1. HYBRID PARALEL MPI-PTHREAD RENDERING PERFORMANCE

Rendering images in-situ requires access to a working OpenGL stack. Many HPC sites do not have GPUs available and of those that do many do not support running X11 on compute nodes which is required by many OpenGL drivers in order to create a rendering context. At HPC sites where GPUs are unavailable for rendering the Mesa OpenGL stack is often used. Mesa's OS Mesa driver implements OpenGL in software and provides an OpenGL context without X11. In the Mesa 9.2.0 release[1] the OS Mesa driver will be replaced by Mesa's Gallium llvmpipe software rasterizer. The Gallium llvmpipe driver is threaded and uses LLVM for just-in-time(JIT) OpenGL GLSL shader compilation. In this section we investigate the threading performance of the new OS Mesa llvmpipe driver when rendering Surface LIC in parallel in order to identify the proper ratio of llvmlpipe rendering threads to MPI ranks per node.

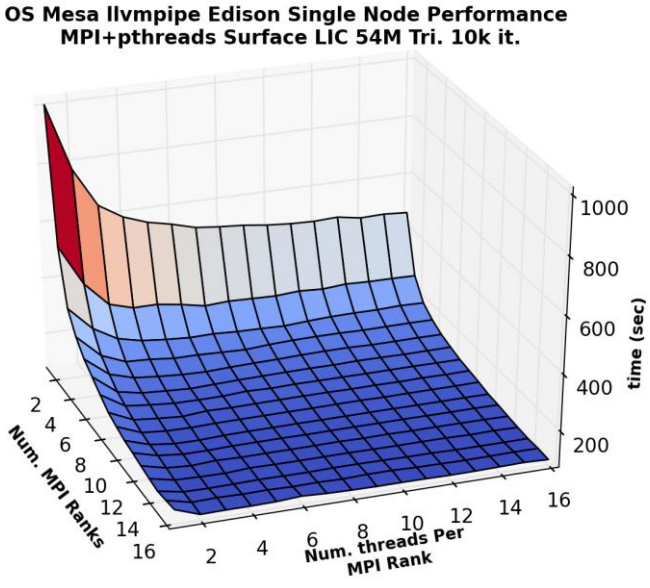

Figure 6: Single node total rendering time of a 54M triangle surface as a function of two independent variables, MPI ranks and llvmpipe rendering threads. The results of all 256 run combinations are shown.

The rendering benchmarks were run on Phase 1 of the Edison supercomputer, NERSC's Cray XC30 Cascade system. Phase I Edison houses 664 compute nodes with each compute node equipped with two 8 core Xeon E5-2670 CPUs and 64GB of DDR3 ram. Because we are interested in determining reasonable choice of number of threads per MPI rank for in-situ rendering we focus on measuring single node rendering performance of our GPGPU Surface LIC implementation on a 54 million triangle simulation dataset using 10 thousand integration steps as we co-vary the number of MPI ranks and rendering threads per rank each up to the number of physical cores on the compute node. We have disabled Cray's default CPU affinity binding and have enabled hyper-threading as some of the runs we made use more rendering threads than there are physical cores on the node.

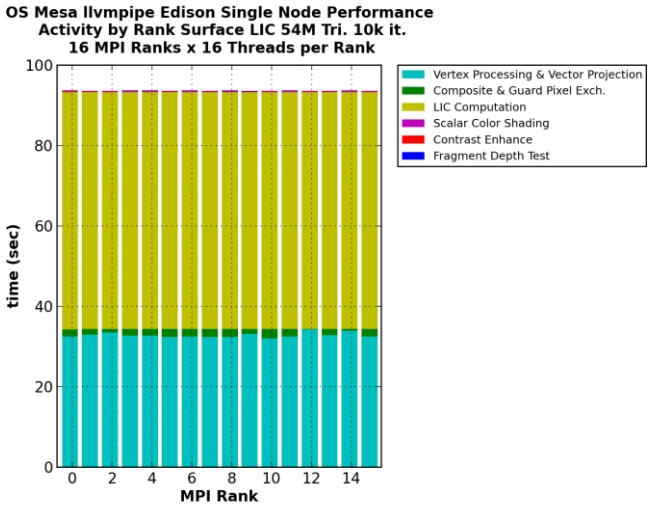

Figure 7: Gant chart showing relative time each process spent in various stages of the Surface LIC algorithm for the run where 16 MPI ranks each with 16 rendering threads were used.

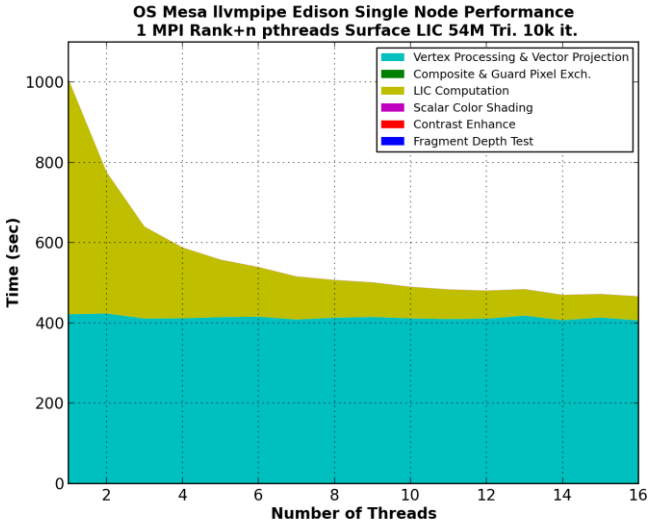

Figure 8: Relative times of each stage in the Surface LIC algorithm for a single MPI rank while the number of llvmpipe rendering threads is increased.

Figure 6 shows the results of the 256 benchmark runs with the total single node rendering time plotted on a surface as a function of number of MPI ranks on the x-axis and number of threads per rank on the y-axis. The absolute fastest render time was attained by the run with 16 MPI ranks and 12 rendering threads per rank.

The Gant chart in figure 7 shows, for the run where we used 16 MPI ranks each with 16 threads, how each process spent its time relative to the other processes. The algorithm's run time is dominated by the vertex processing and LIC computation stages. Time spent in inter-process communication during compositing and guard pixel exchange is a relatively small part of the whole

[1] As of this writing Mesa 9.2.0 is not released. We used a checkout of Git sha 062317d6 obtained from Mesa's Git repo hosted at freedesktop.org.

run time while the remaining stages contribute a negligible amount of time to the overall result.

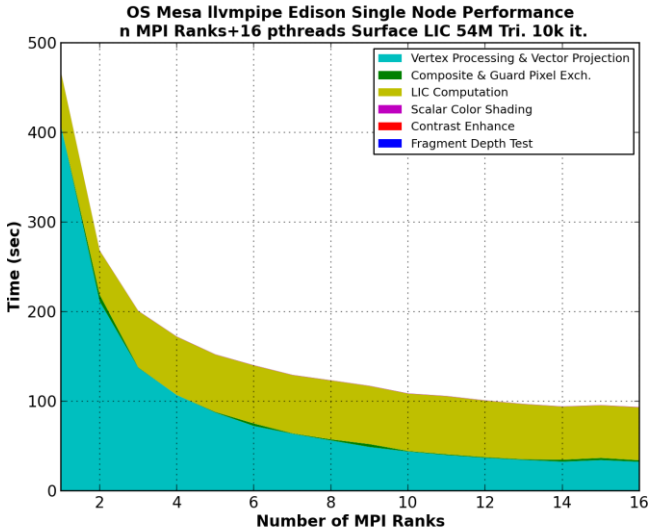

Figure 9: Relative times of each stage in the Surface LIC algorithm for rank 0 of the set of runs where the number of MPI ranks is varied while the number of llvmpipe rendering threads is fixed at 16 threads per rank.

Figure 8 shows, for the set of runs where 1 MPI rank was employed and the number of rendering threads was varied, how the single process spent its time while rendering the surface LIC. This figure shows the speedup attributed to the llvmpipe driver's threading. Note how as the number of rendering threads is increased the time spent computing the LIC decreases while the time spent in the vertex processing stage remains fairly constant. The important take away from this is that the llvmpipe driver's fragment pipeline is threaded while its vertex pipeline is not. This is an important factor in predicting the benefit of the driver's threading for a given rendering algorithm. For algorithms where rendering time is dominated by vertex processing the threaded driver will likely result in little or no speedup.

One interesting characteristic visible in figure 6 is that as we approach the fully oversubscribed case of 16 MPI ranks each with 16 threads for a total of 256 rendering threads the overall rendering performance is not negatively impacted by the large ratio of rendering threads to cores. This case's rendering time was within 0.5% of the fastest rendering time.

Figure 9 shows, for the set of runs where the number of MPI ranks is varied with 16 rendering threads, how the MPI rank 0 process spent its time during rendering. The time spent in the LIC computation increases as the total number of rendering threads on the node is increased beyond 32 threads. We are simply running into the limits of the system's processing power. The large ratio of threads to physical cores on the node employed during this set of runs is negatively impacting the performance of the threaded fragment pipeline. However, overall rendering performance is not negatively impacted since the speedup attained by the MPI data parallelism, which reduces the number of vertices each rank processes, more than makes up for the slowdown due to the large number of rendering threads on the node.

Our results show that because the vertex pipeline in the llvmpipe driver is not threaded for large data processing the most important factor in overall rendering performance is MPI data parallelism which reduces the number of vertices processed per process. As a rule of thumb we found that limiting the total number of rendering threads per node to the number of available cores including hyper-threads if they are available produces a reasonable result.

2. SUMMARY

We investigated the hybrid-parallel (MPI-pthread) rendering performance of the new OS Mesa llvmpipe driver which provides optimized JIT compilation of GLSL shaders and is threaded. We found that MPI data parallelism is the key to fast rendering because the driver's vertex pipline is not threaded and that a reasonable choice for the number of rendering threads per node is the number of cores including hyper-threads if available.