SCSUG 2016 – Paper 39

# Spawning SAS® Sleeper Cells and Calling Them into Action: Implementing Distributed Parallel Processing in the SAS University Edition Using Commodity Computing To Maximize Performance

Troy Martin Hughes

## ABSTRACT

With the 2014 launch of SAS® University Edition, the reach of SAS was greatly expanded to educators, students, researchers, nonprofits, and the curious who could, for the first time, utilize a nearly complete instance of Base SAS software for free, enabling SAS to better compete with open source solutions such as Python and R. Because the SAS University Edition allows a maximum of two CPUs, however, performance is curtailed sharply from more substantial SAS environments that can benefit from parallel and distributed processing, such as designs that implement SAS Grid Manager, Teradata, or Hadoop solutions. Even when comparing performance of the SAS University Edition against the most straightforward implementation of SAS Display Manager (running on the same local desktop instance), SAS Display Manager demonstrates significantly greater performance. With parallel processing and distributed computing becoming the status quo in SAS production environments, the SAS University Edition will unfortunately continue to fall behind its SAS counterparts if it cannot harness parallel processing best practices. To curb this performance disparity, this text introduces groundbreaking programmatic methods that enable commodity hardware to be networked so that multiple instances of the SAS University Edition can communicate and work collectively to divide and conquer complex tasks. With parallel processing facilitated, a SAS practitioner can now harness an endless number of computers to produce blitzkrieg solutions with SAS University Edition that rival the performance of those produced on more costly, complex infrastructure.

## INTRODUCTION

With its “Pay nothing. Gain everything.” tagline, the SAS University Edition provides substantial computing power and a generation of Base SAS software advancements and enhancements for free.[i] Its launch by CEO Jim Goodnight was met with jubilation and a standing ovation at the 2014 SAS Global Forum. Although nearly the full functionality of Base SAS is included in the SAS University Edition, the virtual environment implemented through SAS Studio does contain some limitations. For example, because the SAS Display Manager default system option XCMD is disabled, the X command, SYSTASK statement, and similar functionality is lost. With SYSTASK disabled, parallel processing is limited because multiple SAS sessions cannot be spawned to run concurrent, asynchronous batch jobs. For an introduction to asynchronous and parallel design utilizing SYSTASK, consult the “Automation” chapter of the author’s text.[ii]

Another limitation of the SAS University Edition is the threshold that no more than two CPUs can be leveraged at one time: “The SAS University Edition virtual image is configured to use 1GB of RAM and two processors. You can increase the RAM that is assigned to the SAS University Edition virtual image, but you should assign half (50%) or less of the available physical RAM. You cannot increase the number of processors that are assigned to the SAS University Edition virtual image.”[iii] Given the inability to spawn new SAS sessions and dearth of processing power, parallel processing has been noticeably absent from SAS University Edition literature, grossly limiting the potential of the software interface.

By networking commodity hardware—such as desktops or laptops—their separate processors can be collectively harnessed to accomplish collaborative tasks in parallel. The divide-and-conquer methodology is a common parallel processing paradigm in which a resource-laden task (or its data) is distributed across multiple threads or processors to be executed concurrently. Parallel processing is typically not more efficient than serialized processing because additional resources are required to coordinate the separate tasks. However, because parallel processing can distribute task workload, it represents a scalable solution that can dramatically improve software speed.

Although the SAS University Edition lacks the native ability to spawn new SAS sessions—a fundamental capability of both the SAS Display Manager and SAS Enterprise Guide—it can, however, communicate with other instances of the SAS University Edition running on a shared network. Thus, by enabling separate SAS sessions to take turns reading and writing to the same control table (i.e., SAS data set), multiple sessions can share information in real-time to

synchronize and perform coordinated activities. This text introduces the SLEEPERCELL suite of macros that enables commodity hardware to listen for tasking in a busy-waiting cycle from one or more separate SAS sessions—the handlers. After a handler assigns a task to one or more sleeper cells, the sleeper cells accept and perform the task and report task completion to the handler—all coordinated through the shared control table. Once a sleeper cell is finished, it reenters the busy-waiting cycle and awaits subsequent tasking from the same or other handlers. While the use cases of SLEEPERCELL are limitless, a proof of concept implementation demonstrates harnessing multiple sleeper cells to perform a parallel sort to expand the reach of the SAS University Edition. While designed for the SAS University Edition, SLEEPERCELL is portable to and has been validated on the SAS Display Manager for Windows.

## THE DISTRIBUTED SOLUTION

Distributed solutions are nothing new—they're only new to the SAS University Edition. I recall one fateful afternoon in high school when, with the assistance of Mr. Hamasaki's computer lab and a class of 20, we orchestrated an unnerving and unending rendition of four-part Heart and Soul that I had coded in GWBASIC. While the intent of the harmony was to produce a deeper melody (i.e., more notes), data analytic parallel processing strives for similar objectives—greater throughput in the same time period or equivalent throughput in reduced time. The SLEEPERCELL solution facilitates both of these objectives, enabling the SAS University Edition to tackle larger data sets that would otherwise cause functional or performance failure, and enabling the SAS University Edition to perform equivalent work in less time.

The SAS University Edition virtual environment prevents users from spawning batch jobs through new SAS sessions but it doesn't prevent communication between concurrent SAS sessions. The SLEEPERCELL distributed solution relies on one or more parent processes (i.e., handlers) tasking one or more child processes (i.e., sleepers) to distribute workload (i.e., processes or data) through parallel processing. For example, dividing processes among several sleeper cells might direct that one sleeper perform the MEANS procedure, another perform the FREQ procedure, and a third perform some other analysis, and because each process requires only a shared (i.e., read-only) lock, parallelism can be achieved to reduce the critical path (and execution time) of the software. In another parallel processing model, the data themselves are divided among processors and the same function is performed simultaneously on each data subset. This second model is demonstrated later in this text as a divide-and-conquer sort is performed on a data set to improve performance and enable larger data sets to be sorted without failure.

None of this awesomeness would be possible without a control table facilitating synchronization of handlers and sleeper cells. And because the control table is accessed and updated—which requires an exclusive (i.e., read-write) lock—from multiple SAS sessions, the table must be reliably and securely locked each time it is accessed to ensure that concurrent processes do not cause file access collisions and runtime errors. Thus, just as spies and their assets are often unable to meet face-to-face, SAS sessions cannot be in the same place at the same time (e.g., accessing the same data set concurrently). And, just as spies and assets might employ dead drops—such as scrawling inscrutable, chalky messages on mailboxes or lintels—to avoid direct communication, data-driven parallel processing employs control tables to facilitate effective and secure (yet indirect) communication.

Although handlers can be architected through various methods, a common requirement is that each handler should be able to perform its own tasks if no sleeper cells are available to be activated. For example, the distributed scenario demonstrated later divides a data set into multiple chunks so each subset can be sorted in parallel on separate computers before being aggregated into a single, ordered data set. When sleepers exist, they sort separate chunks of the unordered data set, the handler sorts one chunk of the data set, and the handler subsequently aggregates and interleaves all data subsets. However, if no sleepers are available to task, the handler itself sorts the data using a single out-of-the-box Base SAS SORT procedure. In other business cases, however, multiple sleeper cells might be required to process very large data sets or to boost performance, so a minimum number of required sleeper cells can be specified in the macro invocation to ensure that the required army of sleepers exists. A maximum number of processors can also be specified to ensure that a process that only requires three sleeper cells does not inefficiently attempt to task and monopolize 20 computers to do its work.

## SLEEPERCELL SETUP

Because SAS variables and macro variables cannot be passed among SAS programs running concurrently across separate SAS sessions (outside of RSUBMIT or SYSPARM functionality, not described in this text), control tables offer a viable solution to spawn and synchronize tasks. The LOCKSAFE macro represents a reliable, secure solution that

uses mutex semaphores to achieve an exclusive lock on a control table while it is accessed, read, updated, and closed, thus preventing file access collisions and resultant runtime errors. The LOCKSAFE macro is available in a separate text by the author and is required to implement the SLEEPERCELL solution: *Stress Testing and Supplanting the SAS® LOCK Statement: Implementing Mutex Semaphores To Provide Reliable File Locking in Multiuser Environments To Enable and Synchronize Parallel Processing*.[iv]

Once the LOCKSAFE macro has been downloaded, it should be installed and included via the %INCLUDE statement, SAS Autocall Macro Facility, or other method. LOCKSAFE requires initialization of the MUTEX library and SLEEPERCELL requires initialization of the SLEEPER library. Because the MUTEX library contains semaphore and mutex files that are continually being read with the DOPEN and DREAD functions, it is imperative that other files and data sets not be added to this library or logical folder. This occurs because DOPEN and DREAD were not designed for parallel processing environments, thus they can cause a "Rename of temporary member" error that either corrupts or deletes a SAS data set that is being accessed by a separate process from a separate SAS session. In the SAS University Edition, the LOCKSAFE macro can be saved to the SLEEPER folder that is created:

```
/folders/myfolders/sleeper/locksafe.sas
```

The Locksafe.sas program must be updated to include the location of the MUTEX library:

```
libname &mutlib '/folders/myfolders/mutex'; * CHANGE TO ACTUAL LOCATION;
```

The SLEEPERCELL suite comprises several macros that are included in Appendix A and should be saved to the SLEEPER folder as a single SAS program file:

```
/folders/myfolders/sleeper/sleepercell.sas
```

The Sleepercell.sas program must be updated to include the LOCKSAFE macro and initialize the SLEEPER library:

```
%include '/folders/myfolders/sleeper/locksafe.sas';
libname sleeper '/folders/myfolders/sleeper'; * CHANGE TO ACTUAL LOCATION;
```

An example handler program (to demonstrate a divide-and-conquer sort) is included as Appendix B, and can be saved to the PERM folder that is created:

```
/folders/myfolders/sleeper/handler_example.sas
```

The following code is required in the handler program (and should be customized to the environment) to include the SLEEPERCELL macro and, for this example only, to initialize the PERM library:

```
%include '/folders/myfolders/sleeper/sleepercell.sas';
libname perm '/folders/myfolders/perm';
```

Because SLEEPERCELL relies on the system's clock to provide date-time stamps that synchronize processing, it's critical that clocks on all hardware are identical. If a processor is unsynchronized by even one second, resultant race conditions could cause failure. For example, two competing sleeper cells might each believe they are first in line to access the control table and cause a file access collision and corruption of the control table. If a "Rename of temporary member" error occurs during a DATA step attempting to access the control table, a race condition is the culprit and the SAS sessions likely have unsynchronized internal clocks. If the clocks on SAS sessions differ but cannot be manually synchronized, the interpretation of individual session clocks can be shifted by setting the global macro variable &SHIFTCLOCK to the number of seconds (to three decimal places, or milliseconds) that should be added (i.e., positive value) or subtracted (i.e., negative value) to the specific clock. In practice, when all sessions are not synchronized, all sleeper cells should be set to the handler's clock via &SHIFTCLOCK.

## CONTROL TABLE COMMUNICATION

The SLEEPER.Control control table is essential in coordinating inter-session communication among the handler and various sleeper cells. Figure 1 demonstrates the synchronization that occurs between one handler and one sleeper cell and the vital role that the control table plays. The role of the handler is demonstrated on the left, the role of the sleeper

cell on the right, and modifications to the control table from these two respective sessions appear in the center. Because modifications to the control table require an exclusive file lock, the data set must be locked during each access with the LOCKSAFE macro to avoid file access collisions. And because LOCKSAFE requires a one-second delay to assess and coordinate other potential concurrent sessions, each arrow to the control table in Figure 1 represents a one-second delay and slight inefficiency. Despite this overhead, when data sets are large enough or program are flows complex enough, SLEEPERCELL demonstrates tremendous performance improvement over serialized processing models.

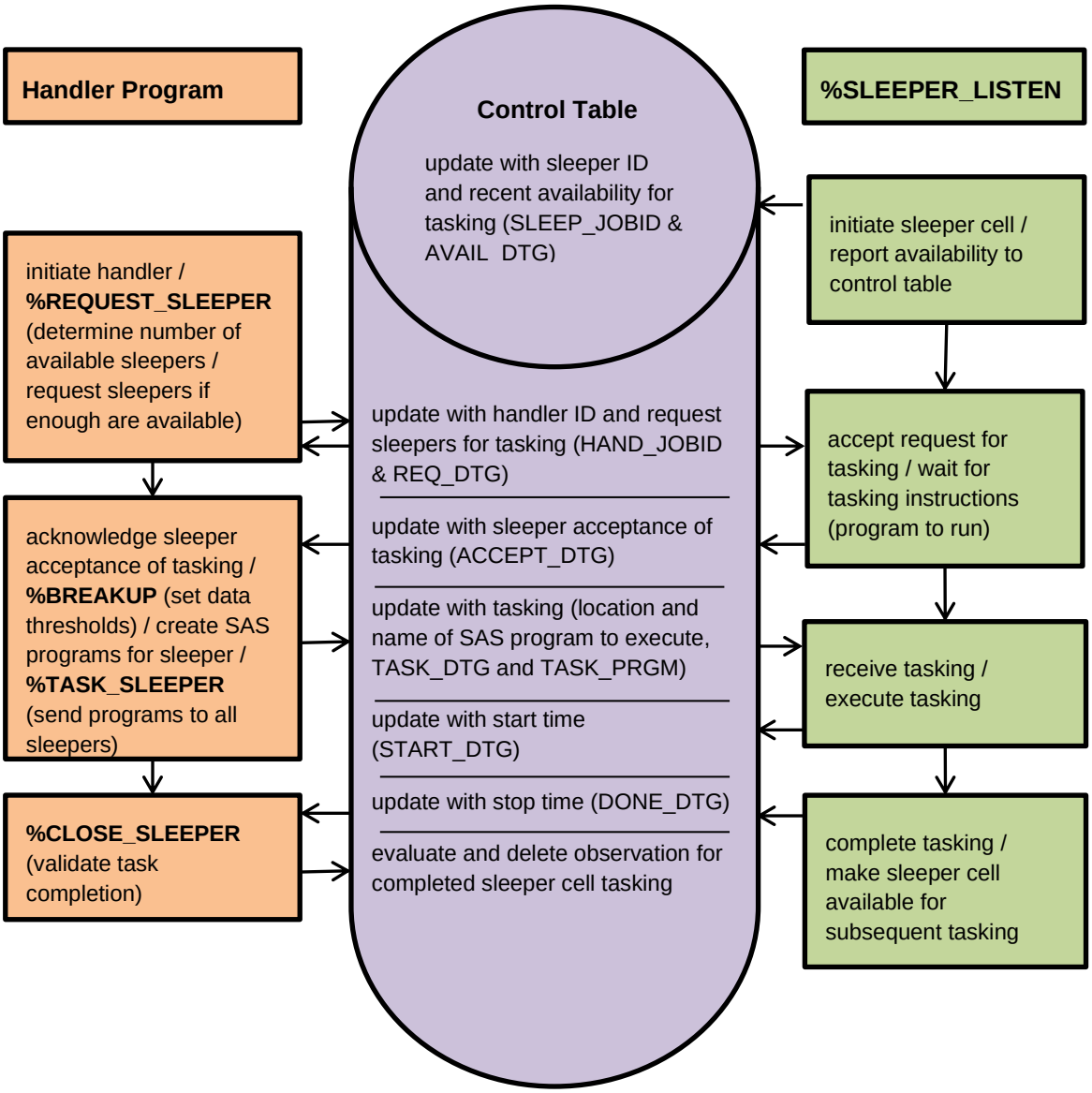


**Figure 1. SLEEPERCELL Task Request and Acceptance Program Flow**

To ensure that tasks are not lost or assigned to sleeper cells that are inactive, a double handshake is required to initiate and validate a sleeper cell tasking. The sleeper initially reports its availability (within the busy-waiting cycle) to the control table, after which the handler requests tasking, and the sleeper subsequently accepts the tasking. This first handshake ensures that all tasked sleepers are active and responsive. After task request and acceptance, the second handshake occurs in which the handler dynamically generates a SAS program for the sleeper cell to perform and the sleeper cell receives and executes the tasking, updating the control a second time with the date-time start of task execution. Without the first handshake to confirm the number of available sleepers, the handler would have no idea

how many dynamic programs to create for various sleeper cells. And, without the second handshake, the handler would not be able to validate that the sleeper had in fact begun the work it had promised to execute.

The double handshake also facilitates timeouts that can act as quality control gates to guard against rogue sleepers or handlers that have terminated with an error or otherwise gone silent. The handler example in Appendix B will only request assistance from sleeper cells whose last report date (AVAIL_DTG) is current (i.e., not greater than the number of seconds delay between busy-waiting cycles, the SEC parameter and variable). This logic is encoded in the REQUEST_SLEEPER macro, and adds an additional 10 seconds to account for lost time that may result due to the competition from multiple sleeper cells simultaneously attempting to access and update the control table:

```
if avail_dtg+sec+10>&dtg then cnt=cnt+1;
```

This check ensures that stale sleepers that have failed to check in (and might possibly have retired or died in the field) are not tasked with work. However, the business logic as coded in REQUEST_SLEEPER is not fail-proof, because it's possible that a sleeper could report its final check in (i.e., before the MAX parameter in the SLEEPER_LISTEN macro is reached and the macro terminates the busy-waiting cycle) and immediately be tasked by REQUEST_SLEEPER to perform some task. Thus, a further quality control (not demonstrated in this example) to improve robustness and reliability would be to validate subsequent task acceptance by the sleeper. For example, built into the control table logic, the ACCEPT_LIM parameter (and variable in the control table) specifies the number of seconds that the control table is willing to wait for a sleeper cell to respond and accept a task. Because the interaction between handler and sleeper cells is carefully choreographed, the SEC parameter as well as four time limit parameters—ACCEPT_LIM, TASK_LIM, START_LIM, and DONE_LIM—can be implemented within future business logic to ensure that deadlocks do not occur and processes timeout appropriately when adequate or timely response is not received.

The SLEEPER.Control control table is created by invoking the CREATE_CONTROL macro, which creates a parameterized control table—PERM.Control in this example—if it does not already exist. The control table contains the following fields:

- **SLEEP_JOBID** — &SYSJOBID of the sleeper, uniquely identifying each sleeper cell. When a sleeper cell accesses the control table (e.g., to receive a task or update completion status), it modifies only its current observation and not data that relate to other sleeper cells, or to its historical tasks that it has already completed.

- **AVAIL_DTG** — Date-time the sleeper cell last checked in to the control table. Because AVAIL_DTG is updated each time the control table is accessed by a sleeper cell, it demonstrates the currency (and availability) of a SAS session. Thus, a fresh AVAIL_DTG value (i.e., one that is not older than the SLEEPERCELL macro SEC parameter number of seconds) indicates that the sleeper cell is current and should be active.

- **SEC** — Number of seconds to wait between pings while the sleeper is listening. This value should be lower than the ACCEPT_LIM and START_LIM or else the handler may timeout between busy-waiting cycles (if those timeout functions are operationalized). A common value is one or two seconds.

- **MAX** — Timeout of listener in seconds. This specifies the total number of seconds that a listener will run. For example, a listener might be turned on in the morning and run for ten hours (or 36,000 seconds). If the MAX parameter expires while a sleeper cell is actively running a task, the task continues, completes, and the control table is updated before the sleeper is turned off.

- **HAND_JOBID** — The &SYSJOBID of the handler requesting or tasking the sleeper, which uniquely identifies each handler. Like sleeper cells, each handler only modifies its respective data in the control table.

- **HAND_PRGM** — Name or logical location of handler program that is tasking the sleeper. In the example, the handler is Handler_example.sas.

- **HAND_PRCS** — Name or description of the specific process (within the program) that is tasking the sleeper.

- **TASK_PRGM** — Name and logical location of SAS code to be executed dynamically by the sleeper. In the example handler, if the divide-and-conquer sort is run across two sleeper cells (in addition to the handler,

which also sorts a subset of the data, and thus creates Sort0.sas), the SAS programs Sort1.sas and Sort2.sas will be created and will be, respectively, in this field.

- **REQ_DTG** — Date-time of task request (by the handler).
- **ACCEPT_DTG** — Date-time of task acceptance (by the sleeper).
- **ACCEPT_LIM** — Time limit (in seconds) for sleeper to accept request (supplied by the handler). This functionality is available, but not included in the SLEEPER_CELL business logic.
- **TASK_DTG** — Date-time of tasking (by the handler).
- **TASK_LIM** — Time limit (in seconds) for handler to provide tasking (supplied by the listener). This functionality is available, but not included in the SLEEPER_CELL business logic.
- **START_DTG** — Date-time of task start (by the sleeper).
- **START_LIM** — Time limit (in seconds) for sleeper to start task (supplied by the handler). This functionality is available, but not included in the SLEEPER_CELL business logic.
- **DONE_DTG** — Date-time of task completion (by sleeper)
- **DONE_LIM** — Time limit (in seconds) for sleeper to complete task (supplied by handler). This functionality is available, but not included in the demonstrated SLEEPER_CELL business logic. In a more complex design, the DONE_LIM parameter and &DONE_LIM macro variable could be used to test for runaway or failed processes that were tasked but which did not complete within a specified time limit. This logic would be akin to implementing the WAITFOR statement with the TIMEOUT parameter following a SYSTASK statement to kill batch jobs that take too long to complete. Thus, if the handler did not receive results back from sleeper cells within a timely fashion, it could continue to subsequent, independent tasks.
- **DONE_ERR** — &SYSCC error code for task. While not implemented within the example, this return code could be utilized to drive exception handling routines that validate successful completion of tasks performed by sleeper cells.

## SLEEPERCELL MACRO SUITE

Several macros comprise the SLEEPERCELL macro suite, including the CREATE_CONTROL macro that creates the control table, the sleeper cell (SLEEPER_LISTENER) that operates in a busy-waiting cycle (either waiting for a task or executing a task), and several macros that the handler utilizes to interact with the listener via the control table. The interaction among these various macros (and their effect on the control table) is demonstrated in Figure 1, with specific functionality including:

- **CREATE_CONTROL** — This macro creates the control table that facilitates communication among the various handlers and sleepers. The control table only needs to be created once, although it may need to be groomed from time to time to remove stagnant observations that can occur when tasks are requested by the handler but never accepted by sleepers, tasked by handlers but never executed by sleepers, or started but never completed by sleepers. This ideally should be accomplished by automated code (not demonstrated) that regularly performs a quality control review of the control table to remove corrupt or old observations.

```
%macro create_control(ctrl= /* control table in LIB.DSN format */);
```

- **SLEEPER_LISTEN** — This macro is the only software run by each sleeper cell, and must be started manually in the SAS University Edition to activate each sleeper. Once executing, SLEEPER_LISTEN will remain in a busy-waiting cycle for the specified number of seconds (i.e., MAX parameter) while running SAS programs when tasked to do so by handlers.

```
%macro sleeper_listen(sec= /* number of seconds to wait in busy-waiting cycle */,
```

```
    max= /* seconds after which the sleeper should stop */,
    task_lim= /* maximum number of seconds to wait after task acceptance */,
    ctrl= /* control table in LIB.DSN format to point toward for tasking */);
```

- **REQUEST_SLEEPER** — This macro is called by the handler to request one or more sleeper cells to perform some task. If sufficient sleeper cells are not available, the &REC_SLEEPER return code will indicate this, and exception handling (as demonstrated in the handler example) can be implemented to cancel the request. The &SLEEPER_CNT global macro variable is set to the number of available sleeper cells, which can be used both in exception handling as well as to drive subsequent dynamic processing to allocate work among various sleeper cells. The macro includes an imbedded SLEEP function that waits the maximum amount of time that available sleepers have designated as their SEC parameter, plus some extra time to account for the number of active sleepers available (since more sleepers can cause delays as they jockey for access to the control table). Thus, the macro first reads the control to ascertain the number of available sleepers, then waits a calculated number of seconds before returning control to the handler.

```
%macro request_sleeper(ctrl= /* control table to query to LIB.DSN format */,
    prgm= /* handler program requesting sleepers */,
    prcs= /* handler process requesting sleepers */,
    min= /* minimum number of sleepers that must be available */,
    max= /* maximum number of sleepers, or ALL to take all available */);
```

- **TASK_SLEEPER** — This macro is called by the handler to task one or more sleeper cells once it has determined that the required number of sleeper cells is available.

```
%macro task_sleeper(ctrl= /* control table to query to LIB.DSN format */,
    prgm= /* handler program requesting sleepers */,
    prcs= /* handler process requesting sleepers */,
    totsleepers= /* number of tasked sleepers */,
    taskprgm= /* folder and file name (no .SAS extension) */);
```

- **EVAL_SLEEPER** — This macro is called by the handler to validate whether the individual sleeper cells have completed their respective tasks. The macro enters a busy-waiting cycle to wait until all tasks are complete or the process times out.

```
%macro eval_sleeper(ctrl= /* control table to query to LIB.DSN format */,
    prgm= /* handler program requesting sleepers */,
    prcs= /* handler process requesting sleepers */,
    totsleepers= /* number of tasked sleepers */,
    taskprgm= /* folder and file name (no .SAS extension) */,
    max= /* seconds to wait for all sleeper cells to complete */);
```

- **BREAKUP** — This optional macro supports SLEEPERCELL and creates a global macro variable (&BREAKUP_PAIRS) that includes a space-delimited list of breakpoints to be used in FIRSTOBS and OBS parameters during dynamic, iterative task program creation. Thus, when a data set is being subdivided to facilitate parallel processing, each respective subset can be created by setting FIRSTOBS to an odd number in the &BREAKUP_PAIRS set and setting OBS to the following even number in the set. For example, if BREAKUP is run on a data set with 100 observations and 3 SUBSETS, &BREAKUP_PAIRS will be initialized to: 1 34 35 67 68 100. Thereafter, the handler will specify (FIRSTOBS=1 OBS=34) in a SET statement to extract a subset of the data set while the first sleeper will specify (FIRSTOBS=35 OBS=67) in a SET statement to extract the second subset of the data set.

```
%macro breakup(dsn= /* data set in LIB.DSN format */,
    subsets= /* number of subsets to create */);
```

## DISTRIBUTED SORT EXAMPLE

SAS 9 enabled multithreaded sorting in which subsets of a data set are sorted in parallel behind the scenes. Because of this advancement, sorts are optimized across multiple threads and a subsequent divide-and-conquer distributed sort provides little to no performance advantage and can actually increase rather than decrease execution time. Within the SAS University Edition, however, since the number of processors is limited for each instance, performance improvements are common when executing a coordinated sort across multiple computers. Moreover, by breaking a large data set into discrete chunks, larger data sets that would otherwise cause functional failure (due to out-of-memory errors) can be successfully processed in the SAS University Edition.

The DISTRIBUTED_SORT macro definition includes the following parameters:

```
%macro distributed_sort(ctrl= /* control table to query to LIB.DSN format */,
    taskprgm= /* folder and file name (no .SAS extension) */,
    prgm= /* handler program requesting sleepers */,
    prcs= /* handler process requesting sleepers */,
    dsnin= /* data set to sort in LIB.DSN format */,
    dsnout= /* data set to sort into in LIB.DSN format */,
    byvars= /* space-delimited list of sort by variables */,
    minsleeper= /* minimum sleeper cells to run (0 or greater */,
    maxsleeper= /* maximum sleeper cells to run (MINSLEEPER
       or greater, or ALL for all available sleeper cells*/);
```

The distributed sort example is included in Appendix B, follows the program flow depicted in Figure 1, and is described in the following steps:

1. The PERM.Control control table is created with the CREATE_CONTROL macro if it does not already exist. This must be done separately because the EXIST function (utilized within CREATE_CONTROL) can fail in a concurrent processing environment and generate spurious results.

2. One or more SAS University Edition sessions are started and the SLEEPER_LISTEN macro is executed on each one. As soon as they are started, these sessions start relaying their availability to the control able so that handlers will know how many sleeper cells are active and available. Note that each of these sessions must utilize the %INCLUDE function (or other functionality) to reference Sleepercell.sas so its macros are available. Thus, running SLEEPER_LISTEN can be as straightforward as the following invocation:

   ```
   %include '/folders/myfolders/sleeper/sleepercell.sas';
   %sleeper_listen(sec=2, max=600, task_lim=5, ctrl=sleeper.control);
   ```

3. A separate SAS University Edition session is started and the Handler_example.sas program is run, which initializes the session, references Sleepercell.sas, and creates the PERM.Bigdata data set. After this setup, the DISTRIBUTED_SORT macro executes and runs the REQUEST_SLEEPER macro (included in SLEEPERCELL) to determine 1) how many sleeper cells are active and available, and 2) whether the required number of sleeper cells is available. After making the request, the session enters a busy-waiting cycle while it waits for sleeper cells to acknowledge and accept the tasking.

4. After the request for sleeper cells is updated in the control table, the sleeper cells (running SLEEPER_LISTEN on one or more SAS sessions) will take turns acknowledging and accepting the task by updating the control table.

5. When the handler program busy-waiting cycle ends, the program assesses whether the minimum number of sleeper cells are available to perform the task. If sleepers are available, the BREAKUP macro (included in SLEEPERCELL) determines the breakpoints (i.e., observation numbers) by which to divide the data set.

6. The handler program next iteratively generates the tasking programs. For example, if two sleeper cells were available, Sort0.sas would be generated to be run by the handler, and Sort1.sas and Sort2.sas would be generated to be run by the two sleepers, respectively. If no sleeper cells were available, the handler program

would still run Sort0.sas to execute a normal SORT procedure. The handler executes the TASK_SLEEPER macro (included in SLEEPERCELL) to task the sleeper cells by updating the control table to include the names and locations of the dynamically generated SAS programs.

7. After the control table has been updated via TASK_SLEEPER, the handler executes Sort0.sas to sort the first chunk of data. For example, in a 100 observation data set, observations one to 33 would be sorted by the handler itself.

8. While the handler is executing its sort, the sleeper cells take turns accessing the control table, receiving the tasks, and executing the respective SAS programs Sort1 and Sort2. Thus, within a couple seconds, all three SAS sessions—the handler and two sleepers—are sorting their respective data chunks.

9. When a sleeper cell completes its sort, it reports its task completion to the control table, creates a new observation to report its availability for new tasking, and reenters the busy-waiting cycle until tasked by the same or a different handler. Historic observations that reflect completed jobs remain in the control table until they are deleted by the SLEEPER_EVAL macro, called by a handler program.

10. When the handler completes its sort, it executes the SLEEPER_EVAL macro (included in SLEEPERCELL) to evaluate whether all sleeper cells completed their tasks correctly. Within a busy-waiting cycle, SLEEPER_EVAL repeatedly tests the control table to determine both that the sleeper cell task was completed and that it completed without warnings or runtime errors. Once evaluated, each observation in the control table referencing a completed task is deleted.

11. Once the evaluation is complete, the handler interleaves all sorted data sets into a single, ordered data set with the DATA step. In this example, the DISTRIBUTED_SORT macro exits and the program terminates.

## GENERALIZABILITY OF SLEEPERCELL

The distributed sort example demonstrated in this text represents just one of an endless number of parallel solutions that can be supported with SLEEPERCELL. Rather than breaking data sets into chunks and dividing processing among separate processors, another common parallel processing paradigm involves sending different tasks to different processors. For example, the handler might perform some analysis while one sleeper runs a MEANS procedure and a second sleeper runs the FREQ procedure. Thus, through modular software design and appropriate critical path analysis, discrete tasks can be run in parallel to achieve a faster overall solution.

Because all distributed solutions require remote sleeper cells to reach across a network connection both to communicate with a shared control table and to retrieve (and write) data sets, heavy input/output (I/O) processing can impede SLEEPERCELL performance. For example, the DISTRIBUTED_SORT example requires that a subset of the PERM.Bigdata data set be read from a centralized location by all sleeper cells and subsequently written back to this network location. A network could represent an actual network infrastructure such as a local area network (LAN), or it could represent several laptops connected to the same Wi-Fi hotspot with one laptop (i.e., the handler) sharing login information with the other laptops (i.e., the sleepers) to enable them to reach across the hotspot to pull and push data remotely. In all models, however, because data are pushed and pulled across a network, the extent to which I/O processing is minimized will benefit SLEEPERCELL performance.

For example, other distributed solutions that could more efficiently be divided across processors might include distributed MEANS and FREQ procedures. Whereas a SORT procedure is I/O intensive because it must both read and write the full data set, MEANS and FREQ procedures only read the full data set while providing a limited report (or output data set) that summarizes data. With this reduced I/O consumption, tasks such as distributed MEANS or FREQ procedures would be expected to be more relatively efficient than the distributed SORT procedure.

## CONCLUSION

The SLEEPERCELL macro extends parallel processing to the SAS University Edition by facilitating communication and synchronization of concurrent SAS sessions through a shared control table. By enabling SAS sleeper cells to exist in busy-waiting cycles, minimal resources are expended while unused commodity hardware such as desktops or laptops wait for tasking from handler programs. When SAS programs do encounter tasks that require greater processing power

or which could benefit from parallel processing, they can selectively activate sleeper cells to perform isolated or coordinated tasks, executing dynamically generated SAS programs on behalf of the handler. With this newfound processing power, overall performance of the SAS University Edition is improved and big data are rendered less of an obstacle to software that otherwise limits users to two processors.

## REFERENCES


[i] The SAS University Edition. SAS Institute. Cary, NC. Retrieved from http://www.sas.com/en_us/software/university-edition.html

[ii] Hughes, Troy Martin. SAS Data Analytic Development: Dimensions of Software Quality. New York, NY: John Wiley and Sons Publishing, 2016.

[iii] Knowledge Base / SAS Products & Solutions. SAS University Edition: Help Center. SAS Institute. Cary, NC. Retrieved from https://support.sas.com/software/products/university-edition/faq/prerequisites.htm.

[iv] Hughes, Troy Martin. 2016. "Stress Testing and Supplanting the SAS® LOCK Statement: Implementing Mutex Semaphores To Provide Reliable File Locking in Multiuser Environments To Enable and Synchronize Parallel Processing." *Proceedings of the Twenty-Seventh Annual Midwest SAS Users Group (MWSUG) Conference.*


## CONTACT INFORMATION

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


Name: Troy Martin Hughes
E-mail: troymartinhughes@gmail.com



SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

## APPENDIX A. THE SLEEPERCELL MACROS

```
%include '/folders/myfolders/sleeper/locksafe.sas';
libname sleeper '/folders/myfolders/sleeper';

%macro breakup(dsn= /* data set in LIB.DSN format */,
    subsets= /* number of subsets to create */);
%let syscc=0;
%global breakup_pairs;
%let breakup_pairs=;
%local obstot low high;
proc sql noprint;
    select count(*) format=15.
    into :obstot
    from &dsn;
quit;
%if &syscc=0 %then %do i=1 %to &subsets;
    %if &i=1 %then %let low=1;
    %else %let low=%sysevalf(&high+1);
    %if &i=&subsets %then %let high=&obstot;
    %else %let high=%sysevalf(&i*&obstot/&subsets,ceil);
    %if %length(&breakup_pairs)=0 %then %let breakup_pairs=&low &high;
    %else %let breakup_pairs=&breakup_pairs &low &high;
    %end;
%mend;

%macro request_sleeper(ctrl= /* control table to query to LIB.DSN format */,
    prgm= /* handler program requesting sleepers */,
    prcs= /* handler process requesting sleepers */,
    min= /* minimum number of sleepers that must be available */,
    max= /* maximum number of sleepers, or ALL to take all available */);
%let syscc=0;
%global req_sleeperRC sleeper_cnt interval_max;
%let req_sleeperRC=GENERAL FAILURE;
%let sleeper_cnt=0;
%let interval_max=0;
%local dtg;
%let dtg=%sysfunc(datetime());
%if %upcase(&max)=ALL %then %let max=9999;
%locksafe(dsn=sleeper.control, sec=1, max=10);
%if %length(&locksafeRC)=0 %then %do;
    data sleeper.control (drop=cnt);
       set sleeper.control end=eof;
       length cnt 8;
       if _n_=1 then cnt=0;
       if missing(hand_jobid) and not missing(avail_dtg) and not missing(sec)
             then do;
          if avail_dtg+sec+10>&dtg then cnt=cnt+1;
          end;
       if eof then call symput('sleeper_cnt',strip(put(cnt,8.)));
       retain cnt;
    run;
    %if %eval(&sleeper_cnt>=&min) %then %do;
       data sleeper.control (drop=cnt interval_max);
          set sleeper.control end=eof;
```

```
            length cnt 8 interval_max 8;
            if _n_=1 then do;
               cnt=0;
               interval_max=0;
               end;
            if missing(hand_jobid) and not missing(avail_dtg) and
                   not missing(sec) and cnt<&max then do;
               if avail_dtg+sec+10>&dtg then do;
                  cnt=cnt+1;
                  hand_jobid=&sysjobid;
                  hand_prgm="&prgm";
                  hand_prcs="&prcs";
                  req_dtg=&dtg;
                  if sec>interval_max then interval_max=sec;
                  end;
               end;
            if eof then do;
               call symput('sleeper_cnt',strip(put(cnt,8.)));
               call symput('interval_max',strip(put(interval_max,8.)));
               end;
            retain cnt interval_max;
         run;
         %if &syscc=0 %then %let req_sleeperRC=;
         %end;
      %else %let req_sleeperRC=NOT ENOUGH SLEEPERS AVAILABLE: You requested &min
         and only &sleeper_cnt are available;
      %end;
%locksafe(dsn=sleeper.control, terminate=YES);
%if %length(&req_sleeperRC)=0 %then
%sleep(sec=%eval(&sleeper_cnt+&interval_max+2));
%mend;

%macro task_sleeper(ctrl= /* control table to query to LIB.DSN format */,
      prgm= /* handler program requesting sleepers */,
      prcs= /* handler process requesting sleepers */,
      totsleepers= /* number of tasked sleepers */,
      taskprgm= /* folder and file name (no .SAS extension) */);
%let syscc=0;
%local i sleeperlist;
%do i=1 %to &totsleepers;
      %if &i=1 %then %let sleeperlist=&taskprgm&i..sas;
      %else %let sleeperlist=&sleeperlist, &taskprgm&i..sas;
      %end;
%locksafe(dsn=sleeper.control, sec=1, max=10);
%if %length(&locksafeRC)=0 %then %do;
      data &ctrl (drop=cnt);
         set &ctrl;
         length cnt 8;
         if _n_=1 then cnt=0;
         if cnt<&totsleepers and not missing(sleep_jobid) and not
                missing(hand_jobid) and
                hand_jobid=&sysjobid and hand_prgm="&prgm" and hand_prcs="&prcs"
                and not missing(req_dtg) and not missing(accept_dtg) and
                missing(task_prgm) and missing(task_dtg) then do;
            cnt=cnt+1;
```

```
            task_dtg=datetime();
            task_prgm=scan("&sleeperlist",cnt,",");
            end;
        retain cnt;
    run;
    %locksafe(dsn=sleeper.control, terminate=YES);
    %end;
%else %let req_sleeperRC=COULD NOT ACCESS SLEEPER CONTROL TABLE;
%mend;

%macro eval_sleeper(ctrl= /* control table to query to LIB.DSN format */,
    prgm= /* handler program requesting sleepers */,
    prcs= /* handler process requesting sleepers */,
    totsleepers= /* number of tasked sleepers */,
    taskprgm= /* folder and file name (no .SAS extension) */,
    max= /* seconds to wait for all sleeper cells to complete */);
%let syscc=0;
%global eval_sleeperRC tasks_complete;
%let eval_sleeperRC=;
%let tasks_complete=0;
%local starttime dtg;
%let starttime=%sysfunc(datetime());
%let dtg=&starttime;
%do %until(%sysevalf(&dtg>&starttime+&max) or &sleeper_cnt=&tasks_complete);
    %let dtg=%sysevalf(%sysfunc(datetime()));
    %locksafe(dsn=sleeper.control, sec=1, max=10);
    %if %length(&locksafeRC)=0 %then %do;
        data &ctrl (drop=cnt eval_sleeperRC);
            set &ctrl end=eof;
            length cnt 8 eval_sleeperRC $1000;
            if _n_=1 then do;
                cnt=&tasks_complete;
                eval_sleeperRC='';
                end;
            if hand_jobid=&sysjobid and hand_prgm="&prgm" and hand_prcs="&prcs" and
                    not missing(done_dtg) then do;
                cnt=cnt+1;
                if done_err^=0 then
                    eval_sleeperRC=catx(",",eval_sleeperRC,"&taskprgm","&syscc");
                call symput('tasks_complete',strip(put(cnt,8.)));
                delete;
                end;
            if eof then do;
                call symput('eval_sleeperRC',strip(eval_sleeperRC));
                end;
            retain cnt eval_sleeperRC;
        run;
        %locksafe(dsn=sleeper.control, terminate=YES);
        %if %eval(&tasks_complete < &sleeper_cnt) %then %sleep(sec=1);
        %end;
    %end;
%mend;

%macro create_control(ctrl= /* control table in LIB.DSN format */);
%if %sysfunc(exist(&ctrl))=0 %then %do;
```

```
    data &ctrl;
       length sleep_jobid 8 avail_dtg 8 sec 8 max 8
          hand_jobid 8 hand_prgm $200 hand_prcs $32 task_prgm $200 task_name $32
          req_dtg 8 accept_dtg 8 accept_lim 8 task_dtg 8
          task_lim 8 start_dtg 8 start_lim 8 done_dtg 8 done_lim 8 done_err 8;
       format sleep_jobid 8. avail_dtg datetime17. sec 8. max 8.
          hand_jobid 8. hand_prgm $200. hand_prcs $32. task_prgm $200. task_name
          $32.
          req_dtg datetime17. accept_dtg datetime17. accept_lim 8.
          task_dtg datetime17. task_lim 8. start_dtg datetime17. start_lim 8.
          done_dtg datetime17. done_lim 8. done_err 8.;
    run;
    %end;
%mend;

%macro sleeper_listen(sec = /* number of seconds to wait in busy-waiting cycle */,
    max = /* seconds after which the sleeper should stop */,
    task_lim = /* maximum number of seconds to wait after task acceptance */,
    ctrl = /* control table in LIB.DSN format to point toward for tasking */);
%let syscc=0;
%global sleeper_listenRC;
%let sleeper_listenRC=GENERAL FAILURE;
%local starttime dtg tasked task;
%let starttime=%sysfunc(datetime());
%let dtg=&starttime;
%do %until(%sysevalf(&dtg>&starttime+&max));
    %let dtg=%sysevalf(%sysfunc(datetime()));
    %locksafe(dsn=&ctrl, sec=1, max=10);
    %if %length(&locksafeRC)=0 %then %do;
       %let tasked=no;
       data &ctrl (drop=found);
          set &ctrl end=eof;
          length found 8;
          if _n_=1 then found=0;
          if sleep_jobid=&sysjobid and missing(done_dtg) then do;
             found=1;
             avail_dtg=&dtg;
             * ACCEPT TASKING;
             if not missing(hand_jobid) and not missing(hand_prgm) and
                       not missing(hand_prcs) and not missing(req_dtg) then do;
                if missing(accept_dtg) and missing(task_dtg) and
                             missing(start_dtg) and missing(done_dtg) then do;
                       accept_dtg=avail_dtg;
                       end;
                else if not missing(accept_dtg) and not missing(task_dtg) and
                             not missing(task_prgm) and
                             missing(start_dtg) and missing (done_dtg) then do;
                       * START TASK;
                       start_dtg=avail_dtg;
                       call symput('tasked','yes');
                       call symput('task',strip(task_prgm));
                       end;
                end;
             output;
             end;
```

```
            else output;
            retain found;
            * ADD NEW LISTENER;
            if eof and found^=1 then do;
               sleep_jobid=&sysjobid;
               avail_dtg=&dtg;
               sec=&sec;
               max=&max;
               hand_jobid='';
               hand_prgm='';
               hand_prcs='';
               task_prgm='';
               req_dtg=.;
               accept_dtg=.;
               task_dtg=.;
               task_lim=.;
               start_dtg=.;
               start_lim=.;
               done_dtg=.;
               done_lim=.;
               done_err=.;
               output;
               end;
         run;
         %locksafe(dsn=&ctrl, terminate=YES);
         %if &tasked=no %then %do;
            %if &sec>1 %then %sleep(sec=%eval(&sec-1));
            %end;
         %else %do;
            * EXECUTE TASK;
            %include "&task";
            %let tasked=no;
            %locksafe(dsn=&ctrl, sec=1, max=10);
               data &ctrl;
                  set &ctrl;
                  if sleep_jobid=&sysjobid then do;
                         done_dtg=datetime();
                         done_err=&syscc;
                         end;
               run;
            %locksafe(dsn=&ctrl, terminate=YES);
            %end;
         %end;
      %else %do;
         %locksafe(dsn=&ctrl, terminate=YES);
         %let sleeper_listenRC=LOCKSAFE FAILURE: &locksafeRC;
         %end;
      %end;
%mend;
```

## APPENDIX B. HANDLER_EXAMPLE.SAS

```
%include '/folders/myfolders/sleeper/sleepercell.sas';
libname perm '/folders/myfolders/perm';
%let taskprgm=/folders/myfolders/perm/sort;

%macro make_data(dsn=, obs=);
data &dsn (drop=i j);
    length char1 $10;
    do i=1 to &obs;
       char1='';
       do j=1 to 10;
          char1=cats(char1,byte(int(rand('uniform')*10)+65)); *A through J;
          end;
       output;
       end;
run;
%mend;

%make_data(dsn=perm.bigdata, obs=10000);


%macro distributed_sort(ctrl= /* control table to query to LIB.DSN format */,
    taskprgm= /* folder and file name (no .SAS extension) */,
    prgm= /* handler program requesting sleepers */,
    prcs= /* handler process requesting sleepers */,
    dsnin= /* data set to sort in LIB.DSN format */,
    dsnout= /* data set to sort into in LIB.DSN format */,
    byvars= /* space-delimited list of sort by variables */,
    minsleeper= /* minimum sleeper cells to run (0 or greater */,
    maxsleeper= /* maximum sleeper cells to run (MINSLEEPER
       or greater, or ALL for all available */);
%let syscc=0;
%let starting=%sysfunc(datetime());
%local i lib pathlib low high;
%let lib=%scan(&dsnin,1,.);
%let pathlib=%sysfunc(pathname(%scan(&dsnin,1,.)));
%request_sleeper(ctrl=&ctrl, prgm=&prgm, prcs=&prcs, min=&minsleeper,
max=&maxsleeper); *creates SLEEPER_CNT;
%if %length(&req_sleeperRC)>0 %then %do;
    %put EXITING MACRO: &req_sleeperRC;
    %return;
    %end;
%breakup(dsn=&dsnin, subsets=%eval(&sleeper_cnt+1));
%do i=0 %to %eval(&sleeper_cnt); *starts at 0 if the handler runs a sleeper session
as well;
    %let low=%scan(&breakup_pairs,%eval(((&i+1)*2)-1),,S);
    %let high=%scan(&breakup_pairs,%eval((&i+1)*2),,S);
    data _null_;
       file "&taskprgm&i..sas";
       put "libname &lib ""%str(&pathlib)"";";
       put " ";
       put "proc sort data=&dsnin (firstobs=&low obs=&high) out=&dsnin.&i;";
       put "   by &byvars;";
       put "run;";
    run;
```

```
    %end;
%let tasking=%sysfunc(datetime());
%task_sleeper(ctrl=&ctrl, prgm=&prgm, prcs=&prcs, totsleepers=&sleeper_cnt,
    taskprgm=&taskprgm);
%include "&taskprgm.0.sas";
%eval_sleeper(ctrl=&ctrl, prgm=&prgm, prcs=&prcs, totsleepers=&sleeper_cnt,
    taskprgm=&taskprgm, max=15)
%let combining=%sysfunc(datetime());
data &dsnout;
    set
    %do i=0 %to &sleeper_cnt;
       &dsnin.&i
       %end;;
    by &byvars;
run;
proc sort data=&dsnin out=&dsnout;
    by &byvars;
run;
%mend;

%distributed_sort(taskprgm=&taskprgm, prgm=distributed_sort, prcs=sort,
    ctrl=sleeper.control, dsnin=perm.bigdata, dsnout=perm.bigdata_out,
    byvars=char1, minsleeper=1, maxsleeper=3);
```