Experiments Report

January 20, 2017

Abstract

This report will include the discussion for the experiments. The experiments section will have data plotting and an initial analysis (model and discussion) based on the developed understanding. A Q & A subsection will follow after the discussion. I will add questions there that still need answering. It would be nice if others contributed with questions!

CPPTraj RMSD

The data reported are CPPTraj comparing experiments between Vanilla (MPI) execution and the task parallel execution of CPPTraj via RADICAL-Pilot. The experiments setup is the following:

- RMSD over 160000 frames as a single trajectory and as an ensemble of 2 trajectories that contain 80000 frames each. (105GB filesize)
- RMSD over 320000 frames as a single trajectory and as an ensemble of 4 trajectories that contain 80000 frames each. (209GB filesize)
- RMSD over 640000 frames as a single trajectory and as an ensemble of 8 trajectories that contain 80000 frames each. (418GB filesize)

The configuration was from a core per 80000 trajectories up to a node per 80000 frames. All experiments were done on Stampede using the Scratch filesystem.

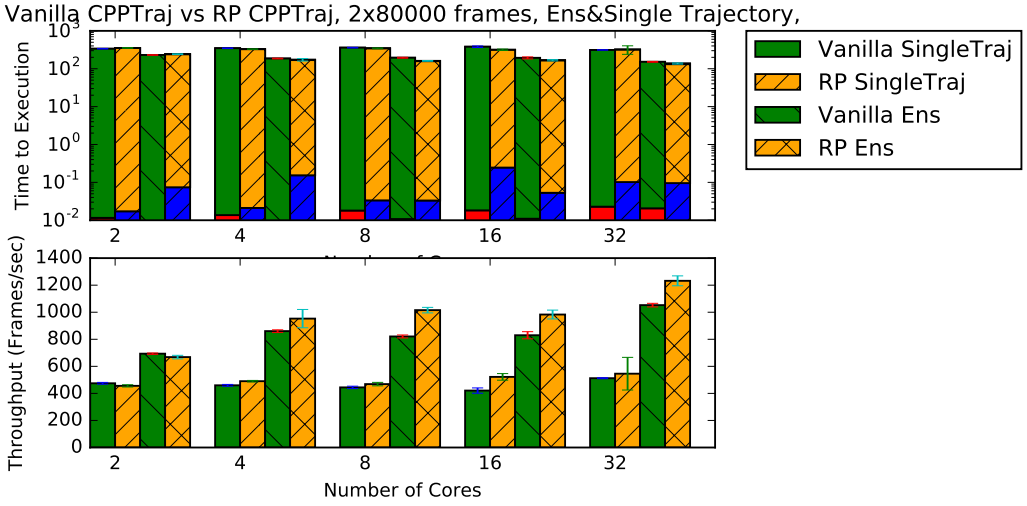


Figure 1: Time to Execution and Throughput comparison between different ways of executing the same CPPTraj analysis. There are in total 160K frames organized as a single trajectory file for the Single trajectory case and as an ensemble of 2 trajectories for the ensemble case

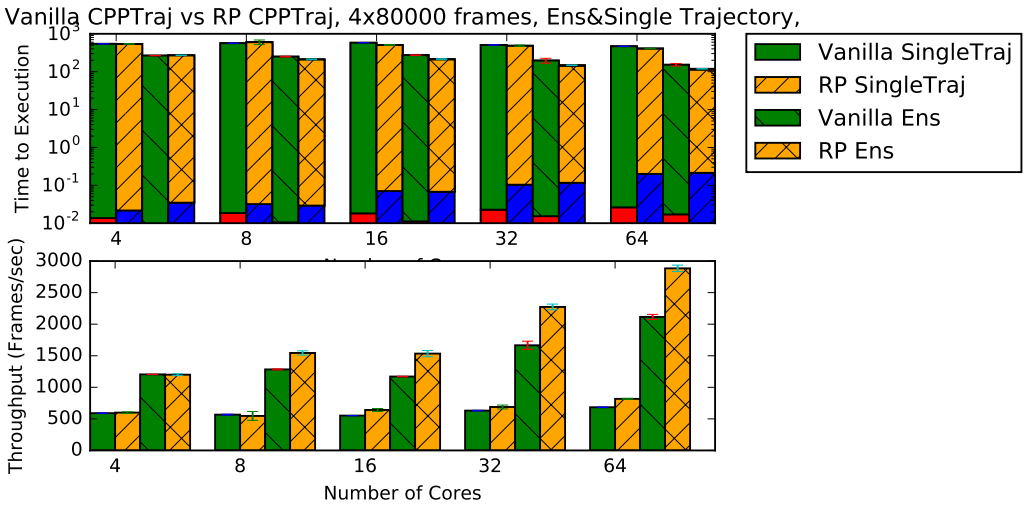


Figure 2: Time to Execution and Throughput comparison between different ways of executing the same CPPTraj analysis. There are in total 320K frames organized as a single trajectory file for the Single trajectory case and as an ensemble of 4 trajectories for the ensemble case.

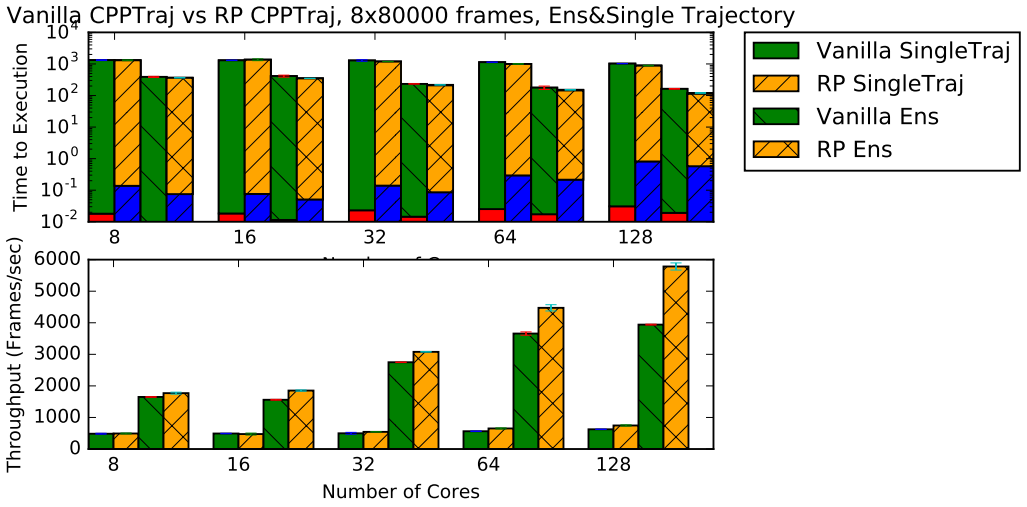


Figure 3: Time to Execution and Throughput comparison between different ways of executing the same CPPTraj analysis. There are in total 640K frames organized as a single trajectory file for the Single trajectory case and as an ensemble of 8 trajectories for the ensemble case.

The top subplot show the Execution time for Vanilla and RADICAL-Pilot. The bottom subplot shows the Average Throughput. In all figures the order of the bars is from right to left:

1 Single Trajectory Vanilla,

2 RP-CPPTraj single trajectory,

3 CPPTraj Vanilla Ensemble and

4 RP-CPPTraj Ensemble

One important note to make, is that as the core count increases, the MPI implementation does not scale

in ensemble case as the task level parallel for the 320K, Figure 2, and 640K frames, Figure 3. The main difference between those two is that the CPPTraj execution via RADICAL-Pilot introduces a small delay between the launching of each CPPTraj process. I believe that this delay reduces the strain CPPTraj's MPI implementation puts to the filesystem and the data are read faster. In the next set of experiments with RMSD, I want to find the filesize, or better the system size, where the MPI implementation cannot scale anymore and the task level parallel can.

The reason behind the above statement is the fact that throughput remains relatively stable. Throughput, here measured as frames per second, is the amount of computed data per time unit. We can say it is the computation velocity. Throughput is a function of input rate and the number of computing blocks. By computating blocks, I mean a self contained element that takes an input, does some sort of processing on the input and gives an output. In this case,it can either be a MPI process or a task.

Assuming that the input rate, throughin, is infinite and it can feed continuously and steadily any number of computing blocks, the throughput will increase linearly as we increase the number of computing blocks. Say that such a block can process N inputs per time unit. Adding a second computing block $2N$ inputs per time unit can now be processed. Thus, with K computing blocks the throughput is KN inputs per time unit. It is now established how throughput changes when the computation blocks vary and the input rate is large enough to accomodate any number of them.

Assume now that the input rate is finite to a maximum of M inputs per time unit. In case $M < N$, throughput is dectated by the input rate. In case $M \geq N$, throughput will increase linearly as long as the number of computing blocks is less or equal to $\lfloor \frac{M}{N} \rfloor$. When the number of computing blocks, becomes larger than the previous number, throuput flats to a rate equal to the rate in which the input is produced.

The question that needs to be answered now is what is the rate that CPPTraj reads in data. The experiments will read the file and do nothing else.

[Update 12/9/2016] CPPTraj was executed with the following ways, using the Scratch filesystem:

1 By using numactl and binding the processes in specific cores. Specifically: ibrun -np 4 numactl –physprocbind=0,7,15,23

2 By requesting 1 nodes and 2 tasks and 2 nodes and 4 tasks.

The change in the execution time was as small as already reported. So I thought, what will happen if the tasks used in the RP execution were more than the cores by a factor of 2 and a factor of 4. Experiments are running and will report further the next days.

[Update 12/16/2016] CPPTraj was executed with the following ways, for a 50GB trajectory file and a 100GB trajectory file:

1 1 process per core on Stampede using the Work filesystem

2 1 process per physical processor on Stampede using the Work filesystem

3 1 process per physical processor on Stampede using the Scratch filesystem

The execution produced the following figures.

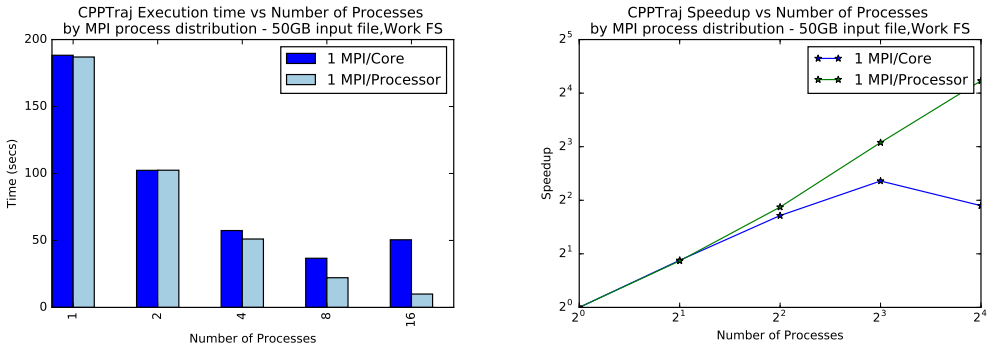


Figure 4: Execution time (L) and Speedup(R) of CPPTraj for a 50GB trajectory file. The figure compares the time to execution between 1 process per core - Stampede has 2 processors with 8 cores each - vs 1 process per physical processor. The Work Filesystem was used.

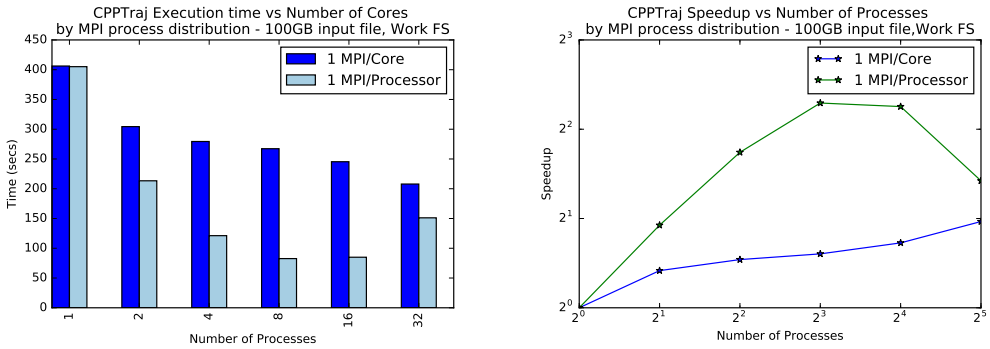


Figure 5: Execution time (L) and Speedup(R) of CPPTraj for a 100GB trajectory file. The figure compares the time to execution between 1 process per core - Stampede has 2 processors with 8 cores each - vs 1 process per physical processor. The Work Filesystem was used.

As we can see in figures 4 and 5 the MPI execution scaling is quite different to the case of the 50GB file and the 100GB file. What is interesting is that for the 100GB file (figure 5) scaling stop at 8 MPI processes equally distributed as one process per physical processor and it is reverted when 32 processes were introduced.

Because these results are different, in how CPPTraj behaves, I compared the behavior of the two filesystems, provided by Stampede. There I only used 1 process per physical processor. If somebody feels that there is a need for comparing 1 process per core, I will run the experiments. In figures 6 and 7, we see how CPPTraj behaves when we change the filesystem used. Although this characterizes the filesystems, it shows us that Workis mainly optimized for execution and Scratch for Storage. It also shows a point where CPPTraj parallel MPI implementation stops scaling, i.e. 100GB, 16 and 32 processes, 1 process per physical core. It would be interesting to see if executing the same exactly configuration with RADICAL-Pilot will offer anything.

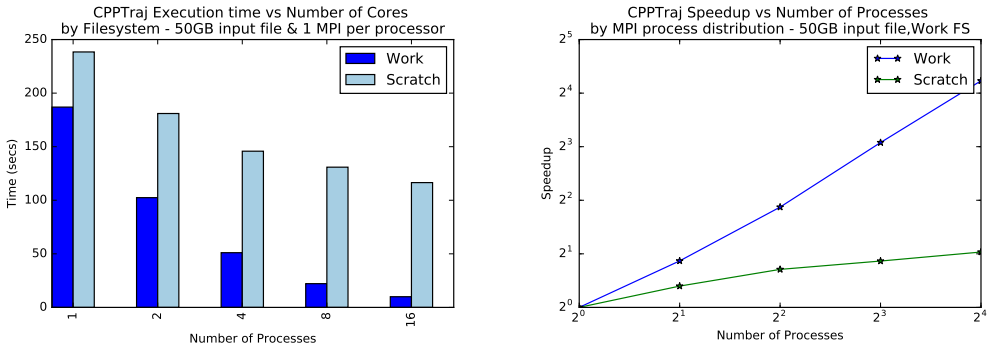


Figure 6: Execution time (L) and Speedup(R) of CPPTraj for a 50GB trajectory file. The figure compares the time to execution between Work and Scratch filesystems. 1 process per physical processor was used.

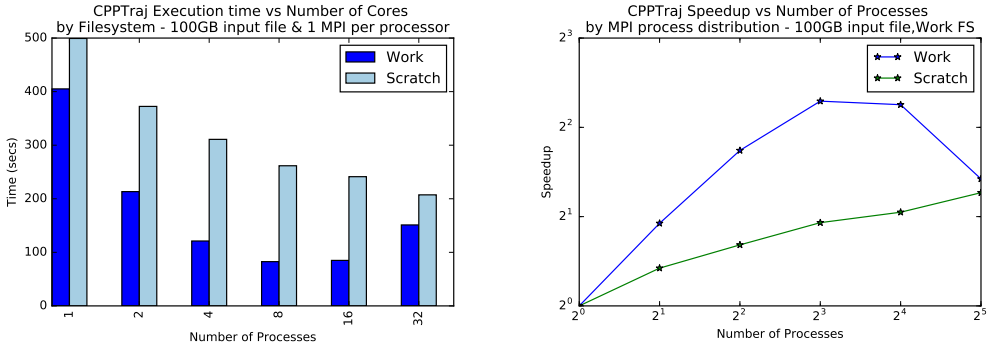


Figure 7: Execution time (L) and Speedup(R) of CPPTraj for a 100GB trajectory file. The figure compares the time to execution between Work and Scratch filesystems. 1 process per physical processor was used.

One way to understand why the two filesystems behave so differently is to see what is the Read Rate (MB/s) that can be achieved when CPPTraj is executed. Figure 8 shows how the filesystems behave. The experiment was done with the 50GB and 100GB trajectory files with 1 MPI process per physical processor. 1 process per physical processor was selected because it is the one that scales as expected over the Work filesystem.

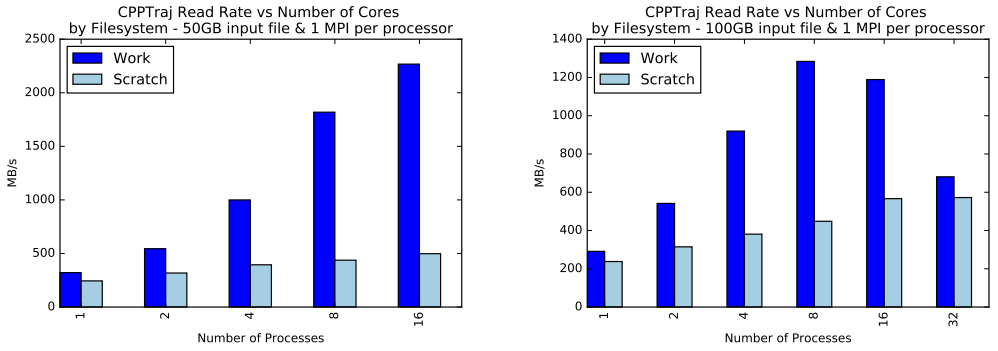


Figure 8: Read Rate in MB per seconds for the 50GB (L) and 100GB (R) trajectory files vs number of processes. The figures compare the read rates between Work and Scratch filesystems. 1 process per physical processor was used.

[Update 12/19/2016]

The following result is extremely interesting. Figure 9 compares the execution of the 100GB file between CPPTraj MPI and RADICAL-Pilot using the single core CPPTraj implementation for the units. The execution was done on the Work Filesystem of Stampede. Since we have established that the execution is depends highly by the filesystem for this type of application, it is reasonable to assume that task-parallel execution relieves the filesystem from the strain the MPI execution creates. It needs though further experimentation to ensure that this is the case and that the results were not just a lucky hit.

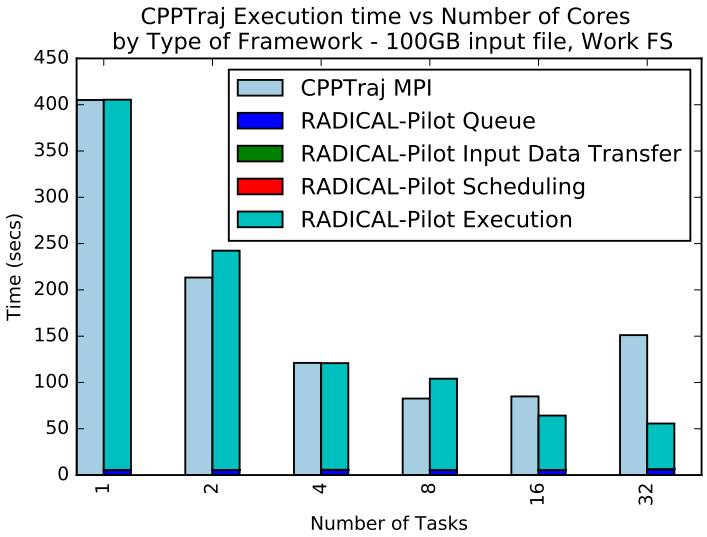


Figure 9: Execution time of CPPTraj MPI and Task level parallel CPPTraj execution through RADICAL-Pilot for a 100GB trajectory file. In the MPI case 1 processes per physical processor was used. For the RADICAL-Pilot execution 1 unit per 8 cores were submitted to achieve the desired 2 units per node.

[Update 1/6/2017]

An experiment was executed where the stripe count was supposed to change in Scratch. According to TACC documentation the stripe count should be set to the number of nodes that are accessing the file. The experiment preserves the 1 MPI process per physical processor. Figure 10 shows the results of the experiment. Despite the fact that the default stripe count was changed by the script that was executed, the actual number of stripes for the file did not change and remain to 2. The number of stripes should be set when a file is written, so that it can be distributed over different storage targets.

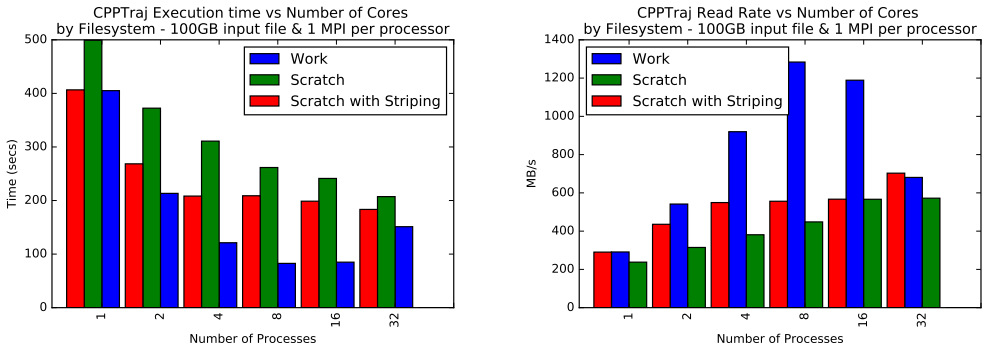


Figure 10: Execution time (L) and Speedup(R) of CPPTraj for a 100GB trajectory file. The figures compare the time to execution and read rate between Work, Scratch with default striping and Scratch with the number of stripes equal to 2. 1 MPI process per physical processor was used. The input data is 100GB

[Update 1/10/2017]

An experiment was executed where the stripe count changes in Scratch. According to TACC documentation the stripe count should be set to the number of nodes that are accessing the file. The experiment preserves the 1 MPI process per physical processor. Figure 11 shows the results of the experiment. The default number of stripes is 2 in Scratch. That means that the file is being stored as two components to two Storage Targets. This number of stripes is enough for execution up to two nodes. For the configuration between 8 and 32 processes, 4 to 16 nodes were acquired. In those cases the number of stripes are equal to the number of nodes.

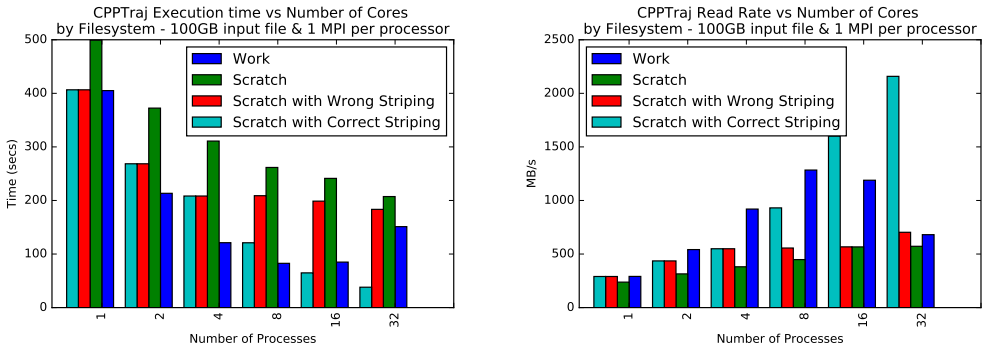


Figure 11: Execution time (L) and Speedup(R) of CPPTraj for a 100GB trajectory file. The figures compare the time to execution and read rate between Work, Scratch with default striping and Scratch with the number of stripes equal to the number of nodes accessing the file. 1 MPI process per physical processor was used. The input data is 100GB

Hausdorff Distance

The data reported here are comparing the Hausdorff Distance calculation via a RADICAL-Pilot and a Spark implementation. Both use the same implementation for the main function and both use parallel read. The experiments were executed over 192 trajectories of CA atoms on Comet. Figure 12 show the mean time to execution for all three cases of trajectory size for the CA atoms.

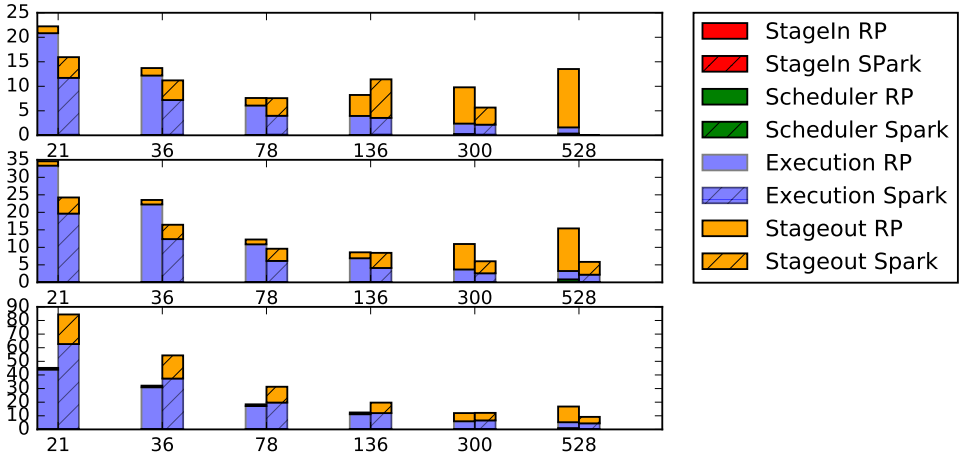


Figure 12: Time to Execution Hausdorff Distance. From top to bottom Short trajectory, Medium Trajectory, Long Trajectory

Figures 13,14 and 15 show in more detail how the execution between the two frameworks differ. Two main differences is that RADICAL-Pilot spends more time to stage in and stage out data to and from a task, where Spark is more expensive in scheduling tasks.

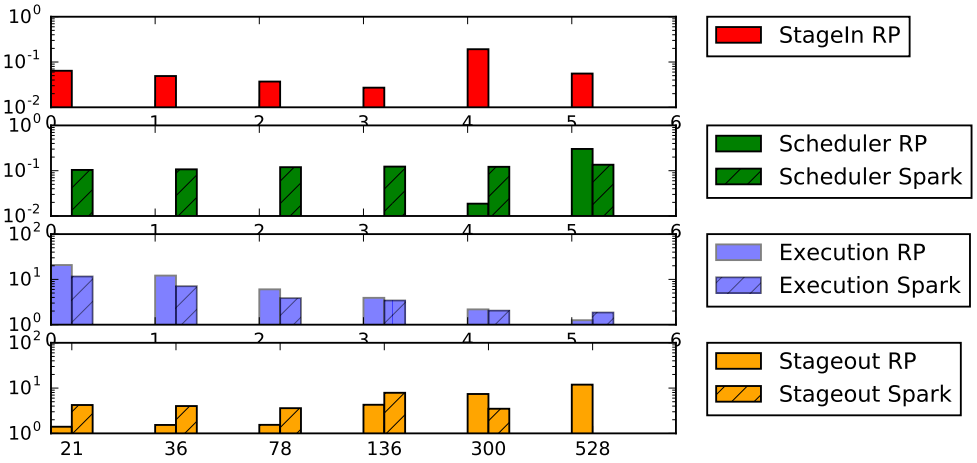


Figure 13: Execution break down between RADICAL-Pilot and Spark execution. The Y axis represents the time spent in each part and the X axis the number of core. This is short trajectory size

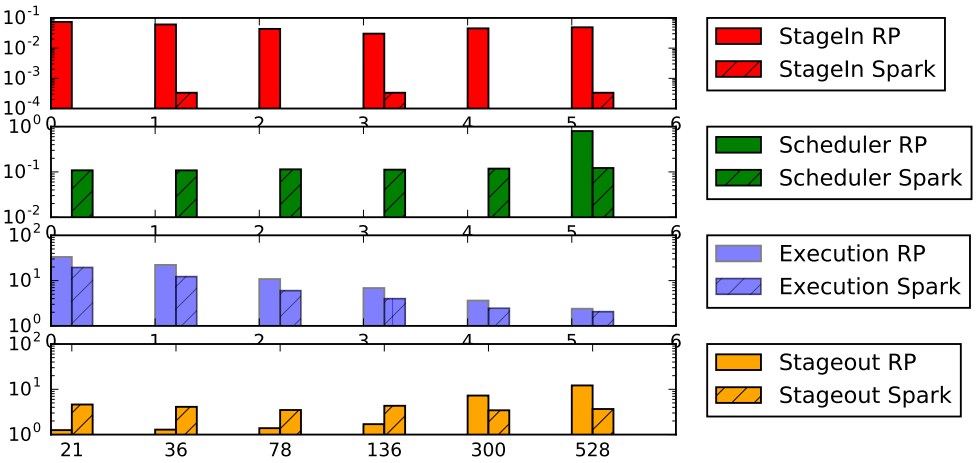


Figure 14: Execution break down between RADICAL-Pilot and Spark execution. The Y axis represents the time spent in each part and the X axis the number of core. This is medium trajectory size

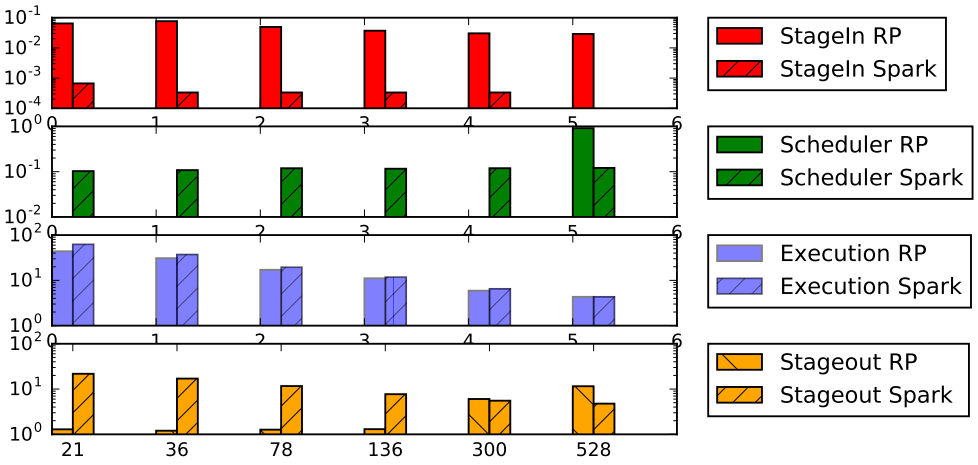
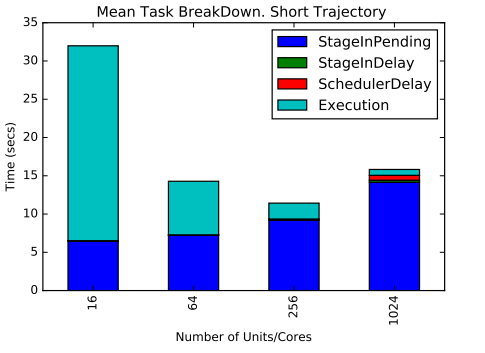


Figure 15: Execution break down between RADICAL-Pilot and Spark execution. The Y axis represents the time spent in each part and the X axis the number of core. This is Long trajectory size

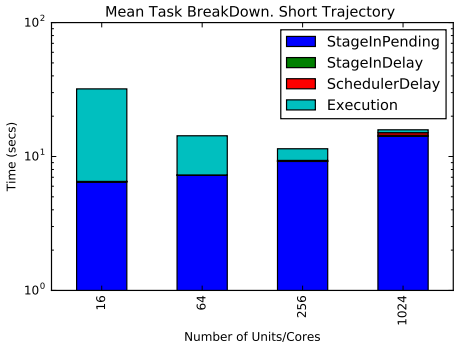
Extra experiments were executed with the following configuration on Comet:

- Number of trajectories is 128
- Number of atoms 214 and frames 102. System sizes are Small, Medium and Large. This is equivalent to 214 atoms per frame, 428 and 856 respectively.
- The whole matrix was calculated
- Core count: 16, 64, 256 and 1024
- Execution was done with RADICAL-Pilot and Pilot-Spark.

The following figures show that comparison.

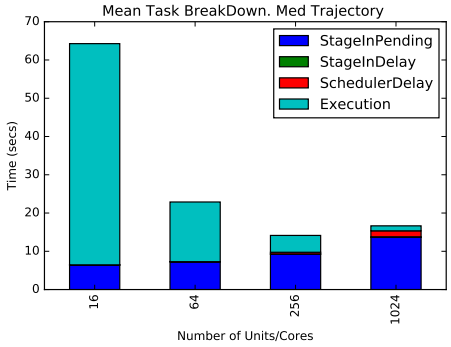


Average Task execution time for four core counts. 128 trajectories were compared. Each trajectory has 102 frames of 214 atoms. The execution partitioned the calculation of the distance matrix into the calculation of a number of submatrices (16,64,256,1024).Resource: Comet,Date:12/1/2016

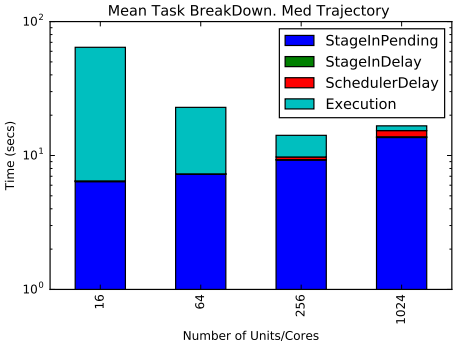


Average Task execution time for four core counts. 128 trajectories were compared. Each trajectory has 102 frames of 214 atoms. The execution partitioned the calculation of the distance matrix into the calculation of a number of submatrices (16,64,256,1024).Resource: Comet,Date:12/1/2016

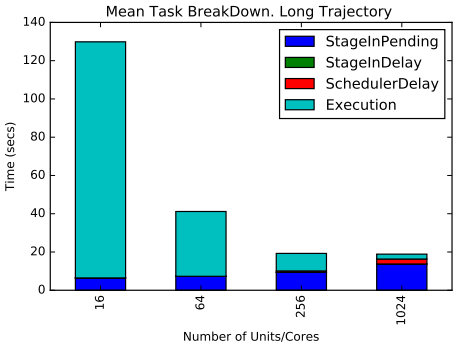
Figure 16:



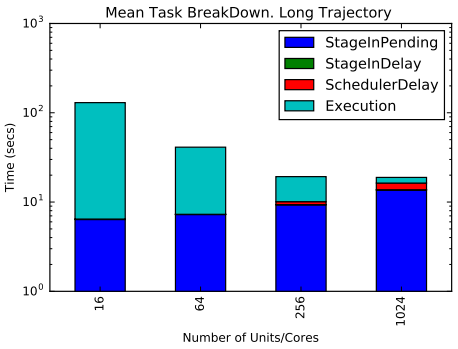
Average Task execution time for four core counts. 128 trajectories were compared. Each trajectory has 102 frames of 428 atoms. The execution partitioned the calculation of the distance matrix into the calculation of a number of submatrices (16,64,256,1024).Resource: Comet,Date:12/1/2016



Average Task execution time for four core counts. 128 trajectories were compared. Each trajectory has 102 frames of 428 atoms. The execution partitioned the calculation of the distance matrix into the calculation of a number of submatrices (16,64,256,1024).Resource: Comet,Date:12/1/2016



Average Task execution time for four core counts. 128 trajectories were compared. Each trajectory has 102 frames of 856 atoms. The execution partitioned the calculation of the distance matrix into the calculation of a number of submatrices (16,64,256,1024).Resource: Comet,Date:12/1/2016



Average Task execution time for four core counts. 128 trajectories were compared. Each trajectory has 102 frames of 856 atoms. The execution partitioned the calculation of the distance matrix into the calculation of a number of submatrices (16,64,256,1024).Resource: Comet,Date:12/1/2016

[Update 12/9/2016]

Extra experiments were executed with the following configuration on Comet:

- Number of trajectories is 128
- Number of atoms 3341 and frames 102. System sizes are Small, Medium and Large. This is equivalent to 3341 atoms per frame, 6682 and 13364 respectively.
- The whole matrix was calculated
- Core count: 16, 64, 256 and 1024
- Execution was done with RADICAL-Pilot and Pilot-Spark.

The following figures show that comparison.

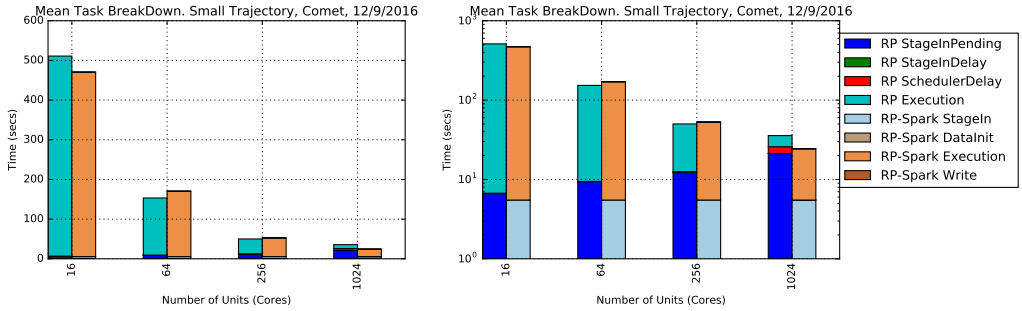


Figure 17: Average Task execution time for four core counts. 128 trajectories were compared. Each trajectory has 102 frames of 3341 atoms(Small size trajectory). The execution partitioned the calculation of the distance matrix into the calculation of a number of submatrices (16,64,256,1024)

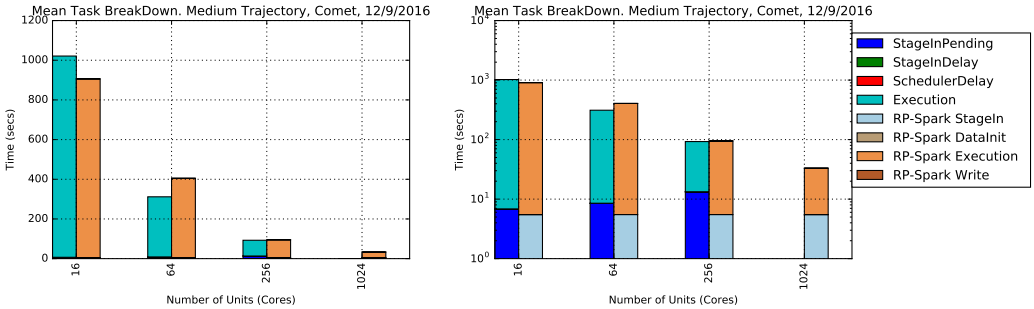


Figure 18: Average Task execution time for four core counts. 128 trajectories were compared. Each trajectory has 102 frames of 6682 atoms(Medium size trajectory). The execution partitioned the calculation of the distance matrix into the calculation of a number of submatrices (16,64,256,1024)

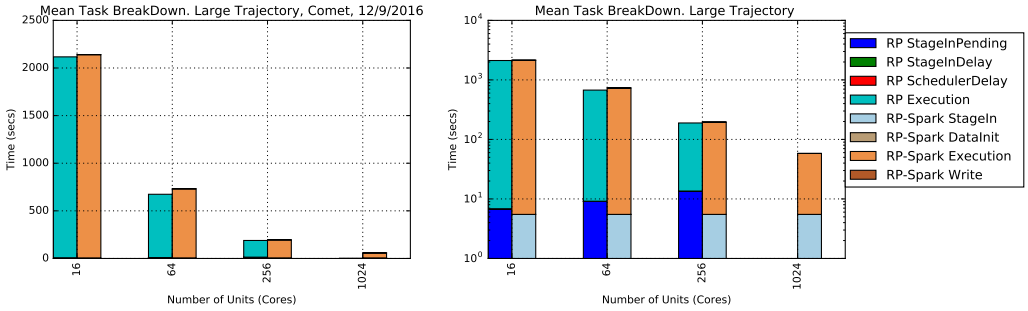


Figure 19: Average Task execution time for four core counts. 128 trajectories were compared. Each trajectory has 102 frames of 13364 atoms (Large size trajectory). The execution partitioned the calculation of the distance matrix into the calculation of a number of submatrices (16,64,256,1024)

Insight: In these type of applications both the task level parallel approach and the Spark approach show significant speed up as the number of cores increase.Although, in the case of the task level parallel approach, we see that speed up flattens when the time needed for the last task to reach the Staging input queue becomes twice as large as the execution time of a task. We realize that by looking at the case of Figure 17. This is due to the usage of the remote database in RADICAL-Pilot. To make matters more clear look at table 1. This table shows the times for the Small sized trajectory of these experiments. For the execution through

Spark, that specific time can be considered as part of the overheads that are introduced by RADICAL-Pilot (actually I would like to write Pilot-Spark, but I was not sure if it was the correct term).

Units	RP StageIn-Pending	RP StageIn-Delay	RP SchedulerDelay	RP Execution
16	6.627337	0.079313	0.003966	504.244539
64	9.361523	0.064966	0.003518	143.894596
256	12.226125	0.035643	0.254092	37.532336
1024	21.220789	0.034938	4.600481	9.847551

Table 1: Average Task execution time for four core counts. 128 trajectories were compared. Each trajectory has 102 frames of 3341 atoms.

The question raised after that was how did Spark tasks spent most of the time. Figure 20 shows that the largest percentage of the execution of the SPark was spent in the computation rather than scheduling or data movement from the workers to the driver.

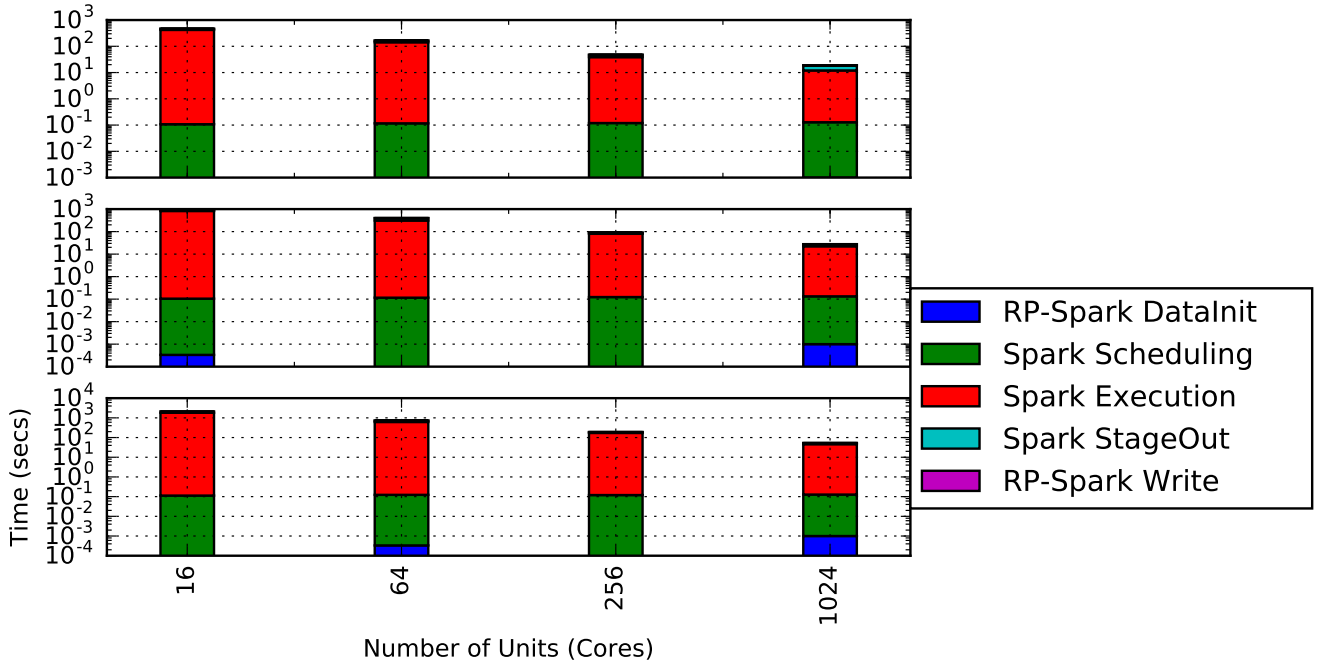


Figure 20: Spark's Average task execution. The top subplot is for the Small trajectory, the middle for the Medium and the bottom one for the Large

Useful Definitions

- N_{Tr}: Number of Trajectories per task
- N_A: Number of atoms in a frame.
- N_F: Number of frames per trajectory. All trajectory files have the same number of frames, 102.
- N_T: Number of tasks
- α: the coefficient of Staging In

- β: the coefficient of the Scheduling delay
- γ: the coefficient of the Execution
- δ: the coefficient of the Staging Out

Analysis

The execution model can be easily broken to different parts. First part of the model is data StageIn. In case of RADICAL-Pilot StageIn is rather easy to undeerstand. In case of Spark, I consider as StageIn the part of the code that is written before partitioning the data. Second part is the time need to schedule a task. Third is the actual execution of the task, which can be broken further more to read, exec and write. Finally, the last part of the model is the time necessary to stage out the data. In case of RADICAL-Pilot it is easy to understand. In Spark, I consider as the time needed from the time that all tasks have returned their data until the end of the script.

***giannis: Why is the need for every parameter?

Essentially, the model will look like:

$$T = \alpha(N_I) + \beta N_T + \gamma Y + \delta\Big(N_O S_O + \frac{k(k+1)}{2}\Big)$$

Y is the tme of the execution of the task.

Task Execution Analysis

That is dependent to the number of trajectories being processed and the number of points in each trajectory. Let T_N be the number of trajectories per task and T_S the size of each trajectory, i.e. the number of points.Thus, the above execution time can be

$$Y = O((N_T r N_F N_A)^2)$$

Let dH be the time to calculate the Hausdorff distance between two trajectories. The following algorithm describes it in pseudocode. The description will help the following analysis:

```
1:  procedure HausdorffDistance(T1,T2)          ▷ T1 and T2 are a set of 3D points
2:      for ∀t1 in T1 do
3:          for ∀t2 in T2 do
4:              Append in D1 calculated d(t1,t2)
5:          end for
6:          Dt1 append max(D1)
7:      end for
8:      N1 = min(Dt1)
9:      for ∀t2 in T2 do
10:         for ∀t1 in T1 do
11:             Append in D2 calculated d(t2,t1)
12:         end for
13:         Dt2 append max(D2)
14:     end for
15:     N2 = min(Dt2)
16:     return max(N1,N2)
17: end procedure
```

Thus, the complexity of dH is

$$dH = \mathcal{O}({T_S}^2) + T_S\mathcal{O}(T_S) + \mathcal{O}({T_S}^2) + T_S\mathcal{O}(T_S)$$

RP Fitting

Before solving any system, I will add some assumptions that will try to simplify the model as well as better understand how the scheduler and data stage in overheads affect the execution.

- RADICAL-Pilot's staging output does not affect the execution pattern of the tasks. All tasks are executed in parallel and there is no data movement from the executing resource to the client. Also, there is no continuation in the execution since only one pass is done over the data.
- Any delay between the moment a Compute Unit goes to the Executing state and starts execute as well as the delay from the moment it returns until it moves to the next stage, are assumed as part of the execution time of the task.
- Parameter γ is set to 1 in this case, because the timing parameters for the execution are captured inside Y.

The execution time of the tasks depend on the number of trajectory files that it gets, the number frames per trajectory and the number of atoms. The number of atoms affects the exeution time linearly, the number of frames and the number of trajectories affect in a squared fashion.

Thus $T_{Ex}(N_{Tr}, N_A, N_F) = e_1 N_{Tr}^2 N_F^2 N_A + e_2$

After appling the least square method the values found for e_1 and e_2 are:

$$e_1 = 1.33271794e-08 \quad e_2 = 8.09404416e-02$$

The χ^2 test was applied to the following points:

Trajectories	Frames	Atoms	Observed Time	Predicted Time
32	102	214	25.46786774	30.46545617
16	102	214	7.00346226	7.67706937
8	102	214	2.0758777	1.97997267
4	102	214	0.76485454	0.5556985
32	102	418	57.81401424	60.84997191
16	102	418	15.59695303	15.27319831
8	102	418	4.39942563	3.87900491
4	102	418	1.3047394	1.03045656
32	102	856	123.41059877	121.61900337
16	102	856	33.88301111	30.46545617
8	102	856	9.16800458	7.67706937
4	102	856	2.56352096	1.97997267

The tested null hypothesis is the observed values belong to the predicted distribution

The χ^2 is calculated with the following formula:

$$\chi^2 = \sum_{1}^{12} \frac{(O_i - P_i)^2}{P_i}$$

The χ^2 value is 2.13474849056. This value is smaller than the $\chi_0^2.05$ for 11 degress of freedom, thus we cannot reject the null hypothesis for the 0.05 significance level.

The Last Unit arrival time is considered as the time the last CU entered the State AgentStagingInput where input staging start to be executed.

The Least Squares Fit was done over the points obtained for 16, 64, 256 and 1024 CUs. The line that was obtained is the following

$$T_A(N_T) = 0.01313076N_T + 7.66860076$$

***giannis: TODO: Update and verify with new dataset.

Spark Fitting

The Spark execution model requires one extra term. That term is shows the time that is needed to for Spark to launch its executors. An executor is the process responsibe to execute the tasks. An assumption that is made is the fact that stagein in the Spark execution does not significantly change the execution time since it creates an RDD with the paths where the file are. So, the model will be

$$T = \beta\frac{k(k+1)}{2} + Y + \delta\Big(N_O S_O + \frac{k(k+1)}{2}\Big) + \epsilon W$$

where W is the number of worker nodes.

***giannis: TODO: Solve the system

1 Leaflet Finder

[Update 1/20/2017]

Despite the above I was able to run the 131K example with 64,256 and 1024 cores in total. A comparison between the execution time of the previous implementation and the use of cdist is shown in Figure 21. As it can be seen, the cdist function from Scipy is quite faster, but 8 times more expensive in terms of memory needs, since it uses double precision floating numbers.

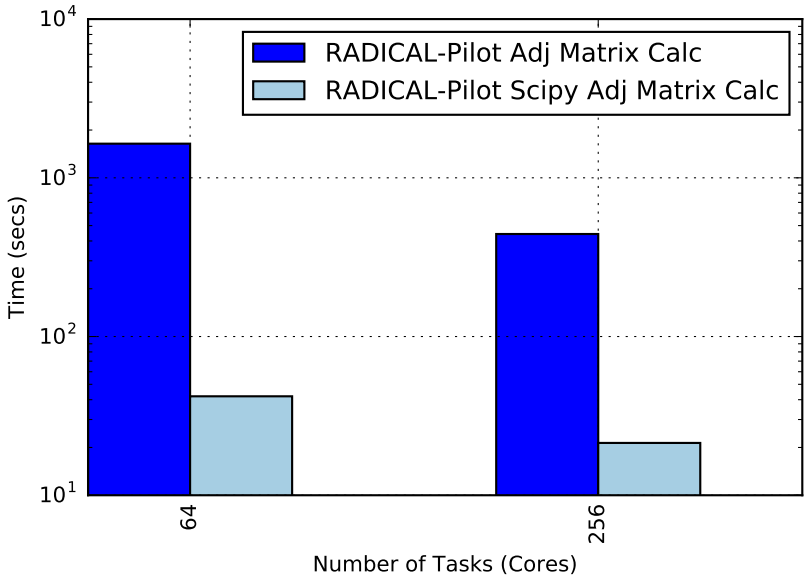


Figure 21: Comparing the execution time for calculating the adjancency matrix for the 131K leaflet finder data points. The comparison is between a numpy custom implementation and the cdist function provided by Scipy

This is the explanation about cdist usage. cdist calculates and returns the pair-wise Euclidean distance of two 2-D vectors. It is doing so by creating a n^2, where n the first dimension of the vectors, double floating point matrix. Each double floating point number is 8 bytes long. Thus, there is a need of $8 * n^2$ bytes to store the result.

When a command line: $a = (cdist(b, b) < c)$ is executed cdist returns a double floating point number and the logical operation in the parenthesis returns boolean. The garbage collector then removes the result that was returned from cdist. The Boolean matrix requires n^2 bytes to be stored.

I will use the leaflet finder distance calculation as an example. Let's assume that we want to calculate the distances of 131072 atoms in one node on Comet, i.e. 24 cores and 128GB of RAM.

- Scenario 1: Let's assume that we want to calculate the whole result concurrently in one node. That means that the memory needs are equivalent to keeping a double floating point matrix of 131072*131072*8 bytes, which is 128GBs.
- Scenario 2: Let's assume that the data are partitioned in blocks of 4096 and each task calculates a 4096*4096 matrix. The memory required at any given time, from cdist, would be 4096*4096*8*24 bytes, which is around 3GBs.
- Scenario 3: Let's assume that the data are partitioned in blocks of 4096 and each task calculates a 4096*131072 matrix. The memory required at any given time, from cdist, would be 4096*132072*8*24 bytes, which is around 96GBs.

Comparing the RADICAL-Pilot implementation with the Spark implementation, with the same data set gave the following result, Figure 22

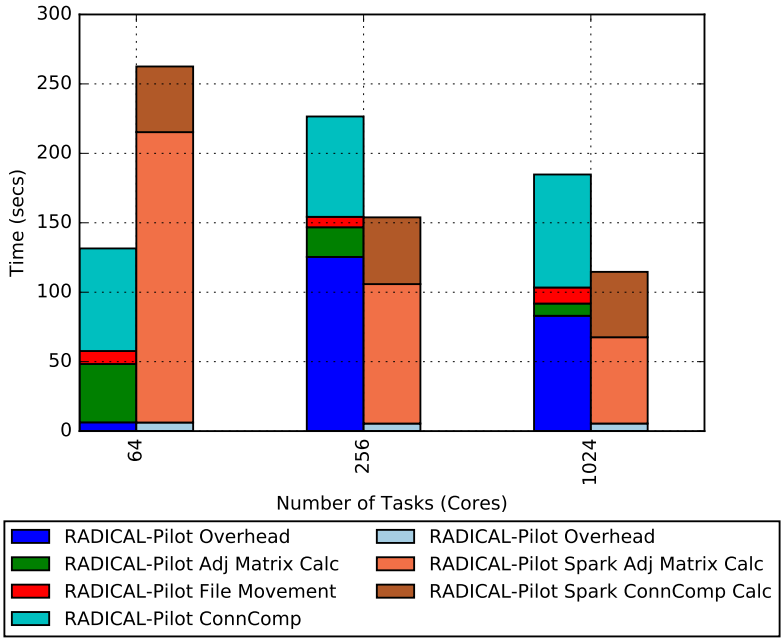


Figure 22: Comparing the execution time for leaflet finder 131K data points. The comparison is between RP-Vanilla and RP-Spark implementations. Both use cdist for calculating the distances