Designing an adaptable framework for highly scalable multidimensional spectral transforms

Dmitry Pekurovsky
UC San Diego

Importance of Spectral Transforms @ Exascale

- Includes FFTs but is more general
- Ubiquitous algorithm in many areas of science/engineering
- Responsible for a large portion of cycles consumed on modern supercomputers
- Can be the limiting factor for bringing codes to Exascale
  - Exascale potential not realized in a number of important fields

WHAT CAN BE DONE TO RECONCILE SPECTRAL TRANSFORMS AND EXASCALE?

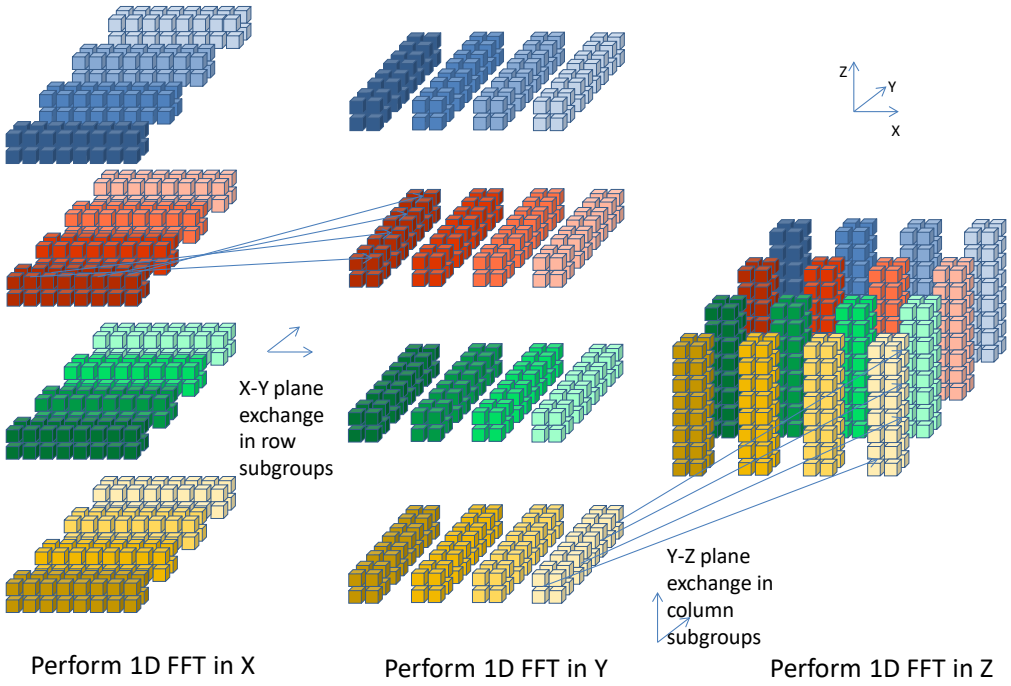
WHAT CAN BE DONE TO INCREASE THE RANGE OF USE BEYOND TRADITIONAL DNS TURBULENCE CODES?

CAN PERFORMANCE AND FLEXIBILITY BE COMBINED?

Common Properties of Spectral Transforms in Multiple Dimensions

- Easily broken down into independent 1D transform components (example: 3D FFT)
- Each 1D component transform is non-local, i.e. uses heavily all data in given dimension
  - Best suited for node-local implementation, i.e. no inter-processor communication during the 1D transform
  - Algorithm typically reshuffles data among processors between consecutive 1D transforms
  - Limited by bisection bandwidth
  - Also limited by node memory bandwidth (see e.g. McClanahan 2011)
- **Recognized as a serious challenge for Exascale**

3D FFT algorithm with 2D decomposition



Existing work

- Many open source implementations available for 3D FFT (also many proprietary ones)
  - FFTW, FFTE, P3DFFT, OpenFFT, AccFFT, PFFT, NBFFT etc
- Each has areas of emphasis
- Traditionally focused on DNS turbulence style codes
- All have restricted data layouts
- Few support overlapping communication
- Few support GPUs
- Few support pruned and/or sparse FFTs
- Few are optimal both in large and small grid range

Expanding functionality

- Spectral Transforms: a diverse ecosystem
  - Varied boundary conditions
  - Varied data types
  - Varied data layout
  - Varied processor decomposition
  - Special cases and potential for optimization:
    - Pruned/sparse transforms
    - Multivariable transforms
    - Higher-order operators: derivatives, convolutions, Laplacian etc
    - Large vs. small grid sizes

Performance Strategy 1

Exascale performance is sub-par, try to minimize the damage.

- Reduce the volume of data sent across the network and to/from memory: employ pruned and/or sparse FFTs where possible

Pruned transform: use same algorithm (e.g. FFT), after 1D transform of size N keep n < N modes.

Example: used in dealiasing methods in turbulence simulations (2/3 rule). Saves volume of communication by close to 50% for cubic grids. Also saves compute time and memory access time.

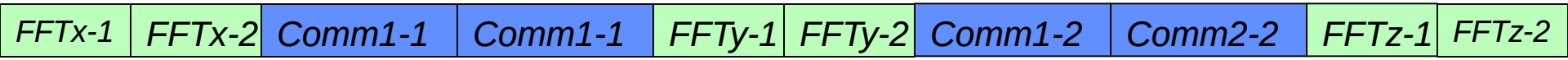
Sparse transform: specialized algorithm for cases with n << N.

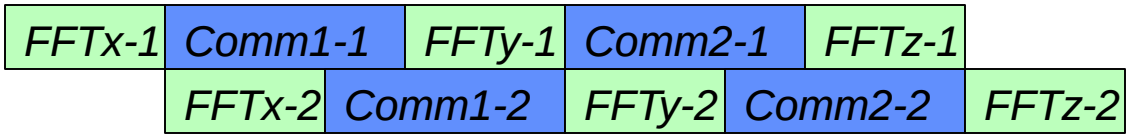
Can be faster than full FFT.

Performance Strategy 2

Combine multiple transforms into a single call

- Aggregate messages
- Overlap communication with computation





Performance Strategy 3

Reduce number of memory reads/writes

- Reduce the number of local memory transpositions
- Optimize cache use for non-unit-stride reads/writes
- Reduce array copies
- Reuse cache by performing several operations in a single loop

Example of cache reuse

- Attempt to combine pairs of the following three operations on slices of the whole array, to reuse cache:
  1. 1D transform (stride-1)
  2. Local memory reordering.
  3. Pack/unpack of send/receive buffer for interprocessor communication.

```
for(k=0;k<N;k++) // Do one k-slice – this often fits in 1 cache line
   for(j=0;j<N;j++) {
      Transform1D(&A[k][j][0]);
      for(ip=0;ip < Ntasks;ip++) {
      Nstart = N/Ntasks*ip; Nend = Nstart + N/Ntasks;
          for(i=Nstart;i<Nend;i++)
             sendbuf[ip][k][i][j] = A[k][j][i];
      }
   }
MPI_Alltoall(sendbuf,…);
```

Performance, contd.

- Strategy 4: Utilize accelerators
  - CUDA implementation
- Strategy 5: Autotuning: select best execution path; communication method; best decomposition etc

P3DFFT++ Framework and design

- Open source package, with examples and documentation
- Distinct code from exiting P3DFFT package
- Object-oriented, modular design, hides platform and implementation details
- Written in C++, with C and Fortran interfaces
- Compute-intensive parts written in C-style code to optimize performance
- Highly flexible data layout

P3DFFT++ Framework and design, contd.

- Uses MPI, eventually combined with OpenMP and CUDA
- Supports multiple transform types (beyond FFT)
- Provides convenience functions for data manipulations
- (Work in progress) Autotuning to select fastest algorithm
- Similar to most libraries, offloads 1D transforms to a specialized library, such as FFTW

Features (current and future)

- Optimized for large core counts (1D, 2D and 3D decomposition)
- Flexible data layout
- Pruned/sparse transforms
- Multivariable transforms (with overlapping communication)
- GPU-enabled
- 3D and 4D transforms
- Autotuning for best algorithm
- Higher-order functions (derivative/Laplacian/convolution etc)

Some details of implementation

- Data types: real/complex, single/double precision
- Functions and classes expressed as C++ templates, with interfaces for C and Fortran
- Low level functions: combinations of 1D transform/derivative, local transposition, interprocessor transposition – accessible to the user.
- 1D transform types: designed as general as possible, with FFTW currently serving as reference implementation.
- Higher level functions: 3D transforms (future: 4D transforms, correlation functions, Laplacian etc)
- Execution model: planner and execution functions, similar to FFTW, plans expressed as C++ classes

Data layout specification

Metadata descriptor `grid`:

- Data grid size and properties
- Processor grid definition (1D, 2D, 3D)
- Local memory layout (order of dimension storage)
  - E.g. (x,y,z), (z,y,x) etc
- Mapping of data grid onto processor grid
  - Example: $P_{grid} = P_1 \times P_2 = 4 \times 8$. Data grid 32^3. Map Y dimension onto P_1, Z dimension onto P_2. Local dimensions: 32 x 8 x 4. Can be stored as array A[4][8][32] (default layout (0,1,2)), or A[32][8][4] (transposed layout (2,1,0)), or any other order

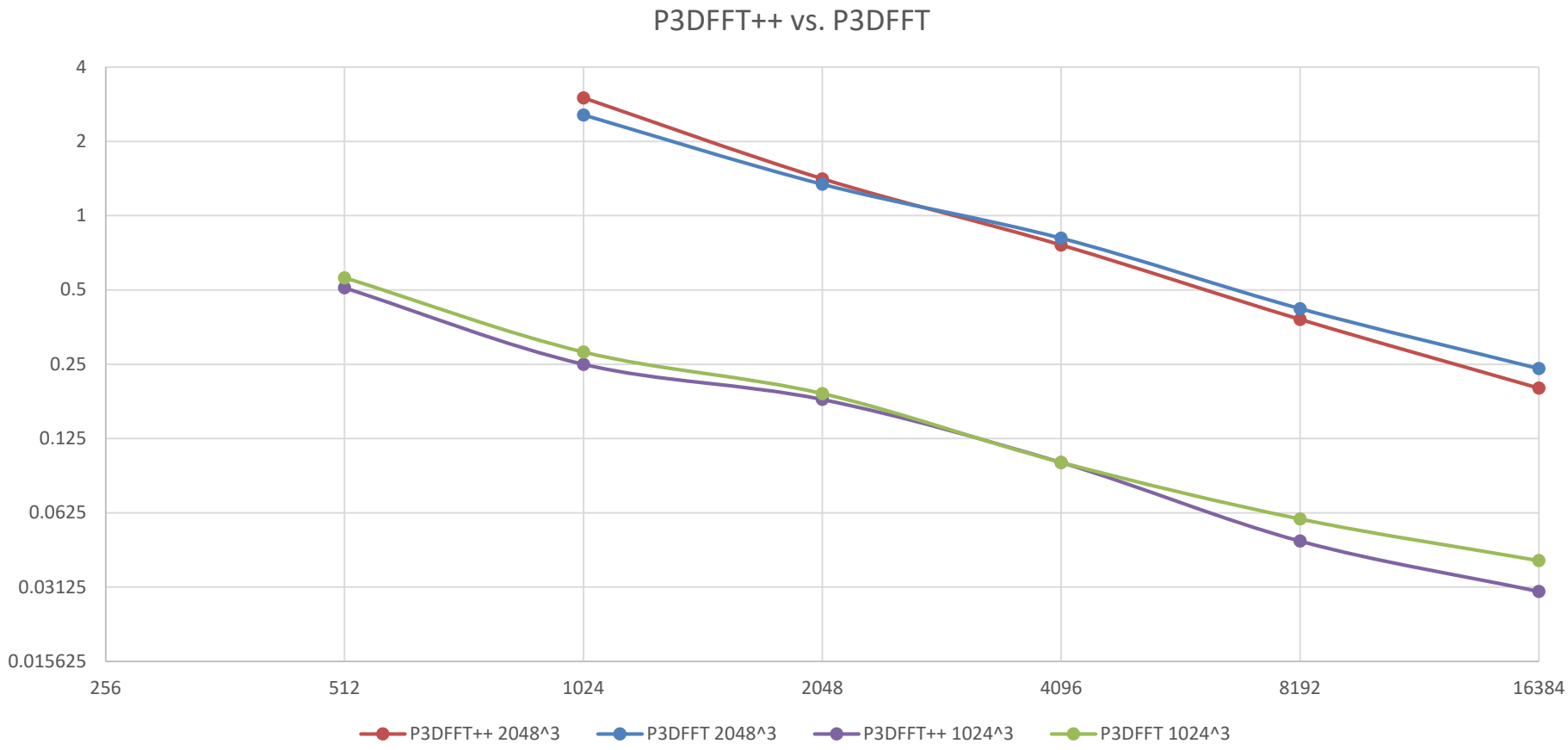
Example of usage

```
int pgrid1[]={1,2,4};
int mem_order1[] = {0,1,2};
int pgrid2[] = {2,4,1};
int mem_order2[] = {2,1,0};
int proc_order[] = {0,1,2};
int gdims[] = {128,128,128};

grid Grid1(gdims,-1,pgrid1,proc_order,mem_order1,mpicomm);
grid Grid2(gdims,-1,pgrid2,proc_order,mem_order2,mpicomm);

int type_ids[3] = {R2CFFT_D,CFFT_FORWARD_D,CFFT_FORWARD_D};
trans_type3D type_rcc(type_ids1);

transform3D<double,complex_double> trans_rcc(grid1,grid2,&type_rcc);
```

Example of usage, contd.

```
int sdims1[3],sdims2[3],glob_start1[3];
for(i=0;i<3;i++) {
sdims1[mem_order1[i]] = grid1.ldims[i];
sdims2[mem_order2[i]] = grid2.ldims[i];
glob_start1[mem_order[i]] = grid1.glob_start[i];
}
int size1 = ldims1[0]*ldims1[1]*ldims1[2];
int size2 = ldims2[0]*ldims2[1]*ldims2[2];
double *IN=new double[size1];
double *OUT=new complex_double[size2];
init_ar(IN,sdims1,glob_start1);
trans_f.exec(IN,OUT,false);
```

Preliminary performance study (Stampede2 @ TACC)



Conclusions and future work

- A new open source package P3DFFT++ (http://www.p3dfft.net) aims to bridge high performance with ease of use and increased use range.
- New object-oriented design
- Preliminary performance on par with P3DFFT and similar libraries.
- Future/in progress work
  - GPU/CUDA
  - Pruned transforms
  - Overlap of communication
  - Higher order functions/combinations
  - 4D transforms
  - Autotuning

Acknowledgements

- **Supported by NSF OAC grants**
- XSEDE resources at TACC and SDSC have been used in this work