------------------------
* Rsubmit programs to grid session grid3 project=idb02
*-------------------------------------------------------------*/
rsubmit grid3 cmacvar=rsubstatus wait=no connectpersist=YES;
  %put pgmversion=&pgmversion;
  %inc "&execloc";
  ---- include more programs here for this grid session ----
endrsubmit;
```

RSUBMIT to more projects/subprojects and grid nodes as needed. As you get to the end, you start changing the CONNECTPERSIST option to NO, so the session is closed when the RSUBMIT completes.

```
/*-------------------------------------------------------------
* Rsubmit programs to grid session grid6 for project=idb04
*-------------------------------------------------------------*/
rsubmit grid6 cmacvar=rsubstatus wait=no connectpersist=NO;
  %put pgmversion=&pgmversion;
  %inc "&execloc";
  ---- include more programs here for this grid session ----
endrsubmit;
```

RESULTS

For those jobs that were running for several hours when executing programs sequentially, we were able to cut job execution times to about 25% - 33% of the sequential times through parallel processing. With time, experience, and reusable methods, we were able to reduce the effort required to achieve this reduction in job execution time.

CONCLUSION

Given a stepwise and measurable approach to optimizing build jobs for integrated databases, you can gain considerable efficiencies through parallel processing with the SAS Grid platform and through analysis of the parallel distribution of programs with analytic techniques such as CPM. To gain the most value, you must:

1. Determine if shorter run-time is worth the investment involved in optimizing parallel processing
2. Analyze the logs to:
    a. Determine per program execution times
    b. Find dependencies between programs
3. Run PROC CPM to calculate the shortest execution time achievable with parallel processing
    a. If the gain is not significant, stop.
    b. If the gain is worth it, continue.
4. Evaluate the Gantt charts and output results to determine your best distribution of programs.
    a. You can manually distribute to gain the benefits
    b. Auto-generate the driver based on the results from CPM.
5. If “dependent-groups” become evident, rerun the CPM analysis with projects and subprojects.

Overall, we were very pleased with the results of our experiment. We continue to refine the techniques and make the code more reusable and generalized. We plan to apply the techniques to other optimization methods.

Note that we have not discussed how to optimize within a given program, but that is also part of the process.

REFERENCES

1. Clinical Data Interchange Standards Consortium (CDISC). www.cdisc.org
2. Kelley, James; Walker, Morgan. Critical-Path Planning and Scheduling. 1959. Proceedings of the Eastern Joint Computer Conference
3. Phillips, Don T.; Ravindran, A.; Solberg, James J.1976. Operations Research Principles and Practice. John Wiley & Sons, Inc. ISBN 0-471-68707-3
4. SAS Institute Inc. 2015. Grid Computing in SAS 9.4, Fourth Edition. Cary, NC: SAS Institute Inc.
5. SAS Institute Inc. 2015. SAS/OR® 14.1 User’s Guide: Project Management. Cary, NC: SAS Institute Inc.
6. Woolf, Murray B. 2012. CPM Mechanics: The Critical Path Method of Modeling Project Execution Strategy. ICS-Publications. ISBN 978-0-9854091-0-4

ACKNOWLEDGMENTS

This investigation in optimization was performed on a project with d-Wise Technologies, Inc. I thank them for the opportunity and the collaboration.

CONTACT INFORMATION

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

Eric C. Brinsfield, President
Meridian Analytics
5412 Season Ln
Virginia Beach, VA 23455 USA
Work Phone: 919-302-3747
Email: eric.brinsfield@meridiananalytics.com

.

Principal Consultant
d-Wise Technologies, Inc.
1500 Perimeter Park Dr.
Morrisville, NC 27560 USA
Email: eric.brinsfield@d-wise.com