

# Pipeline Parallelism Performance Practicalities

M. Michelle Buchecker, SAS Institute Inc., Chicago, IL

## ABSTRACT

Pipeline Parallelism is a great new feature of SAS 9. This presentation will describe what pipeline parallelism is and how it differs from traditional SAS operations. The purpose of pipeline parallelism is to increase your speed of your program and decrease your I/O operations. This paper will take you behind the scenes to show you what's going on and focus on what NOT to do that could actually cause your program to behave improperly.

## THE "GOOD OLD DAYS" - SEQUENTIAL EXECUTION AND INDEPENDENT PARALLELISM MULTIPROCESSING

In SAS Version 6 and earlier releases, in a single SAS program an individual step had to complete before processing of the next step could begin. This is what is known as sequential execution.

Starting in SAS Version 8, you could use MP CONNECT which is a component of SAS/Connect software to allow steps to execute simultaneously as long as no data were being shared. So for instance you could read a raw data file using a DATA step at the same time you read a different raw data file using a different DATA step. However if those DATA steps then each fed a PROC PRINT, the PROC step could not execute until it's corresponding DATA step had finished. This is called Independent Parallelism.

## WHAT IS PIPELINE PARALLELISM

Prior to SAS 9, steps that had dependent data

- had to write the data to disk between steps, typically to the WORK library
- the second step had to wait for the first step to complete before it could start processing that data.

Pipeline parallelism is when multiple steps depend on each other, but the execution can overlap and the output of one step is streamed as input to the next step. Piping is a SAS System 9 extension of the MP CONNECT functionality whose purpose is to address pipeline parallelism. The pipeline can be extended to include any number of steps and can even extend between different physical machines.

Pipeline parallelism is possible when Step B requires output from Step A, but it does not need all the output before it can begin. Because the data flows in a continual stream from one task into another through a TCP/IP socket, program execution time can be dramatically shortened.

For example, PROC SORT really does not need all the data output by the DATA step before it can start. Sorting can start with only two records and then continue by adding and sorting more records as they become available.

## WHAT'S IN IT FOR ME? - BENEFITS OF PIPING

The benefits of piping include

- overlapped execution of PROC and/or DATA steps
- elimination of intermediate writes to disk
- improved performance
- reduced disk space requirements.

## IS IT TOO GOOD TO BE TRUE? - CONSIDERATIONS AND REQUIREMENTS

- You must have sufficient CPU and I/O resources when implementing piping. If you do not have sufficient resources (like multiple processors) on a single machine, you can pipe between remote machines.
- Piping, like all scalability solutions, is most effective when the execution time of an application is substantial.
- Piping supports only single-pass steps (more on this below).
- Piping requires a SAS/CONNECT software license because it is part of the SAS/CONNECT product.
- Piping requires SAS 9. If you are piping across machines, both machines must be running SAS 9.
- Piping introduces a small overhead from doing a SIGNON, CPU overhead of additional processes and a complexity overhead to the application.

- The data passes sequentially through the socket and is not buffered in memory. So, after it is processed, it is no longer available for a second pass.

## SAS DOESN'T MAKE PASSES AT DATA IN MASSES - SINGLE PASS OF DATA

A limitation of piping is that it supports single-pass, sequential data processing. Because piping stores data for reading and writing in TCP/IP ports instead of disks, the data is never permanently stored. Instead, after the data is read from a port, the data is removed entirely from that port and the data cannot be read again. If your data requires multiple passes for processing, piping cannot be used.

Examples of SAS steps that process single-pass, sequential data:

- DATA step
- SORT
- SUMMARY
- GANT
- PRINT
- COPY
- CONTENTS.



Certain options cause the step to perform multiple passes of the data and, therefore, cannot be used with piping. These multiple pass factors are listed below.

Step	Option
DATA	<ul style="list-style-type: none"> <li>• KEY= option</li> <li>• POINT=option</li> </ul>
PROC PRINT	UNIFORM option

## DRATS! NONE OF MY MACHINES HAVE MULTIPLE CPUS

What if your machine only has a single processor and you attempt to run pipeline parallelism? The piping test case will probably take longer than the sequential test case. This is a result of contention for the I/O channel, the CPU, and memory. If you work in an environment such as this, piping can still work for you. Simply farm the work out to other machines!

## HOW DO I IMPLEMENT PIPING?

The piping functionality is specified in the form of the SASESOCK engine on the LIBNAME statement.

General form of the SASESOCK engine:

```
LIBNAME libref SASESOCK "port-specifier" TIMEOUT=time-in-seconds;
```

The TIMEOUT= option is optional and specifies the time in seconds that a SAS process waits to successfully connect to another process. The default value is ten seconds. More information on the TIMEOUT= options is discussed later in this paper.

port-specifier can be represented in these ways:

- ":explicit-port " specifies an explicit port on the machine where the asynchronous RSUBMIT is executing. It is signified by a hard-coded port number.

```
libname payroll sassock ":256";
```

You do not have to configure a port number in the SERVICES file in order to use it. Be aware that if the port that you specify is already in use, you will be denied access.

- ":port service" specifies the name of the service on the machine where the asynchronous RSUBMIT is executing. It is signified by the name of the port service.

```
libname payroll sassock ":pipe1";
```

The port service must be configured in the SERVICES file in order to use it. Be aware that if the port that you specify is already in use, you will be denied access. Your SERVICES file is typically found in a location where the

operating system is installed. For Windows NT it may be found in C:\WINNT\system32\drivers\etc\services. An example of what would be placed in your SERVICES file is shown below:

```
pipe1  13001/tcp
pipe2  13002/tcp
pipe3  13003/tcp
```

- "machine-name:port-number " specifies an explicit port number on the machine specified by machine-name.

```
libname payroll sasesock "orion.finance.com:256";
```

- "machine-name:port service" specifies the name of the service on the machine specified by machine-name.

```
libname payroll sasesock "orion.finance.com:pipe1";
```



When specifying a port on a remote machine you can only read, not write, from that pipe.

## EXAMPLES

In the first example the DATA step will output the data not to disk but to a TCP/IP pipe named pipe1. A PROC SORT step will then execute simultaneously reading from that pipe, sorting the data and creating a permanent SAS data set.

The two steps must be placed in two separate RSUBMIT blocks, or one RSUBMIT and the parent process. This is required so that the two steps can run simultaneously.

The first DATA step writes the results to the TCP/IP pipe. The job of the second step is to remove this data from the TCP/IP pipe. Because of the limited amount of buffer space in the pipe, the processes must run at the same time.

```
/* start session READTASK and execute first data step asynchronously to a pipe */
signon readtask sascmd='sas';
rsubmit readtask wait=no;
  libname outlib sasesock ":pipe1";
  data outlib.equipment;
    /* data step statements */
  run;
endrsubmit;

/* start session SORTTASK and execute second step which gets its input from a pipe */
signon sorttask sascmd='sas';
rsubmit sorttask wait=no;
  libname inlib sasesock ":pipe1";
  libname newlib 'c:\workshop\winsas\mpdp';
  proc sort data=inlib.equipment out=newlib.final;
    by costprice_per_unit;
  run;
endrsubmit;
waitfor _all_ readtask sorttask;

signoff readtask ;
signoff sorttask;
```

In the second example the DATA step will output the data to two different pipes named pipe1 and pipe2 based on some criteria of the data. One PROC SORT steps will then execute simultaneously reading from one pipe, while a second PROC SORT reads from the second pipe.

Once again the steps must be placed in separate RSUBMIT blocks so that they can run simultaneously.

The first DATA step writes the results to the TCP/IP pipe. The job of the second step is to remove this data from the TCP/IP pipe. Because of the limited amount of buffer space in the pipe, the processes must run at the same time.

```
signon readtask sascmd='sas';
rsubmit readtask wait=no;
```

```

libname outlib1 sasesock ":pipe1";
libname outlib2 sasesock ":pipe2";
data outlib1.equip outlib2.clothes;
/* data step statements */
  if type='EQUIPMENT' then output
    outlib1.equip;
  else if type='CLOTHES' then output
    outlib2.clothes;
run;
endrsubmit;

signon clthsort sascmd='sas';
rsubmit clthsort wait=no;
  libname inlib sasesock ":pipe2";
  libname newlib 'c:\workshop\winsas\mpdp';
  proc sort data=inlib.clothes out=newlib.final;
    by costprice_per_unit;
run;
endrsubmit;

signon equpsort sascmd='sas';
rsubmit equpsort wait=no;
  libname inlib sasesock ":pipe1";
  libname newlib 'c:\workshop\winsas\mpdp';
  proc sort data=inlib.equip out=newlib.final;
    by costprice_per_unit;
run;
endrsubmit;

waitfor _all_ readtask clthsort equpsort;
signoff readtask;
signoff clthsort;
signoff equpsort;

```

## ADDITIONAL CONSIDERATIONS

- The benefits of piping should be weighed against the cost of potential CPU or I/O bottlenecks. If execution time for a SAS procedure or statement is relatively short, piping is probably counterproductive.
- Remember that piping is a 1-way street. The data can only be written once to the pipe and read once from the pipe. In the first example we would not be able to leave off the OUT= option in the PROC SORT as that would cause an attempt to write back to the pipe we are reading from. Once the pipe has completed being written to and read from and the pipe closes (more discussion on this later in the paper), you may then use that same pipe again.
- If you have only a few I/O channels, I/O intensive code (like PROC SORT) will clog up the I/O channels.
- In the second example above both PROC SORTs are writing to the same physical path/disk drive. This could cause some slowdown as one step waits for the other to relinquish some I/O control.
- Ensure that each SAS procedure or statement is reading from and writing to the appropriate port. For example, a single SAS procedure cannot have multiple writes to the same pipe simultaneously or multiple reads from the same pipe simultaneously.
- You might minimize port access collisions on the same machine by reserving a range of ports in the SERVICES file.
- Be sure that the task reading the data does not complete before the task writing the data. For example, if the DATA step produces a large number of observations and PROC PRINT only prints the first few observations specified by the OBS= option, this can result in the reading task closing the pipe after the first few observations are printed. This would cause an error for the DATA step because it would continue to try to write to the pipe, which was closed.

For example:

1. DATA step opens port and begins writing data.
2. PROC step begins reading data.
3. PROC step finishes reading data and closes port.
4. DATA step can no longer write to port and generates an error message.

While the task that does the writing generates an error and will not complete, the task that reads will complete successfully. You can ignore the error in the writing task if the completion of this task is not required (as is the case with the DATA step and PROC PRINT example in the item). Alternatively, move the OBS=10 option to the DATA step so that the reader does not finish before the writer.

Ensure that the port that the output is written to is on the same machine that the asynchronous process is run on. However, a SAS procedure that reads from that port can run on another machine.

## TIMING

Be aware of the timing of each task's use of the pipe.

If the task that reads from the pipe

- opens the pipe to read and
- there is a delay before the task doing the writing actually begins to write to the pipe,
- the reader might time out and close the pipe prematurely.

By default, the reader waits 10 seconds to receive data or it will close the pipe.

1. DATA step opens port and processes for a long time; no data written out yet.
2. PROC step waits for data in port; gives up after value specified with the TIMEOUT= option.
3. PROC step closes port.
4. DATA step can no longer write to port and generates an error message.

Use the TIMEOUT= option in the LIBNAME statement to increase the timeout value for the task that is reading. This causes the reading task to "wait" longer for the writing task to begin writing to the pipe. This enables the initial steps in the writing task to complete and the DATA step or SAS procedure to begin writing to the pipe before the reader times out.

## PERFORMANCE GAINS

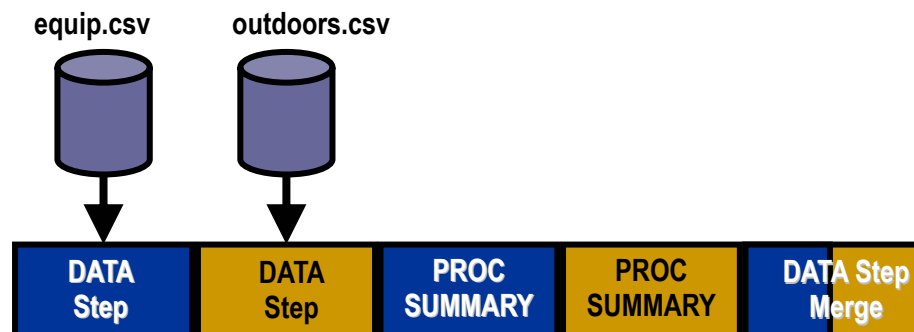
MP CONNECT, pipeline parallelism, and threading improve performance, but by how much?

Look at an example scenario to compare

- traditional sequential processing
- MP CONNECT + Piping
- MP CONNECT + Piping + Threading.

This example was executed using the following criteria:

- eight-way 900 MHz UNIX box
- two raw input files (~1.5G each)
- two DATA steps, two SUMMARY procedures, and a DATA step merge.



two raw input files

- two data steps to read input and calculate revenue and profit
- two PROC SUMMARYs to summarize by region and classify data by employee number
- final merge of projected and actual sales by region to determine those employees who met/exceeded goals.

Sequential implementation

- 1210 seconds

MP CONNECT and Piping (allowing for threading in the PROC SUMMARYs)

- 382 seconds (70% improvement)

The first iteration makes use of MP CONNECT to execute the independent steps in parallel and also to pipe from one step to the next. The two DATA steps are independent of each other and the two summaries are independent of each other.

Piping works well here because there is no need to persist the temporary data sets created by the two DATA steps and the subsequent PROC SUMMARY steps. Therefore, you can use piping to pipe the data from one step to the next. This

- enables overlapped execution of all of the steps including the final merge
- eliminates write to disk of intermediate results, which minimizes disk space requirements.

All steps start at the same time and can process data as it comes through the pipeline.

Final iteration takes advantage of the fact that PROC SUMMARY is threaded in SAS System 9. PROC SUMMARY turns threading off by default for BY-group processing. However, because your BY-groups are fairly large, you can turn on threading by adding the THREADS option to the PROC SUMMARY statement. This final implementation gives you the best performance with regard to elapsed time.

## CONCLUSION

Performing tasks in parallel using pipeline parallelism can dramatically reduce overall elapsed time for your program as well as decrease disk space. SAS/CONNECT provides this capability starting in SAS 9 of the SAS System. However, it must be used judiciously. Improper usage of pipes can result in increased I/O usage, slower elapsed time, or premature pipe closing.

## CONTACT INFORMATION

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

M. Michelle Buchecker  
SAS Institute Inc.  
180 N. Stetson Ave, Suite 5200  
Chicago, IL 60613  
Michelle.Buchecker@sas.com