Data Caching vs. GPFS Results

09-15-07

Ioan Raicu

For each given number of nodes (1, 2, 4, 8, 16, 32, 64), I have 12 experiments for 8 different file sizes, summing to a total of 7*12*8 = 672 experiments. I have finished (last night) running 8 of the 12 experiments (448 runs in total). The experiments I have finished are, which were done for a varying number of nodes (1, 2, 4, 8, 16, 32, 64) and varying number of file sizes (1B, 1KB, 10KB, 100KB, 1MB, 10MB, 100MB, 1GB)... the data lived on GPFS initially for all experiments:

1. Read Caching 0% Locality: Data Caching read with no locality and round-robin scheduler
2. Read Caching 100% Locality: Data Caching read with 100% locality (ran the same workload 4 times in a row) and round-robin scheduler
3. Read+Write Caching 0% Locality: Data Caching read+write with 0% locality and round-robin scheduler
4. Read+Write Caching 100% Locality: Data Caching read+write with 100% locality and round-robin scheduler
5. Read GPFS: read from GPFS
6. Read+Write GPFS: read+write from GPFS
7. Read Wrapper GPFS: read from GPFS through the wrapper script; the wrapper script creates a temp scratch directory on GPFS, makes a symbolic link to the input file, performs the operation, and finally removes the temp scratch directory from GPFS... the Swift wrapper actually does even more things that could slow things down, such as writing logs 10~20 times to a file on GPFS, which I did not do, as these loging is no essential for the wrapper script to function; however, the temp scratch directory and symbolic linking is, and therefore it is what I tested
8. Read+Write Wrapper GPFS: read+write from GPFS through the wrapper script

The following set of figures (1 – 11) are showing the performance of one of the above experiments, and have throughput (either tasks/sec or Mb/s) on the x-axis and the file size on the y-axis.

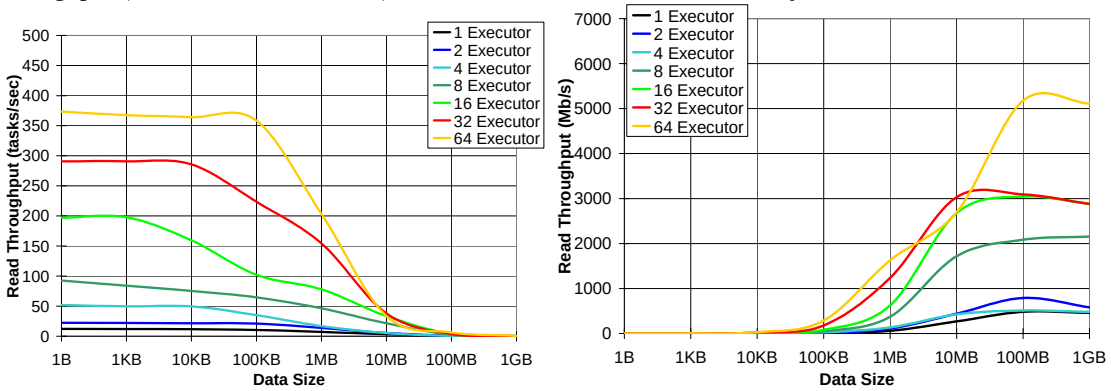

Figure 1: Read Caching 0% Locality

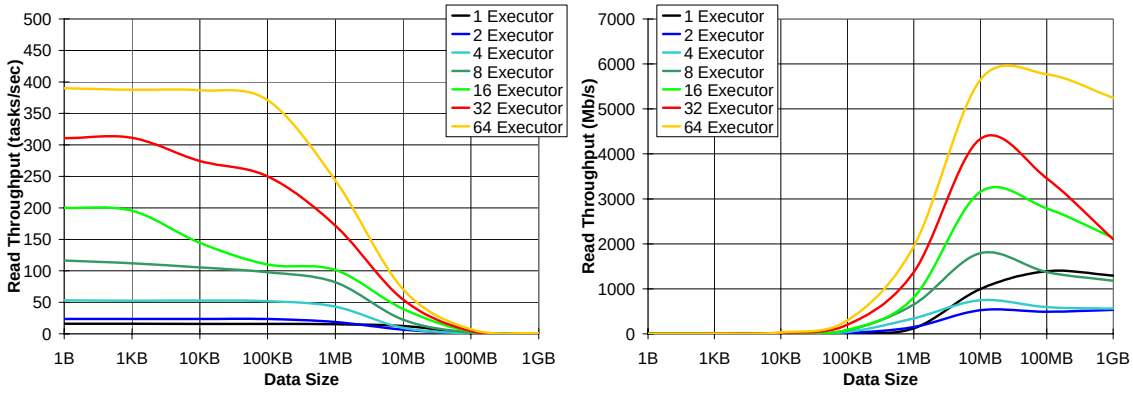

Figure 2: Read Caching 100% Locality

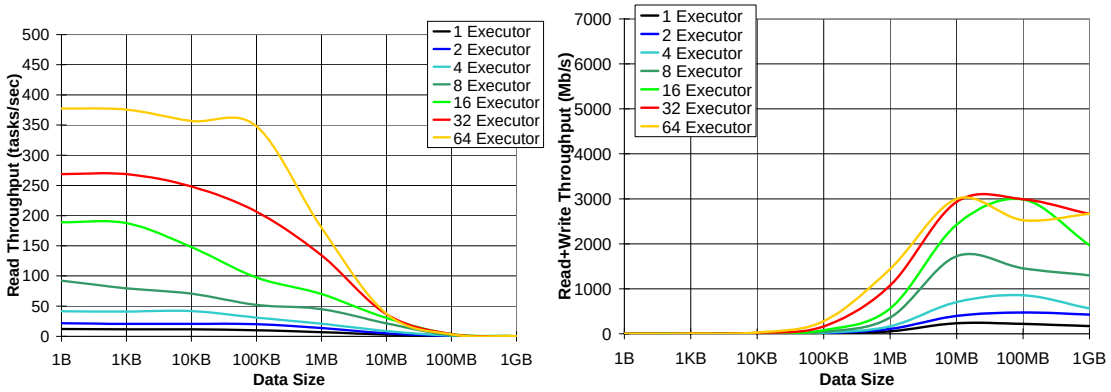

Figure 3: Read+Write Caching 0% Locality

Figure 4: Read+Write Caching 100% Locality

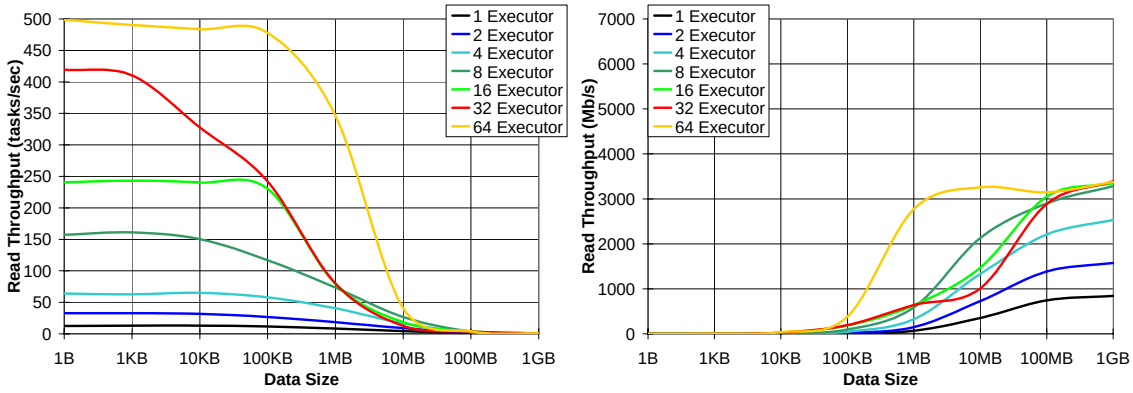

Figure 5: Read GPFS

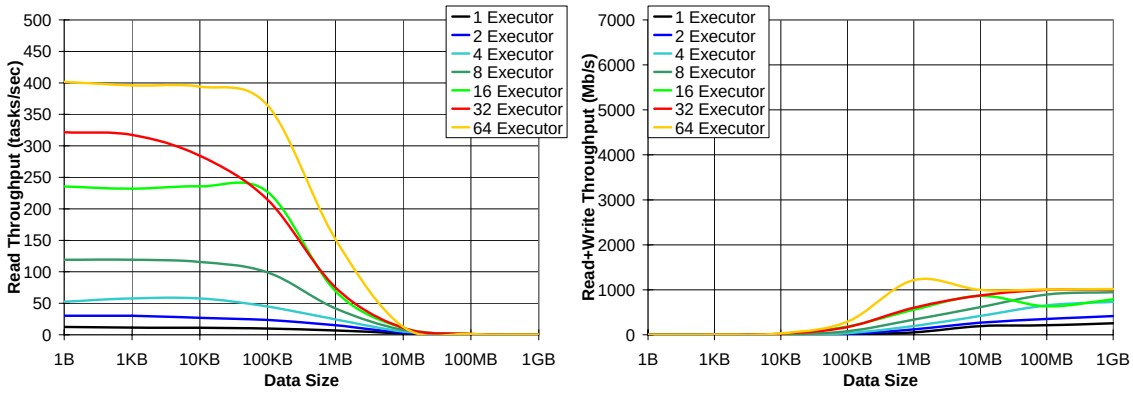

Figure 6: Read+Write GPFS

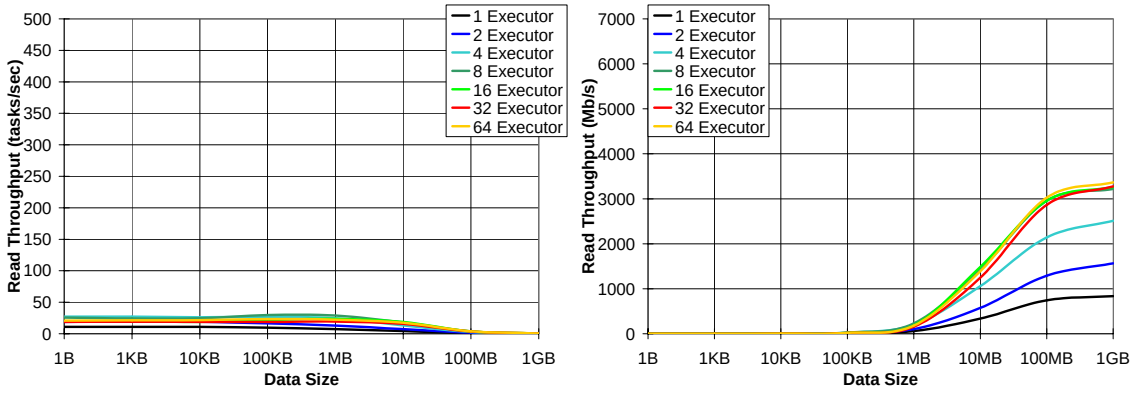

Figure 7: Read Wrapper GPFS

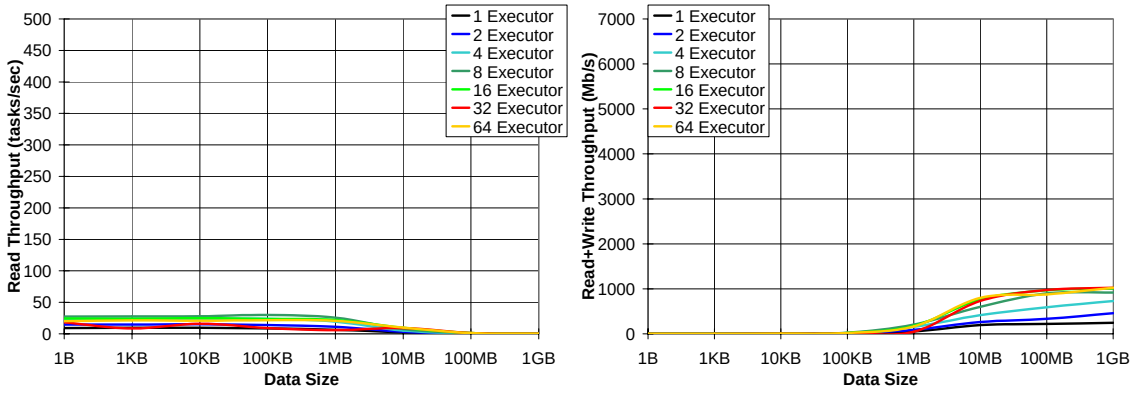

Figure 8: Read+Write Wrapper GPFS

The experiments which I still have left to do are (these tests might take a few days, as I haven't made any large scale runs yet with this part of the code, hopefully I don't have to mess with the code... which is why I left this to the end):

1. Read Caching Data-Aware Scheduler 0% Locality: Data Caching read with 0% locality and data-aware scheduler
2. Read Caching Data-Aware Scheduler 100% Locality: Data Caching read with 100% locality and data-aware scheduler
3. Read+Write Caching Data-Aware Scheduler 0% Locality: Data Caching read+write with 0% locality and data-aware scheduler
4. Read+Write Caching Data-Aware Scheduler 100% Locality: Data Caching read+write with 100% locality and data-aware scheduler

These tests will populate Figures 8, 9, 10 and 11, and add 4 more lines to figures 12 through 18.

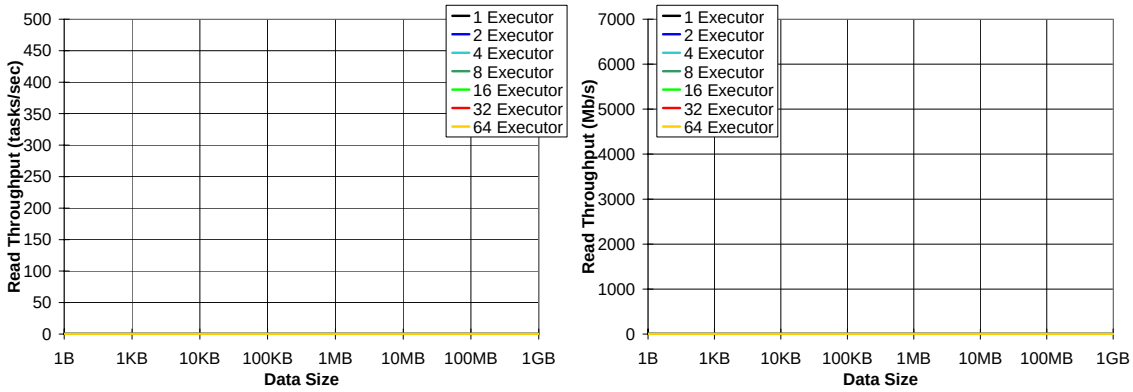

Figure 9: Read Caching Data-Aware Scheduler 0% Locality

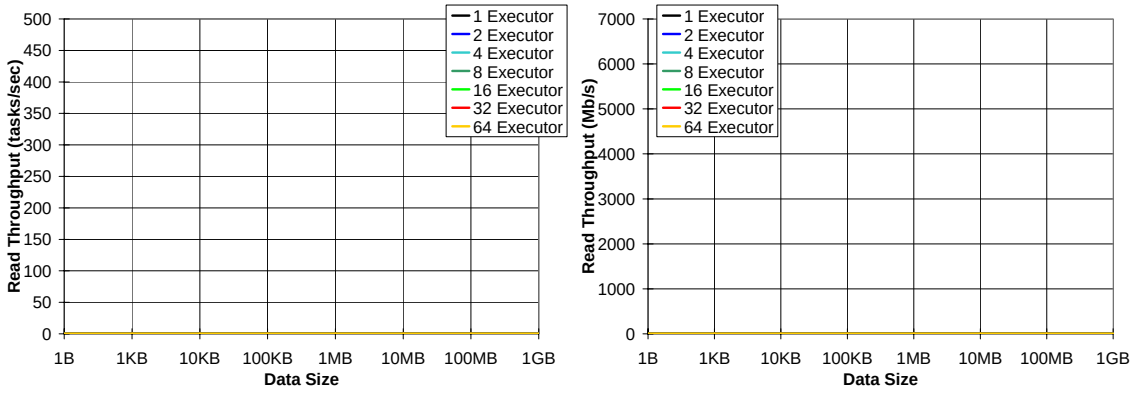

Figure 10: Read Caching Data-Aware Scheduler 100% Locality

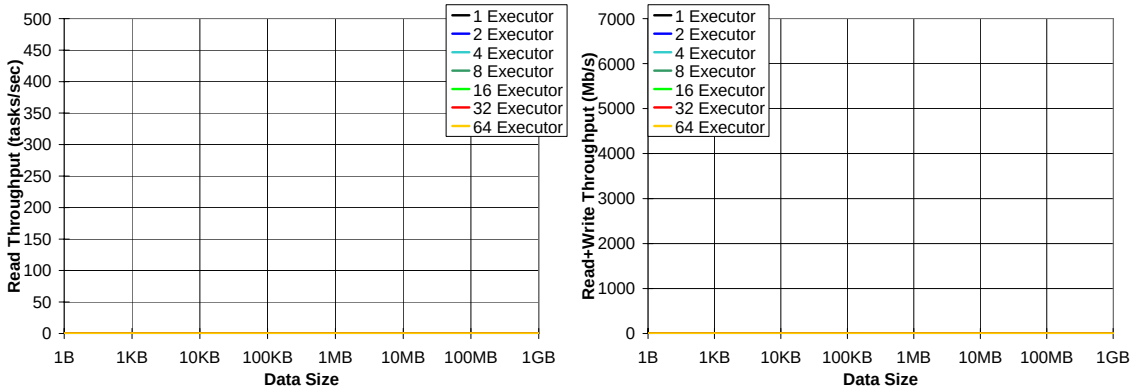

Figure 11: Read+Write Caching Data-Aware Scheduler 0% Locality

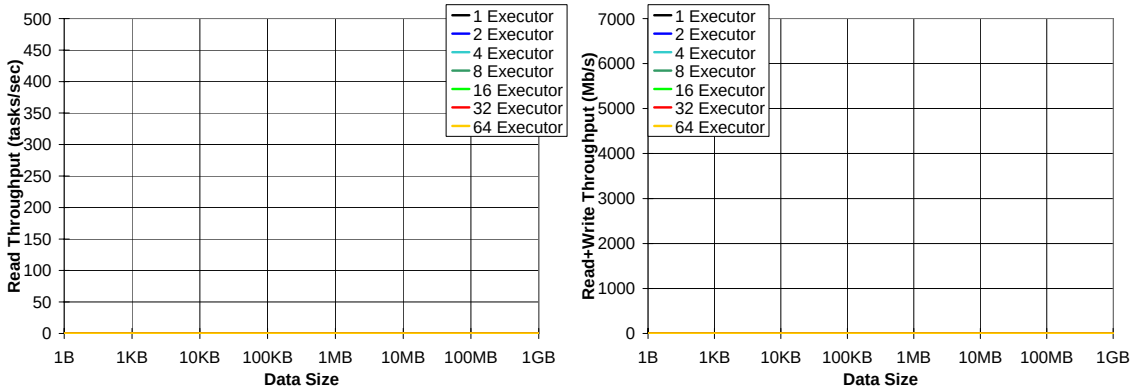

Figure 12: Read+Write Caching Data-Aware Scheduler 100% Locality

The rest of the graphs (Figures 13 through 19) show the same data from Figures 1 through 12, but organized by the number of executors, rather than experiment (i.e. caching, GPFS, etc). The axis remain the same as the previous experiments, only the lines change their meaning from the number of executors to the experiment name. These graphs allow the ease of inspection to see which method is best for particular file sizes and particular deployment size.

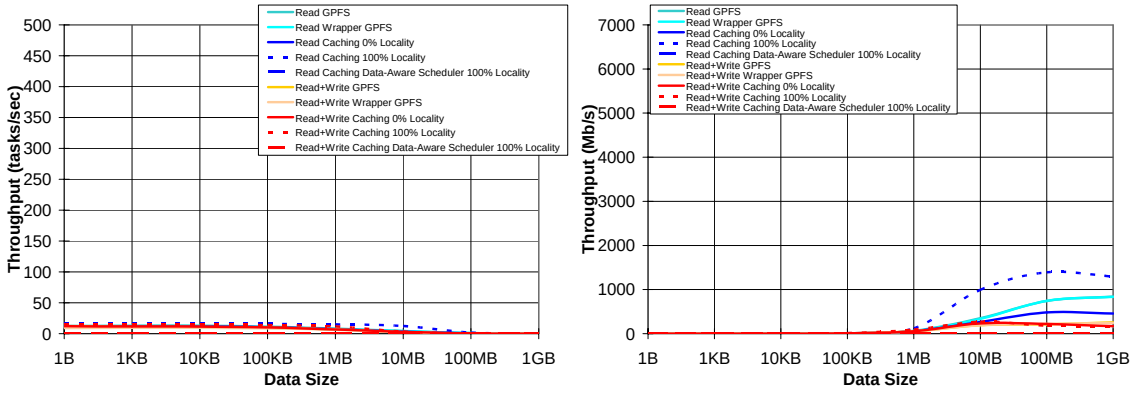

Figure 13: Comparing performance for 1 executor

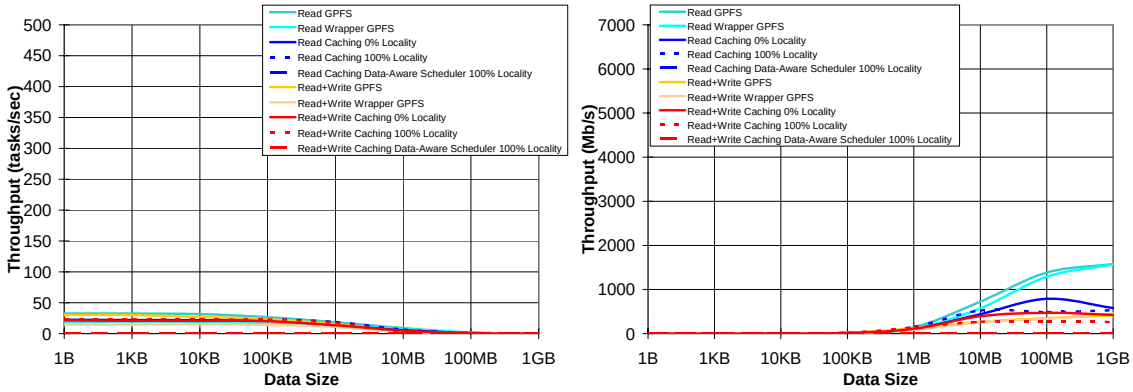

Figure 14: Comparing performance for 2 executors

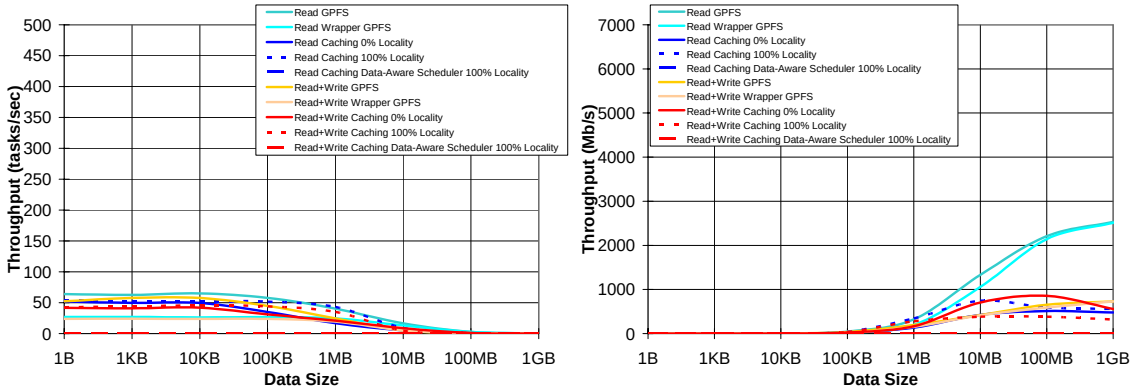

Figure 15: Comparing performance for 4 executors

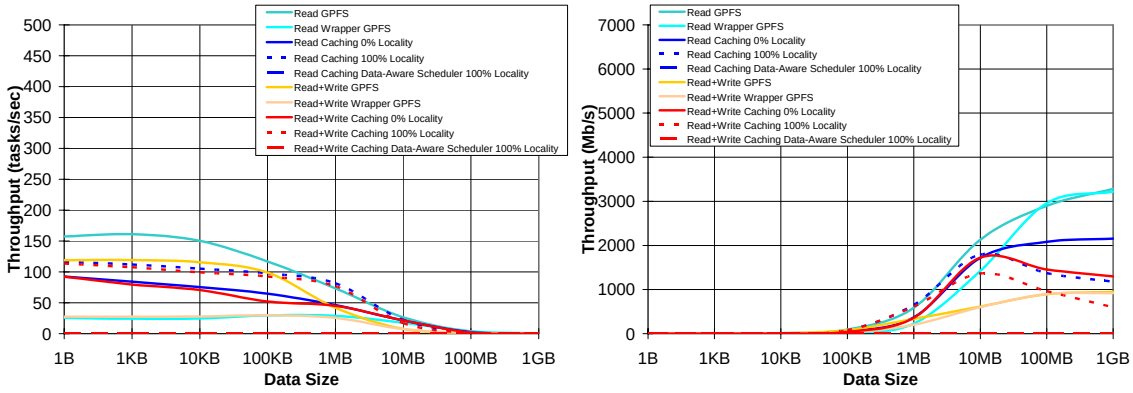

Figure 16: Comparing performance for 8 executors

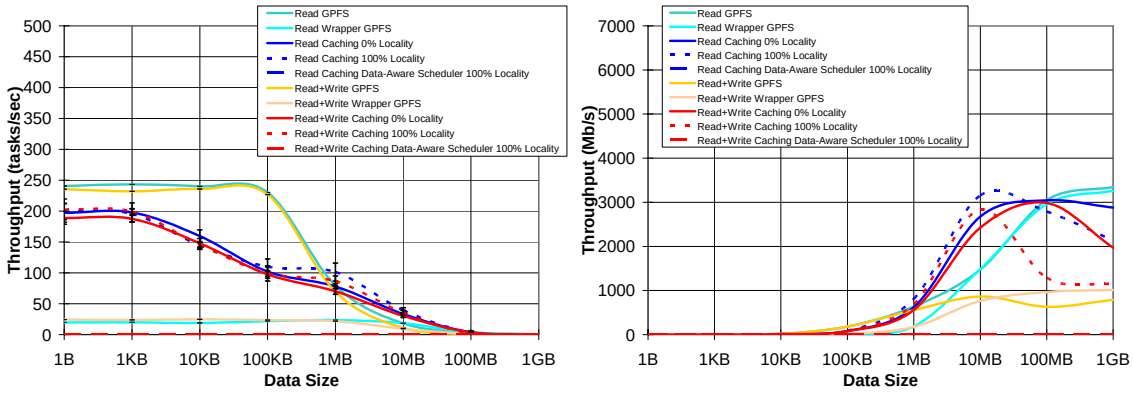

Figure 17: Comparing performance for 16 executors

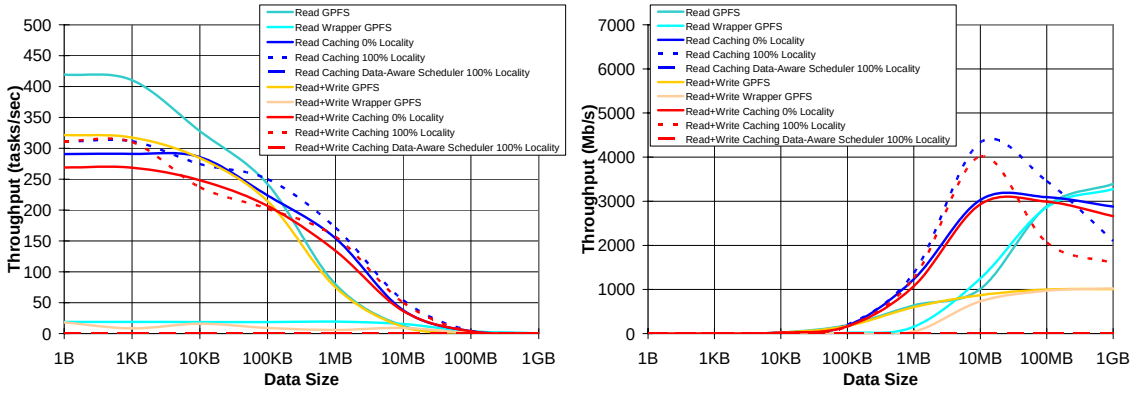

Figure 18: Comparing performance for 32 executors

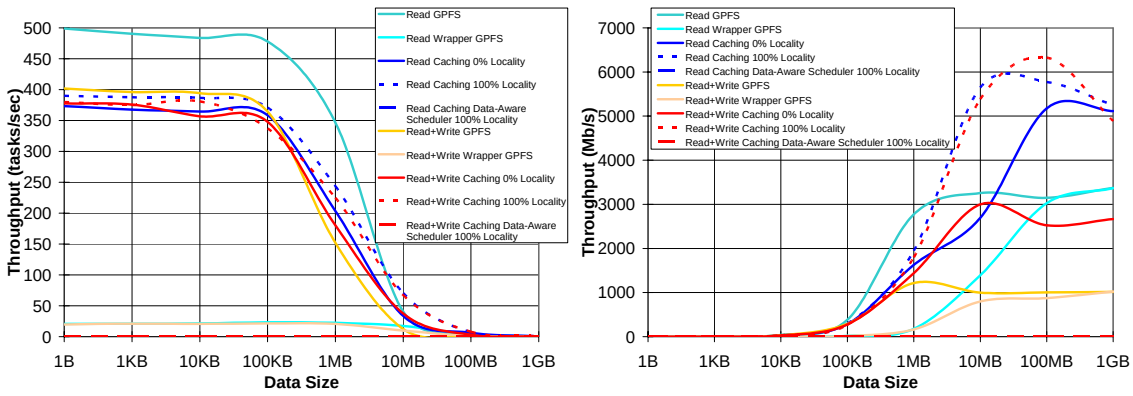

Figure 19: Comparing performance for 64 executors

Overall, I think the most interesting tests are the larger scale ones, so now that we have nice baselines from these tests, I think we could concentrate (and dig deeper into other scenarios) on just 2 different cases, one extreme (1 worker), and the other extreme (64 executors, or ever 128 if we could get the entire ANL/UC site). We could also take the 3rd case, 8 executors, as that should perform similarly to GPFS (as far as I know, GPFS has 8 I/O nodes supporting GPFS calls, so having 8 executors in our experiment makes sense to have comparable performance to GPFS). Notice that the biggest gains were with 64 executors, and I bet the gains will grow further with 128 executors. Also, none of these results reflect the data-aware scheduler, but I'll have those results ready soon. What I expect to see is more poor performance for small file sizes, but better performance for large file sizes when compared to the data caching approach with the round-robin scheduler. Finally, I have experienced poor performance of the data caching code when running the experiment with large file sizes (100MB and 1GB) and with data locality at 100%; this is evident in the graphs. I need to investigate this further…

There is still lots of work to be done here, detailed explanation of each experiment's results, perhaps pick out a few from this bunch to concentrate on, compute the standard deviations and integrate them into the graphs, etc… Then, I need to start putting the paper together, and at the same time, start testing some real apps to see what kind of performance improvement we get.