Paper 15-2018

Parallel Processing with Base SAS

Jim Barbour, Experian Information Solutions, Inc.

ABSTRACT

Parallel processing is a potentially powerful tool for SAS performance tuning. There's a myth out there that one needs to license SAS/Connect in order to conduct parallel processing. Not true! Parallel processing is eminently practicable using just the Base SAS product.

The simplest form of parallel processing relies on the use of SAS option statements alone. More complex parallel processing involves design changes and requires coding in support of those changes as well as the use of the SYSTASK and WAITFOR commands.

This paper will examine three examples of parallel processing: 1) The reduction in run time by a factor of eight for a specific example of Proc Means using just SAS options, 2) The running of multiple non-interdependent SAS procedures in parallel by means of launching subordinate processes, and 3) a comparatively more complex example of breaking a single DATA step into multiple processes in order to read a large data set (200+ million rows, 300+ columns) in less than a quarter of the single-threaded run time. Examples 2 and 3 will then be combined and used in a SAS process whose overall run time went from **9 hours** to **3 hours.**

All examples were run with SAS 9.4 in a multi-core UNIX environment. SAS Enterprise Guide 7.15 was used as the program editor and primary means of submission, but other editors and UNIX command line submission may be used in lieu of Enterprise Guide.

INTRODUCTION

One morning at about 2:00 A.M., I was monitoring the run of one of our principal jobs, a job that takes a series of four raw data files, turns them into SAS data sets, and conducts some QC checks. A full run of roughly 250 million records takes about 9 hours. Even test file runs of 15 to 20 million records take on the order of 45 minutes to run. I couldn't move on to the next phase of my work until I had all four SAS data sets. I struck me that we didn't read even *one* record from the fourth raw file until we had read *all* records in *all* of the previous files. On that late, late shift, one thought ran through my mind: There's *got* to be a better way. Why can't we read all four files at the same time instead of each one waiting for all prior files? Thus my quest for parallel processing was born.

What exactly is parallel processing? Essentially, parallel processing is breaking up a task into multiple components and *simultaneously* working on those components. One form of parallel processing is multi-threading. Parallel processing and multi-threading mean essentially the same thing. However, for the technically minded, there is a distinction. A process runs at a higher level than a thread. Multi-threading occurs within a single process. If one launches multiple processes to accomplish parallel processing, one is indeed processing in parallel, but, strictly speaking, one is no longer multi-threading. This technical distinction will matter little for the purposes this paper and matter not at all for the average SAS user.

Now, consider a business. The business needs to make 100 widgets per day. There are four workers; let's call them Roberto, Michael, Maria, and David. Well this seems obvious, right? Just assign each worker 25 widgets per day, and "job done." Most likely you wouldn't tell Michael to not make any widgets until Roberto had finished his widgets and Maria to not make any widgets until both Roberto and Michael had finished their widgets, and tell David to wait for everyone else before making even one widget. That'd be the dumbest way in the world to run a business. But that's just what you're doing if you run multiple non-interdependent SAS steps one after another. It's going to be more productive if Roberto, Michael, Maria, and David work in *parallel*, that is, all four workers making widgets concurrently, without waiting for one another. Likewise, unless you've got some primitive single core processor, it's going to be a lot more productive if we run SAS steps in parallel instead of making independent steps wait for one another.

This paper is primarily concerned with speed, not efficiency. Here, I am not considering memory use, the amount of i/o, cpu time, or other potential measures of efficiency. My job is to get SAS data sets to our analysts as quickly as possible. Speed therefore is my primary concern and the consumption of resources, within reason, to obtain such speed is considered acceptable. The reader should bear in mind that the speed discussed in this paper does come with some cost and should consider whether said cost is appropriate for the reader's specific circumstances.

In this paper, I will discuss:

- The most basic form of parallel processing, through the use of SAS options
- An "intermediate" example of running multiple independent whole SAS steps simultaneously.
- A more advanced example of parallel processing that breaks up a single SAS step into multiple sub-processes.
- An example job that went from 9 hours run time down to 3 hours when redesigned with parallel processing.

BASIC PARALLEL PROCESSING

What are the SAS statements necessary to conduct basic parallel processing? OK, here you go:

```
OPTION THREADS;
```

Really, that's all there is to it. OK, thanks for reading my very technical SAS paper. Have a nice day, and I'll see you at next year's conference…

What's that? Oh. You want just a tad more detail. Very well, if you insist, but, seriously, that simple option statement is all you need to begin parallel processing. That option is best combined with another:

```
OPTION CPUCOUNT=ACTUAL;
```

What I've done is to turn on multi-threading, and, by using "ACTUAL," I've instructed SAS to interrogate the operating system to see just how may logical cores are available for use. How many cores are available in your environment? Well, there are a number of ways to find out, but perhaps the easiest is:

```
OPTIONS   THREADS     CPUCOUNT=ACTUAL;
PROC      OPTIONS     OPTION=CPUCOUNT;
RUN;
```

After running the above, your log should look something like this:

```
CPUCOUNT=80       Specifies the number of processors that thread-enabled applications
should assume are available for concurrent processing.
```

The above log statements tell me that I have 80 logical cores available in the environment in which I ran the program. Note that I said "logical" cores. The number of physical cores is typically less than the number of logical cores. The number of *logical* cores is the number of cores actually available for use by SAS programs.

I generally run with CPUCOUNT=ACTUAL, which in fact is the default in our shop, but if there were a concern about the performance impact of multi-threaded processes, one could always set the CPUCOUNT to some number less than the actual number and thereby restrict the number of threads available to any one SAS process. As a general rule, I design my multi-threaded jobs to use less than 20% of the available cores, with 20% being the absolute maximum even the highest priority SAS job should use.

OK, this THREADS option sounds all well and fine, but what does it really do for us?

Ah. Fair question. Let's look at some examples. The following is a quarterly report that I run:

```
PROC      MEANS DATA=&Data._&Period.
    (OBS         =      &Means_Obs)
    MAXDEC       =      2
    QMETHOD      =      P2
    n nmiss min max mean p1 p5 p25 p50 p75 p90 p95 p99
    nway;
RUN;
```

This report normally takes four hours to run, single threaded. Multi-threaded, it runs in just 30 minutes. I had to run it a couple of times before I could believe that multi-threading could make that much of a difference, but it does. Multi-threading saves 3.5 hours *per run* (eight times faster) – and that *just by setting the options*.

Multi-threading can run on a number of SAS procedures. Let's look at another: The SORT procedure. I set up some macros to run sorts of different types with both single and multi-threading. Here are some of the results (in figure 1):

Run Times by Type of Sort			
Type of Sort	**Sample Size**	**Run Time**	**CPU Time**
Single Thread	5,000,000	10:11:00	2:32
SQL	5,000,000	**6:01:00**	2:52
Multi-Thread	5,000,000	8:22:00	2:42
Tag	5,000,000	10:19:00	**1:42**
Single Thread	15,000,000	17:53:00	7:02
SQL	15,000,000	17:04:00	8:57
Multi-Thread	15,000,000	**12:31:00**	7:26
Tag	15,000,000	20:10:00	**4:58**
Single Thread	25,000,000	36:47:00	13:10
SQL	25,000,000	25:29:00	14:44
Multi-Thread	25,000,000	**24:16:00**	13:20
Tag	25,000,000	38:09:00	**9:14**
Single Thread	35,000,000	56:29:00	18:33
SQL	35,000,000	**32:50:00**	20:57
Multi-Thread	35,000,000	33:40:00	18:57
Tag	35,000,000	47:30:00	**12:47**
Single Thread	45,000,000	1:03:48:00	24:37
SQL	45,000,000	51:12:00	27:00
Multi-Thread	45,000,000	**40:38:00**	25:21
Tag	45,000,000	1:58:15:00	**20:02**

Figure 1. Run times by sample size and type of sort

From these and many other runs, I draw the following conclusions:

1. Single threaded sort (OPTION NOTHREADS) is never the fastest.
2. Multi-threaded sort is most frequently the fastest of all types of sorts.
3. In those cases where multi-threaded was not fastest, it was only a matter of a minute or two slower. In other words, there's not much penalty for using multi-threaded sort.
4. The larger the dataset, the more multi-threaded sort pays off. In some cases, multi-threaded sort was over 20 minutes faster than single threaded.

Some of these results I've described are fairly dramatic. Can I guarantee that you'll see similar gains in your environment? Well, no, but turning on multi-threading has had no negative effects in our shop, and, in those cases where there have been performance improvements, those improvements have been significant. Do note that I work in an analytical environment. The impact of multi-threading in another type of environment may differ. My suggestion would be to experiment with multi-threading gradually, assessing any impact as you conduct more and more parallel processing over time.

INTERMEDIATE PARALLEL PROCESSING

The jump from basic multi-threading to intermediate parallel-processing is a fairly sizeable one. I'm going to differentiate intermediate parallel-processing from advanced by saying that intermediate parallel-processing consists of running whole SAS steps in a sub process whereas advanced parallel-processing breaks what would normally be a single step into multiple sub-processes.

Note that for the purposes of this paper, I am just using the Base SAS product. If you have a license for SAS/Connect, you can also conduct parallel processing using RSUBMIT. Indeed, it's actually easier with SAS/Connect than with SAS Base alone, but inasmuch as there are many papers on how to parallel process with SAS/Connect and relatively few with SAS Base, I will restrict my remarks herein to just the Base product.

Let's look at an example. Say we have a series of six Proc FREQ's that we'd like to run. There are no interdependencies between these FREQ's. We could run six SAS jobs, manually launching and manually monitoring each, and then manually taking whatever steps are appropriate subsequent to the running of the FREQ's. Of course, babysitting a bunch of little SAS jobs isn't much fun, consumes a lot of time, and is generally to be avoided.

More typically, a SAS job consists of multiple steps. For example, we might have a DATA step to read some raw input, then do some type of manipulations or calculations, and then conduct a variety of steps in order to evaluate and get a sense of our results (Proc REPORT, Proc MEANS, Proc FREQ, etc.). In our example, we have six Proc FREQ's we'd like to run. The most typical way to set this up is to run one after another, each step waiting for all previous steps to complete before beginning execution. Here's a diagram showing the typical set up of a series of Proc FREQ's:

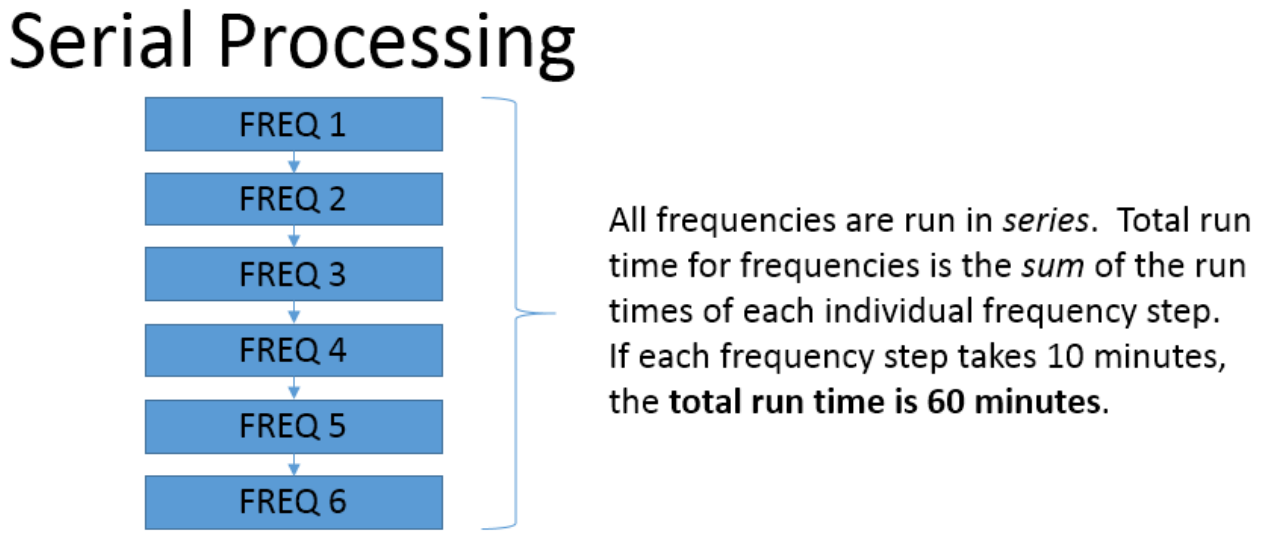

Figure 2. A serially processed set of Proc FREQ's

The problem of course is that with larger datasets a FREQ step can run for a fairly long time. In the example above, each FREQ takes 10 minutes, giving us an overall run time of 60 minutes. What if each takes 20 to 30 minutes? Now we're talking about a 2+ hour run, just for the frequency steps. Not so good.

By contrast, let's look at the same six FREQ steps set up for parallel-processing.

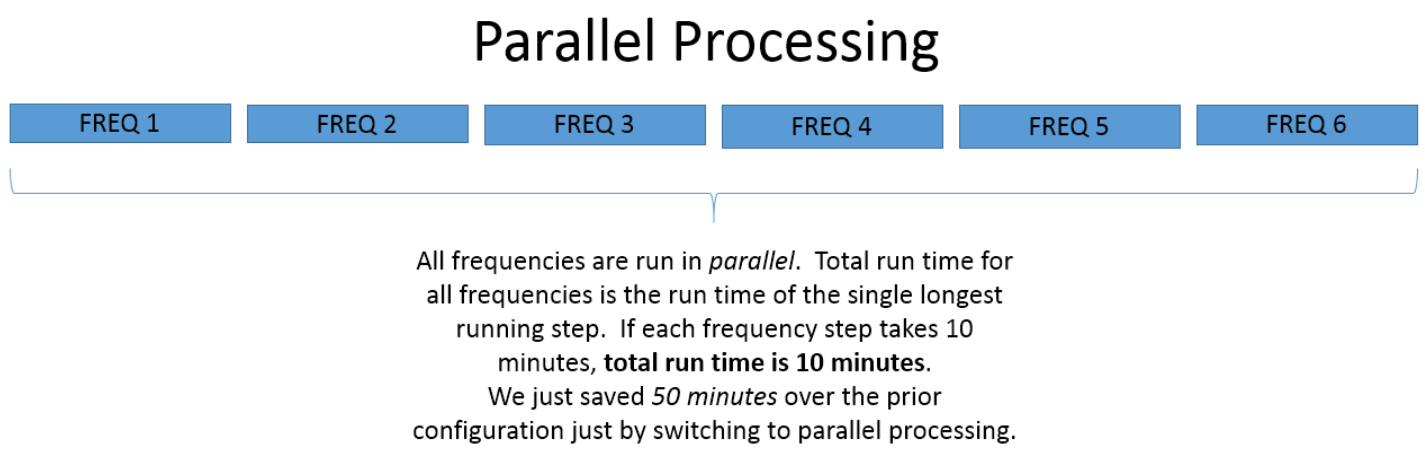

Figure 3. Parallel processing a set of Proc FREQ's.

With parallel-processing, all six FREQ's run simultaneously. The overall run time then becomes the run time of the single longest running step. If each FREQ takes about 10 minutes, the overall run time is just that, 10 minutes. We just saved 50 minutes simply by re-configuring the steps to run in parallel.

What if each takes 20 to 30 minutes? Here is where the value of parallel-processing comes to the fore. Instead of a 2+ hour run as in traditional serial processing, our parallel processing run is no longer than 30 minutes – at least four times faster.

Thus far, I've diagrammed only the FREQ steps. Now, let's look at a parallel processing job as a whole. Conceptually, a parallel processing SAS job might look something like this:

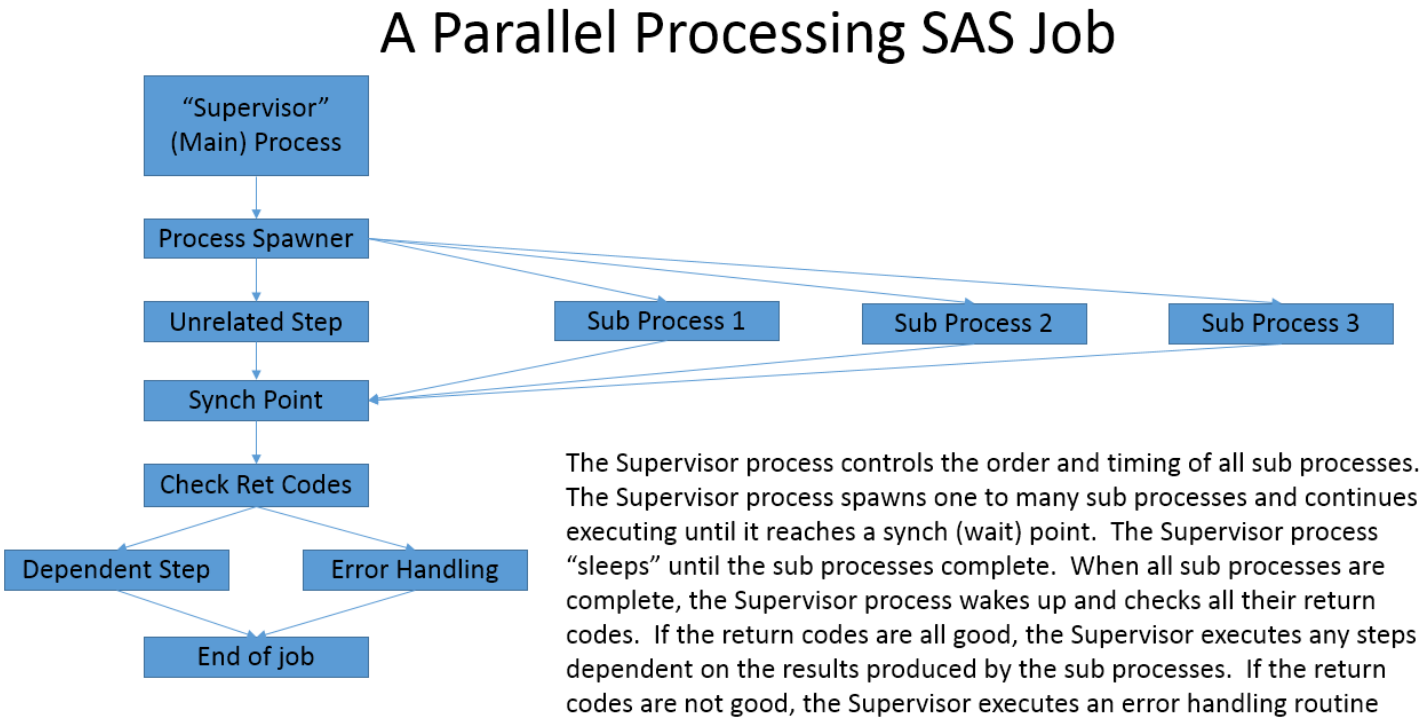

Figure 4. A parallel processing SAS job

SUPPORTING SAS CODE – REQUIREMENTS

In order to launch and control parallel processes, we will be required to do several things:

1. We have to be able to launch a subordinate process (i.e. a sub process).
    a. We must be able to pass parameters to a sub process such that the sub process can perform work relevant to and coordinated with the main (supervisor) process.
    b. We have to be able to control the log files of sub processes lest the logs of the identically named sub-processes overwrite one another or come into contention.
2. The supervisor process must be able to continue to execute concurrently with any sub processes (and the sub processes concurrently with the supervisor).
3. We have to be able to track the end of execution of a sub process
4. We have to be able to query the return code of a completed sub process
5. We have to be able to suspend and then reactivate the supervisor process if any steps in the supervisor process are dependent on results from a sub process.

SUPPORTING SAS CODE – SUPERVISOR PROCESS

Let's look at the SAS code needed to support the above requirements. First and foremost, we need to launch a sub process. We're going to do that by passing commands from SAS to UNIX, commands that start another SAS process. The commands to run a SAS program in UNIX take the general form:

```
nohup sas amazing_pgm.sas &
```

These are the commands that anyone typing at the UNIX command line would use to launch a SAS job. The commands we'll use to launch a sub-process from within our SAS program will follow the same general form as the commands that would be typed in on the UNIX command line – with a few exceptions which I'll discuss later.

Now, we need a way to pass these commands from SAS to UNIX. There are a number of ways to pass commands to UNIX including the "X" command, but I prefer SYSTASK. SYSTASK allows me to give each sub process a name by which that sub process can be tracked and a macro variable that will hold a return code from the sub process. SYSTASK takes the general form:

```
SYSTASK …commands… [NO]WAIT TASKNAME=task name STATUS=a return code macro variable;
```

where "…commands…" represents text commands passed to UNIX (as discussed previously).

In our example, we want to run six Proc FREQ's simultaneously. Let's cleverly give our first FREQ sub process the task name Freq1. We'll then use the macro variable Freq1_RC to hold the return code from the Freq1 sub process. Our SYSTASK might look something like this:

```
SYSTASK …commands… NOWAIT TASKNAME=Freq1 STATUS=Freq1_RC;
```

Notice the use of NOWAIT. Typically, when SAS passes a command to UNIX, SAS waits until the command completes. NOWAIT instructs SAS not to wait, at least not at this juncture. A supervisor process issuing the SYSTASK command with NOWAIT will pass commands to UNIX and continue to execute. If any downstream processing in the supervisor process is dependent on a sub process, we'll have to include SAS code that instructs the supervisor process to hold off executing that dependent code until the sub process completes. The particular SAS code that instructs a supervisor process to hold execution is WAITFOR. Given that our sub process is named Freq1, we'd say:

```
WAITFOR Freq1;
```

When SAS encounters a WAITFOR, it goes to sleep (suspends execution) until the task(s) named in the WAITFOR complete. In our case, we have six FREQ's to run. If we wanted to wait for them all to complete, we'd use _ALL_ in conjunction with WAITFOR which basically tells the supervisor to wait for all six to complete before resuming execution:

```
WAITFOR _ALL_ Freq1 Freq2 Freq3 Freq4 Freq5 Freq6;
```

I often will use a macro to create the wait points in my supervisor processes:

```
**  WAITFOR statement causes main process to sleep until all sub processes are done.
    **;
%MACRO    Create_Wait(Thread_Name=, Seq_No=);
    %IF   %UPCASE("&WAIT")  =     "WAIT"      %THEN
          %RETURN;

    WAITFOR     _ALL_
    %DO         i     =     1     %TO         &Seq_No.;
          &Thread_Name&i
    %END;
    ;
%MEND     Create_Wait;
```

When all six FREQ sub processes complete, the supervisor process will "wake up" (resume execution). At this point, we'll want to know if our sub processes worked properly. I use a little macro to check the return codes:

```
**  Check all of the individual return codes from each of the sub processes.      **;
%MACRO    Check_Return_Codes(Thread_Name=, Seq_No=);
    %DO         i     =     1     %TO         &Seq_No.;
          %IF   "&&&Thread_Name&i."     >     "0"   %THEN
                %DO;
                      %LET        SYSCC =  8;
                      %PUT        ERROR:  &Thread_Name&i.=&&&Thread_Name&i.;
                %END;
          %ELSE
                %DO;
                      %PUT        NOTE-  &Thread_Name&i.=&&&Thread_Name&i.;
                %END;
    %END;
    ;
%MEND     Check_Return_Codes;
```

Notice that I'm putting my macro variables inside double quotes. Strictly speaking, quotation marks aren't required inside a macro (where all values are assumed to be character even if they appear to be a number). The use of double quotes here prevents errors when a macro variable resolves to something that might cause an error such as the state code for Oregon ("OR") which SAS will interpret as the logical operator "OR". Notice also that I called my first macro variable "Thread_Name" really this would more properly be called "The_Return_Code_For_Process_Name," but that's a but unwieldy, so I just used "Thread_Name". Please understand that I'm really talking about a return code here not a task (process) name.

At this juncture, I think that's enough discussion on the various components of SYSTASK and a subsequent WAITFOR. Let's now turn our attention to the text commands we need to pass to UNIX. Recall the general form of SYSTASK is:

```
SYSTASK …commands… NOWAIT TASKNAME=Freq1 STATUS=Freq1_RC;
```

What goes into "commands?" Well something along the lines of `nohup sas amazing_program.sas &` goes into "commands," but we're going to need to just a bit more in order to meet requirements 1a and 1b. Recall that our requirement 1 is:

1. We have to be able to launch a subordinate process (i.e. a sub process).
    a. We must be able to pass parameters to a sub process such that the sub process can perform work relevant to and coordinated with the main (supervisor) process.
    b. We have to be able to control the log files of sub processes lest the logs of the identically named sub-processes overwrite one another or come into contention.

Regarding 1a, above, we need to be able to pass parameters to the sub-process so that it knows things like Lib refs, data set names, etc. There are any number of ways to pass parameters to a sub-process including SAS data sets, raw data files, a SYSPARM on the command line, SET statements, etc. I like using SET statements to pass parameters to sub-processes because I can cut and paste the text out of Enterprise Guide and into a Notepad or other text editor where I

can modify the commands until they work when submitted via a UNIX command line, and then, knowing a working command string, go back and change my program such that the program produces the same text. For more on using SET, see Rick Wicklin's excellent blog post: How to pass parameters to a SAS program

Here's some code:

```
**    Compose a command string.  The parameter names are case sensitive.      **;
**    Coded with tabs for readability.  TRANWRD changes tabs to spaces.       **;
**    For Enterprise Guide, use initstmt, or the log files will disappear.    **;
**    For UNIX, use the log statement and remove the initstmt.                **;
**    Sample commands:                                                        **;
  /*  -log  &dQuote.&dQuote.&sas_path./&Program_Name._Trade_Freq" ||
      STRIP(PUT(_Thread_No,15.))    ||     ".log&dQuote.&dQuote.*/
  /*  -initstmt   &sQuote.&sQuote.PROC PRINTTO
  LOG=&dQuote.&dQuote.&sas_path./&Program_Name._Trade_Freq"       ||
      STRIP(PUT(_Thread_No,15.))    ||     ".log&dQuote.&dQuote.
  NEW;RUN;&sQuote.&sQuote.*/

**    Actual commands:                                                        **;
  _Commands =     TRANWRD("SYSTASK COMMAND 'nohup
  /sas/sas_9.4/install/SASFoundation/9.4/sasexe/sas
                                  &Freq_Log
                                  -noterminal
                                  -set  debug
      &dQuote.&dQuote.&DeBug.&dQuote.&dQuote.
                                  -set  Macro_Name  &dQuote.&dQuote."       ||
      STRIP(_Macro_Name)                ||    "&dQuote.&dQuote.
                                  -set  sas_lib
      &dQuote.&dQuote.&SAS_Lib.&dQuote.&dQuote.
                                  -set  Data_Set
      &dQuote.&dQuote.&SAS_Out.&dQuote.&dQuote.
                                  -set  SAS_Path
      &dQuote.&dQuote.&SAS_Path.&dQuote.&dQuote.
                                  -set  Fmt_Path
      &dQuote.&dQuote.&Fmt_Path.&dQuote.&dQuote.
                                  -set  Pgm_Path
      &dQuote.&dQuote.&Pgm_Path.&dQuote.&dQuote.
                                  -set  MsgLvl
      &dQuote.&dQuote.&MsgLvl.&dQuote.&dQuote.
                                  -set  ErrLvl
      &dQuote.&dQuote.&ErrLvl.&dQuote.&dQuote.
                                  -set  Obs
      &dQuote.&dQuote.&Obs.&dQuote.&dQuote.
                                  -set  Thread            &dQuote.&dQuote."       ||
      STRIP(PUT(_Thread_No,15.))    ||    "&dQuote.&dQuote.
                                  -sysin
      &pgm_path/trade_freq_thread_ex_DA_Export.sas'
                                  &Wait                   STATUS=Trade_Freq_RC"   ||
      STRIP(PUT(_Thread_No,15.))    ||     " TASKNAME=Trade_Freq_Thread"      ||
      STRIP(PUT(_Thread_No,15.))    ||     " shell",'09'X,' ');

            **    Set values cannot be blank.  Change "" to NULL to avoid errors.   **;
            _Commands   =     TRANWRD     (_Commands,'""','NULL');

            **    Compress multiple blanks to single blanks.      **;
            _Commands   =     COMPBL      (_Commands);

            **    Submit the commands to the queue.   **;
            CALL  EXECUTE     ('%NRSTR(%Process_Commands(Commands='           ||
      STRIP(_Commands)              ||    ');)');
```

That's a bit long, but basically what we're doing is using a SAS program to create the exact same text that we might enter on a UNIX command line to run a SAS job. All of this text is stored in the character variable _Commands.

Comments on Commands

In a short paper like this, I can't cover every last aspect of the above presented SAS code that formats the text of the commands that SYSTASK will send to UNIX.

```
SYSTASK ...commands... NOWAIT TASKNAME=Freq1 STATUS=Freq1_RC;
```

However, a few comments are in order:

1. &sQuote resolves to a single quote. &dQuote resolves to a double quote. I use paired single and double quotes to get everything to resolve to the final command text that I need to launch my sub process.
2. &Freq_Log resolves to either a -log parameter or to an -initstmt parameter. The -log parameter should be used when submitting the supervisor process from the UNIX command line. The -initstmt parameter should be used when submitting the supervisor process from SAS Enterprise Guide. The initstmt issues SAS commands that execute before the SAS program itself begins execution. I use the initstmt in this case to run a Proc PRINTTO that keeps my sub processes' logs separate and prevents them from being obliterated by Enterprise Guide.
3. Note that when I execute SAS, I fully qualify the executable with the path name. SAS doesn't have automatic access to your $PATH variable.
4. Note also that I suffix the command string with "shell" so as to make the submission more like a user's command line submission.
5. All parameters in UNIX must come *before* -sysin.

I leave it to the reader to seek out the remaining intricacies and options of each of the components that goes into "commands." Having actual working code, as in the above, should go a long way toward providing the reader with guidance.

Executing the Commands

After the text is all formatted and stored in _Commands, I call execute a simple Macro to process the commands. The little macro looks like this:

```
**  This macro processes any commands passed to it. **;
%MACRO    Process_Commands(Commands=);
    &Commands.;
%MEND     Process_Commands;
```

SYSTASK is executed as soon as SAS encounters it. By using a macro, I can programmatically append all of my parameters before SAS executes the SYSTASK.

Launching a Subordinate Process – Recapitulation

So, basically, to launch a sub process, I format some text to be identical to what I would enter on a UNIX command line – with certain modifications relevant to in-program submission. I then submit those commands to UNIX via SYSTASK followed by a &Wait macro variable and TASKNAME= and STATUS= parameters. The &Wait macro variable is typically set to NOWAIT. When NOWAIT is used, the supervisor process does not *immediately* stop execution and wait for the sub-process to finish. Instead, the supervisor process continues to execute, and *you* have to tell the supervisor process at what point it ought to wait by using the WAITFOR command. If you don't use a WAITFOR, the supervisor process will run to end of job, regardless of what any sub process does and irrespective of any return codes issued by any sub process.

TASKNAME specifies an identifier to be used in conjunction with WAITFOR. For example, if I launch a sub-process with TASKNAME=Freq1, I would code WAITFOR Freq1 further down in the Supervisor process. When the Supervisor process encounters WAITFOR Freq1, the supervisor process will go to "sleep" until sub-process Freq1 completes.

STATUS specifies a macro variable that will contain the return code from the task. For example, if I code SYSTASK …*commands*… NOWAIT TASKNAME=Freq1 STATUS=Freq1_RC, then the return code from Freq1 will be stored in macro variable Freq1_RC. When the supervisor process encounters WAITFOR Freq1, the supervisor process will sleep until Freq1 completes. When Freq1 completes, the supervisor process can interrogate Freq1_RC to see if the sub process executed successfully (zero) or unsuccessfully (non-zero) and take appropriate action.

What we are doing with SYSTASK … NOWAIT is essentially the same as if we had executed a series of SAS jobs from the UNIX command line *except* that we've now automated the launching and monitoring. What would have been a bunch of little SAS jobs requiring babysitting now becomes a single, unified process, a process that can be integrated into a job stream. The reader should note that SYSTASK can be used to launch just about any command that can be executed from the UNIX command line. SAS can therefore be used as a powerful and flexible scripting language for building complex jobs and systems in a UNIX environment – those jobs and systems consisting of a variety of UNIX commands, SAS programs, and other programs and utilities. The possibilities are restricted only by your creativity and your programming skill.

SUPPORTING SAS CODE – THE SUBORDINATE PROCESS

The SAS code for a subordinate process is simplicity itself: You write a SAS program just as you would any other SAS program. The great majority of the code in a subordinate process is just ordinary SAS code. One does however need to add a little code to the front end of the program in order to grab the parameters that the supervisor process has passed along. In order to receive parameters passed by a SET statement, one uses %SYSGET as shown below.

```
*-------------------------------------------------------------------------------*;
**       Retrieve parameters.  The argument to SYSGET is case sensitive.   **;
%LET     DeBug       =     %SYSGET(debug           );
%LET     Macro_Name  =     %SYSGET(Macro_Name      );
%LET     SAS_Lib     =     %SYSGET(sas_lib         );
%LET     Data_Set    =     %SYSGET(Data_Set        );
%LET     SAS_Path    =     %SYSGET(SAS_Path        );
%LET     Pgm_Path    =     %SYSGET(Pgm_Path        );
%LET     Fmt_Path    =     %SYSGET(Fmt_Path        );
%LET     MsgLvl      =     %SYSGET(MsgLvl          );
%LET     ErrLvl      =     %SYSGET(ErrLvl          );
%LET     Obs         =     %SYSGET(Obs             );
%LET     Thread      =     %SYSGET(Thread          );
```

I changed all blank parameters to the text literal "NULL" before passing them. A simple macro can be used to convert them back to blank:

```
**  Parameters cannot be passed if they are blank, so I set them           **;
**  to the text literal NULL.  This macro converts them back to blank.     **;
%MACRO    ReSet_Null(Variable);
    %IF   %UPCASE("&&&Variable")        =     "NULL"      %THEN
          %LET  &Variable               =     ;
%MEND     ReSet_Null;

%ReSet_Null(debug           );
%ReSet_Null(Macro_Name      );
%ReSet_Null(sas_lib         );
%ReSet_Null(Data_Set        );
%ReSet_Null(SAS_Path        );
%ReSet_Null(Pgm_Path        );
%ReSet_Null(Fmt_Path        );
%ReSet_Null(MsgLvl          );
%ReSet_Null(ErrLvl          );
%ReSet_Null(Obs             );
%ReSet_Null(Thread          );
```

ADVANCED PARALLEL PROCESSING

Intermediate parallel processing involves running whole SAS steps concurrently. Advanced parallel processing breaks a single step into multiple sub processes. The means of instantiation, tracking, error checking, flow of control, etc. is the same between intermediate and advanced parallel processing. In other words, your SAS code will look largely the same between intermediate and advanced parallel processing in terms of the technical mechanics of the supervisor and sub processes. The chief differences in coding will come from how one divides up the work of a single step into multiple sub-processes and how one recombines the results of multiple sub-processes into a single result.

BREAKING A SINGLE SAS STEP INTO MULTIPLE SUB PROCESSES

Recall the 2:00 A.M. inspiration behind my delving into parallel processing: A long running job that reads in four raw files. One of those four files is many orders of magnitude larger than the others. Indeed, a serially processed read of just that one file takes about **four hours**. While I was interested in putting parallel processing to work on the overall process, my particular interest lay in that one large file. How to do it?

SAS gives us some very nice parameters that can control the reading of files: FIRSTOBS and OBS. There's a bit more to it, but at a high level, the way in which I implemented parallel processing for this one large file was by dividing it up into *n* parts, with *n* being controlled by a macro variable parameter as follows: `%LET THREADS=15;`

In the above example I am setting *n* to 15. I will break the read of the large file into 15 pieces, each "piece" being a sub process. Each sub process will read its assigned share of the file into a SAS data set and then the supervisor process will combine the results to produce the same output as the original, serially processed version.

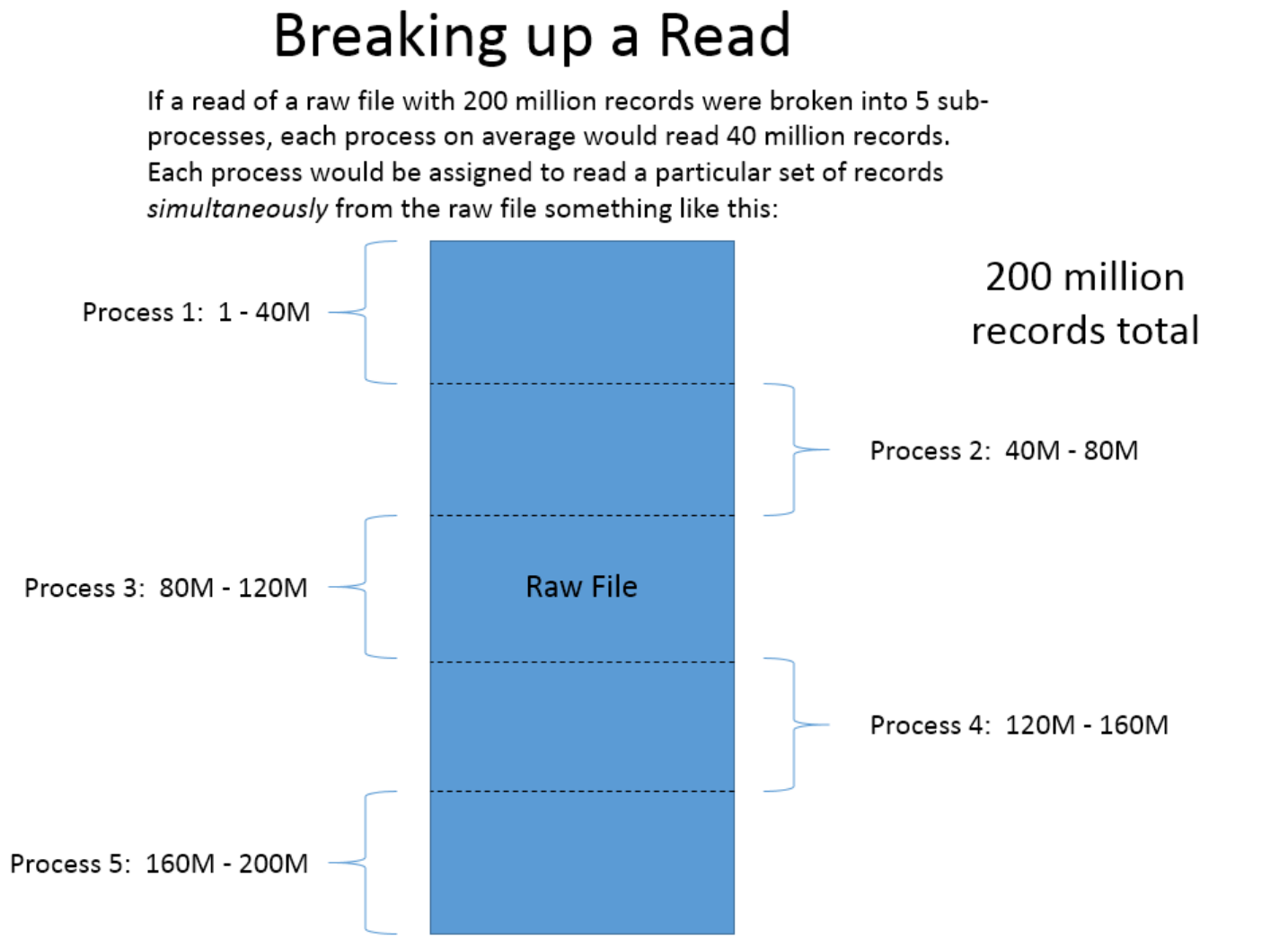

Figure 5. Reading a single raw file with multiple concurrent sub processes (parallel processing).

But won't the combining of results take time? Might not the time required to combine the results offset any gains made while reading the raw data? As a matter of fact, the combining does take time, but in my specific case, the single processed version of my job already must combine four files. I'm not adding a new step. Combining the results takes about the same amount of time with the original four files as it does with the multiple smaller files from the new sub-processes – the total number of records is the same whether I single process or multi process the job. So, in my case, there's no time penalty for combining the files.

Even in jobs where there was no previous combining of results going on, it's far faster to read a SAS data set than a raw file. If one can reduce the time it takes to read the raw files sufficiently, even when taking into account the additional time consumed by combining the results, run times will still be less overall.

In my case, the read of the large raw file in a single process takes about **4 hours**. With parallel processing, I can get my read down to less than **45 minutes**, depending on how many sub processes I allocate for the purpose. Even if combining the results were to take an additional 90 minutes, I'd still come out 1 hour and 45 minutes ahead.

LOAD BALANCING

I've already mentioned my means of divvying up the work among my *n* subordinate processes: FIRSTOBS and OBS. As I instantiate each subordinate process, I will pass in a FIRSTOBS and an OBS parameter that will tell the sub process which portion of my large raw file to read.

So, no problem, right? Take the total record count and divide by, in this case, fifteen. Each sub process gets 1/15th of the work. Well, that approach would work, but recall how FIRSTOBS works. In raw text files, there isn't some magical way that a sub process can instantaneously jump to, say, record number 150 million and begin reading its assigned share of the file. SAS is actually reading all the records before FIRSTOBS, but SAS quickly discards them without fully processing them. If we assign 1/15th of the total number of records to each sub process, sub processes reading portions of the file near the beginning of the file will quickly get to their assigned portion and start reading. Sub processes that are assigned portions of the file near the end of the file will have to read through hundreds of millions of records before they get to their assigned portion. With a large file, processes reading portions of the file toward the beginning will complete their work before processes reading portions toward the end even begin. In other words, you won't be running fifteen processes simultaneously. In fact, you might only have 3 or 4 processes running at any given time. Having only 3 or 4 processes running at any one time isn't going to give us the best performance. Ideally, we want all 15 sub processes to be executing throughout the entire period the raw file is being read. How can we keep all 15 processes continually busy? Front loading. That is, assigning more records to the processes that read portions of the file toward the beginning than processes processing portions toward the end.

As I tuned my input job, I knew that I needed to assign more records to the processes reading the beginning of the file. What I wound up doing is iteratively:

1. Dividing the total number of records to be parceled out by the number of processes (15 for the first iteration in this case), thereby obtaining a "base" number of records to assign to the sub-process.
2. Multiplying the base number by a weighting factor, thereby obtaining a weighted number of records to assign to the sub-process.
3. Instantiating a sub process with FIRSTOBS=1 and an OBS parameter corresponding to the weighted number of records.
4. Incrementing FIRSTOBS by adding OBS to it.
5. Subtracting the weighted number of records obtained in 2, above, from the original total number of records, giving me a new number of records that need to be parceled out.
6. Decrementing the number of processes by 1.
7. Going back to number 1, above, until all records have been parceled out to all of the sub processes.
8. In the last and final process, I set the OBS to MAX so as to make sure that every record is processed.

What's the ideal weighting factor? Well, the ideal factor is going to vary with your particular circumstances. I came up with a factor of 4.5%, and that seems to be working in my environment, but I can't imagine that 4.5% is going to be universally applicable. My advice would be to try different means of weighting your beginning processes such that all processes finish within a few minutes of each other. It's going to be very difficult to get every process to finish simultaneously in every run. Variations in system load and other external factors are going to change the individual

execution times of each sub process such that complete coordination of stop times will be elusive. If all the sub processes finish within a few minutes of each other, that's pretty good. Bottom line: You're just going to have to experiment and see what works best in your particular environment.

POTENTIAL PROBLEMS

Potential problems include dropping a record from where one OBS leaves off and the next FIRSTOBS begins. This is a pretty obvious problem that can be easily uncovered during the validation process. See Validating the Results, below.

A trickier problem occurs when there are inter-record dependencies. For example, any type of LAG() processing would have serious problems at the "boundary" where one OBS leaves off and the next FIRSTOBS begins. One will have to do some coding to get around cases like this.

As another example, say you have a raw file whose records are grouped by customer. Perhaps you could have something like this:

Header record, customer 1
Transaction record, customer 1
Transaction record, customer 1
Transaction record, customer 1
Transaction record, customer 1
Header record, customer 2
Transaction record, customer 2
Transaction record, customer 2
Transaction record, customer 2
Header record, customer 3
Etc.

If there's any customer level logic that is performed on change in customer number, you'll need to make sure that all records for a given customer fall into the same sub process. This might be accomplished by having each process continue to read when it reaches its OBS limit until it reaches a change in customer number. Likewise, at the start of each sub process, the sub process reads but discards all records until it reaches a change in customer number whereupon it would start actually processing the records. Obviously, you're going to have to do some coding to support something like this. The OBS functionality that SAS provides doesn't allow one to read beyond the limit set by the OBS parameter. You'll need to write your own code here.

COMBINING THE RESULTS

In order to combine the results, we will simply use a DATA step with a SET statement specifying multiple input SAS data sets. Normally, if a series of input SAS data sets are specified in SET statement, the output SAS data set will have the records in the order in which they were read. For example, if there were four input SAS data sets, then our output file would be ordered as follows:

1. All the records from the first SAS data set in the order in which they were read.
2. All the records from the second SAS data set in the order in which they were read.
3. All the records from the third SAS data set in the order in which they were read.
4. All the records from the fourth SAS data set in the order in which they were read.

But what there were some type of key in each file and further what if each file were already in sort order? Wouldn't it be nice, especially if there were hundreds of millions of records, if we could preserve the sort order and avoid having to re-sort the data?

Well, actually, we can. There's a special form of a DATA step known as an interleave. Basically, the only difference between an interleave and a "normal" DATA step is the presence of a BY statement. For example:

```
DATA SAS_Out.Combined_Data_Set;
    SET   SAS_In.Data_In1
          SAS_In.Data_In2
          SAS_In.Data_In3
          SAS_In.Data_In4
          ;
    BY    Sort_Key_Var;
RUN;
```

VALIDATING THE RESULTS

The simplest validation is to run you job as is and save your output SAS data set. Then, run your new parallel processing version and again save your SAS data set. After you have the two output SAS data sets, simply run a Proc COMPARE. There should be no variable value differences, the record counts should be the same, and all columns should be present in both data sets. In other words, the files should be completely identical.

PUTTING IT ALL TOGETHER

My discussion here is all based on real world experience. Let's look at the overall transformation that my "inspirational" long running job went through, one that reduced run time from 9 hours to 3 hours.

The original, serially-processed version of my SAS job looked like this:

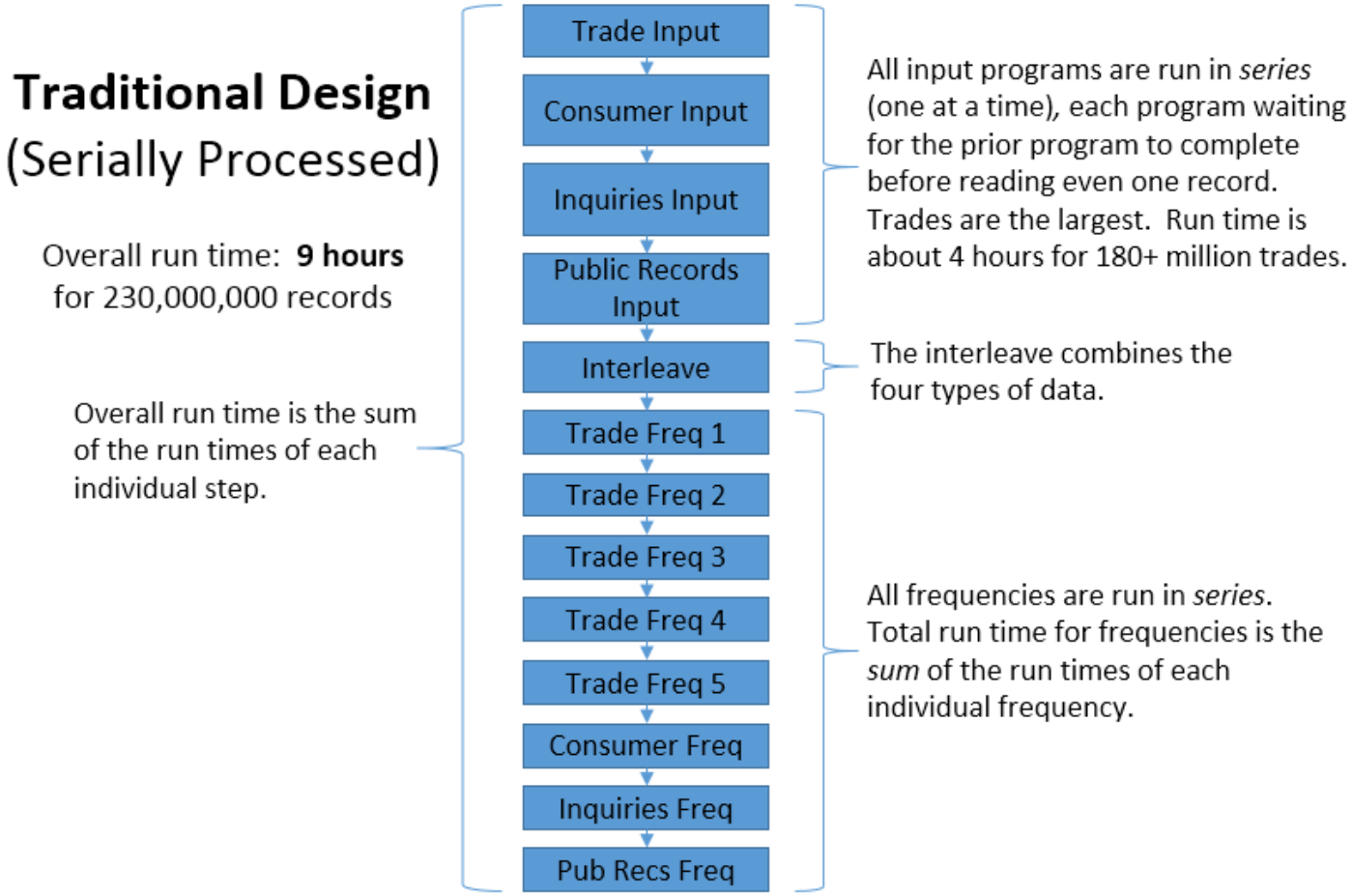

Figure 6. Original serially processed design.

Every step waited for every other previous step before it began execution. In the reading of our four types of input raw data, we read the four types serially, one type at a time, each subsequent type waiting for each prior type to completely finish. Total run time was 9 hours. Reading the largest file required four hours.

After redesign, our parallel processing version looks like this:

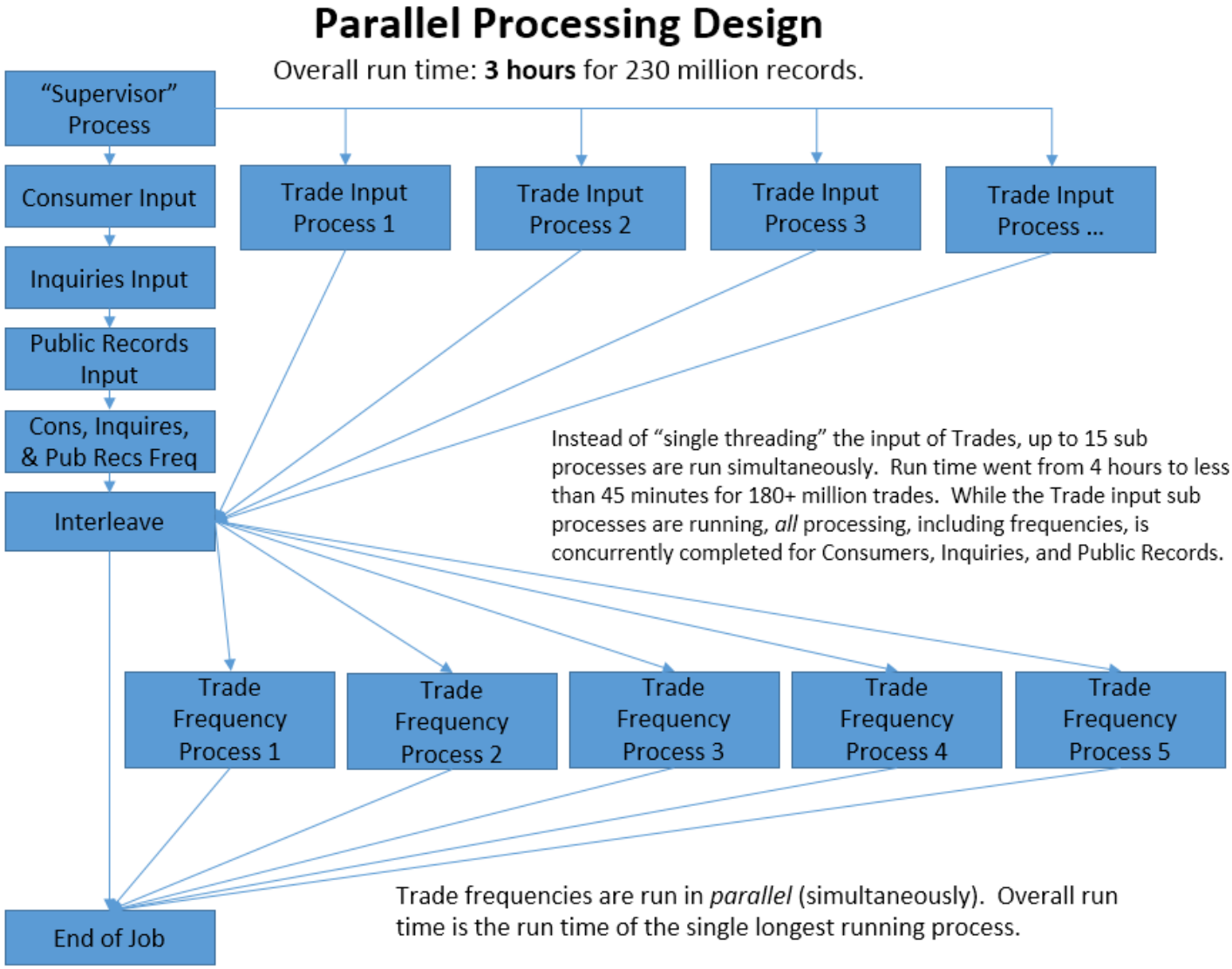

Figure 7. New parallel processing design.

In the re-designed, parallel processing job, the Consumer, Inquiry, and Public Records input programs – and their respective frequency checks – are all run as part of the supervisor process while the Trade inputs are run in up to 15 simultaneously executed subordinate processes. By the time the Trade inputs finish, processing for the other types of records has already been completed. The four data types are interleaved once all the Trade sub-processes complete, and then all frequency steps associated with Trades are run simultaneously. Run time went from 9 hours to 3 hours, with **6 hours saved per run.**

CONCLUSION

If one needs to improve performance, I can think of few SAS techniques more powerful than parallel processing, a technique that can be implemented with just the Base SAS product. To really do a make-over on your SAS jobs, you'll have to do some programming, but even just setting a couple of SAS options can make SAS procedures run significantly faster. You will need to master some SAS coding techniques in order to fully implement parallel processing. Ultimately though it is the overall design of your process flow from which you will gain the greatest performance improvements. You must lay out which portions of your processing are independent and may executed concurrently. Choose the jobs and job steps in which you intend to implement parallel processing well. Techniques, even powerful ones like parallel processing, are really only successful when combined with good, well thought out design. Parallel-processing increases significantly the complexity of a job and that job's consumption of resources. One should weigh carefully the costs vs. the benefits of parallel-processing – but there are cases where the extra work and complexity of parallel processing are worthwhile.

I'll leave you with this one last thought: If you've got a powerful multi-core SAS environment, why on earth aren't you making full use of it? Typically, analytical environments don't have high CPU utilization. Capacity is lying idle. If you've got critical but long-running jobs, put that capacity to work by employing parallel processing.

ACKNOWLEDGMENTS

I wish to thank Mr. Kirk Paul Lafler for his encouraging me to write this paper, Mr. Daniel Zidaru for his leadership of the process redesign we're currently undergoing and his permitting me to try the techniques herein described, and, lastly, I wish to thank my boss, Ms. Luz Torrez, for assigning me to do the process redesign work, work I have found both challenging and satisfying.

CONTACT INFORMATION

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

Jim Barbour
Experian Information Solutions
jim.barbour@gmail.com