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.

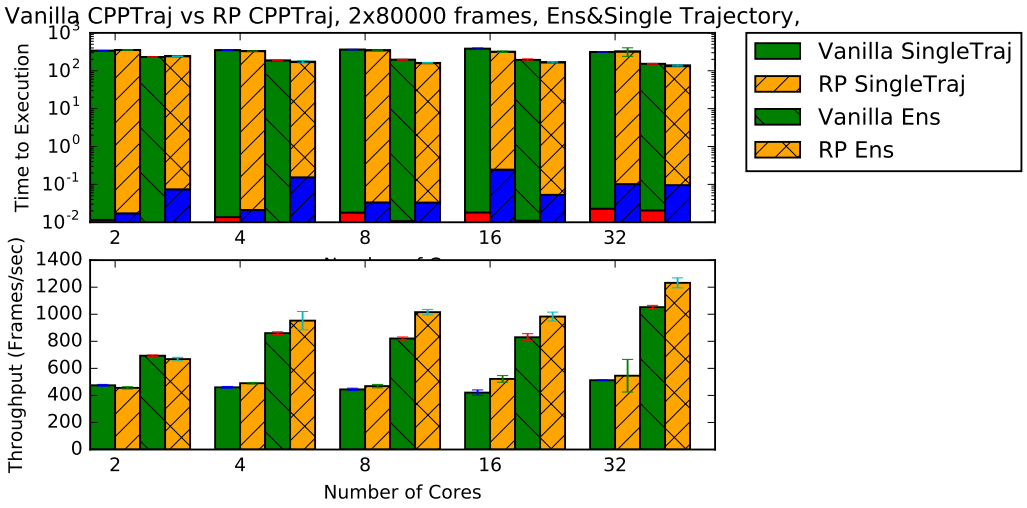

Figure 1: Time to Execution and Throughput comparison between different ways of executing the same CPPTraj analysis. There are in total 160K frames organized as a single trajectory file for the Single trajectory case and as an ensemble of 2 trajectories for the ensemble case

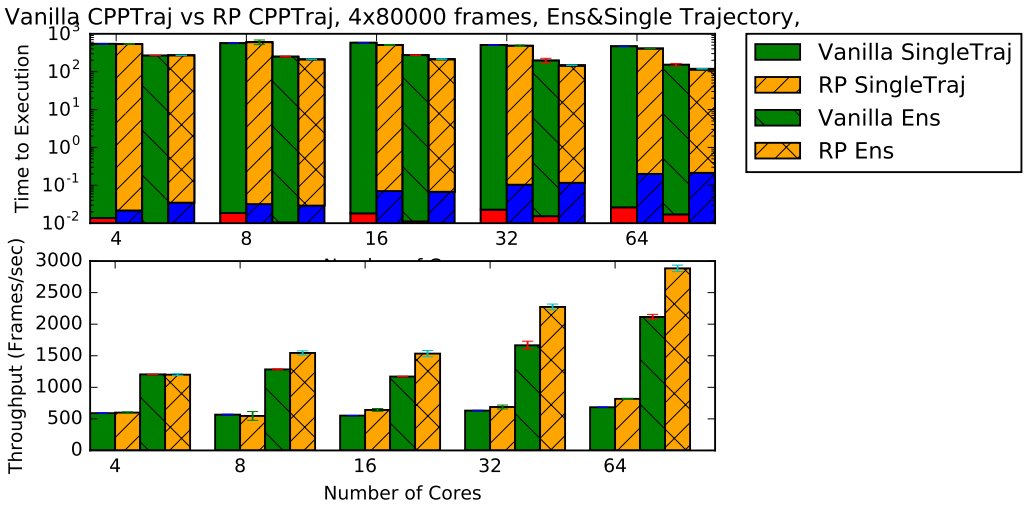

Figure 2: Time to Execution and Throughput comparison between different ways of executing the same CPPTraj analysis. There are in total 320K frames organized as a single trajectory file for the Single trajectory case and as an ensemble of 4 trajectories for the ensemble case.

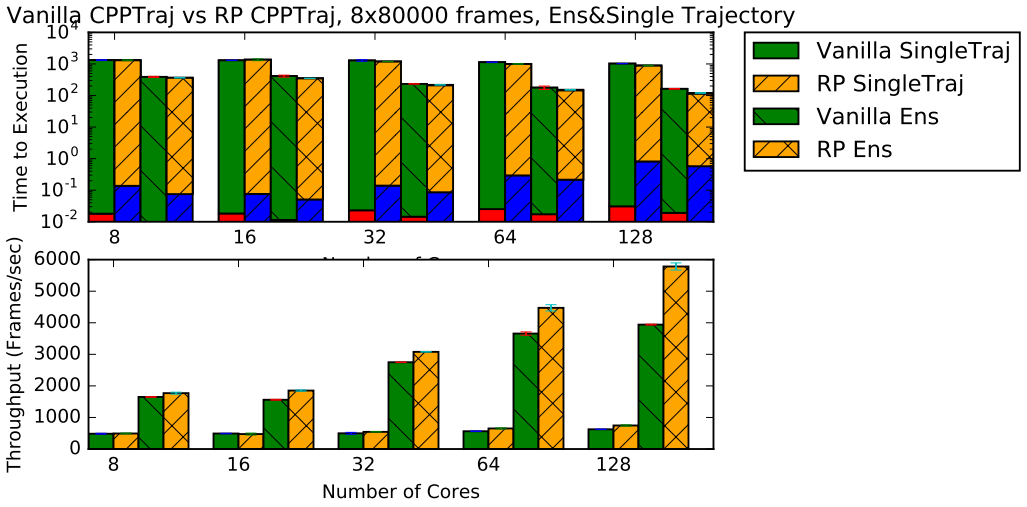

Figure 3: Time to Execution and Throughput comparison between different ways of executing the same CPPTraj analysis. There are in total 640K frames organized as a single trajectory file for the Single trajectory case and as an ensemble of 8 trajectories for the ensemble case.

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.

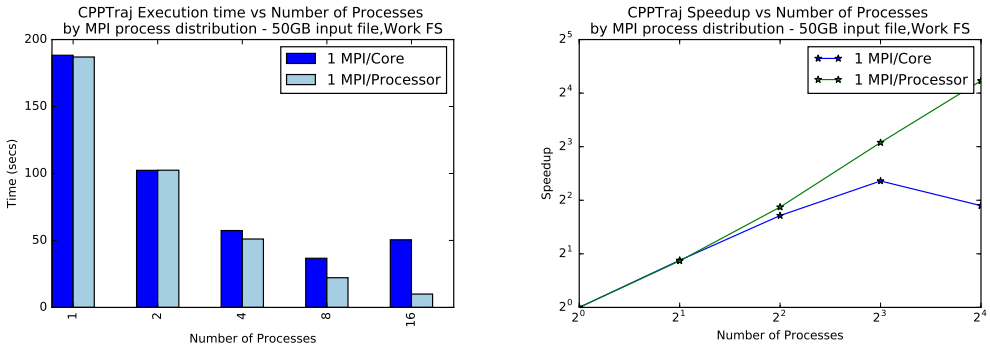

Figure 4: Execution time (L) and Speedup(R) of CPPTraj for a 50GB trajectory file. The figure compares the time to execution between 1 process per core - Stampede has 2 processors with 8 cores each - vs 1 process per physical processor. The Work Filesystem was used.

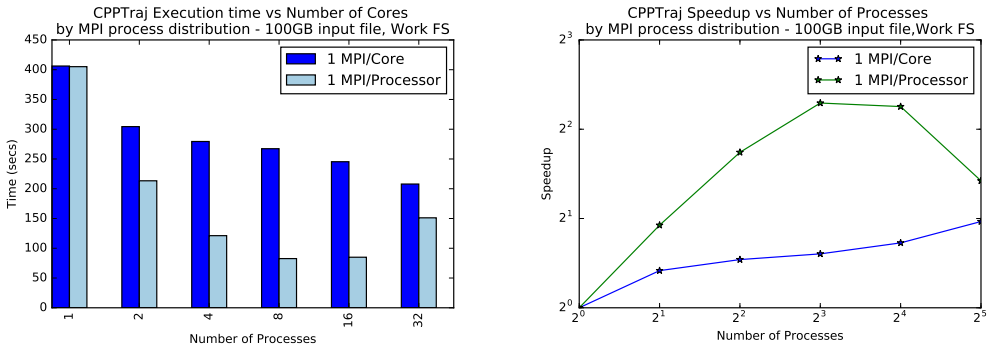

Figure 5: Execution time (L) and Speedup(R) of CPPTraj for a 100GB trajectory file. The figure compares the time to execution between 1 process per core - Stampede has 2 processors with 8 cores each - vs 1 process per physical processor. The Work Filesystem was used.

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.

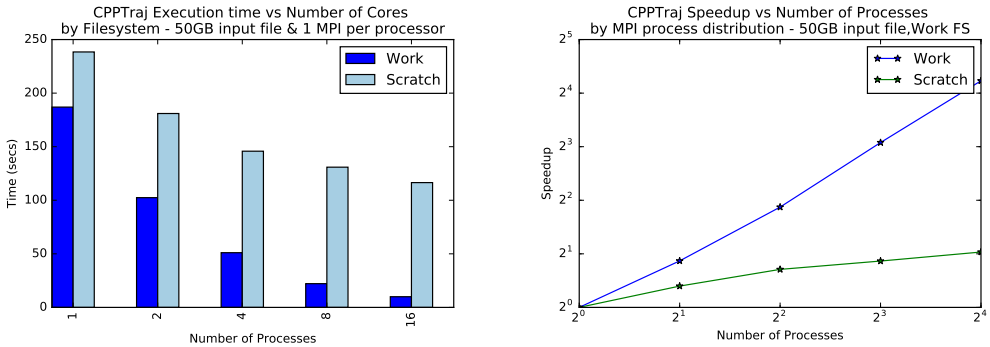

Figure 6: Execution time (L) and Speedup(R) of CPPTraj for a 50GB trajectory file. The figure compares the time to execution between Work and Scratch filesystems. 1 process per physical processor was used.

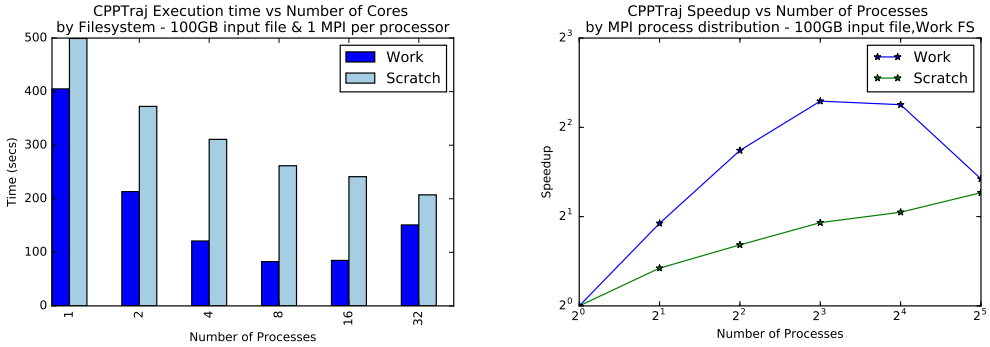

Figure 7: Execution time (L) and Speedup(R) of CPPTraj for a 100GB trajectory file. The figure compares the time to execution between Work and Scratch filesystems. 1 process per physical processor was used.

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.

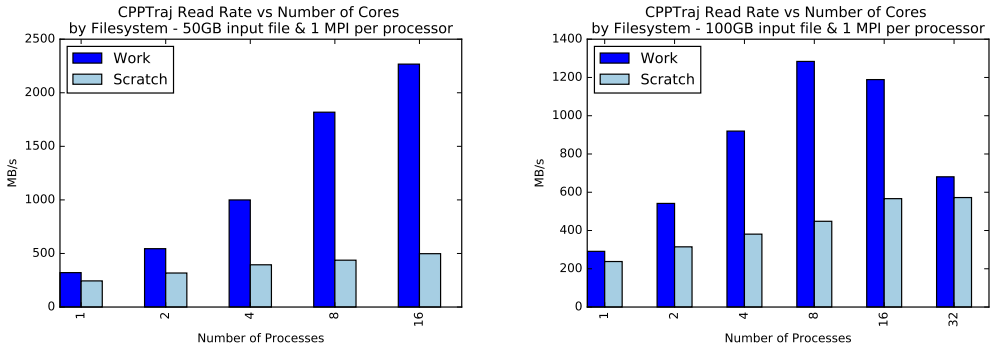

Figure 8: Read Rate in MB per seconds for the 50GB (L) and 100GB (R) trajectory files vs number of processes. The figures compare the read rates between Work and Scratch filesystems. 1 process per physical processor was used.

[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.

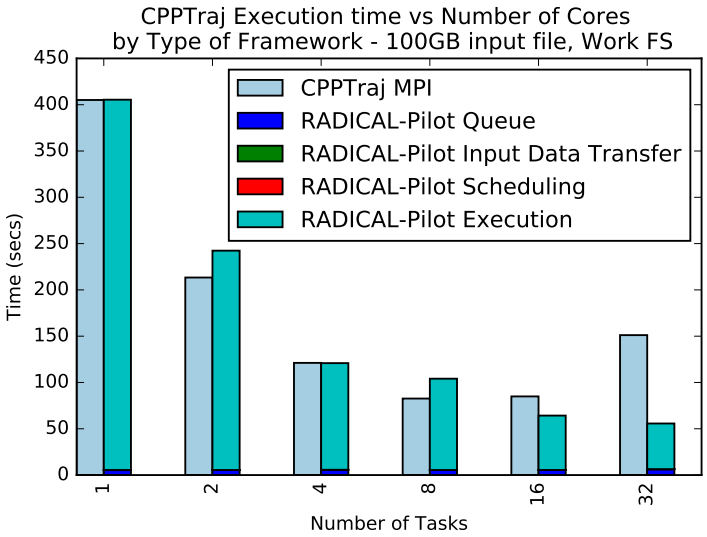

Figure 9: Execution time of CPPTraj MPI and Task level parallel CPPTraj execution through RADICAL-Pilot for a 100GB trajectory file. In the MPI case 1 processes per physical processor was used. For the RADICAL-Pilot execution 1 unit per 8 cores were submitted to achieve the desired 2 units per node.

[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.

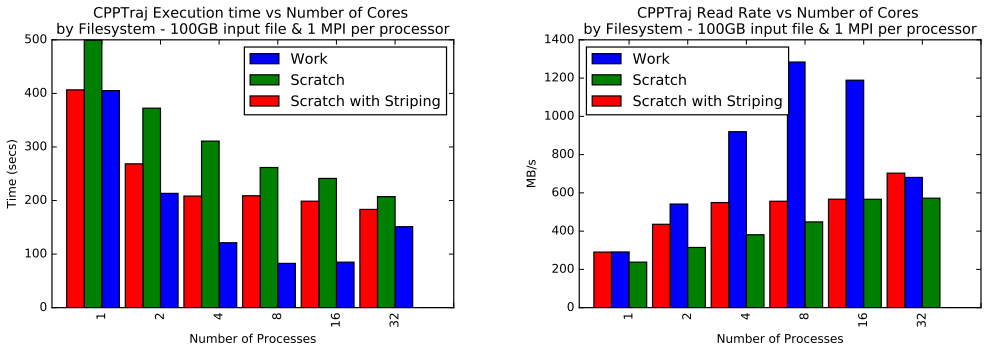

Figure 10: Execution time (L) and Speedup(R) of CPPTraj for a 100GB trajectory file. The figures compare the time to execution and read rate between Work, Scratch with default striping and Scratch with the number of stripes equal to 2. 1 MPI process per physical processor was used. The input data is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

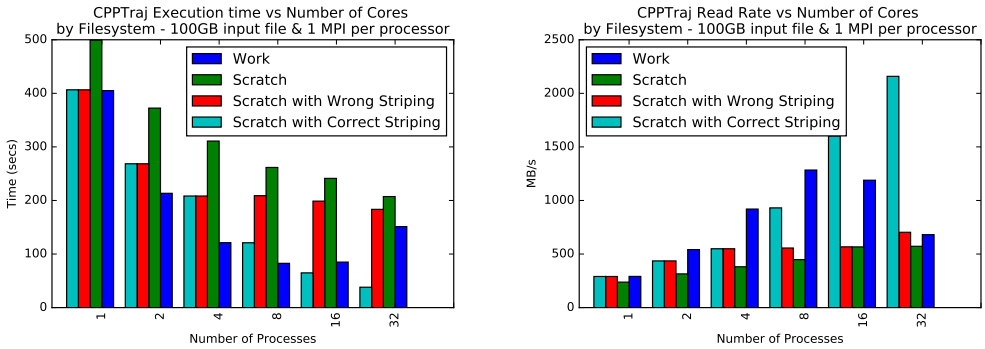

Figure 11: Execution time (L) and Speedup(R) of CPPTraj for a 100GB trajectory file. The figures compare the time to execution and read rate between Work, Scratch with default striping and Scratch with the number of stripes equal to the number of nodes accessing the file. 1 MPI process per physical processor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