opendss parallel feature using python

<https://sourceforge.net/p/electricdss/code/HEAD/tree/trunk/Version8/Distrib/Examples/Parallel_Processing/Python/Example_1/Example_1.py>

**import** **win32com.client**

**from** **win32com.client** **import** makepy

**import** **sys**

**from** **tkinter** **import** \*

**import** **gc**

**import** **numpy** **as** **np**

**from** **IPython.display** **import** clear\_output

**import** **time**

**from** **os** **import** system, name

*#import tkmessagebox*

*# Initialize OpenDSS (early binding)*

sys.argv = ["makepy", "OpenDSSEngine.DSS"]

makepy.main()

DSSObj = win32com.client.Dispatch("OpenDSSEngine.DSS")

DSSText = DSSObj.Text

DSSCircuit = DSSObj.ActiveCircuit

DSSSolution = DSSCircuit.Solution

DSSParallel = DSSCircuit.Parallel;

DSSBus=DSSCircuit.ActiveBus

DSSCtrlQueue=DSSCircuit.CtrlQueue

DSSObj.Start(0)

DNumActors = 6

DSSText.Command='ClearAll'

DSSText.Command='compile "C:/Program Files/OpenDSS/IEEETestCases/13Bus/IEEE13Nodeckt.dss"'

DSSText.Command='set maxiterations=1000 maxcontroliter=1000'

DSSSolution.Solve *# Solves Actor 1*

DSSText.Command = 'Clone ' + str(DNumActors - 1)

DSSText.Command = 'set ActiveActor=\*'

DSSText.Command = 'set mode=time controlmode=time number=525000 stepsize=1s hour=0 sec=0 miniterations=1 totalTime=0'

DSSText.Command = 'set ActiveActor=1'

DSSText.Command = 'Set Parallel=Yes'

DSSText.Command = 'SolveAll'

BoolStatus = 0;

**while** BoolStatus == 0:

ActorStatus = DSSParallel.ActorStatus

BoolStatus = all(Status == 1 **for** Status **in** ActorStatus) *#Checks if everybody has ended*

ActorProgress = DSSParallel.ActorProgress

system('cls')

**for** i **in** range(1, DNumActors):

print('Actor ' + str(i) + ' Progress(' + str(ActorProgress[i-1]) + ') @ CPU ' + str(i - 1))

time.sleep(0.5); *# A little wait to not saturate the Processor*

print('Simulation finished by all the actors')

<https://sourceforge.net/p/electricdss/code/HEAD/tree/trunk/Version8/Distrib/Examples/Parallel_Processing/Python/Example_2/Example_2.py>

**import** **win32com.client**

**from** **win32com.client** **import** makepy

**import** **sys**

**import** **gc**

**import** **numpy** **as** **np**

**import** **time**

**from** **os** **import** system, name

sys.argv = ["makepy", "OpenDSSEngine.DSS"]

makepy.main()

DSSObj = win32com.client.Dispatch("OpenDSSEngine.DSS")

DSSText = DSSObj.Text

DSSCircuit = DSSObj.ActiveCircuit

DSSSolution = DSSCircuit.Solution

DSSParallel = DSSCircuit.Parallel

DSSBus=DSSCircuit.ActiveBus

DSSCtrlQueue=DSSCircuit.CtrlQueue

DSSObj.Start(0)

NCPUs=DSSParallel.NumCPUs

print(' total number of NCPUs', NCPUs )

DSSText.Command='ClearAll'

DSSText.Command='set parallel=No'

EndArray=[] *# for checking when all the actors are done*

yDelta=8760/(NCPUs-1)

ActorCPU =[]

**for** x **in** range(0, (NCPUs-1)):

print('Core Number',x)

**if** x != 0:

DSSParallel.CreateActor()

DSSText.Command='compile "C:/Program Files/OpenDSS/EPRITestCircuits/ckt5/master\_ckt5.DSS"'

DSSText.Command='Solve'

**if** x==NCPUs-1:

yDelta=8760-(NCPUs-2)\*yDelta

DSSText.Command='set mode=yearly totaltime=0 number=' + str(yDelta) + ' hour=' + str(x\*yDelta)

EndArray.append(1)

DSSText.Command='set parallel=Yes'

DSSText.Command='SolveAll'

BoolStatus = **False**

time.sleep(1);

**while** BoolStatus == **False**:

ActorStatus = list(DSSParallel.ActorStatus);

BoolStatus = ActorStatus == EndArray

ActorProgress=DSSParallel.ActorProgress

system('cls')

print('BoolStatus = ',BoolStatus,' ActorStatus = ',ActorStatus,' ActorProgress = ', ActorProgress)

**for** i **in** range(1,(NCPUs-1)):

DSSParallel.ActiveActor = i;

CHour = DSSSolution.dblHour;

print('Actor Time(hours)',CHour);

time.sleep(0.5)

print('Simulation finished');

PROBLEM STATEMENT:

We were having the ability to process a single circuit using opendss direct library at the moment. The time taken by the process to run multiple circuits sequentially, was relatively high.

INTRODUCTION

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.

EPRI has evolved OpenDSS into a more modular, flexible and scalable parallel processing platform we are calling OpenDSS-PM (opendss parallel machine).

The parallel machine

To create the parallel machine, OpenDSS-PM uses the actor model [[1-4](#_ENREF_1)]. 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 is choosed as the Direct DLL API for our use case. We 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.

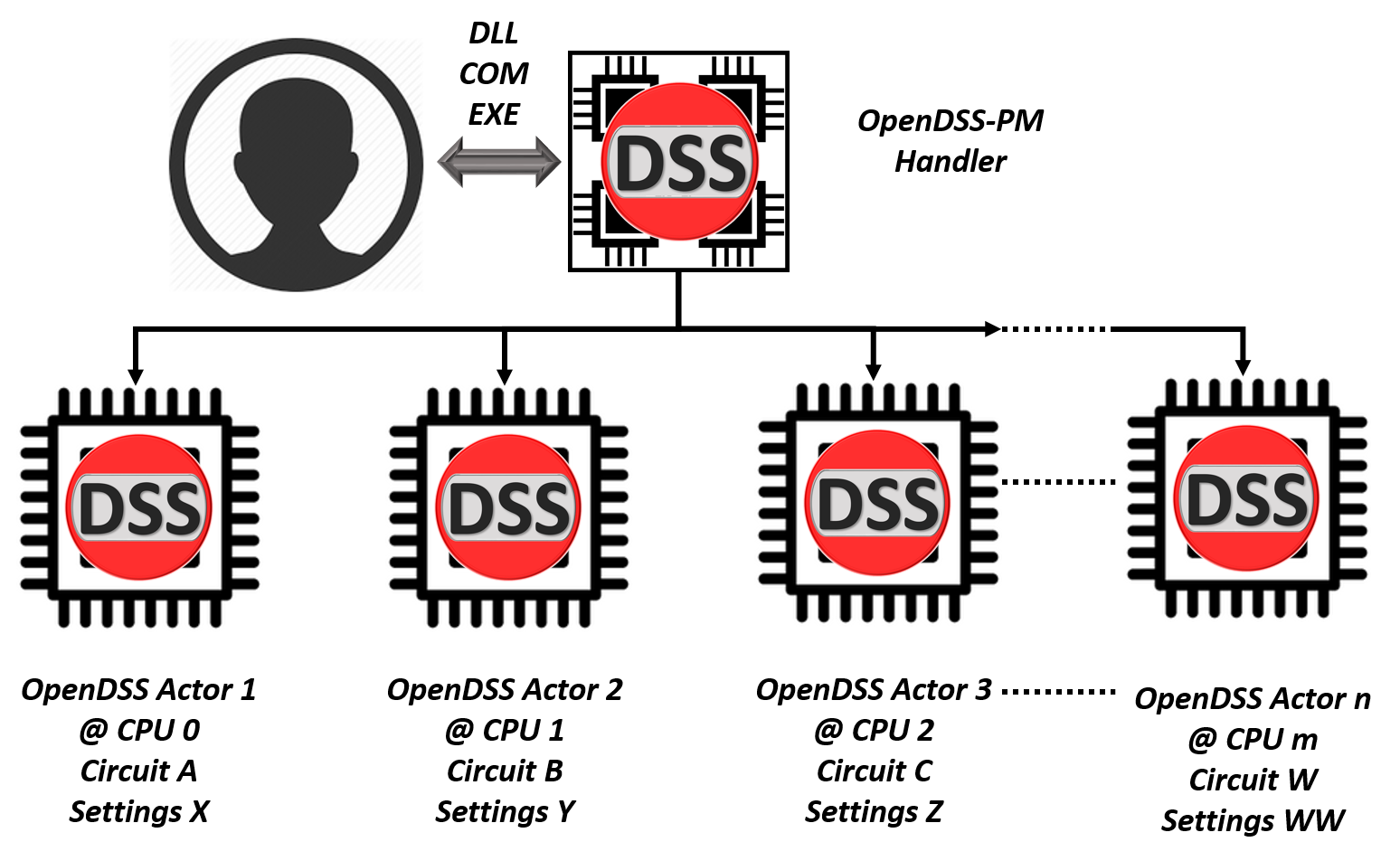


Figure 1. Operational scheme proposed for evolving OpenDSS into a parallel processing machine based on actors

As can be seen in Figure 1, the interface will work as the communication medium between the different actors on the parallel machine.

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:
   1. Solve the active actor
   2. 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.

1. 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). The number of CPUs will be equal to the number of threads of the active processor multiplied by the number of NUMA nodes.

### *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 number of Cores will be equal to the number of number of physical cores of the active processor multiplied by the number of NUMA nodes.

### *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.

### *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. If the value assigned is the star character (\*), all the commands after setting ActiveActor=\* will be executed for all the actors. This feature only works with the scripting tool or when using the Text interface in COM/DLL.

### *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. It 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.

### *Abort*

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

### *Clone*

This command clones the active circuit as many times as specified in the argument, for example, Clone 4 will create 4 actors with the same model as the one in memory at the time “Clone” is invoked. The argument needs to be a positive number greater than 0. If the number of clones requested exceeds the number of CPUs, OpenDSS will deliver and error message (7001). If the number of clones requested is invalid the error message will be 7004.

## ***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). |
| 7004 | The number of clones requested is not valid, this number cannot be 0 or a negative number. |

## ***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

Clone 2

set parallel=Yes

set activeActor=\*

set mode=yearly number=2000 totaltime=0

Set ActiveActor=1

Set hour=0

set activeActor=2

set hour = 2000

set activeActor=3

set hour = 4000

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*](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.

WHY MULTITHRADING DOESNT WORK

The process invoking the "parallel for" will multi-thread a separated code segment for handling variables within the local scope. Each thread has "all CPU" affinity, which means that you don't know where each iteration is taking place, the OS takes care of that based on CPU workload.

Why it doesn't work with OpenDSS? because OpenDSS is a process itself and creates threads internally. This means that for parallelizing multiple instances of OpenDSS, which is the worse way to go, you'll have to create a process for OpenDSS separately, increasing the number of threads used as a multiple of 2.  
Instead, using the parallel suite in OpenDSS the number of threads can increase linearly and the CPU control handled using OpenDSS commands.

Even if we try to create multiple instance of opendss, since the same application is opened several times (instances), not necessarily means that it will be assigned to a separate CPU. You have to allocate the thread using another mechanism. If you don't allocate threads on different processors, no matter how many times you try to clone you application, all these copies will be executed using the same processor affecting negatively the processing time.

If we copy the dll multiple times and try to access using multithreading, than also, windows can easily recognize the same app replicated and to avoid crashing it will put them on a queue. I tried what you propose several years ago (like 5) so, my advice is, look for a thread allocation mechanism using your programming language and clone your instances within your program, copying the DLL on disk that many times is not the answer

PERFORMANCE CONSIDERATION AND DEPENDENCY

Well, as a wise man I know says: "It all depends". IT depends on what you are trying to do and how are you doing it, let’s evaluate your system features:

OTHER CONSIDERATIONS AND RECOMMENDATION:

1) each actor has its own thread and the interface to the actors (the one you use to communicate them from the external program and gives you control over COM/DLL) has a separate thread. So, for example, lets assume you have created 5 actors, that means that your application is actually using 6 threads: 5 actors + the interface. So, when creating the actors we always recommend to leave at least 2 threads free for the OS and the calling application, otherwise, the PC will freeze and many processes may collapse. It is very easy to do parallel processing in OpenDSS, but you need to know what you are doing.  
Normally, problems related to convergence in a parallel environment are related to the way the models are initialized on each actor, the best way to initialize them is sequentially and then go for full parallel. Another source of errors is when modifying the Y Bus matrix, that requires to be done sequentially, so we recommend to do a snap solution on the actor after a modification has been done (no parallel, just do the snap on the actor that changed), for later keep going with the parallel simulation. This doesn't happen always, but is a good way to control those problems.

2) Yes, why not? It all depends on what you are looking for. As mentioned above, we always recommend to leave at least 2 threads free for the OS and the calling interface (your program). In standard PC architectures a processor core can handle up to 2 threads concurrently, however, it comes with a cost, lets suppose you have a processor with nominal operating frequency of 3.10 GHz and max (turbo) 5 GHz, which is what hyper-threading is intended for. So, with one thread per core, the thread will execute at 3.1GHz, but when using 2 threads in the same core you'll use the turbo freq 5GHz.  
Nevertheless, it's 5GHz for 2 threads, which ideally means that each thread will execute at 2.5 GHz, going slower. Additionally, your processor is contained in a very limited space, having to deal with the heat generated while operating. The computer knows that, and if the temperature goes up drastically, it will reduce the speed to keep the core at good temperature, affecting the simulation performance. That's why the super computers and servers are so expensive and complex, they require a very fascinating cooling infrastructure to keep them operating at nominal values.

DESIGN CONSIDERATION:

In our case a substation contains multiple feeders, and when we process multiple feeders in parallel feature operation, few considerations we have made to improve on processing time,

The processing time is directly proportional to the size of the network. So if we take a big network and a relatively smaller network, both the circuits will take roughly around the time taken by the big network, because although the smaller network will finish processing, it will be lying idle and waiting for the other one to finish. To overcome this situation, we are arranging the number of network to be run in parallel, in order of their sizes in descending order, and process them in batches (each batch contains 4 circuits, i.e. The number of cpu count).

SOME MORE RECOMMENDATION FROM OPENDSS TEAM:

For each instance of OpenDSS, the circuit solution has to go through the sparse matrix solver. Apart from the sparse solver time, the remaining time taken is proportional to the problem size.

there are a couple of good choices for exploiting the "low-hanging fruit" in the OpenDSS techniques without going to some completely different solution technique:

1. Divide the circuit into a bunch of smaller circuits and assign a separate processor to each instance of OpenDSS. This is what Davis is doing with some success. Of course, you have to write some code for solving the problem of interconnections between the smaller circuits and have a platform that can manage parallel processes such as LabView, or some other custom-written system.
2. Divide the problem in time (or in the case of Monte Carlo, in specific instances) and assign a processor to each time segment. If the problem has several independent segments of time, such as a day or a week, you could assign each processor to solve for one day or one week, etc. Of course, this depends on the ability of the operating system to keep instances truly independent. Standard Windows does not seem to be particularly good at this when there is I/O occurring for each time step, but a large server might work. Several on this forum have promised to look into it, but have not got back with their results.

PRECAUTIONS:

When working in parallel (set parallel=yes) if you start a simulation task you need to make sure that is finished before sending a new simulation task, otherwise all the data will collapse, please take that into consideration the next time (remember: everything is happening at the same time). You have to be careful when working in parallel, otherwise, just disable the feature to work sequentially.

Also the hardware resources are limited. The actors created in OpenDSS run with the highest priority in the core, which means that no other thread can be executed in the dedicated thread of the processor. for that reason we always recommend to leave 1 or 2 threads (CPUs) free for handling the graphics, OS tasks and others.