A Portable Framework for Multudimensional Spectral-like Transforms At Scale

Dmitry Pekurovsky
San Diego Supercomputer Center
University of California at San Diego
La Jolla, California, USA
dpekurovsky@ucsd.edu

Abstract— **We report progress in an ongoing effort to develop a versatile and portable software framework for computing muti-dimensional spectral-like transforms at large scale. The design covers Fast Fourier Transforms and other algorithms that can be broken down into line operations. This class of algorithms covers a wide range of scientific applications and are notoriously challenging to scale on largest supercomputers. Another challenge addressed by this project is the fast pace of change in the field of High Performance Computing, with new systems and paradigms appearing every few years, demanding great adaptability and effort on behalf of software developers. To this end we have developed a flexible software framework as an open source package named P3DFFT++. It is written in C++ in a highly object-oriented fashion, with interfaces for Fortran and C. The goal is to shield the user from details of low-level mechanisms of communication and computation by providing a portable high-level API for commonly used algorithms. The framework will incorporate many modern HPC programming features, such as GPU implementation, overlapping communication with computation and GPU data transfer, as well as algorithm autotuning. We cover design choices of the package and early results.**

Keywords—Fast Fourier Transforms, Parallel Programming, Scientific Computation, Numerical Algorithms, Open Source Software

I. Introduction

Fast Fourier Transforms (FFTs) is a ubiquitous algorithm in computational science and engineering, second only perhaps to linear algebra in terms of the universality of impact. Codes from a variety of disciplines rely on FFTs, often through the use of third-party libraries, to simulate a wide range of phenomena. This paper deals with a challenging case of multidimensional (3D and 4D) FFTs computed repeatedly during simulations running on the medium to high end of High Performance Computing (HPC) architectures, in terms of size and power. This is the typical way they are used for many applications, including (but not limited to) Direct Numerical Simulations of turbulence, simulations of the ocean and atmosphere, astrophysics, acoustics, seismic simulations, material science, medical imaging and molecular dynamics.

The challenge of FFTs at large scale is well-studied and has to do with dependence on the system's **bisection bandwidth**, as well as on-node **memory bandwidth** [1-3]. This is a known issue on high-end HPC systems today and, as we approach the Exascale, this limitation is only going to become worse and will become a significant bottleneck for many applications. A number of past-decade implementations of Multi-dimensional (M-D) FFT aim to optimize performance at large scale, utilizing strategies such as two-dimensional (2D) decomposition [4-11]. While this was a significant step forward compared to one-dimensional (1D) decomposition, now there is a greater need than ever to work on evolving M-D FFT algorithms and software to keep pace with the system evolution and application requirements.

Fourier Transforms come in many flavors. In addition, there are algorithms sufficiently similar to FFT. They can be placed in the same category, which we call **spectral transforms**. When creating a general use package, it makes sense to make it as general as possible, without losing performance. Many existing spectral transforms implementations have a rigid or specialized **user interface**, limiting their usability**.** Past packages (such as P3DFFT, the predecessor of P3DFT++ in terms of key ideas) have been created in response to the largest demand at the time, namely Direct Numerical Simulations of turbulence (DNST) field [12-26], where distinct applications typically have a fairly consistent set of data structures. These packages focused on achieving high performance, while staying within the narrow range of data structures and problems of a typical DNST application. The natural next step of evolution might encompass other possible use scenarios, including data structures, feature sets and opportunities for application-specific optimizations. This is where we perceive the need for a universal approach that would allow a **high degree of flexibility** in terms of features and usage, at the same time **maximizing scalable performance** by incorporating lessons learned from earlier work on M-D FFT packages. This is precisely what the new framework P3DFFT++ aims to accomplish. It provides an **adaptable**, **portable implementation** of M-D FFT and related spectral-like algorithms as an open source package. This is reminiscent of the role FFTW [27] has played for spectral transforms in the past, and the way packages like BLAS and LAPACK have been the standards for interface and implementation in the field of linear algebra computations.

P3DFFT++ provides both a software package and a universal API for using M-D FFT and related spectral-like transforms. It allows for user's choice of data layout in terms of

memory storage and processor space decomposition. It supplies an abstraction level that hides optimization from the user, thus making it easily adaptable to a number of architectures. P3DFFT++ is built in a modular manner, providing opportunities for expansion, for example by adding more transform types in addition to FFTs (e.g. sine/cosine, wavelets, high-order finite difference schemes etc). It also aims to streamline calculations of constructs that rely on FFTs (such as derivatives and convolutions), in order to improve performance and maximize usability. In short, P3DFFT++ is designed to be a **universal toolbox** for a broad range of uses of spectral transforms, while maintaining competitive performance at scale.

In this paper we start by defining the problem in the context of previous work in the field. We proceed to describe design elements of P3DFFT++, then to demonstrate its performance and ease of use.

II. Previous Work and the Scope of the Project

Throughout this paper we refer to spectral-like transforms when we mention any multi-dimensional transform algorithm on a structured grid that has the following properties:

1. It can be reduced to a sequence of 1D transforms for an entire array, one for each dimension, independent of other dimensions.

2. Each such 1D transform is compute and memory-bandwidth intensive. In terms of data decomposition, it is best to have all data in that dimension to reside locally in memory for each core/task. The reason for the above is avoidance of heavy communication that would be necessary to exchange data between the stages of the 1D algorithm if the data were not all local.

Clearly, all flavors of 3D FFT and sine/cosine transforms fall under this category. In addition, high-order finite difference schemes and wavelets can also be considered spectral-like transforms and supported by P3DFFT++ without loss of generality.

As mentioned in the introduction, P3DFFT package [4] has been an important cornerstone in the evolution of this software niche. P3DFFT was written in Fortran90 and MPI/OpenMP, encapsulating a well-performing formulation of spectral algorithms suitable for extreme scale computation. In particular, P3DFFT implements 2D decomposition in processor space, which allows in principle to scale a N^3 problem up to N^2 tasks. P3DFFT follows the most common sense and efficient path of computing 3D FFT (for details see [4] and Fig. 1): starting with data distributed as pencils local in X dimension, we do 1D FFT in X, Y and Z dimensions, in turn over the entire array, interspersed with two each of local and inter-processor transposes. The end result consists of data distributed as a Z-pencil. It is assumed that after performing necessary operations in Fourier space (such as taking derivatives, convolution etc) a user might want to do an inverse transform from Z- to X–pencils.

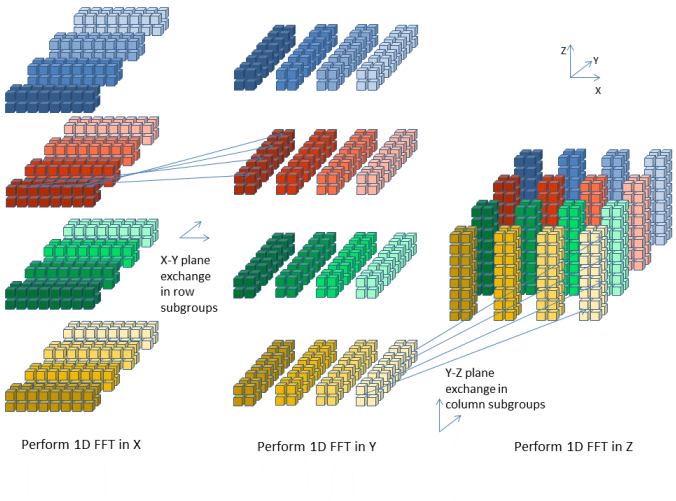

Figure 1. 3D FFT implementation with 2D decomposition typically involves a sequence of 1D transforms in X, Y and Z dimensions, interspersed with two sub-communicator all-to-all exchanges.

Thus only X-pencils are supported in physical space and Z pencils in the Fourier space, out of many other potentially useful configurations. In contrast, as we shall see below, P3DFFT++ envisions many other possible choices for data layout. In addition, the way 1D FFTs and the transposes are combined in P3DFFT is fixed and the user has no control in the process. Finally, P3DFFT provides only real-to-complex transforms, as well as a few related real-to-real transforms such as sine and cosine. The implementation of the algorithm is fixed and thus not adaptable to multiple architectures.

While performance of a package like P3DFFT on a suitable problem and hardware may be impressive (see Fig. 2), even such solutions will not be sufficient as we step into Exascale computing, in view of both problem and architectures adaptability. Exascale computing is very much a moving target, in terms of both hardware and software paradigms. Therefore one hopes for a framework portable enough to avoid rewriting scientific software every 1-2 years.

A number of other 3D FFT solutions have been published, both as open source third-party libraries [5-11] and as parts of proprietary codes [28-34]. Although a thorough review of 3D FFT packages is out of the scope of this paper, it is fair to say that most existing solutions implement an approach similar to P3DFFT, with some variations of features and specialization for certain use cases and platforms. For example, some of these packages may have features like autotuning, 3D decomposition, GPU implementation, pruned transforms and overlap of computation with communication, while no one package provides all of them. None of the existing implementations, to the author's knowledge, provide flexible data layout options. The abundance of packages may be daunting for a new user, especially without a clear information contrasting them. It takes substantial time and commitment to thoroughly evaluate ten or so libraries and compare their performance to make an informed choice. Once committed to a library, the user is unlikely to switch.

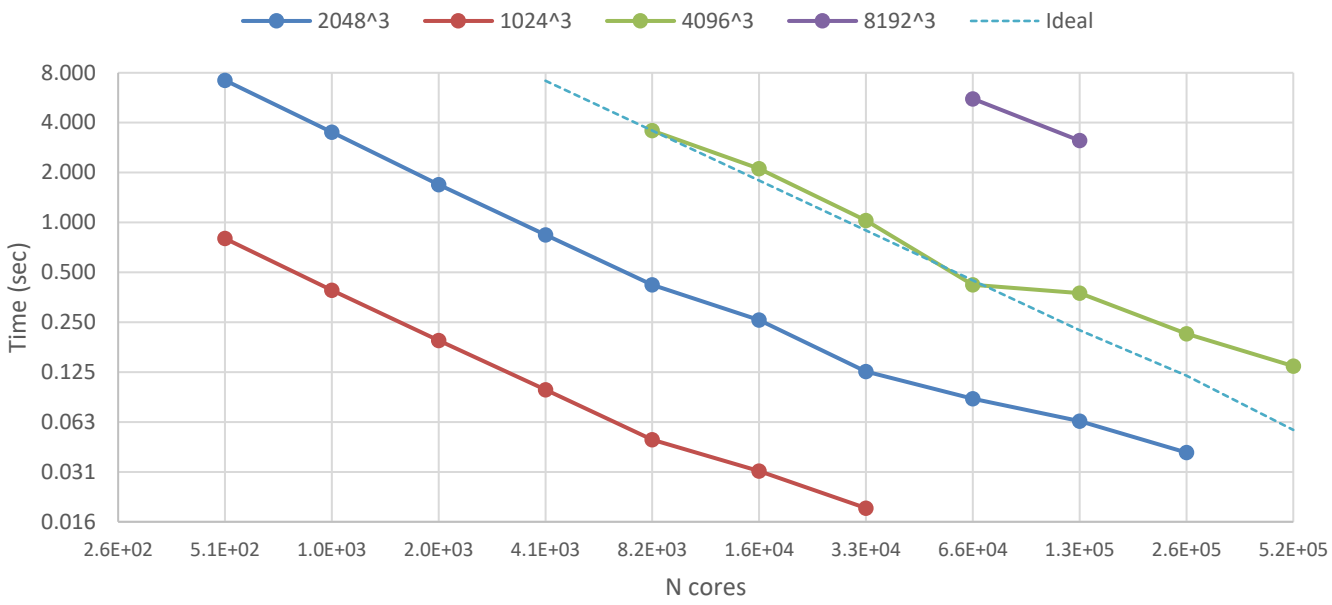

Figure 2. Strong scaling of P3DFFT on Mira (IBM BlueGene/Q at Argonne National Lab).

Therefore, they may be missing useful features and/or performance and in some cases may not even realize it.

P3DFFT++ aims to combine the best features mentioned above under a "one roof" approach. In addition, it expands the context of a spectral transform in modern computing by giving more choice about what the user can do. Using C++ object-oriented features, it encapsulates many options in a convenient interface.

P3DFFT++ building blocks are 1D transforms, local transposes and interprocessor transposes. By combining these blocks in any desired way the user has a high degree of control over the execution of higher-dimensional transform. The user can do any combination of 1D transforms and transposes, going all the way to 3D FFT. This framework is extremely flexible and can be customized, for example, for cases such as de-aliasing in computational fluid dynamics, where only parts of the 3D spectrum are needed for the computation, and the rest can be discarded, with resulting savings in compute time. P3DFFT++ can be thought of as a FFTW-like standard for multidimensional spectral transforms on pre-Exascale and Exascale machines.

In designing P3DFFT++, special attention is paid to minimizing expensive operations, such as inter-processor communication, local memory transposes and other operations leading to cache misses. Even though P3DFFT++ uses a higher-level language and has a flexible interface, its performance is on par with that of existing packages like P3DFFT, for cases that both can handle. Continued work is aimed at achieving even higher performance by utilizing modern optimizations with potential to make a difference at Exascale level.

In addition to the low-level functions ("building blocks" mentioned above), P3DFFT++ provides high level 3D (and, in the future, higher dimensional) transform functions, both for convenience and optimization. In particular, an autotuning framework is going to be used for the planning stage, choosing the best execution path for a given combination of platform and problem. The framework also includes utilities for derivative and convolution calculations.

III. P3DFFT++ Framework and Design

P3DFFT++ is implemented in C++ and uses MPI for inter-processor communication. 1D transforms are delegated to standard libraries such as MKL, FFTW, ESSL, CUFFT, or alternatively to a user-defined implementation. The package includes C++, C and Fortran interfaces, and documented through examples, tutorials and reference pages. The home page for this package is `http://www.p3dfft.net`. The package is released through `github.com` with an open source license.

P3DFFT++ uses an object-oriented design to encapsulate various data structures and transforms into classes and class templates, providing a clear and concise interface for the user. It presently supports four datatypes: single-precision real and complex, and double-precision real and complex. Most classes are defined as C++ templates in terms of input and output datatype, for example:

```
template <class Type_in,class Type_out>
class transform3D;
```

P3DFFT++ borrows the idea of transform planners from FFTW. Namely, each transform (be it 1D or 3D) has a planner function (usually contained in a C++ class constructor) that gets called once when the transform is initialized, and contains any setup arrangements for execution (as well as possibly trial execution runs within an autotuning framework). Once a transform has been "planned", it can be executed multiple times in a fast call. Using C++ classes is a convenient way to encapsulate all the information and functions referring to a transform.

A. Data layout descriptors

As mentioned above, P3DFFT++ is intended to support very general data layout, both in terms of grid decomposition and

memory arrangement. We begin with discussion of 3D grids, with generalization for 4D straightforward. A 3D data grid can be mapped onto a 1D, 2D or 3D processor grid. (Most 3D FFT implementations use 1D or 2D decomposition, which is well-suited for the algorithm as it preserves at least one entirely local dimension. However, some applications have inherent 3D decomposition, and it is necessary for a package like P3DFFT++ to deal with such cases, so this layout choice will be included in future versions). Let P_1, P_2 and P_3 be dimensions of the processor grid. (In case of 2D decomposition one of these will be equal to 1, and in case of 1D decomposition two of these will be equal to 1.) The processor grid is constructed via MPI Cartesian communicators. If the dimensionality of the processor grid is greater than 1, there are multiple ways the processor grid can be constructed from the (linear) global MPI space of ranks 1 through N_p. For example, P_1 can be mapped onto the fastest-changing index of the 3D processor grid, so the corresponding communicator contains adjacent MPI ranks. Alternatively, it can be mapped to next-fastest-changing index, so the stride in terms of MPI ranks is P_2. Finally, it can be mapped onto the slowest-changing index, so the stride is P_2P_3. Such topological choices may in some cases be embedded in the application calling P3DFFT++. In other cases choosing a different topological mapping may potentially make a performance difference. Therefore, to adequately describe data decomposition among MPI tasks, for each of the three array dimensions (rank `i`) the following information is required:

1. Size of the global grid, G_i

2. Size of the MPI communicator subdividing this dimension, P_i (with the value of 1 implying a local dimension).

3. Topological rank of the communicator among the three dimensions (represented as an integer 0 through 2, with 0 corresponding to fastest-changing index/adjacent MPI ranks, and 2 corresponding to slowest-changing index/largest stride).

The above description describes the size and location of each local portion of the grid in the global grid, as well as assigns each MPI task its own position in the multidimensional processor grid. The local portion of the grid for each task has dimensions easily computed as $L_i = G_i/P_i$. Next consider how a three-dimensional array is stored in memory. The simplest case would be to store the array simply following the logical dimensions ordering, namely first dimension (`X`) is stored with stride-1 access, followed by `Y` and `Z` (this is sometimes called Fortran, or row-major, storage, although the name is misleading as it can be used both in Fortran and C). However, more generally, each logical dimension `i` (range 0 through 2) can be stored as rank M_i in the mapping of the 3D local array onto the one-dimensional RAM of the node. Each M_i can have values from 0 to 2, with 0 being the stride-1 access dimension and 2 being the largest-stride dimension. The reason we include a generalized memory ordering is that spectral 1D algorithms are fastest when the data to be transformed is arranged in stride-1 pattern. Thus it is necessary to transpose the data locally in memory (in addition to transposing it in MPI space) when going from `X` to `Y` and then to `Z` 1D transforms. In addition, the calling application may have its own storage conventions, and P3DFFT++ attempts to describe the most general case.

The above descriptions of data decomposition in MPI space and local storage scheme are enough to avoid most ambiguities in data layout representation. Every word in memory location for every MPI rank is assigned its place in the global data and processor grids, and vice versa. The original 3D array is assumed to be contiguous, although in the future it is possible to expand the description to include non-contiguous arrays, such as subarrays embedded in larger arrays.

B. Base classes

The `grid` class includes all information about data layout for a given variable, as explained above. It includes the following fields: global and local (per MPI task) grid dimensions G_i and L_i; MPI information (such as task ID, number of tasks and MPI communicator); processor grid information (size of communicator P_i and how it maps onto the global communicator space); local storage layout information, namely memory storage ordering M_i. In short, this class describes all relevant aspects of data layout, and is used as metadata for array variables. For example, when defining a 3D FFT, the grid objects for input and output arrays may have the same global grid dimensions but different local dimensions, distribution among MPI tasks and local storage layout. The grid class is oblivious to datatype of the data array, as that information is encoded in the class templates for transform classes.

P3DFFT++ defines a number of the most common spectral 1D transform types, such as complex-to-complex FFT, real-to-complex FFT, cosine transform etc. This list includes an empty transform, implying simply a copy from input to output, as a convenience feature. Each 1D transform type is defined as a class containing basic information such as the size of the transform N, number of transforms in a bunch m, data types for input and output, a pointer to the planner for this transform (such as `fftw_plan_dft_many` in FFTW), and a pointer to the execution function (such as `fftw_execute_dft`).

A general multidimensional transform is defined as a sequence of 1D transforms of suitable types as well as local and interprocessor transposes. In P3DFFT++ this is expressed as a linked list of classes of type `stage`. Each `stage` can be of the following three varieties (programmed through derived classes in C++):

1. The 1D transform class includes necessary information to execute 1D transform for a 3D array, including the size of the transform, number of elements needing to be transformed independently, strides, and the transform type. The number of elements `m` can control whether we transform just one line ($m=1$), one plane ($m=L_2$, dimension of local array in the plane NL_2) or the entire volume ($m=L_2L_3$). Notice that in contrast to the transform type class, which only describes the type of transform and is oblivious of the data, this class includes all relevant information about the data, such as grid metadata descriptors. The constructor for this class includes a call to planning functions (for example if FFTW is used for 1D FFTs, then a call to fftw_plan_dft_many or a similar planner is included). An execution member function executes the 1D transform that has been planned by the constructor and can be called multiple times. This class also includes local memory

transposes, called independently or in conjunction with the transform to optimize memory access.

2. The interprocessor transpose implements an exchange equivalent to `MPI_Alltoall` in a Cartesian subcommunicator, including all the needed packing and unpacking, as well as local memory transposes. Its constructor initializes fields such as the MPI communicator handle and dimensions, and buffer sizes for the all-to-all exchange. The exchange itself may be implemented via `MPI_Alltoall`, or an alternative method such as pairwise exchange. Autotuning mechanism is planned in future versions to help establish the best performing option for a given platform and problem.

3. Another class combines interprocessor transpose with 1D transform, providing opportunities for optimization by minimizing memory access.

These classes form the backbone of P3DFFT++. An arbitrary sequence of these three types of stage classes defines an execution path for a multidimensional transform (with the limitation that exactly three 1D transforms are included, for example, in the case of 3D transform). A user can arrange them in any manner (as long as the data descriptors are consistent between the consecutive stages).

C. *Higher-level classes*

P3DFFT++ also provides a higher-level class transform3D, which combines individual stages needed for a given 3D transform in an optimized fashion. This class takes as input the metadata descriptors (grid objects) for input and output arrays, as well as the three transform types to be used in X, Y and Z dimensions. Constructor for this class includes planning of the 3D transform algorithm, including necessary calls to 1D transform planners. Since there are multiple execution paths possible for a given 3D transform, this class also includes an autotuning framework to choose the best possible execution path (this is described in more detail below). The execution member function executes the path that the autotuner found to be the best. In addition, a derivative execution function is provided, in order to combine the spectral transform with derivative calculation in the Fourier space (such as multiplication by the suitable wavenumber).

Future work will include more higher-order classes, for example a 4D transform, more derivative options (e.g. Laplacian, divergence, curl) and convolutions.

IV. Performance considerations

While some might expect a loss in performance of P3DFFT++ due to the use of C++, as well as expanded feature set, this loss turns out to be negligible. According to its desan ign, the overhead from higher-level C++ features is small compared to the bulk of the computation, which is done either by specialized libraries such as FFTW, or a C-style code in critical portions of the software. In addition, ongoing work includes a GPU interface.

A. *Interprocessor communication*

Interprocessor communication is the main bottleneck for performance of spectral algorithms at large scale. It involves all-to-all exchanges, done repeatedly, either within the global communicator, or within Cartesian sub-communicators. In either case, this is an expensive operation with high data volume, that tends to stress the system interconnect's bisection bandwidth. As the size of high-end HPC architectures grows, bisection bandwidth is typically not growing at the same rate, so performance of all-to-all communication is likely to become an even bigger bottleneck as time goes on.

P3DFFT++ employs several strategies to minimize the impact of this problem. Firstly, an autotuner can select the best path of execution, as explained below, minimizing the cost of such all-to-all exchanges. In addition, an overlap of communication with computation will be implemented for certain types of transforms. Earlier results [10, 35-38] suggest this can partially hide the cost of the expensive all-to-all exchanges. Finally, providing the pruned transforms option (where only a part of the Fourier spectrum is kept) helps reduce the volume of data in such exchanges, with proportionate decrease in cost.

B. *Memory access*

In addition to bisection bandwidth, spectral algorithms typically stress local memory bandwidth on each compute node. In fact, some authors predict this bottleneck may even overshadow the bisection bandwidth limits in future systems [1,2]. In particular, this arises in three types of situations:

1. Executing local 1D transforms, such as FFT.

2. Transposing the data locally in memory between 1D transforms.

3. Packing/unpacking send/receive buffers for an interprocessor exchange.

P3DFFT++ design is concerned with minimizing the number of memory reads and writes. Especially concerning are non-stride-1 reads and writes. Such patterns of access are a known source of inefficiency, due to a high number of cache misses involved. Unfortunately, such operations are an integral part of spectral algorithms. P3DFFT++ follows the design choice common to most MD FFT implementations, namely calling an established 1D transforms via established library such as FFTW. Since these transforms are cache-intensive, for best performance these calls are done for data arranged in stride-1 pattern. P3DFFT++'s task, therefore, is to rearrange the data in stride-1 pattern before calling each of the 1D transforms. This is done by reorder functions implementing the loop blocking method to minimize the price for cache misses. Since no assumptions are made about the input and output data layouts, such reordering may be needed in the beginning and in the end of the run as well.

P3DFFT++ takes advantage of opportunities for optimization in transition between the three main situations listed above, by combining two of them in a way maximizing cache reuse. Below is an example (in pseudocode) of a combined call to 1D transform with memory transpose. The code in this example transposes memory ordering from (0,1,2) to (1,0,2), meaning that only the first two indices are interchanged. A temporary array for each `k` value is used first to transpose the input array, then to do a transform (in the dimension of the first index `j`, for all `i`), writing results into the

k's space of the output array. The transposition inherently is not cache-friendly (and could be further optimized by loop blocking, as in other parts of the code), however compared to the separate transpose and transform operations, we save one read+write equivalent of the entire array.

```
for all k:
  for all j and i:
    temp(j,i)=In(i,j,k)
    transform1D(Temp,Out(1,1,k))
```

C. GPU capability

Most modern pre-exascale computers include GPUs as part of system design, and therefore it is important for competitive software packages to make use of this capability. Since GPU technology changes rapidly and different vendors often have incompatible interfaces, it is imperative for any long-term package to include an interface that is portable and general enough.

Work on GPU interface in P3DFFT++ is ongoing. Currently a trial version using CUDA implementation for NVIDIA GPUs is in place for evaluation. It uses CUFFT and CUTENSOR libraries from NVIDIA for 1D FFTs and memory transposes, respectively. Ongoing work includes asynchronous transfers, which will partially overlap compute and communication time with the time of data transfer to/from the GPU. In addition, it will include wrapper functions that are blind to the underlying GPU programming semantics. A good candidate for this is HIP interface from AMD, which bridges AMD and NVIDIA GPU syntaxes.

D. Autotuning 3D transforms

Consider the case of 3D transform with 2D decomposition. Input and output arrays are defined according to grid object descriptors, as explained above. This includes both decomposition in processor space and a mapping to memory storage. The algorithm must go through three stages of 1D transform, with two or more inter-processor transposes and two or more local memory transposes. The course of the algorithm consists of an assembled sequence of basic P3DFFT++ building blocks (see above, namely 1D transforms combined with reordering, inter-processor transpose combined with reordering, and inter-processor transpose combined with 1D transform and reordering). These blocks have to be assembled in a way respecting the consistency of data layout between them, i.e. the output of one stage must be the same as the input for the next one.

In general, there is more than one combination of such assembly paths yielding the needed output (as was observed in [39]). Although some heuristics may be used to determine more optimal paths, in practice this has turned out to be unreliable and not sufficient, considering the variety of problems and architectures. Therefore P3DFFT++ will adopt an autotuning framework for measuring each of the best candidate assembly paths. Each path consists of a linked list of stages, and these paths are stored in a vector list. The autotuner goes through each path, measuring its execution time for a given number of repetitions. This is done in the planning call (constructor) of 3D transform class. The best-performing path is saved and used in the execution step.

E. Performance experiment

Here we compare performance of the latest CPU version of P3DFFT++ with P3DFFT v. 2.7.9. We used Stampede2 platform at TACC (using Intel's Knights Landing Nodes with 68 cores per node, of which 64 were used, and 100 Gb/sec Intel Omni-Path network with a fat tree topology employing six core switches). We have used a pair of real-to-complex and complex-to-real 3D FFT, which is relevant to many applications. We tested grid sizes 1024^3 and 2048^3, with 2D processor decomposition. Reported numbers in Fig. 3 are the timing for the forward/inverse transform pair, with the optimal processor grid dimensions for each case. We see comparable performance of P3DFFT++ and P3DFFT, with P3DFFT++ slightly winning in most cases. Note that these results were obtained with a version of the code without the autotuner, which can be expected to further improve performance. Also we note that performance on other platforms is quite similar in nature to these results.

V. USING P3DFFT++

In this section we demonstrate the use of P3DFFT++ for calculation of complex-to-complex 3D FFT. In this case, 2D decomposition is used. The input is in X-pencils, with the most basic memory ordering, while the output is in Z-pencil, with memory ordering such that Z dimension is stored with stride 1. We will use C++ code for this demonstration, but C and Fortran interfaces are also available, and example programs are provided in the distribution.

First, call P3DFFT++ initialization function setup once before any use, remembering to use p3dfft namespace:

```
using namespace p3dfft;
setup();
```

Next, set up input and output grid objects. For this we will need global grid dimensions, processor grid information and memory ordering map.

```
gdims[] = {nx,ny,nz};
mem_order1[] = {0,1,2};
pgrid1[] = {1, p1, p2};
proc_order[] = {0,1,2};
```

Construct the input grid object:

```
grid grid1(gdims,-1,
pgrid1,proc_order,mem_order1,MPI_COMM_WORLD);
```

Now define and construct the output grid object:

```
mem_order2[] = {2,1,0};
pgrid2[] = {p1,p2,1};
grid grid2(gdims, -1,pgrid2,proc_order,
mem_order2,MPI_COMM_WORLD);
```

Now define which type of 3D transform we want. In this case, all three dimensions will have complex-to-complex forward FFT in double precision:

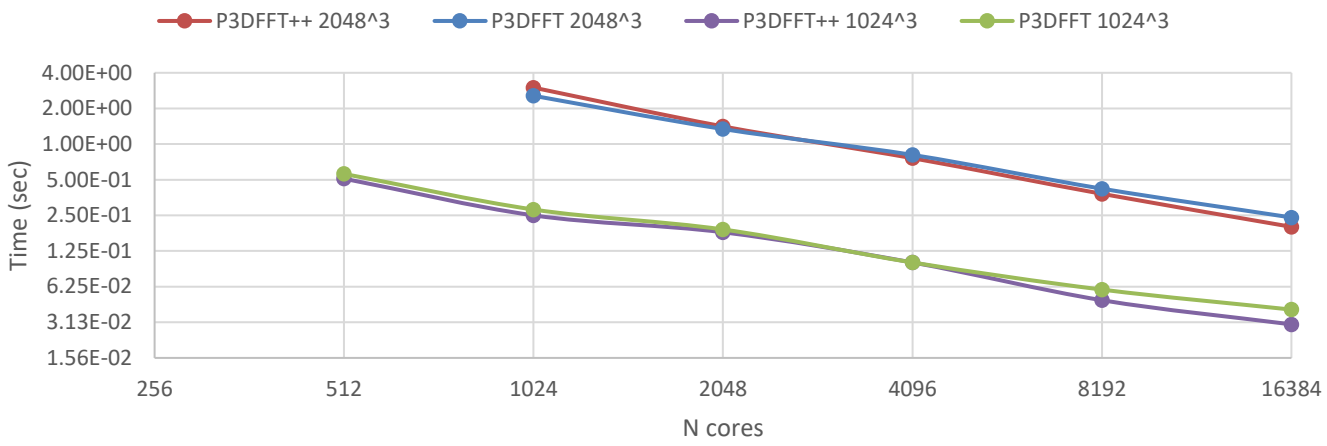

Figure 3. Performance comparison of P3DFFT++ and P3DFFT v. 2.7.9 on Stampede2 at TACC. Reported on the vertical axis is timing for a forward-inverse pair of real-to-complex/complex-to-real 3D FFT in seconds.

```
int type_ids[3] = {CFFT_FORWARD_D,
CFFT_FORWARD_D,CFFT_FORWARD_D};
trans_type3D type_cft_forward(type_ids);
```

Now find local dimensions of the input array and allocate space for it (note that mem_order mapping is used to translate from logical to physical storage indices):

```
int sdims1[3];
for(i=0;i<3;i++)
   sdims1[mem_order1[i]] = grid1.ldims[i];
int size1 = sdims1[0]*sdims1[1]*sdims1[2];
complex_double *IN=new
complex_double[size1];
```

Now do the same for output array:

```
int sdims2[3];
for(i=0;i<3;i++)
   sdims2[mem_order2[i]] = grid2.ldims[i];
int size2 = sdims2[0]*sdims2[1]*sdims2[2];
complex_double *OUT=new
complex_double[size2];
```

Next we construct the 3D transform, including planning and finding the best execution path through autotuning, as described above.

```
transform3D<complex_double,complex_double>
trans_f(grid1,grid2,&type_cft_forward);
```

Now the input can be initialized. Then we are ready to execute the transform, as many times as necessary.

```
for(i=0;i < nRep;i++) {
…   trans_f.exec(IN,OUT);   …   }
```

After P3DFFT++ is done, call cleanup() to deallocate temporary variables P3DFFT++ uses:

```
cleanup();
```

Using the same testing framework, many other kinds of transforms could have been defined, such as real-to-complex, cosine, sine, or a user-defined transform, in any reasonable combination for three dimensions. Also various alternative data layout options are possible simply by changing pgrid, proc_ordering and mem_order. More details can be found in the user guide and tutorial at `http://www.p3dfft.net`.

VI. Conclusions and Future Work

We have provided a motivation for a new adaptable software framework for multidimensional FFTs and other spectral transforms. We have listed the desirable characteristics of such framework, such as adaptability in terms of problem scope and architecture features, extending far beyond the existing multidimensional FFT libraries. This list formed the basis for creation of an open source P3DFFT++ library package. We have provided an overview of its design choices, discussed performance features and demonstrated using the package for a common 3D FFT case. P3DFFT++ (available at http://www.p3dfft.net) is written in C++ with interfaces for C and Fortran. It has been documented and tested on a variety of problems. At present, the functionality of P3DFFT++ includes most features present in P3DFFT and other comparable libraries, while far surpassing them in terms of the data options. Certain advanced features are still being developed and tested.

Performance considerations are of primary importance in this discussion. At this time P3DFFT++ performance is comparable to that of P3DFFT for the cases we have tested. Ongoing work includes integrating more performance-improvement features, such as asynchronous GPU operations, pruned transforms and overlap of communication with computation, in order to make the package practical for Exascale platforms. 4D transforms and 3D decomposition are features of interest to the community and will also be incorporated into the package.

ACKNOWLEDGMENT

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. In particular, it has used Comet platform at San Diego Supercomputer Center/UCSD, and Stampede2 platform at TACC/U. Texas at Austin.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

This work was supported by U.S. NSF grant OAC-1835885.

REFERENCES

[1] K. Czechowski, C. Battaglino, C. McClanahan, K. Iyer, P.K. Yeung and R. Vuduc, "On the communication complexity of 3D FFTs and its implications for exascale", Proceedings of the 26th ACM international conference on Supercomputing, pp. 205-214, June 2012.

[2] C. McClanahan, K. Czechowski, C. Battaglino, K. Iyer, P.K. Yeung and R. Vuduc, "Prospects for scalable 3D FFTs on heterogeneous exascale systems", ACMIEEE conference on supercomputing, SC. 2011

[3] H. Gahvari, and W. Gropp, "An introductory exascale feasibility study for FFTs and multigrid", IEEE International Symposium on Parallel & Distributed Processing (IPDPS), pp.1-9, April 2010.

[4] D. Pekurovsky, "P3DFFT: A framework for parallel computations of Fourier transforms in three dimensions", SIAM Journal on Scientific Computing, 34(4), pp.C192-C209, 2012.

[5] N. Li, and S. Laizet, "2decomp & fft-a highly scalable 2d decomposition library and fft interface", Cray User Group 2010 conference, pp. 1-13, May 2010.

[6] D. Takahashi, "An implementation of parallel 3-D FFT with 2-D decomposition on a massively parallel cluster of multi-core processors", International Conference on Parallel Processing and Applied Mathematics, pp. 606-614, Springer, Berlin, Heidelberg, September 2009.

[7] A. Gholami, J. Hill, D. Malhotra and G. Biros, "AccFFT: A library for distributed-memory FFT on CPU and GPU architectures", arXiv preprint arXiv:1506.07933, 2015, unpublished.

[8] T.V.T. Duy and T. Ozaki, "Hybrid and 4-D FFT implementations of an open-source parallel FFT package OpenFFT", The Journal of Supercomputing, 72(2), pp.391-416, 2016.

[9] M. Pippig, "An efficient and flexible parallel FFT implementation based on FFTW", Competence in High Performance Computing, pp. 125-134, Springer, Berlin, Heidelberg, 2011.

[10] J.H. Göbbert, H. Iliev, C. Ansorge and H. Pitsch, H., "Overlapping of Communication and Computation in nb3dfft for 3D Fast Fourier Transformations", In Jülich Aachen Research Alliance (JARA) High-Performance Computing Symposium, pp. 151-159, Springer, Cham., September 2016

[11] S. Plimpton, R. Pollock and M. Stevens, "Particle-Mesh Ewald and rRESPA for Parallel Molecular Dynamics Simulations", PPSC, March 1997.

[12] D. Donzis, P.K. Yeung and K.R. Sreenivasan, "Dissipation and enstrophy in isotropic turbulence: resolution effects and scaling in direct numerical simulations", Physics of Fluids, 20(4), p.045108, 2008.

[13] H. Homann, O. Kamps, R. Friedrich and R. Grauer, "Bridging from Eulerian to Lagrangian statistics in 3D hydro-and magnetohydrodynamic turbulent flows", New Journal of Physics, 11(7), p.073020, 2009.

[14] S. Laizet, E. Lamballais and J.C. Vassilicos, "A numerical strategy to combine high-order schemes, complex geometry and parallel computing for high resolution DNS of fractal generated turbulence", Computers & Fluids, 39(3), pp.471-484, 2010.

[15] L. Thais, A.E. Tejada-Martínez, T.B. Gatski, G. Mompean and H. Naji, "Direct and Large Eddy Numerical Simulations of Turbulent Viscoelastic Drag Reduction", Wall Turbulence: Understanding and Modeling, pp. 421-428, Springer, Dordrecht, 2011.

[16] P.K. Yeung and C.A. Moseley, "A message-passing, distributed memory parallel algorithm for direct numerical simulation of turbulence with particle tracking", Parallel Computational Fluid Dynamics 1995 (pp. 473-480), 1996.

[17] T. Engels, D. Kolomenskiy, K. Schneider, F.O. Lehmann and J. Sesterhenn, "Bumblebee flight in heavy turbulence", Physical review letters, 116(2), p.028103, 2016.

[18] A. Beresnyak, "Spectra of strong magnetohydrodynamic turbulence from high resolution simulations", The Astrophysical Journal Letters, 784(2), p.L20, 2014.

[19] S. Lange and F. Spanier, "Evolution of plasma turbulence excited with particle beams", Astronomy & Astrophysics, 546, p.A51, 2012.

[20] L. Arnold, C. Beetz, J. Dreher, H. Homann, C. Schwarz and R. Grauer, "Massively Parallel Simulations of Solar Flares and Plasma Turbulence", Parallel Computing: Architectures, Algorithms and Applications, Bd, 15, pp.467-474, 2008.

[21] N. Peters, L. Wang, J.P. Mellado, J.H. Gobbert, M. Gauding, P. Schafer and M. Gampert, "Geometrical properties of small scale turbulence", Proceedings of the John von Neumann Institute for Computing NIC Symposium, Juelich, Germany (pp. 365-371), February 2010.

[22] J. Schumache and M. Pütz, "Turbulence in Laterally Extended Systems", PARCO, pp. 585-592, 2007.

[23] P.J. Ireland, T. Vaithianathan, P.S. Sukheswalla, B. Ray, and L.R. Collins, "Highly parallel particle-laden flow solver for turbulence research", Computers & Fluids, 76, pp.170-177, 2013.

[24] P. Fede and O. Simonin, "Numerical study of the subgrid fluid turbulence effects on the statistics of heavy colliding particles", Physics of Fluids, 18(4), p.045103, 2006.

[25] S. Banerjee, A.G. Kritsuk, "Energy transfer in compressible magnetohydrodynamic turbulence for isothermal self-gravitating fluids", Phys. Rev. E, v.97, no. 2, p. 023107, 2018.

[26] J. Bodart, L. Joly and J.B. Cazalbou, "Large scale simulation of turbulence using a hybrid spectral/finite difference solver", Parallel Computational Fluid Dynamics: Recent Advances and Future Directions, pp.473-482, 2009.

[27] M. Frigo and S.G. Johnson, "The design and implementation of FFTW3", Proceedings of the IEEE, 93(2), pp.216-231, 2005.

[28] E.J. Bylaska et al, "Transitioning NWChem to the Next Generation of Manycore Machines", Exascale Scientific Applications: Scalability and Performance Portability, ch. 8., 2017.

[29] A. Canning, J. Shalf, N. J. Wright, S. Anderson and M. Gajbe, "A hybrid MPI/OpenMP 3d FFT for plane wave first-principles materials science codes", Proceedings of the International Conference on Scientific Computing (CSC), p. 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2012.

[30] M. Gajbe, A. Canning, L.W. Wang, J. Shalf, H. Wasserman and R. Vuduc, "Auto-tuning distributed-memory 3-dimensional fast Fourier transforms on the Cray XT4", Proc. Cray User's Group (CUG) Meeting,May 2009.

[31] M. Lee, N. Malaya and R.D. Moser, "Petascale direct numerical simulation of turbulent channel flow on up to 786k cores", Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 61, September 2013.

[32] S. Song and J.K. Hollingsworth, J.K., "Computation–communication overlap and parameter auto-tuning for scalable parallel 3-D FFT", Journal of Computational Science, 14, pp.38-50, 2016.

[33] J. Jung, C. Kobayashi, T. Imamura and Y. Sugita, "Parallel implementation of 3D FFT with volumetric decomposition schemes for efficient molecular dynamics simulations", Computer Physics Communications, 200, pp.57-65, 2016.

[34] A.G. Chatterjee, M.K. Verma, A. Kumar, R. Samtaney, B. Hadri, B. and R. Khurram, "Scaling of a Fast Fourier Transform and a pseudo-spectral

fluid solver up to 196608 cores", Journal of Parallel and Distributed Computing, 113, pp.77-91, 2018.

[35] K. Kandalla, H. Subramoni, K. Tomko, D. Pekurovsky, S. Sur and D.K. Panda, "High-performance and scalable non-blocking all-to-all with collective offload on InfiniBand clusters: a study with parallel 3D FFT. Computer Science-Research and Development, 26(3-4), p.237, 2011.

[36] [35] K. Kandalla, H. Subramoni, K. Tomko, D. Pekurovsky and D.K. Panda, "A Novel functional partitioning approach to design high-performance MPI-3 non-blocking alltoallv collective on multi-core systems", Parallel Processing (ICPP), 2013 42nd International Conference on (pp. 611-620). IEEE, October 2013.

[37] H. Subramoni, A.A. Awan, K. Hamidouche, D. Pekurovsky, A. Venkatesh, S. Chakraborty, K. Tomko and D.K. Panda, "Designing non-blocking personalized collectives with near perfect overlap for RDMA-enabled clusters", International Conference on High Performance Computing (pp. 434-453). Springer, Cham, July 2015.

[38] D. Pekurovsky, A. Venkatesh, S. Chakraborty, K. Tomko and D.K. Panda, "Designing Non-blocking Personalized Collectives with Near Perfect Overlap for RDMA-Enabled Clusters", High Performance Computing: 30th International Conference, ISC High Performance, Frankfurt, Germany, Proceedings, Vol. 9137, p. 434, Springer, July 2015.

[39] T.V.T. Duy and T. Ozaki, "A decomposition method with minimum communication amount for parallelization of multi-dimensional FFTs", Computer Physics Communications, 185(1), pp.153-164, 2014.