Paper 269-26

SAS Multi-Process Connect: What, When, Where, How, and Why

John E. Bentley, First Union National Bank, Charlotte, North Carolina

Abstract

SAS Software® Version 8 supports parallel processing on symmetrical multiprocessor (SMP) systems. Part of SAS/CONNECT®, Multi-Process Connect® (MP CONNECT) exploits multiple processors on a host system by a process analogous to Version 6.12's RSUBMIT/ENDRSUBMIT approach to remote compute services. It allows the user to start a "remote" SAS session on each processor on the system and then submit code to each of those remote sessions; the results are gathered back into the originating SAS session. Although conceptually clear, certain questions remain. How does the program need to be structured? When should MP CONNECT be used? What impact does MP CONNECT have on other user's SAS sessions? This paper starts with a short review of SMP system architecture and then proceeds to explain what MP CONNECT is, why it's needed, and how and when to use it. The orientation is towards the UNIX environment. Users running SAS Version 8 or 8.1on multi-processor systems and experienced with SAS/CONNECT and remote compute services will find the paper most immediately useful.

Disclaimer: The views and opinions expressed here are those of the author and not those of First Union National Bank. First Union National Bank does not endorse, recommend, or promote any of the computing architectures, platforms, software, programming techniques or styles referenced in this paper.

Introduction

In many fields--banking, insurance, retail catalog sales, and government statistics to name only a few--data sets containing millions or tens of millions of records were common long before the term "data warehouse" was coined. They were called "Very Large Databases" then. Because of their size, they were stored on tape and processing was done on mainframe-class computers. Despite the best efforts of dedicated programmers, data access was not easy and processing time was measured in hours.

In today's disk-based data warehouse and data mart environments, we have easier access to massive amounts of data. It's quite common to work with several million or tens of millions of records. Unless highly specialized software is used, however, processing time is still measured in hours. SAS Version 8 changes that, though.

Version 8 of SAS/CONNECT includes the Multi-Process Connect facility (MP CONNECT) to exploit multi-processor capabilities of symmetrical multiprocessor (SMP) and massively parallel processor (MPP) systems "by allowing parallel processing of self-contained tasks and the coordination of all the results in the original SAS session". (SAS/CONNECT User's Guide, On-line version.) MP CONNECT runs in all parallel environments, but this author has experience with MP CONNECT only in UNIX so all examples, references, and benchmarks presented here will be UNIX-oriented.

Without MP CONNECT, SAS Software, including Versions 8 and 8.1, executes *sequentially* one step or procedure at a time on a single processor. (SPDS is an exception.) Even when extracting from parallel relational database engines, the results of the SQL query must be returned to a single processor where the remaining SAS code executes sequentially.

MP CONNECT is the key for SAS users to enter the world of parallel processing. With it, the DATA step, PROC SQL, and some PROCs such as SORT can execute in parallel on SMP UNIX servers like Sun Enterprise Servers, the HP-9000 Servers, Compaq's AlphaServers, and IBM's RS/6000s as well as MVS and CMS mainframe systems. Parallelization results in increased in program efficiency. As programs run faster, more timely information is provided to decision makers and that directly affects profitability. Return on investment in the computer system is increased because the computer that's running parallel SAS programs will be able to handle more work in the same amount of time.

Parallel Processing

The authors of Parallel Systems in the Data Warehouse compare parallel computing to building a house. The house corresponds to the problem to be solved and workers are the CPUs. There are many different tasks involved in building the house, and to get the job done efficiently the workers must work on the separate tasks at the same time but in the proper order.

The term *parallel* in the computing context used in this paper refers to simultaneous or concurrent execution—individual tasks being done at the same time. Without MP CONNECT, processing is restricted to s*equential dependency*, which is the condition in which Task B can't begin until Task A is finished.

Consider a simple SAS program that reads in a flat file to create a SAS data set, transforms the variables, sorts the data set, and then calculates grouped summary statistics. PROC SORT doesn't begin until the DATA step ends, and PROC MEANS won't run until the sort finishes. The total execution time for the job is the sum of the times for each of the separate steps.

MP CONNECT provides *independent parallelism*. This occurs when there are no dependencies between tasks and they can therefore be run concurrently. Suppose our example has two flat files that need to be read, manipulated, sorted, and then merged before calculating group summary statistics.

If we read in and manipulate both data sets at the same time and have the output data sets sorted at the same time, then we've implemented independent parallelism for this part of the processing. The processes of gathering the sorted data sets into a single data set and calculating the statistics are done sequentially. Execution time is calculated by adding the time required for the lengthiest series of parallel processes plus the time needed for sequential processing.

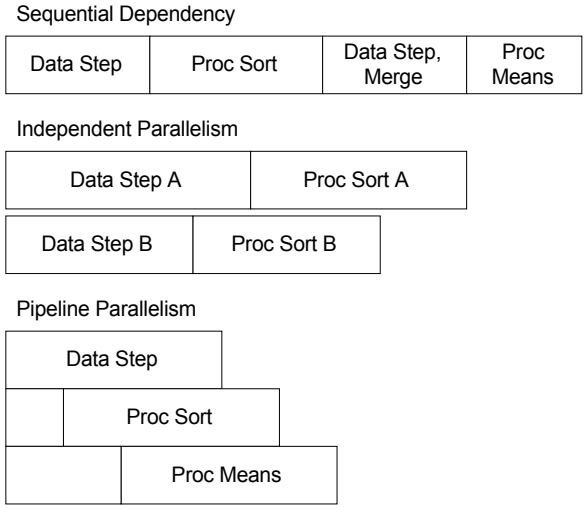
Program structure is critical in parallel programming. In our example the merge step and summary statistics calculations

were be done sequentially. But if the business user was comfortable using grouped statistics, the statistics could have been calculated in parallel, the results output to a data set, and then those data sets merged and used as input for final calculations. In another case, we could have calculated the summary statistics in parallel if none of the variables for which statistics are needed were created in the merge step and none were shared between the data sets

The most efficient form of parallel processing is *pipeline parallelism*, but this isn't provided by SAS yet. Pipeline parallelism is possible when Task B requires output from Task A but it doesn't need all the output before it can begin. In our example, PROC SORT really doesn't 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. PROC MEANS can start its calculations as soon as two records of the same group are available from PROC SORT.

Because the data flows in a continual stream from one task into another, program execution time can be dramatically shortened with pipeline parallelism. Execution time is calculated by adding the time required for the final process to complete plus the time that process had to wait before beginning.

Figure 1. Types of Program Execution



Why use Symmetrical Multiprocessor (SMP) Systems?

Probably the most widely used parallel-capable hardware architecture today is the *symmetrical multiprocessor* (SMP) system, an incredible advance over single CPU systems. In an SMP box, multiple CPUs and associated resources run under the control of a single instance of the operating system. Memory and disk resources are shared. All major computer manufacturers make UNIX-based SMP machines, but SAS users may be most familiar with those from Sun and Hewlett-Packard.

Decision support systems (DSS) are well suited for parallel processing on SMP platforms because of the nature of the data flow. DSS refers to the extraction, analysis, and presentation of data from historic databases to enable knowledge-based decision-making. OLAP, on-line analytical processing, is a subtype of DSS. An example is examining weekly sales over the past quarter by product, city, and state. SAS/EIS® is software for building Executive Information Systems, also a type of DSS.

In a DSS application, data is read, manipulated, analyzed, and output. The data may be read from multiple source data sets, relational databases, or flat files. Business rules may require that different manipulations and transformations be applied to different data sets. The analysis, reporting, or presentation requirements may specify detail based on location, time, product or some other categorical or grouping variables.

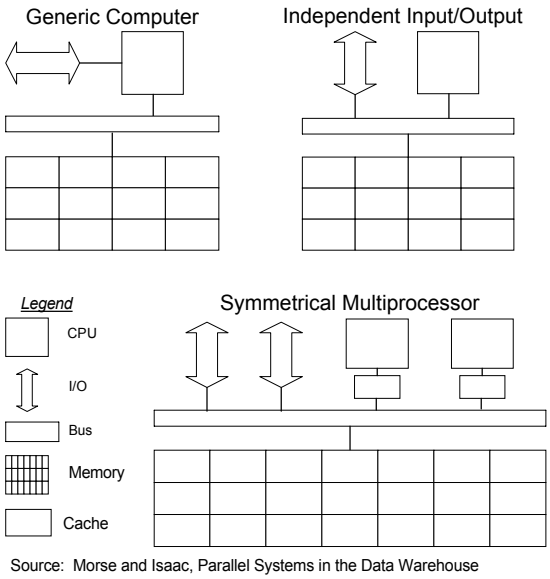
When any of these situations exist, we can often process in parallel by splitting the program into independent pieces for simultaneous execution, thereby reducing overall execution time. Ideally, and not considering system overhead, the execution time for a parallel program will equal the execution time for the same program run sequentially divided by the number of processors being used. That is, if a program takes an hour to run sequentially then splitting it among four processors will reduce the run time to 15 minutes.

Background: SMP Hardware

In the original single-CPU systems, the processor was forced to assist in the data input/output operations. While I/O processing was taking place, the CPU couldn't perform calculations and vice versa. Data could only move between the memory and I/O interface via the CPU. The first evolution was to offload I/O to a separate controller. This controller directly interfaces with memory without assistance from the CPU. The CPU only contacts the I/O controller in short bursts to tell it where in memory to place or fetch data from the disk.

When the CPU and I/O controller are simultaneously active, a simple form of independent parallelism is taking place—different, separate operations are occurring concurrently. An important point here is that both components are accessing different portions of the same memory space. This means that a *shared memory architecture* is in place.

Figure 2. Evolution of the SMP system



Source: Morse and Isaac, Parallel Systems in the Data Warehouse

Once it became possible for more than one device—CPU or I/O controller—to access the memory at the same time, the

obvious next step was to allow a more than one processor or I/O controller to share a single, larger memory space.

In a *shared memory* architecture, the operating system enforces a strict separation between the concurrent operations. Each of the components—CPU and controller—has a portion of the memory assigned to it and the operating system ensures the logical separation of the assigned spaces so that one component cannot step on the memory allocated to another.

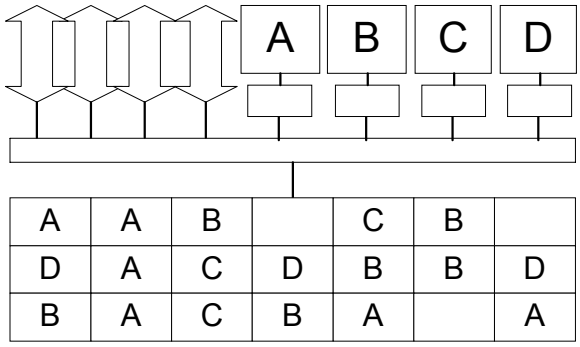
It is important to recognize that in shared memory architecture the CPUs and controllers share the same path or *bus* to the main memory. This means that only one CPU or controller can be reading or writing to memory at a time. While the bus speed is fast, there is still a finite data transfer rate, called *memory bandwidth*. The operating system manages access to memory, and a technique called *caching* reduces memory contention and improves overall performance.

Today, the most common way that SMP is used is for *multitasking*. While one CPU is executing one program, another CPU executes another. Because the operating system keeps the processes separate and invisible from each other, it appears to each program (and user) that it is running on a single-CPU sequential processing machine. This is what allows SAS to run so effectively on multi-CPU servers. The operating system allows each processor to multitask as well, swapping programs in and out so that each processor can be working on more than one program. But the number of programs actually running at once is directly proportional to the number of CPUs.

But what if we have a four CPU system and only one big production job is running? Three of the CPUs are, in effect, being wasted. Clearly, while multitasking is useful it doesn't inherently take advantage all of the power of the SMP architecture. The logical solution is to let all of the CPUs cooperate on a single program (or application). The goal is to make the program execute faster, so it is reasonable to assume that by putting all four CPUs to work on the same program, the program will execute four times faster. In the same manner, by better use of the system's resources a problem four times as large should complete in the time it takes a single-CPU to solve the original problem. Although there is a little overhead lost to the operating system, this *scalability* is a basic fact of parallel processing.

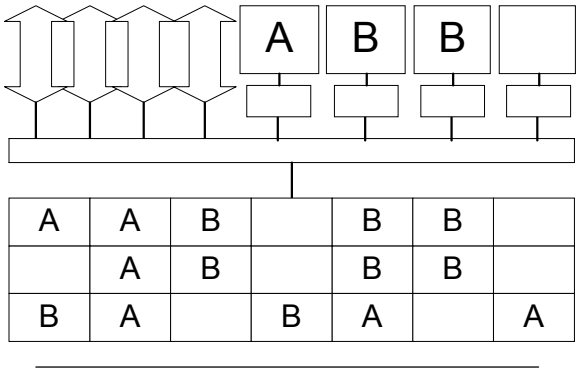
Figure 3 shows multitasking on a four-processor SMP system, and Figure 4 shows multitasking and parallel processing on the same system. In Figure 3, portions of memory have been assigned to each of the four jobs being run.

Figure 3. Multitasking on an SMP System



In Figure 4, only two jobs are running: one CPU is running one job and two are working on another. The fourth is idle. Notice that the single CPU has its own memory addresses but the cooperating CPUs share the same memory space. The logical separation between the two jobs is being maintained by the operating system. Allocation and control of the CPUs, however, is through both the operating system and application software.

Figure 4. Parallel processing on an SMP system



SMP Strengths and Weaknesses

The commercial success of SMP systems is due largely to their ability to multitask existing (legacy) sequential applications and programs with little or no modification. They do this by not running the application in parallel; in effect, the application thinks it is on a single CPU machine and executes sequentially just like it does on an ordinary single-CPU machine. While one application is executing on one CPU, another application is executing on another CPU. Given the large amount of memory installed on SMP systems—often a gigabyte (GB) or more plus a megabyte or more of cache memory—these systems are highly effective at supporting many users running many applications.

To run in parallel, an application must be specifically written to support it. The point was made earlier that the operating system and the application cooperation in controlling CPUs during parallel execution. Writing parallel application software is no trivial matter, and this at least partially explains why it was Version 8 before SAS released MP CONNECT.

Most relational databases have long had parallel versions of their software. Because a data warehouse is essentially a massive database, it is important that the RDBMS support *multiple threads* so that the CPUs can cooperate as in Figure 4 to speed up query execution. This is accomplished by breaking the application code into *parallel* and *sequential regions*. Multiple processors work on the code in the parallel regions but only one works when the program is in a sequential region.

In effect, this is what MP CONNECT allows SAS programmers to do: write a program containing parallel and sequential regions. MP CONNECT provides a middle tier between the local and remote environments that protects the programmer from the complexity of interacting with the operating system.

SMP Scalability

The most significant limitation of the SMP architecture is its *scalability*. In short, scalability is the improvement to be had by adding an additional processor to the system. The goals of adding more processors are to (1) execute a job faster (*speed up*) or (2) increase the size of the job that can be completed in a given amount of time (*scale up*).

To measure scalability, we first calculate/estimate a ratio of the job size divided by completion time for a single CPU machine. The hope is that the ratio will scale in proportion to the number of processors. We want to see that doubling the number of processors will double either the speed or the scale of the machine—ideally, it will do both.

Although memory management impacts an SMP machine running in parallel, bus bandwidth limitations are the more important factor that constrains scalability. It is not the amount of memory that does not scale; it's the access to that memory—the bus bandwidth—that does not scale. Both of these factors will be presented next.

SMP Memory Management Limitations

There is a major disconnect between the speed with which a CPU can process data and the speed with which data can be fed to it. Dynamic random access memory and all its variants (SDRAM, EDRAM, and CDRAM) simply can't feed data to the CPU fast enough. A partial solution is to create a *cache* of very fast static random access memory between each CPU and main memory. It's expensive, but better matched to the speed of the CPU.

When the CPU makes a request, the hope is that the data being requested will already reside in the cache, producing a cache hit. If it does, the processor can continue without having to wait for the fetch from main memory. Based on the which cached data is used, the system is constantly refilling the cache with it's "best guess" data without waiting for the CPU to request it. If the data isn't in the cache, though—a cache miss—then the CPU must wait for main memory to be accessed and performance suffers accordingly. Luckily, caching techniques have been refined to the point where hit rates of 95-99% are common for many database applications.

Caching clearly improves performance. At the same time, however, it creates another problem that degrades performance. If each CPU has its own copy of data in its own cache, what happens when one of the CPUs changes a value in its cached copy and then that value is passed through to be written in main memory? The other CPUs must be notified or they will be using dirty data and likely produce an error.

The solution for maintaining *cache coherence* requires that each CPU constantly "listen" to the bus for messages from other CPUs that they are writing to memory. When that happens, the other CPUs have to refresh their cache. As the number of CPUs in the parallel SMP system increases, each CPU must spend more time "bus sniffing" and flushing and refilling its cache.

SMP Bandwidth Limitations

Even in the most cache coherent SMP system, access to the physically shared memory—the bus—is a bottleneck. Once the available bandwidth has been fully utilized, no matter how efficient the operating system is at managing traffic, adding another CPU will not contribute to scalability but will in fact reduce performance. Why? Each additional byte of memory that the new CPU retrieves will come at the expense of a memory reference needed by another CPU.

Picture in your mind hogs at a feeding trough. For each hog added, the portion each one gets is smaller---the law of diminishing returns. Once all the space around the trough is taken, no more hogs can be fed. The physical limit of the trough has been met.

Today, SMP architectures are generally considered not scalable for parallel processing to over twenty-four processors. Most observers agree that SMP systems are more appropriate to hosting 200-gigabyte decision support data marts than terabyte-sized data warehouses.. All SMP vendors are working feverishly to raise the scalability threshold, however. In September 2000, IBM demonstrated near linear scalability in upgrading a 32-processor NUMA-Q system (a sort of massively parallel clustered SMP system) to 48 and then 64 processors while running against a 300 gigabyte DB2 Universal database.

MP CONNECT and Multiprocessing

MP CONNECT is part of the SAS/CONNECT product. But instead of running a SAS session on a remote host, it runs a SAS session on a different CPU on the same host as the local SAS session. This allows a SAS program to exploit multi-processor capabilities of symmetrical multiprocessor (SMP) and massively parallel (MPP) systems. It eliminates the need for executing a spawner program or a TELNET session on the remote host, and it also eliminates the need for a script file on the local host and the need to configure the remote access method.

Although MP CONNECT's syntax is simple and straightforward, the programs must be structured with parallel and sequential code sections—encapsulated—to take advantage of parallel architectures. Because the parallel sections must be self-contained, converting complex production jobs this may require a significant redesign and redevelopment effort. The ancient art of flowcharting can be a big help.

MP CONNECT is limited physically only by the number of CPUs available, but logically there are no limits on the number of SAS sessions that can be spawned. Think about a computer with six processors—a six-way box. The "local" SAS program is run using one of the processors, but if the program itself can be divided at some point into six tasks that can be executed independently of each other, such as sorting six data sets, there's no reason why the other five processors can't be used. The "local" SAS session initializes an additional five "remote" SAS sessions, coordinates their execution, and gathers the results of all the remote sessions back into the local session.

MP CONNECT – Considerations

At SUGI 25 in "Multiprocessing with Version 8 of the SAS System", Cheryl Garner of the SAS Institute listed three questions that need to be answered to determine if a particular program or application can benefit from MP CONNECT. This author adds a fourth question.

1. Can the data source(s) be split so that they can be processed separately?
2. Can the program be split into multiple SAS sessions that can run at the same time and not depend on each other or access non-shared resources, such as output data sets?

3. Can you get a good return on your investment of programmer time needed to write a parallel program vs. the machine time needed to run the program sequentially?
4. What is the impact to other users if one program uses multiple processors simultaneously?

There are no easy answers to these questions because there are many variables that must be considered. For questions 1 and 2, the go/no-go decision is based on the degree of difficulty for doing it, which in turn is based in large part on the expertise of the programmer. The degree of difficulty is also based on whether we are modifying an existing program or writing a new program—it may be more difficult to modify an existing program, determined in part by the presence or absence of program documentation and the use of macro programming. Another factor is whether or not the original programmer is doing the work and how long ago the program was written, that is, how well the programmer remembers why things were done the way they were.

If we assume that the answer is "yes" to the first two questions, then question 3 is the key. It may not be a good return on investment to spend six hours writing (or modifying), testing and debugging a sequential program run monthly if the speed-up is only 20 minutes. If, however, the usage load on the computer is great enough, the resulting scale up of the box may warrant the time and energy.

How MP CONNECT Works

A sequential SAS program is made up of a series of DATA steps and PROCs executes sequentially, requiring that the current step or procedure finish before the next one can begin. A key concept related to sequential processing is *synchronous execution*. With 6.12's Remote Compute Services, even though two processors are used—the local and the remote) the sessions executed synchronously—one at a time--because code bracketed between SUBMIT and ENDSUBMIT run on the remote host had to complete before control returned to the local host and it could continue processing. Only one host could execute at a time and the hand off from one to the other had to be synchronized.

MP CONNECT and parallel processing is based on *asynchronous*—simultaneous—execution. Asynchronous Remote Compute Services were implemented with SAS Version 7. To execute asynchronously, the program must have self-contained blocks of code encapsulated by SUBMIT-ENDSUBMIT. The way it works is that after the first encapsulated block is submitted to run, control immediately returns to the local host. This way, the local program continues to run, submitting more encapsulated blocks of code or local code until told to wait for the remote processing to finish or the local program completes.

As noted earlier, a parallel program can have multiple parallel and sequential regions and most PROCs don't run in parallel. In a complex program, this could translate into a sequential regions followed by multiple parallel regions followed by sequential followed by parallel and on and on. The structure of an analytical or reporting program will largely be dictated by the necessity of having the results of parallel execution available for sequential processing in a PROC or final DATA step. A data transformation program (or a reporting program), though, could perhaps be run entirely in parallel.

This author's experience is that time used to make a simple flowchart can be time well spent. (An unpublished draft of Garner, 1999, "Multiprocessing" also illustrates the utility of flowcharting.) Figure 5 illustrates one usage that combines data from multiple datasources: SAS data set, flat file, and relational database. For each, the data must be sorted before merging. On a four-way system, MP CONNECT allows all four data sets to be read and sorted simultaneously.

Figure 5. Speeding up a Sort and Merge

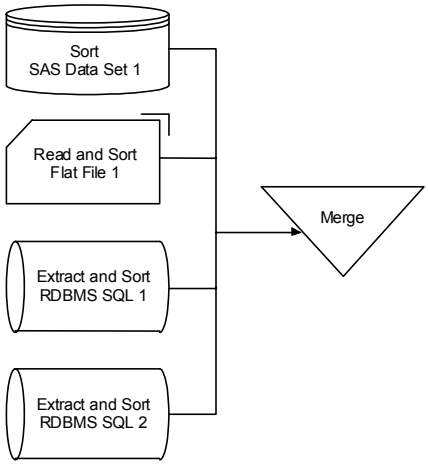
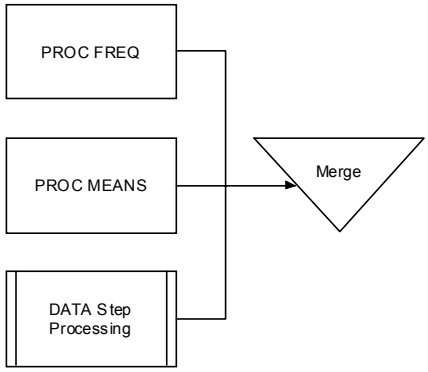


Figure 6 shows another usage that runs multiple PROCs at the same time that a DATA step is running.

Figure 6. Speeding Up Data Summarization



A Sample Parallel SAS Program

Code Sample 1 is a simple program written for MP CONNECT. It sorts three existing SAS data sets, merges them into one, then calculates descriptive statistics and prints the output. If these data sets are large, then the speed up and scale up gained here will be substantial.

Code Sample 1. An MP CONNECT Program

```
options autosignon=yes sascmd='sas';

libname cards '/prod/data/credit_cards';
libname loans '/prod/data/home_loans';
libname loans '/prod/data/auto_loans';
libname common '/saswork';

rsubmit process=loans wait=no;
  libname loans '/prod/data/home_loans' ;
  libname common '/saswork';
  proc sort data=loans.may out=common.loan_data;
    by market district;
  run;
endrsubmit loans;

rsubmit process=cars wait=no;
  libname loans '/prod/data/auto_loans';
  libname common '/saswork';
  proc sort data=cars.may out=common.car_data;
    by market district;
  run;
endrsubmit loans;

proc sort data=cards.may out=common.cards_data;
  by market district;
run;

waitfor _all_ loans cars;

rget loans;
rget cars;

data combined;
  merge common.loan_data common.car_data
        common.cards_data;
  by market district;
  exposure=sum(homeLoanTotal,carLoanTotal,
                cardBalanceTotal);
run;

proc means data=combined;
  by market district;
  var exposure homeLoanTotal,carLoanTotal,
      cardBalanceTotal;
run;
```

MP CONNECT Syntax and Usage

System Options

Using AUTOSIGNON=Yes, the RSUBMIT statement and any options specified with it are automatically executed without an explicit SIGNON. At the same time, any SAS/CONNECT global settings in effect are also executed. Because sign on is automatic, all valid SIGNON options are available for RSUBMIT. The default action is to sign off automatically when ENDRSUBMIT is encountered.

Used with AUTOSIGNON=Yes, the SASCMD=<*'command'*> system option specifies the SAS command used to invoke a "remote" SAS session on another one of the system's CPUs. SASCMD= can be also specified as an option to the RSUBMIT statement. When used in both places, the value specified with RSUBMIT command takes precedence over the value specified as a system option.

The value specified by SASCMD= does not necessarily have to be the usual executable command used to invoke a SAS session—"SAS". If you need to execute additional host commands prior to the SAS invocation, such as setting host environment variables, the SAS/CONNECT User's Guide says that you can write a script that contains your host commands followed by the SAS invocation, and then specify this script as the SASCMD= value.

Notice in Code Sample 1 that there are no quotation marks around the value passed to AUTOSIGNON but that the value passed to SASCMD is quoted. If a quoted value is passed to AUTOSIGNON, no error will be returned but remote SIGNON will fail to complete. Passing a non-quoted string to SASCMD= will return an error.

RSUBMIT Command

The RSUBMIT statement is the key to remote compute services. All SAS code bracketed between RSUBMIT and ENDRSUBMIT is executed remotely on a different processor, not in the local session. The way it works is conceptually similar to client-server processing but physically the sessions are executing on different processors in the same machine.

Each environment exists independently from the other but not in total isolation. As Code Sample 1 shows, each environment has access to the same DASD directories but a separate LIBNAME statement must appear in each environment even though the LIBNAME can be the same. Even though the statements execute in parallel (asynchronously) in multiple environments, all results and output can be gathered back to the local SAS session log and processing can continue either sequentially or new parallel sessions can be started. The RGET and RDISPLAY commands, covered later, retrieve and view the remote logs and output.

Two options are important for MP CONNECT's RSUBMITs. The first is used to assign an identifier to a remote session.

PROCESS=<remote session identifier>
REMOTE=<remote session identifier>
CONNECTREMOTE=<remote session identifier>

PROCESS= was added in Version 8 as an RSUBMIT option to allow differentiation between an MP CONNECT SIGNON (PROCESS=) and a SIGNON to a remote host on a remote host (REMOTE=), but either can be used. Commands such as RGET and RDISPLAY reference a specific session by using the remote session identifier.

The second option specifies whether or not remote submits are to be executed synchronously or asynchronously.

WAIT=<value>
CONNECTWAIT

The valid values for WAIT= are

YES | Y	execute RSUBMIT synchronously (default)
NO | N	execute RSUBMIT asynchronously

WAIT=NO will usually be the desired choice so that the local session regains control immediately to continue local processing or to RSUBMIT to other remote sessions.

If running MP CONNECT interactively or from a windowed environment and WAIT=NO is specified, then a third option will be useful.

MACVAR=<value> | CMACVAR=<value>

MACVAR=<value> allows you to use %SYSRPUT to test the status of a remote session to determine whether processing has completed or is still in progress. The <value> specifies

the name of the macro variable associated with this remote session. Importantly, the value of the macro variable is not set if the RSUBMIT command fails due to a syntax error. Otherwise, the value of the macro variable is set at the completion of the RSUBMIT command to one of the following:

0	The RSUBMIT is complete
1	The RSUBMIT failed to execute
2	The RSUBMIT is still executing

WAITFOR Command

In Code Sample 1, three sessions are run concurrently, two remotely and one locally. Identified by the PROCESS= options, "Loans" sorts the home loan data set remotely and "Cars" sorts the auto loan data set remotely. The local session sorts the credit cards data set. Loans and Cars are processed asynchronously remotely at the same time as the credit cards data set is being sorted locally.

We want to merge all three sorted data sets into a single data set, but MERGE requires sequential processing and so can't start until all sorted three data sets are available. MP CONNECT uses the WAITFOR statement to tell the local session at that point to wait for the remote sessions to complete before proceeding. As the SAS documentation puts it, "The WAITFOR statement allows you to wait for one or more asynchronously executing tasks before returning control to the local session."

The WAITFOR statement must include either the _ANY_ or the _ALL_ specification and a list of the processes (the SAS documentation refers to them as tasks) to be waited on. _ANY_ causes the local session to suspend execution until any of the specified sessions complete. _ALL_ does just what the name suggests, suspending the local session until all the sessions listed complete. Conceptually, behind the scenes SAS monitors a MACVAR-like value and doesn't proceed until a value of 0 is returned for each session.

An option that may be used in conjunction with _ANY_ and _ALL_ is TIMEOUT=<seconds>. TIMEOUT allots the maximum time in seconds for asynchronous task processing to complete. If the tasks do not finish processing in the time allocated for them, then task processing is terminated. If the tasks finish processing before the TIMEOUT time is reached, the WAITFOR returns control to the SAS session.

RGET Command

While the remote sessions execute in parallel, their log and output spool to disk until retrieved into the local environment. If the local session ends before the spooled log and output are retrieved, then they are deleted.

The RGET command causes the spooled log and output to be retrieved. The syntax is

RGET <remote session identifier>;

The remote session identifier is that assigned by PROCESS=. Only one identifier is allowed per RGET command, so one RGET will be needed for each remote session.

It's important that the WAITFOR command precede RGET. The obvious reason is that the log and listing will be more readable. The more important reason, though, is that if RGET executes while an asynchronous remote submit is still in progress the spooled log and output statements are immediately retrieved and merged into local log and output windows, spooling halts of that process halts, and the remote process switches to synchronous (sequential) mode. That is, local processing stops until the remote submit(s) have completed. If multiple remote processes are running and there are dependencies later in the program's local code, such as a merge, then the results are unpredictable.

To avoid this possible switch to synchronous mode in a windowing environment local session, use the MACVAR option in the RSUBMIT statement. This allows you to check the progress of an asynchronous remote submit without causing it to execute synchronously. Alternatively, use the RDISPLAY command described in the next section.

RDISPLAY Command

As previously noted, when an asynchronous remote submit executes, the log and the output are not merged into the local log and the output windows; instead, they are spooled for retrieval at a later time. In a windowing environment local session, RDISPLAY allows you to view the spooled log and output statements created by the remote process. RGET must be still be used to actually merge the spooled and local statements. The syntax is

RDISPLAY <remote session identifier>;

LISTTASK Command

Like RDISPLAY, LISTTASK is used to monitor the status of remote sessions submitted from a windowing environment local session. LISTTASK will report only processes that are still active or that have completed but not specified in a WAITFOR statement. As soon as a remote process is started, it is added to an internal tracking list. If a process is later specified in WAITFOR, however, it is removed from the tracking list. LISTTASK displays information such as the process name and its current status. The syntax is

LISTTASK _ALL_ | <remote session identifier>;

Performance Tests

Three different production simulations were run on two different UNIX symmetrical multiprocessor systems, one IBM and the other HP. On each platform the default installation configuration of SAS Version 8.1 was used. Table 1 lists the key components and specifications of each system.

Table 1. Test Systems Configuration

System	Components
IBM RS/6000 XP Silver node	• Processors: 4 each 332 MHz PowerPC 604e • Level 2 cache: 256K • Memory: 3 GB • Internal bus: 8 GB/sec theoretical max. • Data bus: Fast-Wide SCSI, 16-bit, 10MHz, 20 MB/sec/card • Operating System: AIX 4.3.1
SAS on RS/6000	• Version 8.1 TS1M0 • -mvarsize 32K • -msymtabmax 4MB • -sortsize 64MB • -memsize 128MB • -maxmemquery 6MB
HP 9000	• 6 each 240 MHz PA-RISC PA8200 • Level 2 cache: 1 MB • Memory: 4 GB • Internal bus: HyperPlane Crossbar, 15.36 GB/sec theoretical max. • Data bus: Fast-Wide SCSI, 16-bit, 10MHz, 20 MB/sec/card • Operating System: HU-UX 11

System	Components
SAS on HP 9000	• Version 8.1 TS1M0 • -mvarsize 32K • -msymtabmax 4MB • -sortsize 48MB • -memsize 64MB • -maxmemquery 6MB

Different data sets were run on each platform. All data sets were compressed and not indexed. On the IBM RS/6000, one group of three data sets was representative of those used in production jobs that, for example, score customer households for their propensity to purchase a new product. A second group of three data sets was used to test sorting and generating statistical output. On the HP 9000, six test data sets typical of daily transaction-level data sets used to produce summary reports were used.

Table 2 describes the test data sets. *Vars* is the number of variables in each data set; *Obs* is the approximate number of observations in each data set; *Size* is the approximate size in bytes of each data set.

Table 2. Test Data Sets

Platform	Purpose	Data Set Description (per data set)		
		Vars	Obs	Size
RS/6000	Score 3 data sets, generate stats, output 3 flat files containing scores	25	~9M	~900MB
RS/6000	Sort 3 data sets with the OUT= option. Generate frequencies and stats from each.	14	~11M	~500MB
HP 9000	Read 6 data sets, manipulate and transform data, sort, interleave the final data sets, generate reports, output a data sets and a flat file.	114	~1.8M	650 MB

The tests were run as separate batch jobs, submitted from the UNIX command line. Testing was done on weekends to simulate late night production runs when there are very few OLAP users on the system.

In each test, MP CONNECT was used to run two remote sessions in addition to the local session. For the RS/6000 tests, each of the three sessions processed one data set. Modifying the programs was mostly copy and pasting to create encapsulated parallel code sections.

In the HP test, each processor manipulated two data sets and then all six transformed data sets were combined into a final data set in the local session. Modifying the sequential program for parallel execution required a moderate amount of coding changes that substantially altered the flow of the program.

Table 3 shows the test results. *Time* shows the total elapsed time—real time—in hh:mm:ss for the tests to execute.

Table 3. Test Results

Platform	Task	Sequential Time	3-way Parallel Execution Time
RS/6000	Scoring, stats, and flat file creation	0:46:04	0:17:33
RS/6000	Sort, freqs, and stats	0:30:55	0:10:38
HP 9000	Report and data set creation	0:21:54	0:06:04

Writing SAS Programs for Parallel Processing

The difficulty of writing a parallel program or modifying an existing program for parallel execution is, in this author's short experience, partially dependent on the degree of "elegance" in programming being sought. MP CONNECT syntax and implementation is simple and straight-forward, but without a clear plan, a complex parallel program can turn into some of the worst spaghetti code ever seen.

It's temptingly easy to copy and paste existing code segments and then encapsulate them with RSUBMIT-ENDRSUBMIT, but a program of a few hundred lines with numerous DATA steps, LINKs, GOTOs, and PROCs can quickly balloons to a thousand lines of what looks like unmanageable repetitious code, depending on the number of remote sessions being used. That creates serious problems for debugging and testing and program modification and maintenance.

Documentation, especially program comments, takes on added importance. As was mentioned earlier, flowcharting makes parallel programming much easier to plan and implement. A visual representation of the process clearly identifies which sections must be sequential and which can be parallel.

When modifying an existing sequential program, it's highly important to fully understand what the program is doing and how it is doing it so that the most efficiency can be gained during the modification process. Do not simply copy, paste, and encapsulate inefficient code. Modifying a sequential program for parallel execution provides an opportunity to review the code for efficiency (and accuracy!) and to write program documentation that might have been earlier put off.

Using macros and %INCLUDEs are key coding techniques for reducing the number of lines of code that must be written and maintained. Good programming style, resulting in improved readability, is also critical.

MP CONNECT Programming Tips

- Remote sessions and local sessions exist independently of each other. System options, LIBNAMES, filerefs, and macro variables must be specified for each session.
- Each remote session creates it's own WORK directory, so remote and local sessions can't share work data sets without some clever coding.
- Macro programs must be made available in each session, either by coding, %INCLUDEing, or the SASAUTOS system option.
- Remote sessions do not persist. The default is for the session to SIGNOFF when ENDRSUBMIT is

encountered. It doesn't hurt, though, to use SIGNOFF as a coding technique to make the code more readable.

- A program can move into and out of parallel execution any number of times, not just once per CPU.
- Each time a remote session is started, it is a completely new session and options, LIBNAMEs, etc. must be specified.
- Pay close attention to data set names. It's easy to get confused when different remote processes are working with different data. Use long file names to establish a naming convention.
- Test and debug each remote session one at a time in the local environment with a small test data set. After it's working properly, encapsulate it for remote processing.
- Macros variables are session-specific, even global macro variables defined in the local session before any remote processing is started. Use SYSLPUT and &SYSRPUT to pass macro variables values between local and remote sessions.
- Manage repetitive code by using macros. Use the SASAUTOS=<directory> NOMPRINT system options to specify the macro directory and prevent macro code from printing in the log. Alternatively, use %INCLUDE < > /NOSOURCE2. This keeps the log readable.
- Manage a large program by breaking it into a series of smaller programs and then %INCLUDEing them in the main program. Do this in both the local and remote sessions.
- Use lots of comments. Programmers that come after you may not be familiar with some of the techniques you use, and why you did something a particular way may be important. Comments take only a few seconds to write. Even you will appreciate them in six months!

Summary

SAS's MP CONNECT provides SAS users with the ability to exploit the power of multiple processor systems. By encapsulating programs into parallel and sequential sections, execution time can be dramatically reduced. Speeding up a program reduces the time it takes to put information into the hands of decision makers. The productivity of the computer system itself is improved because it can handle more work in the same amount of time, thereby improving return on investment.

MP CONNECT is simple to code and implement. The concept is the same as using RSUBMIT to take advantage of remote computer services. The only difference is that the remote host is really a CPU different from the one the local host is running on.

MP CONNECT is not a silver bullet, though. Yes, it will reduce the run time of inefficient code, but it's not a substitute for efficient coding. Yes, it will speed up poorly designed processing workflow, but it's not a substitute for a well-designed workflow.

Before writing a parallel program or modifying an existing program for parallel execution, take a hard look at the data sources, the processing that's needed, the programmer time it will take, and the impact on other users. Be sure that the data sources and processing will support parallel execution, that the savings in execution time is worth the additional investment in programmer time, and that other users can live with the fact that their programs may be delayed.

References and Resources

DMB Review Magazine. http://www.datawarehouse.com

Garner, Cheryl (1998) "How to Use Version 7 Features to Optimize the Distributed Capabilities of SAS® Software" in SUGI24 Conference Proceedings. Cary, NC: SAS Institute.

Garner, Cheryl (1999) "Multiprocessing with Version 8 of the SAS® System" in SUGI25 Conference Proceedings. Cary, NC: SAS Institute.

Inmon, W.H., et al. (1999). Data Warehouse Performance. New York: Wiley Computer Publishing.

"SAS/CONNECT User's Guide" in SAS Online Doc, Version 8. Cary, NC: SAS Institute.

Stephen Morse and David Isaac (1998). Parallel Systems in the Data Warehouse. Upper Saddle River, NJ: Prentice-Hall, Inc.

About the Author

John E. Bentley has used SAS Software for fourteen years in the healthcare, insurance, and banking industries. For the past three years he has been with the Enterprise Information Group of First Union National Bank with responsibilities of supporting users of First Union's data warehouse and data marts and managing the development of SAS client-server applications to extract, manipulate, and present information from it. John regularly presents at national, regional, and special interest SAS User Group Conferences and local SAS User Group meetings. He is on the Executive Committee of the Data Mining SAS User Group.

Contact Information

John E. Bentley
First Union National Bank
201 S. Tryon Street, 5th floor, NC-1025
Charlotte, NC 28288
704-383-2686 or John.Bentley2@FirstUnion.Com

Code Sample 2. Parallel Scoring using a Macro Program

```
/*****************************************************************;
** dd_score_3way.  Test scoring using MP CONNECT.
**
** Run two remote sessions and one local session.
**
** For each data set,
**   - Read in an extract data set and 3 parameter data sets.
**   - Manipulate and transform date.  Calculate 14 scores.
**   - Create an output data set.
** Code is in the macro DD_SCORE_MACRO
** Other macros are standard profit scoring macros.
*****************************************************************/

** SYSTEM OPTIONS ;

options
   nocenter errors=3 no$syntaxcheck compress=yes
   sasautos=('/usr/local/sas8/profit/code/macros',
             '/work/tmp/d507201/mp_connect/programs')
   yearcutoff=1940 nomprint
   autosignon=yes   sascmd='/usr/local/sas8/sas';

** CAPTURE START TIME WITH A MACRO ;

%m_stime;

** START-UP MACRO;

%m_stime;

** START-UP MACROS AND LIBRARIES ;

%let legacy=dd;
%let legname=Demand Deposits;

libname mpcon v8 '/work/tmp/d507201/mp_connect/data';
libname ext v8   '/work/tmp/d507201/mp_connect/extracts';
libname parm v8
  '/work/tmp/d507201/mp_connect/parameters';
libname score v8 '/work/tmp/d507201/mp_connect/scores';

** REMOTE PROCESS 1 ;

RSUBMIT process=jan wait=no;

   %include
   '/work/tmp/d507201/mp_connect/mp_options.sas';

   %include '/work/tmp/d507201/mp_connect/mp_libs.sas';

   %dd_score_macro(January,2000,01,01/01/2000,
                   01/31/2000);
ENDRSUBMIT jan;

** REMOTE PROCESS 2 ;

RSUBMIT process=feb wait=no;

   %include
   '/work/tmp/d507201/mp_connect/mp_options.sas';

   %include '/work/tmp/d507201/mp_connect/mp_libs.sas';

    %dd_score_macro(February,2000,02,02/01/2000,
                    02/28/2000);
ENDRSUBMIT feb;

** LOCAL PROCESSING ;

%dd_score_macro(March,2000,03,'1MAR2000'd,'31MAR20
00'd);

waitfor _all_ jan feb;

rget jan;  rget feb;

signoff jan;  signoff feb;

** CAPTURE END TIME WITH A MACRO ;

%m_etime;

** PRINT TIMINGS TO LOG ;

%m_ems;
```