An Overview of FFT Computation towards Exascale - Accelerating the Communication Cost of parallel 3-D FFT

The 9th Annual MVAPICH User Group (MUG) Meeting

Alan Ayala*, Stan Tomov*,
Miroslav Stoyanov†, Azzam HaidarΔ,
Jack Dongarra**□

* University of Tennessee at Knoxville – Innovative Computing Laboratory
+ Oak Ridge National Laboratory
Δ NVIDIA Corporation
□ University of Manchester
CONTENT

1. Fast Fourier Transform
   - Single-device computation
   - Parallel algorithm and libraries capabilities
   - heFFTe library

2. FFT on large scale systems
   - Communication bottleneck
   - Scalability
   - MPI challenges
     - Phase diagrams
     - Effect of MPI distribution

3. Performance results and conclusions
1. FFT COMPUTATION

Algorithms and libraries
1. **Fast Fourier Transform Computation**

In essence, the FFT of $x$, an $m$-dimensional vector of size $N := N_1 \times N_2 \times \cdots \times N_m$ is defined by $y := FFT(x)$, which is obtained as follows,

$$
\tilde{y} := \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \cdots \sum_{n_m=0}^{N_m-1} \tilde{x} \cdot e^{-2\pi i \left( \frac{k_1 n_1}{N_1} + \frac{k_2 n_2}{N_2} + \cdots + \frac{k_m n_m}{N_m} \right)},
$$

where $\tilde{y} = y(k_1, k_2, \ldots, k_m)$, and $\tilde{x} := x(n_1, n_2, \ldots, n_m)$. 
1. **Fast Fourier Transform Computation**

In essence, the FFT of $x$, an $m$-dimensional vector of size $N := N_1 \times N_2 \times \cdots \times N_m$ is defined by $y := FFT(x)$, which is obtained as follows,

\[
\tilde{y} := \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \cdots \sum_{n_m=0}^{N_m-1} \tilde{x} \cdot e^{-2\pi i \left(\frac{k_1 n_1}{N_1} + \frac{k_2 n_2}{N_2} + \cdots + \frac{k_m n_m}{N_m}\right)},
\]

where $\tilde{y} = y(k_1, k_2, \ldots, k_m)$, and $\tilde{x} := x(n_1, n_2, \ldots, n_m)$.

Hence, the FFT could be directly computed by a tensor product; however, this would cost $O(N \sum_{i=1}^{m} N_i)$. The advantage of the FFT is that the cost can be reduced to $O(N \log_2 N)$ operations by exploiting the structure of the tensor.
Let $\tilde{x} \in \mathbb{R}^N$ and $\tilde{y} = \text{FFT}(\tilde{x})$, then:

$$\tilde{y} = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \cdots \sum_{n_m=0}^{N_m-1} \tilde{x} \cdot e^{-2\pi i \left( \frac{k_1 n_1}{N_1} + \frac{k_2 n_2}{N_2} + \cdots + \frac{k_m n_m}{N_m} \right)}$$
### 1.1. State-of-the-art: Single-device FFT libraries

<table>
<thead>
<tr>
<th>Library</th>
<th>Language</th>
<th>Developer</th>
<th>GPU support</th>
<th>Open Source</th>
<th>2D &amp; 3D support</th>
<th>Stride data support</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUFFT</td>
<td>C</td>
<td>NVIDIA</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ESSL</td>
<td>C++</td>
<td>IBM</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FFTE</td>
<td>Fortran</td>
<td>Riken</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FFTPACK</td>
<td>Fortran</td>
<td>NCAR</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFTS</td>
<td>C</td>
<td>U. Waikato</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFTW3</td>
<td>C</td>
<td>MIT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FFTX</td>
<td>C</td>
<td>LBNL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>KFR</td>
<td>C++</td>
<td>KFR</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>KISS</td>
<td>C++</td>
<td>Sandia</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>OneMKL</td>
<td>C</td>
<td>Intel</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>ROCM</td>
<td>C++</td>
<td>AMD</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>VkFFT</td>
<td>C++</td>
<td>D. Tolmachev</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
1.2. Distributed FFT computation (3-D case)
1.2. Distributed FFT computation (3-D case)

3-D FFT parallel algorithm

1: From the computation resources, create grid of processes (heuristic approach)

2: Distribute data, define shapes at every transpose phase

3: Choose algorithms and define or tune parameters

4: For each direction \((x,y,z)\):
   - Compute 1D or 2D FFTs
   - Locally transpose data
   - Pack data (either in contiguous or stride memory)
   - Transfer data (Binary or Collective MPI Communication)

5: At output stage, scale and reorder indices, if required
1.2. Distributed FFT computation (3-D case)

Kernels need:
- 1D FFT (e.g. FFTW3, CUFFT)
- Packers/Unpackers
- A message passing interface (MPI)

Schematically, Processor 1 is doing the following tasks:
1.2. Distributed FFT computation (3-D case)

Different implementations for parallel FFT computation:

- **Slabs (1-D decomposition)**
- **Pencils (2-D decomposition)**
- **Bricks decomposition**

*Input 3-D grid*  
*Output 3-D grid*
### 1.3 Distributed FFT libraries

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2DECOMP&amp;FFT</td>
<td>NAG</td>
<td>Fortran</td>
<td>FFTW3, ESSL</td>
<td>CUFFT</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>AccFFT</td>
<td>Georgia Tech</td>
<td>C++</td>
<td>FFTW3</td>
<td>CUFFT</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Cluster FFT</td>
<td>Intel</td>
<td>Fortran</td>
<td>MKL</td>
<td>GenFFT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRAFFT</td>
<td>Cray</td>
<td>Fortran</td>
<td>FFTW3</td>
<td>ACML</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFTE</td>
<td>U. Tsukuba / Riken</td>
<td>Fortran</td>
<td>FFTE</td>
<td>CUFFT</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFTMPI</td>
<td>Sandia</td>
<td>C++</td>
<td>FFTW3, MKL, KISS</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>FFTMPI-KK</td>
<td>Sandia</td>
<td>C++</td>
<td></td>
<td>CUFFT</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFTW3</td>
<td>MIT</td>
<td>C</td>
<td>FFTW3</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>heFFTe</td>
<td>ICL - UTK</td>
<td>C++</td>
<td>FFTW3, MKL, Stock</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>nb3DFFT</td>
<td>RWTH Aachen</td>
<td>Fortran</td>
<td>ESSL</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3DFFT</td>
<td>UC San Diego</td>
<td>C++</td>
<td>FFTW3</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>spFFT</td>
<td>ETH</td>
<td>C++</td>
<td>FFTW3</td>
<td>CUFFT, ROCM</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>SWFFT</td>
<td>Argonne</td>
<td>C++</td>
<td>FFTW3</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
## 1.4 MPI frameworks on distributed FFT libraries

**MPI ROUTINES USED FOR TENSOR TRANSPOSITION WITHIN DISTRIBUTED FFT LIBRARIES.**

<table>
<thead>
<tr>
<th>Library Name</th>
<th>Language</th>
<th>Developer</th>
<th>Point-to-Point exchange</th>
<th>Collective exchange</th>
<th>Process Topology</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Blocking Non-Blocking</td>
<td>Blocking Non-Blocking</td>
<td></td>
</tr>
<tr>
<td>AccFFT</td>
<td>C++</td>
<td>GA Tech.</td>
<td>MPI_Sendrecv</td>
<td>MPI_Alltoall</td>
<td>MPI_Cart_create</td>
</tr>
<tr>
<td>2DECOMP&amp;FFT</td>
<td>Fortran</td>
<td>NAG</td>
<td>MPI_Send</td>
<td>MPI_Alltoall</td>
<td>MPI_Cart_create</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MPI_Cart_sub</td>
</tr>
<tr>
<td>FFTE</td>
<td>Fortran</td>
<td>U. Tsukuba Riken</td>
<td>-</td>
<td>MPI_Alltoall</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MPI_Isend</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFTMPI</td>
<td>C++</td>
<td>Sandia</td>
<td>MPI_Send</td>
<td>MPI_Alltoall</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFTW</td>
<td>C</td>
<td>MIT</td>
<td>-</td>
<td>MPI_Alltoall</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>heFFTe</td>
<td>C++</td>
<td>UTK</td>
<td>MPI_SendRecv</td>
<td>MPI_Allreduce</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nb3dFFT</td>
<td>Fortran</td>
<td>RTWH Aachen</td>
<td>MPI_SendRecv</td>
<td>MPI_Allreduce</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3DFFT</td>
<td>C++</td>
<td>UCSD</td>
<td>-</td>
<td>MPI_Alltoall</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpFFT</td>
<td>C++</td>
<td>ETH Zürich</td>
<td>MPI_Send</td>
<td>MPI_Allreduce</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWFFT</td>
<td>C++</td>
<td>Argonne</td>
<td>MPI_Send</td>
<td>MPI_Allreduce</td>
<td></td>
</tr>
</tbody>
</table>

7/20
At ICL we are working a benchmark harness for evaluating the performance and scalability of state-of-the-art libraries towards Exascale computing.

Refer to: [Interim Report on Benchmarking FFT Libraries on High Performance Systems](#)  
Innovative Computing Laboratory Technical Report, July 2021  
University of Tennessee
2. heFFTe library: FFTs for exascale
The image contains a diagram highlighting various libraries and frameworks related to Fast Fourier Transform (FFT) computations. The diagram shows a network of components such as P3DFFT, SWFFT, AccFFT, FFTX, and others, connected to libraries and frameworks like cuFFT, OneMKL, FFTW, cuBLAS, rocBLAS, MAGMA, and OpenMP. The ECP Ecosystem is also indicated.
1. Definition of input/output processors grids (normally provided by users):

If user only has their MPI communicator and number of processors, we provide a routine to generate above grid of processors:

```cpp
heffte ::proc_setup_min_surface(my_mpi_comm, nprocs);
```
1. Definition of input/output processors grids (normally provided by users):

```
\[
\text{heffte} :: \text{proc\_setup\_min\_surface}(\text{my\_mpi\_comm}, n\text{procs});
\]
```

2. Distribute data among processors using \textit{box3D} objects at input and output:

```
\[
\begin{align*}
\text{std::vector<box3d<index>>} & \quad \text{inboxes} = \text{heffte}::\text{split\_world}(\text{world}, \text{proc\_i}); \\
\text{std::vector<box3d<index>>} & \quad \text{outboxes} = \text{heffte}::\text{split\_world}(\text{world}, \text{proc\_o});
\end{align*}
\]
```
heFFTe API

4. Create FFT plan:

```cpp
auto fft = heffte::make_fft3d<backend_tag>(inboxes[me], outboxes[me], my_mpi_comm, options);
```

**backend_tag**: Corresponds to the FFT library for local computations (e.g., FFTW3, CUFFT, MKL)

**options**: Contains information from flags set by users
heFFTe API

4. Create FFT plan:

```cpp
auto fft = heffte::make_fft3d<backend_tag>(inboxes[me], outboxes[me], my_mpi_comm, options);
```

*backend_tag*: Corresponds to the FFT library for local computations (e.g., FFTW3, CUFFT, MKL)

*options*: Contains information from flags set by users

5. Compute an in-place parallel 3D FFT:

```cpp
std::complex<my_precision_type> *output_array;
fft.forward(output_array, output_array, workspace.data(), scale::full);
fft.backward(output_array, output_array, workspace.data());
```

*workspace.data()*: Can be given by the user or calculated by heFFTe for establishing a computation workspace

*scale::...*: The scaling options are *full*, *none* and *symmetric*
heFFTe API

4. Create FFT plan:

    auto fft = heffte::make_fft3d<backend_tag>(inboxes[me], outboxes[me], my_mpi_comm, options);

**backend_tag**: Corresponds to the FFT library for local computations (e.g., FFTW3, CUFFT, MKL)

**options**: Contains information from flags set by users

5. Compute an in-place parallel 3D FFT:

    std::complex<my_precision_type>* output_array;

    fft.forward(output_array, output_array, workspace.data(), scale::full);
    fft.backward(output_array, output_array, workspace.data());

**workspace.data()**: Can be given by the user or calculated by heFFTe for establishing a computation workspace

**scale::...**: The scaling options are **full**, **none** and **symmetric**

6. Tracing functionality can be added within your code to generate a runtime trace for performance analysis.

-D_HEFFTE_ENABLE_TRACE=ON

    heffte::add_trace(“Initiating tracing”);
    ---Code to be traced ---
    heffte::add_trace(“Ending tracing”);
Experimental Setup

Our experiments were performed using up to 1,024 Summit nodes, out of a total of 4,608. Each node consists of two sockets, each composed of a 22-core IBM POWER9 CPU and 3 NVIDIA Volta V100 GPUs.

Architecture of Summit nodes: computing units and network connections.
3. Experiments and results
2.1 Performance improvement from CPU to GPU

Fig. 4. Profile of a 3D FFT of size $1024^3$ on 32 Summit nodes with all-to-all communication – using 40 MPI processes per node (left pie) and 6 MPIs per with 1 GPU per MPI (right pie).
2.2 Strong Scalability - Summit

Comparison of pencil and slab decompositions for strong scaling of a 3-D FFT of size $1024^3$. Using *heFFT* with cuFFT backend, 6 MPI processes (1 MPI processes per GPU-V100) per node, and single-precision complex data.
2.3 Communication bottleneck

Comparison of strong scaling (with fixed size $1024^3$) using 6 MPI processes (1 MPI processes per GPU-V100) per node, for different interconnection options, using MPI_Alltoall communication.

Scalability of MPI_Alltoall communication for a double-complex precision of a 3-D FFT of size $1024^3$ with cuFFT backend and 6 V100-GPUs per node.
2.3 Communication bottleneck

Time for MPI_Alltoall communication and number of message sent for a 3D FFT of size $1024^3$. Using 40 MPIs and 6 GPUs per node.

Scalability of MPI_Alltoall communication for a double-complex precision of a 3-D FFT of size $1024^3$ with cuFFT backend and 6 V100-GPUs per node.
3.1 Performance analysis - MPI

Our experiments show that further tuning of MPI parameters and network topologies can help to get faster computation, specially for small size number of resources.

Comparison of achievable bandwidth from two-node exchange via MPI_Send, using of MVAPICH, SpectrumMPI and OpenMPI-UCX on Summit.
3.2 Impact of MPI on scaling

In the following figures we explore the scalability of MPI_AlltoAllv via a heFFTe experiment on a $1024^3$ complex-to-complex 3-D FFT, using 6 NVIDIA V100GPUs per node.

- **Spectrum MPI 10.3**
- **MVPICH-GDR 2.3.5**
- **OpenMPI + UCX**
3.2 Impact of MPI on scaling

![Graph showing impact of MPI on scaling up to 64 nodes (192 GPUs)](image1)

- Spectrum MPI
- MVAPICH
- OpenMPI-UCX

Up to 64 nodes (192 GPUs)

![Graph showing impact of MPI on scaling from 128 to 1024 nodes (6144 GPUs)](image2)

- Spectrum MPI
- MVAPICH
- OpenMPI-UCX

From 128 to 1024 nodes (6144 GPUs)
3.2 Impact of MPI on scaling

MVAPICH is around 10-20% faster than SpectrumMPI 10.3

![Graph showing the impact of MPI on scaling up to 64 nodes (192 GPUs)].

Up to 64 nodes (192 GPUs)

![Graph showing the impact of MPI on scaling from 128 to 1024 nodes (6144 GPUs)].

From 128 to 1024 nodes (6144 GPUs)
3.2 Impact of MPI on scaling

MVAPICH is around 10-20% faster than SpectrumMPI 10.3

No much difference for large number of nodes
3.3 Performance analysis

In the following figure we compare the runtime to compute a 3-D transform of size $1024^3$, using double-precision complex random input.

Strong scalability for a 3-D FFT of size $1024^3$ using 40 MPI processes per node, 1 per IBM Power9 core, 20 per socket. Using 2 reshapes (transpositions) per FFT direction.
3.3 Performance analysis

In the following figure we compare the runtime to compute a 3-D transform of size $1024^3$, using double-precision complex random input.

Strong scalability for GPU libraries using 4 NVIDIA V100 GPUs per node, 2 per socket.
Conclusions

- We presented an analysis of performance and scalability for 3-D parallel FFT libraries and analyzed the computation on upcoming large scale systems.

- Scalability of FFTs is highly impacted by how libraries handle communication for a large number of process units.

- Tuning can help to further accelerate the performance of current FFT software. The choice of the MPI distribution can help to further speedup; however, this is architecture dependent.

- Scalability towards Exascale would require (auto) tuning of algorithmic parameters and bandwidth management.