EPRI | ELECTRIC POWER RESEARCH INSTITUTE

# User Instructions for Parallel Processing with OpenDSS-PM

*Davis Montenegro*
*Roger Dugan*
*Revised 09-29-2017*

The classic OpenDSS program is a simulation platform built for execution in a single, sequential process. Each procedure/function is called from each object sequentially to perform a QSTS simulation. The performance that can be achieved is based on the structure of the low level routines, the simplicity of the routines and the efficiency of the compiler.

EPRI has explored several methods to accomplish parallel processing in OpenDSS, including the parallelization of the whole program using a different interface (the Direct DLL API) and the modification of the solver using other programming languages, among other methods. However, these approaches demand significant extra work for the user and will be always tied to a particular interface. As a consequence, desirable features that users are accustomed to, such as the COM interface, would be at risk of being deprecated for this type of processing.

EPRI has evolved OpenDSS into a more modular, flexible and scalable parallel processing platform we are calling OpenDSS-PM based with the following guidelines:

1. The parallel processing machine will be interface-independent
2. Each component of the parallel machine should be able to work independently
3. The simulation environment should deliver information consistently
4. The data exchange between the components of the parallel machine should respect the interface rules and procedures
5. The user handle of the parallel machine should be easy and support the already acquired knowledge of OpenDSS users

## *The parallel machine*

To create the parallel machine, OpenDSS-PM uses the actor model [1-4]. There are several actor frameworks for Delphi proposed by various authors; however, we already had a framework developed to evaluate Delphi's tools for this purpose so we chose to use it. Each actor is created by OpenDSS-PM, runs on a separate processor (if possible) using separate threads and has its own assigned core and priority (*real-time* priority for the process and *time critical* for the thread).

The interface for sending and receiving messages from other actors will be the one selected by the user, this is, either the COM interface, the Direct DLL API, or a text script using the EXE version of the program From this interface, the user will be able to create a new actor (instance), send/receive messages from these actors, and define the execution properties of the actors such as the execution core, simulation mode, and circuit to be solved, among others. This concept is shown in Figure 1.

Using the existing interfaces, the user can:

1. Request the number of available cores and the number of physical processors available.
2. Create/Destroy actors
3. Execute the simulation of each circuit concurrently and in parallel (hardware dependent)
4. Assign the core where the actor will be executed
5. Modify the simulation settings for the active actor
6. Set the name of the circuit that will be simulated

Basically, the user can do the same things he can do with the classic version plus the operations related with parallel processing.

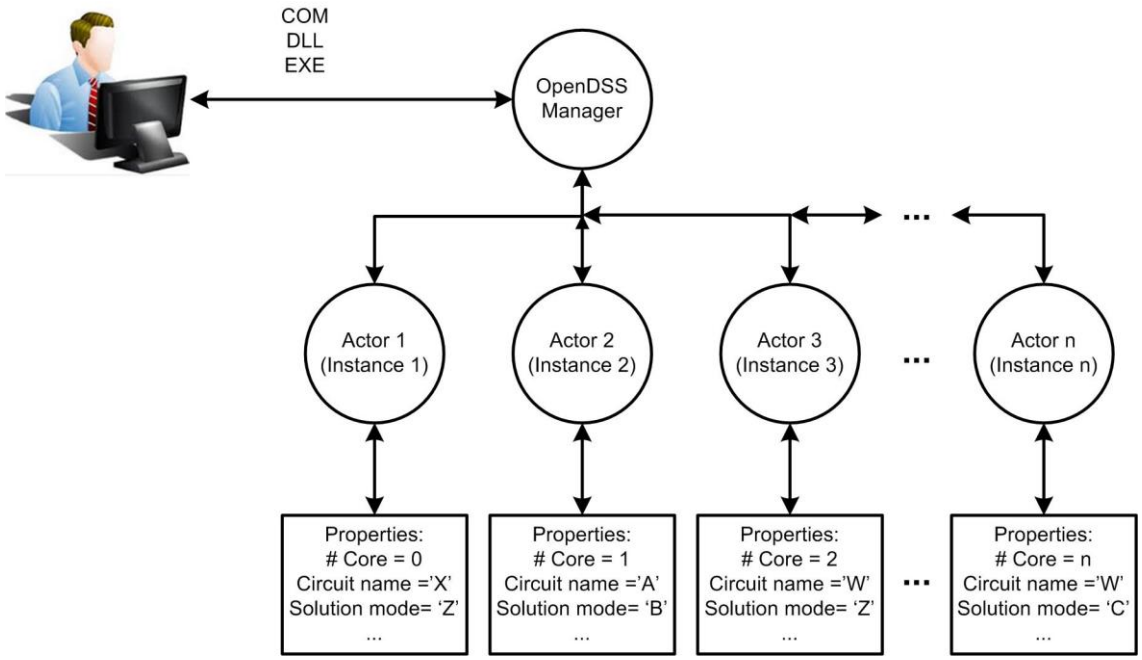


*Figure 1. Operational scheme proposed for evolving OpenDSS into a parallel processing machine based on actors*

As can be seen in Figure 1, the interface will work as the communication medium between the different actors on the parallel machine. This feature enables several simulation modes inspired in parallel computations such as temporal, Diakoptics, among others. These type of simulations will be driven by an external program that will handle the actors within the parallel machine, in keeping with the actor concept as a *message driven state machine*.

To operate the parallel machine the suggested procedure is as follows:

1. The program will create a new default actor every time the *start* function is called in an OpenDSS-PM COM or DLL interface or when the EXE version is started. OpenDSS-PM will create Actor 1, designate a memory space, open an instance for KLUSolve, and define the execution thread. In return, OpenDSS-PM will return to the user an ID (integer) to identify the created actor.
2. After the program has started the user issues the NewActor command to create a new actor.
3. After a new actor is created, the user will designate the core in which the actor is to be executed using the *Set Core=nn* command. This command will apply to the *active actor* using the selected interface. Core 0 will be the default core for the initial actor created at start up, but this can be changed.

4. To change the active actor, the user will issue the *Set ActiveActor=nn* command.
5. After the active actor is set, the user can execute OpenDSS commands as done for the classic version using the selected interface. The commands will apply to the process executed by the active actor.
6. There are two options for solving the systems with actors:
    a. Solve the active actor
    b. Solve all of the actors

    If the user selects to solve only the active actor while there are other actors created, the user can continue to interact with the other actors while the solving actor is working. On the other hand, if the user selects to solve all the actors, the ability to exchange information with an actor will depend on the availability of its core. If there are not enough cores to handle the request the user program will have to wait until one of the actors finishes the solution routine.

7. Each actor can be asked for data and can store its own monitor and energy meter samples locally.

To make this possible it is necessary to clone essential classes of OpenDSS. This cloning process must be done every time the user requests it. By default, there will be at least 1 actor active for performing simulations and the default core will be Core 0.

The configuration of each instance (actor) can be made sequentially, however, the parallel processing of each actor circuit is done using multithreading, defining the process and thread affinity and its priority.

## *Added Instruction set for OpenDSS-PM*

### *NumCPUs*

Returns the number of threads of virtual cores (CPUs) available in the computer. This is a *read-only* value and must be executed using the "*Get*" command (opposite of the Set command). For more information about the processor architecture visit: http://www.howtogeek.com/194756/cpu-basics-multiple-cpus-cores-and-hyper-threading-explained/

### *NumCores*

Returns the number of physical processors (Cores) available in the computer. If the computer has less than 64 CPUs the number of cores will be the half of the number of CPUs (2 threads per Core), otherwise, these numbers will be the same. This is a read-only value and must be executed using the "*Get*" command. The value is returned in the Result string in the EXE version and the *@result* variable for the text scripting interface. The string is also available in the Result property of the Text interface in the COM interface and the result of the DSSPut_Command function of the DLL interface.

### *NewActor*

This command creates a new actor (OpenDSS-PM Instance) and sets the new actor as the active actor. There can be only 1 circuit per actor. The *NewActor* Instruction will increment the variable *NumOfActors*; however, if the number of actors is the same as the number of available CPUs the new actor will not be created, generating an error message. This instruction will return the ID of the active actor. This command does not requires a precedent command.

*NumActors*

Returns the number of actors created, this number cannot be higher than the number of available CPUs. This is a read-only value and must be executed using the "*Get*" command. By default there is at least 1 actor created.

*ActiveActor*

This option can be used to get or set the active actor. The active actor cannot be higher than *NumActors* nor lower than 1. Use the commands *Get* or *Set* to operate on this value.

*CPU*

This option can be used to get or set the CPU assigned to the *active actor*. CPU numbers are indexed from 0..n. That is, the CPU number is 0-based and cannot be greater than (*NumCPUs* – 1). Actors are indexed from 1..n. When a new actor is created it is automatically assigned to the corresponding core in sequence starting from 0 (i.e. Actor 1 will be assigned to CPU 0, Actor 2 to CPU 1, and Actor 3 to CPU 2 and so on). Use the command *Get* or *Set* commands to operating on this value.

*ActorProgress*

This option will show on the summary tab the progress for all the actors when performing a task. For example, if each actor is performing a *yearly* simulation and the user wants to know the progress of each actor (%), this is the instruction that must be used. This is a read-only value and must be executed using the "*Get*" command.

*ClearAll*

Clears all the existing circuits and their classes. After executing this command OpenDSS will create only one actor, the active actor will be actor number 1 and the CPU assigned will be the CPU assigned originally for this actor.

*Wait*

This function freezes the execution of the Frontal Panel Actor until all the other actors are available to receive new messages or start new processes. By using this function, it is probable that the user may not be able to see updates on the summary/results tab, but is a very good mechanism for synchronizing all the actors within the parallel environment and make classical scripts compatible with OpenDSS-PM.

*Parallel*

This function enables/disables the parallel processing functionalities of OpenDSS-PM. By disabling this features, OpenDSS-PM will behave as the classic version of OpenDSS and even if the user can create new actors, each actor will operate sequentially one after the other (no concurrency). By default this option is disabled and it is required to activate it to gain access to the parallel processing features of OpenDSS-PM.

*SolveAll*

This command starts the solution process for all the existing actors.

*ConcatenateReports*

This option can be used to Enable/Disable the report concatenation of the monitor's content. When disabled (default) the user needs to specify the actor to gain access to the monitor's data using the commands show or export. When enabled, this option allows to summarize all monitors content with the same name working at different actors into a single file.

*CalcIncMatrix*

This command can be used to calculate the Branch-To-Node incidence matrix (B2N) of the active circuit. The calculation is performed considering the PDElements as branches (rows) and the buses as nodes (Columns). The order of the PDElements as rows goes as follows: Lines, transformers, capacitors in series and reactors in series.

*CalcIncMatrix_O*

This command can be used to calculate the Branch-To-Node incidence matrix (B2N) of the active circuit. However, this command delivers and optimized matrix that is organized inside the algorithm by using the CktTree class. The calculation is performed considering the PDElements as branches (rows) and the buses as nodes (Columns). Additionally, this algorithm calculates the levels of each bus to populate an internal array called BusLevels. The Bus level is an integer that reveals how far in terms of buses is the Bus respect to the feeder's backbone. The Backbone is a randomly selected continuous path from the feeder head to a point in the feeder selected as feeder end.

*Tear_Circuit*

This command tears apart the circuit in many pieces as CPUs – 1 are available in the local PC. The tearing takes place by using the *CalcIncMatrix_O* command internally. Then, the algorithm estimates the best route for generating a set of sub-Circuits by placing an energy meter at the tearing points selected by the algorithm. As a result, this command will generate a folder called *Torn_Circuit* inside the project's folder. Each sub-Circuit will be contained in this folder starting at the substation (*Torn_Circuit* folder root) and the other Sub-Circuits in folders called *Zone_1, Zone_2* and so on until the total number of sub-Circuits.

*Export IncMatrix*

This command exports in a csv file the B2N matrix using a compressed coordinated format (Row, Column, and Value). This format is used to facilitate uploading this data into a sparse matrix.

*Export IncMatrixRows*

This command exports in a csv file the names of the rows (PDElements) of the latest calculation of the B2N matrix using any of the methods for this purpose.

*Export IncMatrixCols*

This command exports in a csv file the names of the columns (buses) of the latest calculation of the B2N matrix using any of the methods for this purpose.

*Export BusLevels*

This command exports in a csv file the names and levels of the columns (buses) of the latest calculation of the B2N matrix using any of the methods for this purpose.

*Abort*

This command aborts all the simulation jobs running and gives back the control to the caller.

## *Error codes Associated to the Parallel Machine (PM) Operation*

| Code | Description |
|---|---|
| 7000 | This error is generated when the user is trying to create a new circuit but the number of available CPUs is already assigned to other circuits. To avoid this message, before creating a |

| | |
|---|---|
| | new circuit check if there are available CPUs by requesting OpenDSS-PM to deliver the number of CPUs and the number of existing Actors. |
| 7001 | This error is generated when the user tries to create a new actor but there are no more CPUs available. The number of actors cannot exceed the number of available CPUs on the computer. Check the number of actors (*NumActors*) and the number of CPUs (*NumCPUs*) before executing the *NewActor* Command. |
| 7002 | This message is displayed when the user is trying to activate an inexistent actor. To avoid this message check the number of actors (*NumActors*) before activating one. The ID of the actor should be less than or equal to *NumActors*. |
| 7003 | This message is displayed when the user is trying to assign a non-existent CPU to the active actor. To avoid this message, check the number of CPUs (*Get NumCPUs*) before activating one. The CPU ID should be lower to the Number of existing CPUs (starts in CPU 0). |

## *Examples*

The following example will create three actors using a DSS script with the EXE version of OpenDSS-PM. Then, the simulation mode, start time and number of steps is set for each actor. Finally, all the compiled systems are solved concurrently. The system being solved is EPRI's Ckt5 test circuit.

```
clearAll
set parallel=No

compile "C:\Program Files\OpenDSS\EPRITestCircuits\ckt5\Master_ckt5.dss"
set CPU=0
Solve

NewActor
compile "C:\Program Files\OpenDSS\EPRITestCircuits\ckt5\Master_ckt5.dss"
set CPU=2
Solve

NewActor
compile "C:\Program Files\OpenDSS\EPRITestCircuits\ckt5\Master_ckt5.dss"
set CPU=3
Solve

set parallel=Yes

set activeActor=1
set mode=yearly number=2000 hour = 0 totaltime=0

set activeActor=2
set mode=yearly number=2000 hour = 2000 totaltime=0

set activeActor=3
set mode=yearly number=2000 hour = 4000 totaltime=0

SolveAll
Wait

set ConcatenateReports=Yes
show monitor MS2
```

You need to be careful when organizing OpenDSS-PM scripts – remember that now everything is happening at the same time. If the user wants to see the voltages, export monitors or execute any other "report" command, it is necessary to wait until all the processes are executed. Otherwise, OpenDSS-PM will execute the processes immediately. As in the classic OpenDSS version, to have control of the

operations that you are performing in the script, you can select the part of the script that you want to execute, and then, by right-clicking with the mouse, select "*Do selected*" from the pop-up menu as shown in Figure 2.

If you want to verify the execution of the solution in terms of CPU allocation after executing the script presented above, execute the windows *Resource Monitor (http://www.digitalcitizen.life/how-use-resource-monitor-windows-7)* or the *Task Manager*. You will be able to check how each system is being solved on a separate CPU. In the case of the example script, the first actor will be executed on CPU 0, the second on CPU 2 and the third on CPU 3. In the *performance* tab of the *Task Manager* you will see the utilization of the processors by the different actors when executing the solve command as shown in Figure 3.

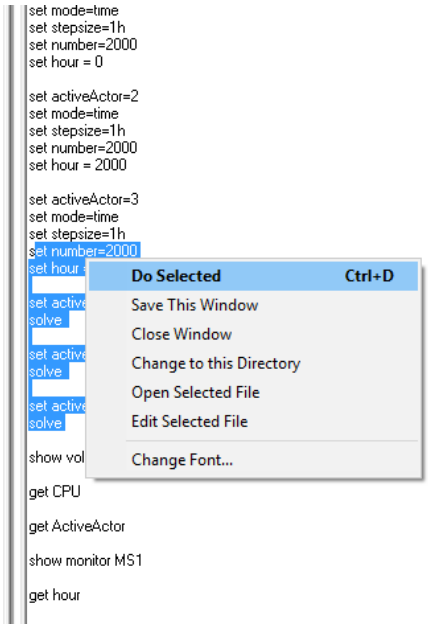


*Figure 2. Selecting the parts of the script that the user wants to be executed*

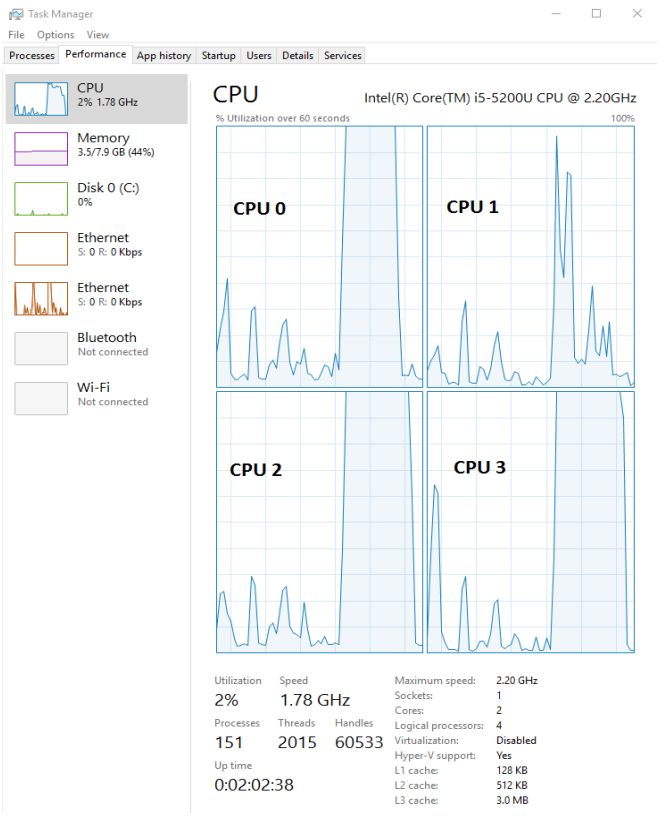


*Figure 3. Processor usage when performing parallel processing with OpenDSS-PM*

As a general recommendation, do not to use all the available CPUs for an OpenDSS simulation. Your system can freeze and will not let you to perform other activities in parallel to the simulation.

## *References*

[1] C. Hewitt, E. Meijer, and C. Szyperski. (2012, 05-15). *The Actor Model (everything you wanted to know, but were afraid to ask)*. Available: http://channel9.msdn.com/Shows/Going+Deep/Hewitt-Meijer-and-Szyperski-The-Actor-Model-everything-you-wanted-to-know-but-were-afraid-to-ask

[2] National Instruments. (2013, 24, 05). *Actor Framework Template documentation*. Available: http://www.ni.com/white-paper/14115/en

[3] L. Jie, J. Eker, J. W. Janneck, L. Xiaojun, and E. A. Lee, "Actor-oriented control system design: a responsible framework perspective," *IEEE Transactions on Control Systems Technology,* vol. 12, pp. 250-262, 2004.

[4] D. Montenegro, "Actor's based diakoptics for the simulation, monitoring and control of smart grids," Université Grenoble Alpes, 2015.