Experiments Report

November 12, 2016

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.

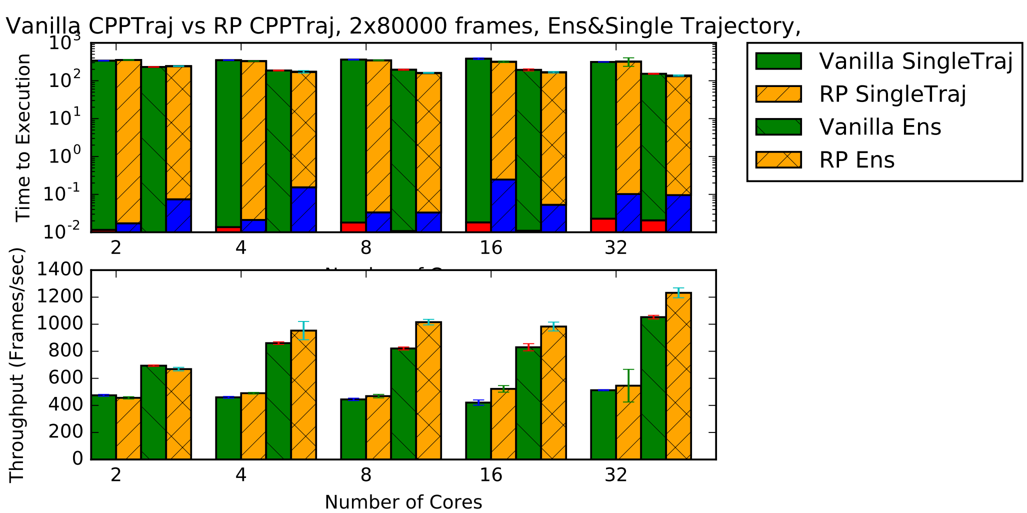

Figure 1: Time to Execution and Throughput comparison between different ways of executing the same CPPTraj analysis. There are in total 160K frames organized as a single trajectory file for the Single trajectory case and as an ensemble of 2 trajectories for the ensemble case

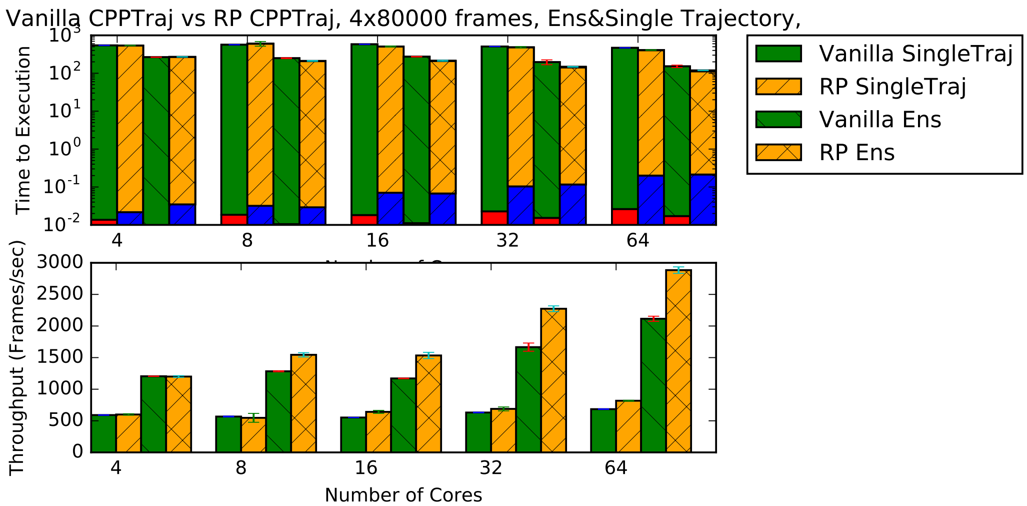

Figure 2: Time to Execution and Throughput comparison between different ways of executing the same CPPTraj analysis. There are in total 320K frames organized as a single trajectory file for the Single trajectory case and as an ensemble of 4 trajectories for the ensemble case.

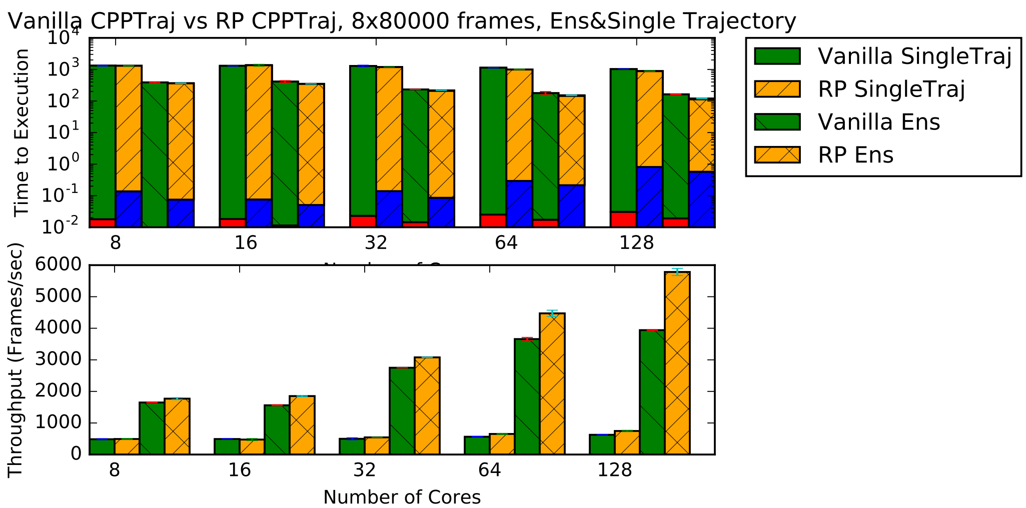

Figure 3: Time to Execution and Throughput comparison between different ways of executing the same CPPTraj analysis. There are in total 640K frames organized as a single trajectory file for the Single trajectory case and as an ensemble of 8 trajectories for the ensemble case.

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.

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 4 show the mean time to execution for all three cases of trajectory size for the CA atoms.

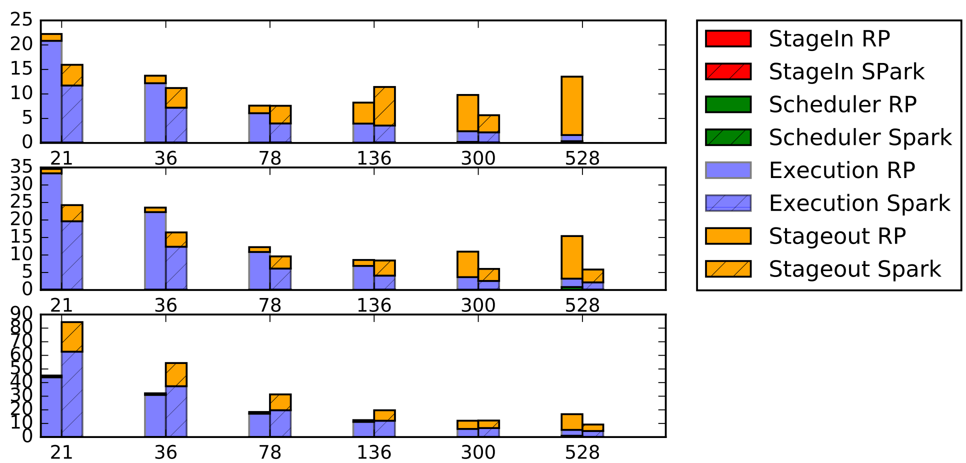

Figure 4: Time to Execution Hausdorff Distance. From top to bottom Short trajectory, Medium Trajectory, Long Trajectory

Figures 5,6 and 7 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.

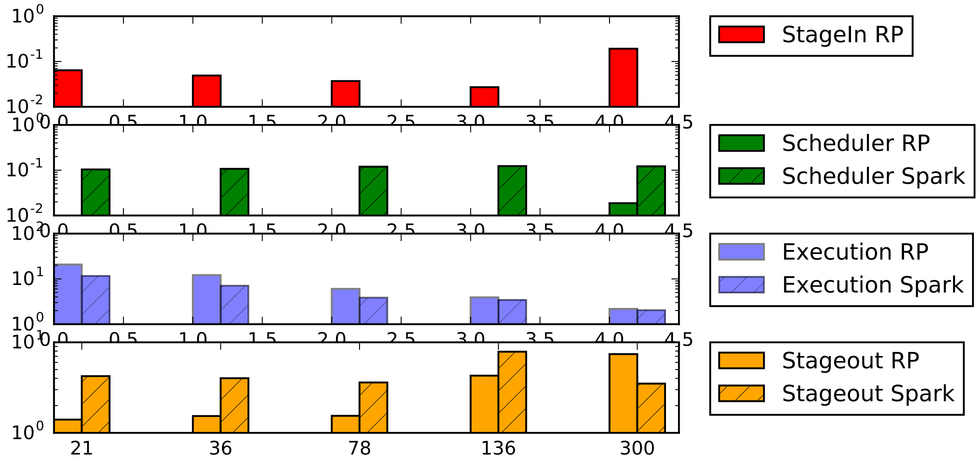

Figure 5: Execution break down between RADICAL-Pilot and Spark execution. The Y axis represents the time spent in each part and the X axis the number of core. This is short trajectory size

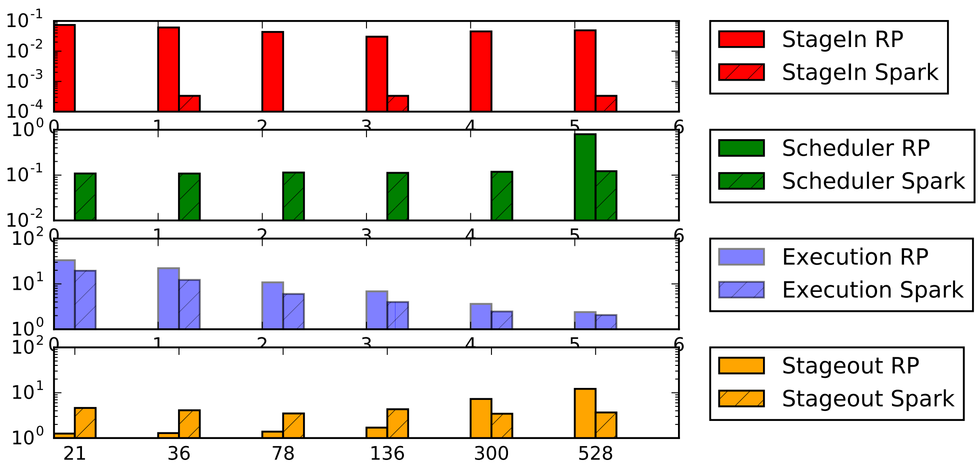

Figure 6: Execution break down between RADICAL-Pilot and Spark execution. The Y axis represents the time spent in each part and the X axis the number of core. This is medium trajectory size

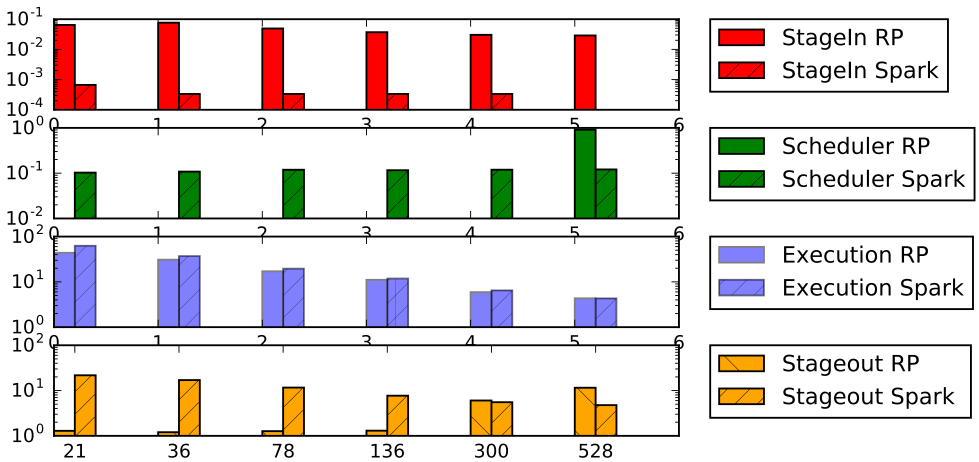

Figure 7: Execution break down between RADICAL-Pilot and Spark execution. The Y axis represents the time spent in each part and the X axis the number of core. This is Long trajectory size

Useful Definitions

- N_I: Number of Input files
- S_I: Size of each file
- k: Number of tasks
- N_O: Number of Output files
- S_O: Size of each file
- 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.

Essentially, the model will look like:

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

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

$$Y = (T_N T_S)r + {T_N}^2 dH + {T_N}^2 w$$

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:

```
procedure HausdorffDistance(T1, T2)            ▷ T1 and T2 are a set of 3D points
    for ∀t1 in T1 do
        for ∀t2 in T2 do
            Append in D1 calculated d(t1, t2)
        end for
        Dt1 append max(D1)
    end for
    N1 = min(Dt1)
    for ∀t2 in T2 do
        for ∀t1 in T1 do
            Append in D2 calculated d(t2, t1)
        end for
        Dt2 append max(D2)
    end for
    N2 = min(Dt2)
    return max(N1,N2)
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

C is the coefficient vector. y is the total execution times and A will be the matrix that holds the several values. The calculations were done for CA short 21,36,78 and 136 tasks and the result is used to verify the rest.

$$y = \begin{bmatrix} 20.83152014 \\ 12.18815162 \\ 6.06983352 \\ 3.94762929 \end{bmatrix}$$

$$A = \begin{bmatrix} 544 & 21 & 20.7635511 & 21 \\ 408 & 36 & 12.1358488 & 36 \\ 272 & 78 & 6.03011796 & 78 \\ 204 & 136 & 3.91804321 & 136 \end{bmatrix}$$

$$C = \begin{bmatrix} 1.34411980e-05 \\ 9.88082352e+11 \\ 9.99759905e-01 \\ -9.88082352e+11 \end{bmatrix}$$

The next table shows the calculated values for the time to execution, the actual measured mean values and the ablsolute error from the calculation.

Trajectory Size	Number of Tasks	Predicted Time	Actual Time	Error
Short	300	2.19505713407	2.38752645466	0.192469320591
Short	528	1.26865175492	1.61307469131	0.344422936393
Medium	21	33.3136668993	33.3253119711	0.0116450718086
Medium	36	22.2328782699	22.2469520392	0.0140737693618
Medium	78	10.8452443487	10.8571666795	0.0119223308157
Medium	136	6.8738059856	6.88058625833	0.00678027273885
Medium	300	3.63918139191	3.67618065543	0.0369992635193
Medium	528	2.40475870352	3.23822765489	0.833468951372
Long	21	43.8873638582	43.8923199669	0.00495610872072
Long	36	30.9028212926	30.9349148759	0.0320935833527
Long	78	17.1266483406	17.1459104994	0.0192621588231
Long	136	11.1204478449	11.1351985826	0.0147507377432
Long	300	4.34514286961	5.26987538991	0.924732520303
Long	528	4.34051945651	5.26987538991	0.929355933395

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. So, the model will be

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

where W is the number of worker nodes.

***giannis: TODO: Solve the system