AccFFT: A New Parallel FFT Library for CPU and GPU Architectures

Amir Gholami
The University of Texas at Austin,
Austin, TX 78712
gholami@accfft.org

George Biros (Advisor)
The The University of Texas at Austin,
Austin, TX 78712
gbiros@acm.org

We present a new distributed-FFT library. Despite the extensive work on FFTs, we achieve significant speedups. Our library uses novel all-to-all communication algorithms to overcome this barrier. These schemes are modified for GPUs to effectively hide PCI-e overhead. Even though we do not use GPUDirect technology, the GPU results are either better or almost the same as the CPU times (corresponding to 16 or 20 CPU cores). We present performance results on the Maverick and Stampede platforms at the Texas Advanced Computing Center (TACC) and on the Titan system at the Oak Ridge National Laboratory (ORNL). Comparison with P3DFFT and PFFT libraries show a consistent $2-3\times$ speedup across a range of processor counts and problem sizes. Comparison with FFTE library (GPU only) shows a similar trend with $2\times$ speedup. The library is tested up to 131K cores and 4,096 GPUs of Titan, and up to 16K cores of Stampede.

1. INTRODUCTION

The most expensive part of distributed-FFT is the transpose phase, which adversely affects the scaling at large core counts. This phase involves all-to-all exchanges, which is essentially a transpose operation between a subgroup of tasks. This exchange is wrapped around a packing/unpacking phase to make the data contiguous in the memory. Generally this accounts for less than 10% of the transpose time. This phase can be performed either by reshuffling of the data as done in P3DFFT, a local transpose as implemented in PFFT library, or eliminated by using MPI Data types. Although optimizing the packing/unpacking is important, but the communication time is the dominating factor. The situation is even worse for the GPU, since the data has to be transferred forth and back to the CPU through PCI-e, which is very expensive.

2. CONTRIBUTIONS

We have implemented novel schemes to overcome the communication bottleneck. Optimizing the communication phase, requires different algorithms for different processor ranges. For small ranges, a point-to-point (P2P) all-to-all is preferred as it has an optimal data transfer of $\mathcal{O}(N)$, and the operation finishes in one stage. However, as the number of processors increase, the overhead associated with sending $p \gg 1$ send/recv directives, will considerably deteriorate the performance. For this reason, a hypercube all-to-all is typically used for large processor counts. In this scheme, each process sends/receives data from another process and splits the problem into two. This requires $\log p$ stages to complete the communication phase. In this method, one transfers $\mathcal{O}(N \log p)$ data, which is higher than P2P. But its smaller overhead compared to sending/receiving from p processes at once, make it faster for medium range counts. However, the larger data transfer will adversely affect its performance for large processor counts.

This is not a limitation in our k-way all-to-all scheme. In this method, each task sends/receives data from k other task and splits the problem into k smaller ones. The communication finishes in $\log p/\log k$ stages, with a total data transfer of $\mathcal{O}(N \log p/\log k)$. The algorithm reduces to a P2P all-to-all for $k = p$, and to a hypercube one for $k = 2$. We use an auto-tuning scheme to select the best choice for the splitting parameter as part of the FFT setup. This allows us to automatically choose different values for k for different core counts. An important advantage of the algorithm is that it works for non powers of two as well.

An important contribution of our library, is the extension of these algorithms to GPUs. The k-way all-to-all scheme is modified to interleave send/recv operations with PCI-e data transfers between GPU and CPU. FFT computations on GPU are upto $10\times$ faster than CPU, and the PCI-e overhead dominates the overall time. Our asynchronous communication scheme effectively masks this bottleneck.

In the pencil decomposition algorithm, one has to perform two all-to-all operations to compute the global FFT. These can be thought as transposing a batched, 2D matrix. However, the costs of the transposes is not the same. The first transpose involves communicating strided elements in the memory which is very expensive, while the last transpose involves contiguous superelements. We use several packing and unpacking algorithms to reduce the overhead of the first

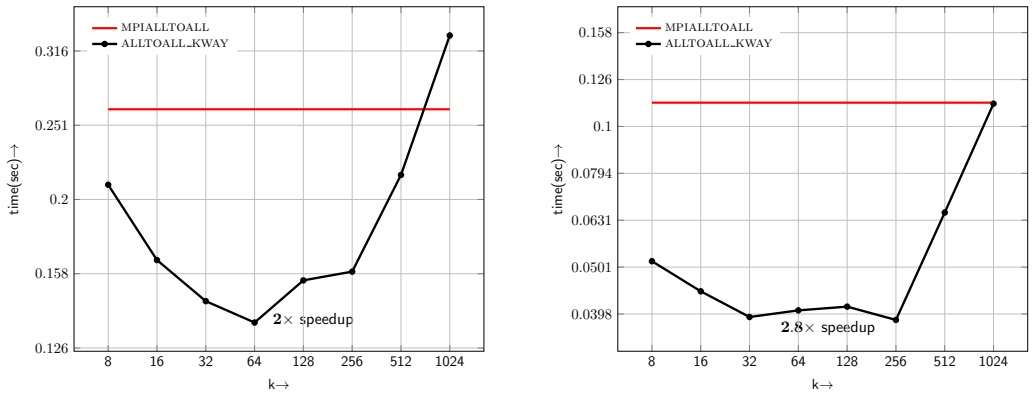

Figure 1: The timings are given for different splitting parameter, k in our k-way all-to-all. The optimal transpose time corresponds to $k = 64$. (Left) Transpose time for $N = 2048^3$ on 16,384 cores of Stampede. (Right) Transpose time for $N = 2048^3$ on cores of 512 GPUs on Titan.

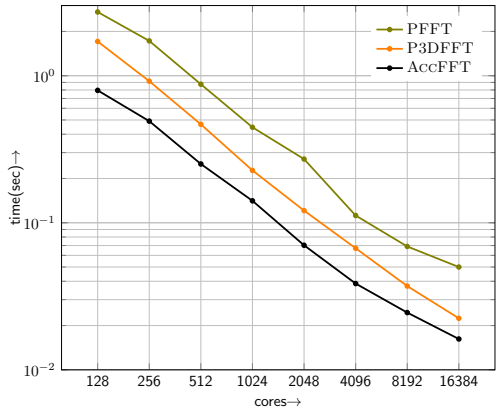

Figure 2: Strong scaling on Stampede for an R2C transform with $N = 1024^3$.

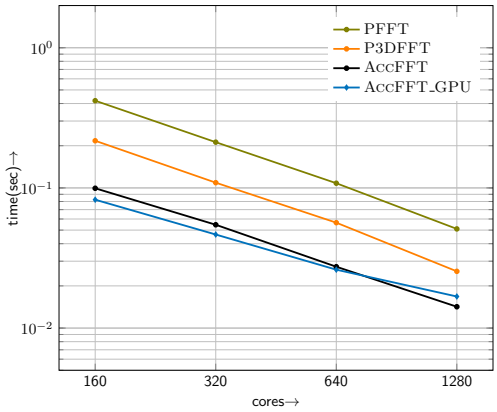

Figure 3: Strong scaling on Maverick for an R2C transform with $N = 256 \times 512 \times 1024$.

phase. Unlike other implementations, we do not change the memory access pattern in the frequency domain, so we do have the overhead of a final local unpacking [1].

3. REPRESENTATIVE RESULTS

We have tested the performance of the library on the Maverick and Stampede platforms at the Texas Advanced Computing Center (TACC) and on the Titan system at the Oak Ridge National Laboratory (ORNL):

- The **Stampede** system at TACC is a Linux cluster consisting of 6400 compute nodes, each with dual, eight-core processors for a total of 102,400 available CPU-cores.
- The **Maverick** system at TACC is a Linux cluster with 132 compute nodes, each with dual 10-core 2.8GHz Intel Xeon E5 (Ivy Bridge) processors and one K40 GPU.
- The **Titan** system is a Cray XK7 supercomputer at ORNL. Titan has a total of 18,688 nodes consisting of a single 16-core AMD Opteron 6200 series processor and a K20 GPU, for a total of 299,008 cores and 18,688 GPUs.

Stampede and Maverick use a fat-tree topology, while Titan uses a torus connection. However, our auto-tuned k-way all-to-all is topology agnostic, and the best splitting parameter is chosen during the setup time. An example of the speedup achieved by our routine is shown in Figure 1. The k-way all-to-all is two times faster than MPI_ALLTOALL (hypercube all-to-all), for transposing an array of size $N = 2048^3$. The gain is even more noticeable on Titan for transposing the same array size on 512 GPUs. A speedup of 2.8× can be achieved by using the tuned k-way all-to-all.

Comparison of AccFFT with P3DFFT and PFFT libraries shows a consistent $2-3\times$ speedup across a range of processor counts and problem sizes. The comparison of the GPU code with FFTE library shows a similar trend with a 2× speedup. Our GPU timings are almost the same as of 16/20 CPU cores on Stampede and Maverick, respectively. This is due to the effective masking of PCI-e overhead. To the best of our knowledge, the latter is the only maintained open source library for GPUs.

[1] An exception to this is P3DFFT library, which provides the same feature.