2DECOMP&FFT - A Highly Scalable 2D Decomposition Library and FFT Interface

Ning Li, *the Numerical Algorithms Group (NAG) and*
Sylvain Laizet, *Imperial College London*

ABSTRACT: *As part of a HECToR distributed CSE support project, a general-purpose 2D decomposition (also known as 'pencil' or 'drawer' decomposition) communication library has been developed. This Fortran library provides a powerful and flexible framework to build applications based on 3D Cartesian data structures and spatially implicit numerical schemes (such as the compact finite difference method or spectral method). The library also supports shared-memory architecture which becomes increasingly popular. A user-friendly FFT interface has been built on top of the communication library to perform distributed three-dimensional FFTs. Both the decomposition library and the FFT interface scale well to tens of thousands of cores on Cray XT systems. The library has been applied to Incompact3D, a CFD application performing large-scale Direct Numerical Simulations of turbulence, enabling exciting scientific studies to be conducted.*

KEYWORDS: 2D decomposition, distributed FFT, Poisson solver, shared-memory

1. Introduction

The Computational Science and Engineering (CSE) support for Cray XT system HECToR, the UK's national supercomputing facility, is provided by a team of HPC experts at NAG. A critical part of the service is the distributed CSE (dCSE) programme, which delivers dedicated software engineering support to research groups to improve their scientific software packages.

The work presented in this paper is part of a dCSE project to modernise Incompact3D, a Computational Fluid Dynamics (CFD) application for Direct and Large-eddy simulation of turbulence. This work is in collaboration with the Turbulence, Mixing and Flow Control group at Imperial College London. The main objective is to update Incompact3D's communication framework, in particular, to implement a new domain decomposition strategy to improve its scalability on modern supercomputers. After some preliminary work, it became apparent that the outcome of this project can benefit many other applications (such as many of the CFD applications that used by members of the UK Turbulence Consortium) and a decision was made to pack the reusable software components into a library.

2DECOMP&FFT is a Fortran library to conduct two major tasks. First of all it implements a 2D domain decomposition algorithm (also known as 'pencil' or 'drawer' decomposition, among other names) for applications using 3D Cartesian data structures. On top of that it also provides a simple and efficient FFT interface to perform three-dimensional FFTs in parallel. The library is optimised for large-scale computations on supercomputers and scales well to tens of thousands of processors on both Cray and non-Cray systems. It relies on MPI but provides a user-friendly programming interface that hides communication details from application developers.

2. 2D Domain Decomposition

For a large category of applications solving differential equations on three-dimensional Cartesian meshes, their numerical algorithms are inherently implicit. For example, a compact finite difference scheme often results in solving a tridiagonal linear system when evaluating spatial derivatives or interpolations; a spectral code often involves performing Fast Fourier Transforms along global mesh lines.

There are two approaches to performing such computations on distributed-memory systems. One can either develop distributed algorithms (such as parallel tridiagonal solver or parallel FFT working on distributed data), or one can dynamically redistribute (transpose) data among processors in order to apply serial algorithms in local memory. The second approach is often preferred due to its simplicity: any existing serial algorithms (hopefully efficiently implemented already for a single CPU) remain unchanged; porting serial code can be quite straight-forward as much of the original code logic still holds, and the only major addition is the data transposition procedures.

In the past, many applications implemented the above idea using 1D domain decomposition (also known as 'slab' decomposition). In Fig. 1, a 3D domain is arbitrarily chosen to be decomposed in Y and X directions. It can be seen that in state (a), any computations in the X-Z planes can be done in local memories while data along a Y mesh-line is distributed. When it is necessary to calculate along Y mesh-lines (say to evaluate Y-derivatives, or to perform 1D FFTs along Y), one can redistribute the data to state (b), in which any computation in Y becomes 'local'. Swapping between state (a) and (b) can be achieved using standard MPI_ALLTOALL(V) library.

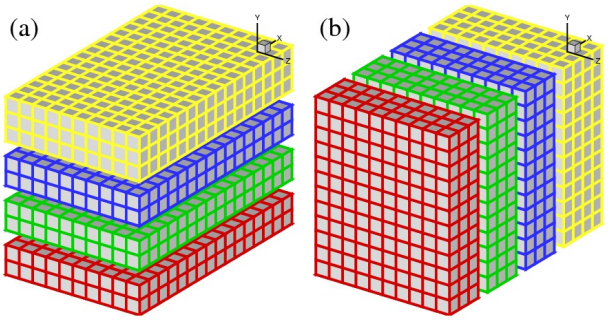

Figure 1: 1D domain decomposition using 4 processors: (a) decomposed in Y; (b) decomposed in X.

A 1D decomposition, while quite simple, has some limitations, especially for large-scale simulations. Given a cubic mesh of size N^3, one obvious constraint is that the maximum number of processors N_{proc} that can be used in a 1D decomposition is N as each slab has to contain at least one plane. For a cubic mesh with 1 billion points, the constraint is $N_{proc}<1000$. This is a serious limitation as most supercomputers today have tens of thousands of cores and some have more than 100,000.[1] Large applications are also likely to hit the memory limit when each processor handles too much workload.

As a result, a 2D decomposition strategy, a natural extension to the 1D idea, becomes practically relevant now with many applications lagging behind. Fig. 2 shows that the same 3D domain as in Fig. 1 can be partitioned in two dimensions. From now on, states (a), (b) and (c) will be referred to as X-pencil, Y-pencil and Z-pencil arrangements, respectively. While a 1D decomposition algorithm swaps between two states, in a 2D decomposition one needs to traverse three different states using four global transpositions ((a)→(b)→(c)→(b)→(a)) to complete a cycle.

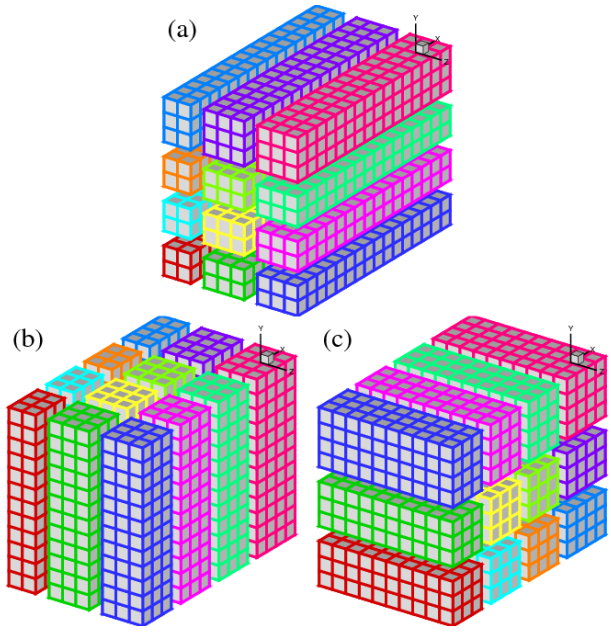

Figure 2: 2D domain decomposition using a 4 by 3 processor grid.

Again MPI_ALLTOALL(V) can be used to realise the transpositions. However it is significantly more complex than the 1D case. There are two separate communicator groups. For a $P_{row} \times P_{col}$ 2D processor grid: P_{row} groups of P_{col} processors need to exchange data among themselves for (a)↔(b); P_{col} groups of P_{row} processors need to exchange data among themselves for (b)↔(c). It is also worth mentioning that the implementation of the communication routines are very sensitive to the orientations of pencils and their associated memory patterns. The packing and unpacking of memory buffers for the MPI library needs to be handled with great

1 The November 2009 TOP500 list shows that all top 30 systems have more than 10,000 cores; 88 of the top 100 systems have more than 5,000 cores; the largest IBM BlueGene has 294,912 cores.

care. These software engineering topics are almost certainly irrelevant to the scientific researches conducted by the applications. This is one of the motivations to create the 2DECOMP&FFT library - to handle these technical issues properly and to hide all communication details from the application developers who can concentrate on their scientific studies.

3. Software Installation

The software is under active development. The source code and documentations are available from the author upon request. There is no special installation procedure as this software is meant to be built from source together with the application code. Simply decompress the package in any location and it is ready to be compiled.

A Fortran 95 compatible compiler is the minimum requirement. In addition, the code also relies on allocatable enhancement features (defined in ISO TR 15581) and Cray pointers (for the shared-memory implementation), which are almost universally supported by modern Fortran compilers.

The library is designed to be very user-friendly. Most features are packed in a black box. Users are able to turn on/off software options from the main Makefile provided. For example, there are options to:

- switch between single and double precision simulations.
- opt to use MPI_ALLTOALL calls instead of MPI_ALLTOALLV for data transpositions while data is evenly distributed. All communication code in this library is written to support general uneven data distribution so MPI_ALLTOALLV is the default mode.
- turn on the shared-memory implementation of the communication code (see Section 5 for details).

These features are supported through preprocessing directives built in the Fortran code. Also supplied are platform dependent Makefiles to support different hardware and compiler combinations, which assist end users to port the library.

For applications using the distributed FFT interface (as will be discussed in Section 6), an FFT engine may be selected in the main Makefile. The multi-dimensional FFT routines delegate 1D transforms to a 3rd-party library. The default FFT engine is called 'generic', which does not rely on any external 3rd-party libraries. Also released are implementations using FFTW3, ACML, FFTPACK5, Intel MKL and IBM ESSL.

A few test applications are bundled with the package which not only validate the code, but also demonstrate the proper use of the library.

4. Using the Decomposition Library

2DECOMP&FFT library functions are provided in several Fortran modules. A base module contains 2D decomposition information and data transposition routines. An FFT module is built on top to provide three dimensional distributed FFT functions. There are also other utility functions such as a MPI-IO module for proper parallel IO. The decomposition library will be discussed in this section while the FFT interface will be covered later in Section 6. As only the key routines are discussed here, readers are referred to the user guild distributed with the software for full coverage of the APIs.

4.1 Basic programming interface

First of all a Fortran module needs to be used:

```
use decomp_2d
```

A set of global variables[2] are defined in the decomposition library for applications to define their data structures. These include a KIND variable to properly define data types in applications (single vs. double precision); a few MPI types should the applications need to call MPI library routines directly; a few MPI variables such as the current MPI rank and the size of the MPI communicator; and most importantly, variables to describe the sub-domain held by the current processor – its starting index, ending index and size for all three pencil orientations. The preferred approach is for applications to define their data structures using allocatable arrays based on these decomposition information. The starting/ending indices, based on the global coordinate of the whole computation domain, may be very useful in many situations (for example to extract a 2D plane from the 3D domain).

All the global variables are initialised by an initialisation routine that needs to be called at the beginning of the application:

2 Defining public global variables in a Fortran module is not always a good practice. However such design is intentional in this library to simplify the building of applications.

```
call decomp_2d_init(nx,ny,nz, &
     P_row,P_col)
```

The global size of the 3D domain and a 2D processor grid need to be supplied. The global variables then should be treated as read-only throughout the lifetime of the application. The library also contains a set of communication routines that actually perform the data transpositions. As mentioned in Section 2, one needs to perform four global transpositions to go through all three pencil orientations. Correspondingly, the library provides four communication subroutines in the form of:

```
call transpose_x_to_y(in, out)
call transpose_y_to_z(in, out)
call transpose_z_to_y(in, out)
call transpose_y_to_x(in, out)
```

where input array *in* and output *out* should have been defined for the correct pencil orientations in the calling program. Note that the library is written using Fortran's generic interface so different data types are supported without user intervention. For many applications both arrays simply contains real numbers. But *in* and *out* can be complex arrays for FFT-type of applications.

Although it is not necessary for application developers to understand the internal logic of these transposition routines, it is important to know that they can be expensive, especially when running on large number of processors. So applications should minimize the number of calls to them, even sometimes at the cost of duplicating computations.

The library is completed by a finalisation routine to properly clean up the memory:

```
call decomp_2d_finalize
```

4.2 Advanced programming interface

While the basic decomposition API is very user-friendly, there may be situations in which applications need to handle more complex data structures. For example, to implement a 3D real-to-complex FFT, applications need to store both the real input (of global size nx*ny*nz) and the complex output (of smaller global size - such as (nx/2+1)*ny*nz - where roughly half the output needs not to be stored due to conjugate symmetry). Different global sizes need to be distributed as 2D pencils and their distribution parameters are very different. Similar situations arise from many CFD applications in which main physical quantities are stored on a staggered mesh, resulting in slightly different global sizes.

In order to handle the above situations, an advanced programming interface is provided. Applications may define a decomposition object (an instance of a Fortran derived data type), initialise it using any global size, and then access to the decomposition information of that particular global size afterwards. Such decomposition information can then be used to define data structures for the distributed data. When global transpositions are required, this decomposition object may be passed into the transposition routines in the basic interface as the third *optional* parameter so that data of this particular global size (rather than the default size nx*ny*nz) is transposed. This is implemented using Fortran generic interface so that users of the basic interface only can safely ignore the complexity here.

5. Shared-memory Implementation

Many modern supercomputers use multi-core processors and cores on same node often have shared local memory. For example, this library's main development platform HECToR is a Cray XT4 system using quad-core processors, sharing 8GB of memory.

For the ALLTOALL type of communication in which each MPI rank has to send/receive messages to/from all other MPI ranks, one can imagine that traffic from cores on the same physical node compete for their network interface. One possible solution is to create shared send/recv buffers on each SMP node. Then only leaders of the nodes participate MPI_ALLTOALL(V), resulting in fewer but larger messages passing within the system, hopefully improving the performance of communication.

The actual shared-memory program was based on code supplied by David Tanqueray of Cray Inc. who initially applied this idea to several molecular dynamics applications. The shared-memory code uses the Unix System V shared memory API which is widely available. Similar to the standard library, the shared-memory library has been implemented as a black box. So users can simply set a pre-processor flag when compiling the library to switch on the function.

Note that this shared-memory implementation is not automatically portable while moving to a new system. There is a piece of low-level code which accesses system-dependent information to gather vital information for the

shared-memory code (such as which MPI rank belongs to which node and who else shares the node). On Cray XT systems, this is done by checking the */proc* file system of the computing nodes. This function needs to be rewritten for different hardware. However, one can reasonably expect that such function can be provided by hardware suppliers or system administrators familiar with the new hardware. A system-independent version of the shared-memory code will be implemented in the future.

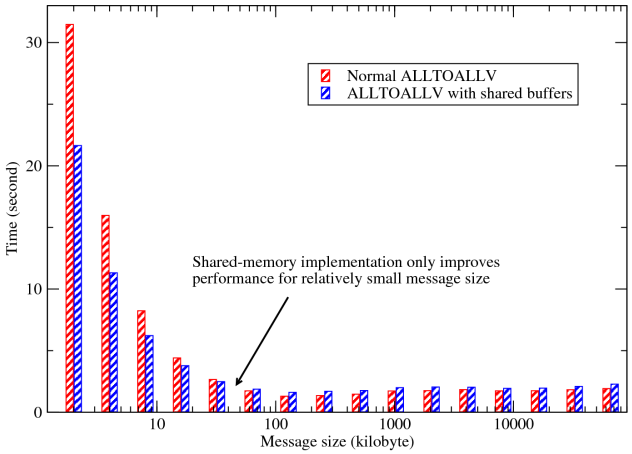

Figure 3: Shared-memory code performance

Finally, the performance of the shared-memory code is shown in Fig. 3. The data was collected on HECToR phase 2a hardware from a series of simulations using 256 MPI ranks over a range of problem sizes. When the problem size is small (so is the message size), the transposition routines were called more times so that the total amount of data moving within the system remains a constant. Fig. 3 shows that when the problem size is small, the overhead of setting up communications is very high and the shared-memory code can improve communication efficiency by up to 30%. As the problem size increases, the benefit of using shared-memory code becomes smaller. In fact for large message size (> 32Kb in this example), the shared-memory code is actually slower due to the extra memory copying operations required to assemble the shared buffer[3]. However, do remember this test was performed on a quad-core system. HECToR is scheduled to be upgraded to 24-core system in 2010 and the general trend across major supercomputing sites is to have more cores per node. Do expect the shared-memory code to play a more significant role in the very near future.

3 Sometimes packing and unpacking of buffers are not required for ordinary MPI_ALLTOALL(V) operation due to the memory pattern of 3D arrays.

6. Fast Fourier Transform

Having developed the decomposition library, one can apply it to many applications using 3D Cartesian data structures, in particular those whose algorithms can be split into a sequence of one dimensional operations. It is well known that a multiple-dimensional FFT is mathematically equivalent to a series of 1D FFTs. As a demonstration, a distributed FFT library is implemented using this framework.

6.1 Review of parallel FFT libraries

Being an extremely useful research tool, FFT software packages are available everywhere. Almost all hardware vendors produce their own FFT products tuned for particular hardware. There are also many open-source codes implementing various algorithms. Unfortunately, when working on large-scale distributed systems, the options are very limited and many famous scientific packages have to implement their customised FFT libraries. In this section, only those libraries closely related to this project, especially those available on Cray XT systems are reviewed.

FFTW[2] is one of the most popular FFT packages available and is officially supported by Cray. It is open-source, supporting arbitrary input size, portable and delivers good performance due to its self-tuning design (planning before execution). There are two major versions of FFTW. Version 2.x actually has a reliable MPI interface to transform distributed data. However, it internally uses a 1D (slab) decomposition which, as discussed earlier, limits the scalability of large applications. Its serial performance is also inferior to that of version 3.x, which has a redesign that better supports the SIMD instructions on modern CPUs. Unfortunately, the MPI interface of version 3.x is in unstable alpha stage and has been so for many years. So there is no reliable way to compute multi-dimensional FFTs in parallel.

Cray has its own CRAFFT (CRay Adaptive FFT), for example, as part of xt-libsci library on XT systems. It provides a simplified interface to delegate computations to other FFT kernels (including FFTW). Being 'adaptive' means that it can dynamically select the fastest FFT kernels available on a system. However, it only supports very limited distributed routines (only 2D/3D complex-to-complex transforms are supported as of version xt-

libsci/10.4.0) and those routines are based on an evenly-distributed 1D decomposition.

As XT systems are based on AMD CPUs, obviously one can use the FFT routines provided by AMD Core Math Library (ACML) which is specially tuned for the AMD processors. AMD does provide a multi-threading version of the library but there is no distributed support.

There are several open-source packages available which implement 2D-decomposition based distributed FFTs. For example, Plimpton's parallel FFT package[6] provides a set of C routines to perform 2d and 3d complex-to-complex FFTs together with very flexible data remapping routines for data transpositions. The communications are implemented using MPI_SEND and MPI_IRECV.

Takahashi's FFTE package[1] in Fortran contains both serial and distributed version of complex-to-complex FFT routines. It supports transform lengths with small prime factors only and uses MPI_ALLTOALL to transpose evenly distributed data. There is no user callable communication routines.

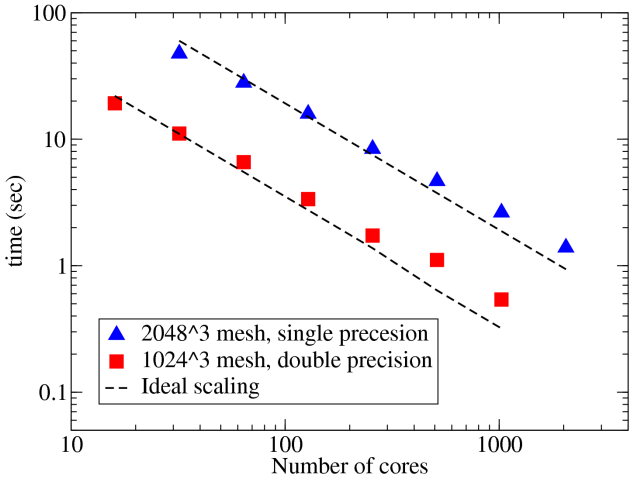

Figure 4: P3DFFT scaling on HECToR.

Finally there is the well-known open-source package P3DFFT[7], which has been widely adopted by scientists doing large-scale simulations in many research areas such as cutting-edge turbulence studies. The P3DFFT project was initiated at San Diego Supercomputer Center by Dmitry Pekurovsky. It is quite efficient, as can be seen in Fig. 4 showing the parallel scaling of its sample application on HECToR's phase 1 dual-core hardware.

However, initial attempts to adapt this library to use in the present project proved to be impractical. P3DFFT is targeting purely spectral applications[4], with its public API performing only real-to-complex and complex-to-real FFTs and its communication routines handling complex data type. There are FFT-specific features (such as the padding of real input for in-place transforms) built in its logic and data structure which are not relevant to general applications. The CFD application being refactored in this project, based on a compact finite difference schemes, require a more general domain decomposition library to handle its global data transpositions. This motivated the authors to develop the 2DECOMP&FFT library from scratch and provide a two-layer design – with a general-purpose 2D decomposition library as the foundation and a distributed FFT library built on the top. Taking advantage of the advanced programming interface discussed in section 4.2, the FFT library is also a general-purpose one, supporting both complex-to-complex and real-to-complex/complex-to-real transforms.

6.2 FFT API

Like the decomposition library, the FFT API is also designed to have user-friendliness as one of the top priorities. First of all, one additional Fortran module has to be referred to in order to use the FFT interface:

```
use decomp_2d_fft
```

Then one needs to initialise the FFT interface by:

```
call decomp_2d_fft_init
```

The initialisation routine handles planing for the underlying FFT engine (if supported) and defines globally work spaces for the FFTs. By default, it is assumed that physical-space data is stored in X-pencil. The spectral-space data is then stored in transposed Z-pencil format after the forward FFT. To give applications more flexibility, the library also supports the opposite direction, if an optional parameter is passed to the initialisation routine. Physical-space data in Y-pencil is not an option as it would require additional expensive transpositions which does not make economical sense.

The main functionality this FFT package provides is to perform 3D FFTs where the distributed input data is stored in ordinary ijk-ordered 3D arrays across processors. For complex-to-complex (c2c) FFTs, the user interface is:

4 Fourier-based spectral applications only require a distributed FFT library and no other explicit data transposition.

```
call decomp_2d_fft_3d(in, out,    &
     direction)
```

where direction can be either forward or backward. The input array *in* and output array *out* are both distributed 3D complex arrays and have to be either in a X-pencil/Z-pencil combination or vice versa, depending on the direction of FFT and how the FFT interface is initialised.

While the c2c interface is already in the simplest possible form, for many applications involving real quantities, the 3D FFT interface can be used in a more compact form:

```
call decomp_2d_fft_3d(in, out)
```

Here if *in* is a real array and *out* a complex array, then a forward FFT is implied. Similarly a backward FFT is computed if *in* is a complex array and *out* a real array.

When real input is involved, the corresponding complex output satisfies so-called `Hermitian redundancy' - i.e. some output values are complex conjugates of others. Taking advantage of this, FFT algorithms can normally compute r2c and c2r transforms twice as fast as c2c transforms while only using about half of the memory. Unfortunately, the price to pay is that application's data structures have to become slightly more complex. For a 3D real input data set of size nx×ny×nz, the complex output can be held in an array of size (nx/2+1)×ny×nz, with the first dimension being cut roughly in half (for Z-pencil input, the complex output is of size nx×ny×(nz/2+1) instead). 2DECOMP&FFT library uses the advanced programming interface discussed in section 4.2 to distribute both global arrays and hides as many details as possible from end users. One utility subroutine is provided to help applications properly allocate memory space to store the complex data when using the r2c/c2r interface. Please note that the complex output arrays obtained from X-pencil and Z-pencil input do not contain identical information. However, if `Hermitian redundancy' is taken into account, no physical information is lost and the real input can be fully recovered through the corresponding inverse FFT from either complex array.

Finally, to release the memory used by the FFT interface:

```
call decomp_2d_fft_finalize
```

It is possible to re-initialise the FFT interface in the same application at the later stage after it has been finalised, if this becomes necessary.

6.3 FFT engines

The distributed FFT interface only performs data management and communications. The actual computations – a lot of 1D FFTs – are delegated to a 3rd-party FFT library and are always performed using data in local memory.

Users have the freedom to choose from 6 different FFT engines supported by 2DECOMP&FFT: a generic algorithm, FFTW (version 3.x), ACML, FFTPACK (version 5), Intel MKL and IBM ESSL. The advantages and disadvantaged of these FFT libraries are summarised in Table 1.

Library	Open-source	Hardware tuned	Complete & general API[5]	Easy parallel coding[6]
generic	Y	N	N	Y
FFTW3	Y	Y	Y	N
ACML	N	Y	N	Y
fftpack	Y	N	N	Y
MKL	N	Y	Y	N
ESSL	N	Y	N	N

Table 1: Comparison of FFT libraries

The generic implementation is extended from an algorithm proposed by Glassman[3][9]. 'fftpack' is included because it is widely used by legacy applications and is quite relevant to the PDE solver discussed in Section 7. FFTW (version 3.x) is the most popular open-source packages. Others are all highly optimised vendor libraries with ACML specially tuned for AMD CPUs, MKL for Intel CPUs, and ESSL widely used on IBM Power-x based systems and BlueGenes.

All vendor libraries provide highly efficient APIs to computed multiple 1D FFTs in one call, a feature very useful in the parallel implementation. These all involve

5 ACML has much limited r2c and c2r support with its storage format inconvenient to use in the parallel library. ESSL assumes stride-1 storage in its r2c/c2r interface.

6 FFTW's planning, although very powerful, is not easy to use in the parallel implementation because it requires proper memory alignment which can not be guaranteed in Fortran. So one either plans every transform before execution (time consuming) or plans on globally defined data arrays (memory consuming). MKL has improperly designed API which does not directly accept multi-dimensional arrays as input/output.

specification of a *stride* parameter and a *distance* parameter to describe the memory pattern of the data structures. When functions are missing from the FFT engines (such as ESSL's lack of support to arbitrary transform length), they are implemented with the parallel code so that 2DECOMP&FFT can be universally applied.

7. Distributed Poisson Solver

The 2D decomposition library and the FFT interface are not only themselves useful in applications, but also serve as building blocks to develop higher-level libraries. This section discusses a general-purpose elliptic PDE solver, which applies a Fourier-based spectral method to solve Poisson's equations. This is particularly relevant to the current CFD application solving the incompressible Navier-Stokes system in which mass conservation is enforced by solving a pressure Poisson equation.

There are various numerical algorithms solving Poisson's equations, broadly classified into two categories: iterative solvers and direct solvers. The Multigrid approach is often considered most efficient iterative method while FFT-based solvers are the most efficient direct methods. In the context of parallel computing, FFT-based methods are often relatively easier to implement.

There are actually two types of FFT-based approaches to solve Poisson's equations. The first type, sometimes referred to as matrix decomposition, uses Fourier's method to treat the finite difference discretisation of Poisson's equations. Using ordinary central differencing, a finite difference discretisation of a 3D Poisson's equation results in a linear system with seven diagonal lines. To solve the system quickly, one can apply Fourier analysis in one dimension, reducing the system to a number of pentadiagonal systems. Fourier analysis in a second dimension further reduces the problem to many tridiagonal systems which can be solved efficiently. The mathematical formulations of such method, in particular the proper treatment of non-periodic boundary conditions and the applications on different mesh configurations (staggered vs. collocated mesh), were established in 1970's[13][14] and there are open-source software implementations available, such as FFTPACK[4], containing optimised FFT, fast sine and cosine transform routines, and FISHPACK[5], containing several Poisson solvers among other algorithms. This method actually fits in the present library framework very well – one can apply 1D FFTs and tridiagonal solver direction by direction in local memory, given that the data involved is properly redistributed using the transposition routines.

The numerical method adopted in this project is of a second type - a fully spectral treatment of the Poisson's equation, in order to directly take advantage of the 3D distributed FFT library developed. The algorithm involves the following steps:

- Pre-processing in physical space
- 3D forward FFT
- Pre-processing in spectral space
- Solving the Poisson's equation by a division of modified wave numbers
- Post-processing in spectral space
- 3D inverse FFT
- Post-processing in physical space

The forward and backward transforms are standard FFTs (even for data sets with non-periodic boundary conditions). Depending on different boundary conditions, some of the pre- or post-processing steps may be optional. Without giving any mathematical details, the pre- and post-processing involves operations which evaluate wave numbers and package the Discrete Cosine Transforms into a suitable form so that standard FFT routines can be applied. The Discrete Cosine Transforms are required to represent data with symmetric boundary conditions: $\partial p/\partial n=0$, exactly the boundary condition that describes the pressure boundary in CFD applications.

This pre- and post-processing can be either *local* – meaning that operations can be done regardless of the parallel distribution of data, or *global* – meaning that calculations are only possible when data sets involved are all local in memory (i.e. the operations have to be done in a particular pencil-orientation). Fortunately, for the global case, whenever the data required is not available, the global transposition routines provided by the base library can be used to redistribute the data. The number of global transpositions required is dependent on the boundary conditions, which are summarised in Table 2.

B.C.	# Global Transpositions	1024^3 128 cores	2048^3 1024 cores	4096^3 8192 cores
000	FFT	4.81	6.26	7.59
100	FFT+8	7.38	10.38	14.41
010	FFT+6	6.81	8.86	12.63
110	FFT+12	8.23	11.56	16.31
111		8.41	11.67	16.48

Table 2: Poisson solver performance

Here the boundary type 0 represents periodic boundary conditions and type 1 represents homogeneous Neumann (or symmetric) conditions. As can be seen, as many as 12 additional transpositions may be required for the pre- and post-processing while the 3D Fast Fourier Transforms themselves (one forward and backward pair) contains only 4 global transpositions. The number of communication calls appears to be high. However, because FFT is such as a computational intensive algorithm, the actual benchmark results on three large problem sizes (1024^3, 2048^3 and 4096^3) show that the communication cost is only a small proportion of the total runtime.

It is possible to extend this solver to additional boundary conditions which would involve discrete sine transforms and quarter-wave transforms.

8. Library Performance

8.1 FFT library performance

The performance of a distributed FFT library is determined by both the computation efficiency of the underlying 1D FFT algorithm and the communication efficiency of the data transposition algorithm. Table 3 shows the speed-up that this library can achieve over the serial runs using FFTW's 3D FFT interface. The times reported (in seconds) are for forward c2c transforms and all the transforms were planned using FFTW_ESTIMATE.

N^3	Serial FFTW		Distributed FFT		
	plan	execution	16-core	64-core	256-core
64^3	0.359	0.00509	0.00222	*	*
128^3	1.98	0.0525	0.0223	0.00576	0.00397
256^3	8.03	0.551	0.179	0.0505	0.0138
512^3	37.5	5.38	1.74	0.536	0.249
1024^3	#	#	-	4.59	1.27
2048^3	#	#	-	-	17.9

Table 3: Serial vs. distributed performance of the FFT interface

It can be seen that due to the communication cost, the absolute speed-up over the serial library isn't great (only about 20-40 times on 256 cores). However, the parallel library does allow much larger problems to be computed efficiently. In particular, for smaller core count (16 and 64), each time the problem size is increased by 8 times, the computing time increases by 8-10 times, following the trend of the underlying serial library very well.

Large-scale parallel scaling benchmarks of the FFT interface were done on HECToR and Jaguar, using problem size up to 8192^3 . The timing results presented here are the time spent to compute a pair of forward and backward transforms on a random signal, both c2c and r2c/c2r. The underlying FFT engine is version 4.3 of ACML FFT. In all cases, the original signals were recovered to machine accuracy after the backward transforms - a good validation for the library itself. Up to 16384 cores were used on HECToR and each case was repeated 3 times and the fastest results were recorded. On Jaguar, the world No. 1 system at the moment, fewer but larger tests were arranged using up to 131072 cores. Please note that runtime does vary a lot for communication intensive jobs on busy production systems.

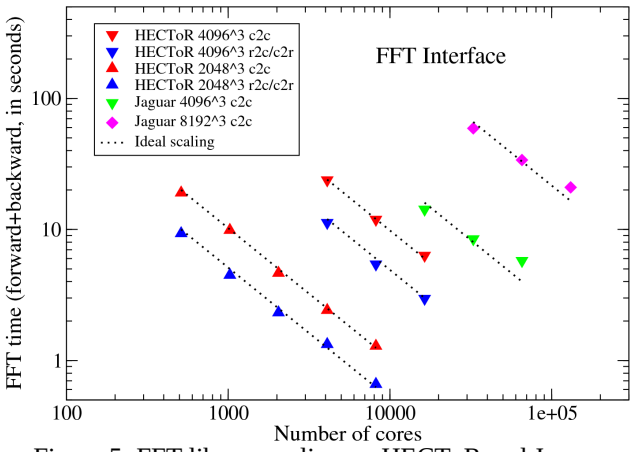

Figure 5: FFT library scaling on HECToR and Jaguar

It can be seen that the FFT interface scales almost perfectly on HECToR for all the tests done. As expected, the r2c/c2r transforms are nearly twice as fast as the corresponding c2c ones. On Jaguar, the scaling is less than perfect for larger core counts but the efficiency is still at a respectable 81% for the largest test. For one particular problem size, 16384-core job on the 4096^3 mesh, Jaguar took twice as much time to run. This is no surprise. While HECToR has quad-core processors at the moment, Jaguar has two 6-core chips built on each node. The problems set up for these benchmarks really prefer a power-of-2 core count to run efficiently as the communication network can be used in a balanced way. For example, a communicator with 4 members always sits

on the same physical chip on HECToR XT4 when using the system's default SMP-style rank placement. This is not the case on either Jaguar or the 24-core HECToR phase 2b XT6 system currently being built.

8.2 Practical advices

Application performance can be affected by many factors. For example, depending on the network hardware, the MPI library might prefer certain message size. Some runtime tuning through system environmental variables can often improve code performance. Application users are encouraged to run tests before engaging in any large-scale simulations. This section discusses some performance issues which are known to affect the behaviour of 2DECOMP&FFT.

The global transpositions and MPI_ALLTOALL type of communications are known to be very demanding for network bandwidth. So it is almost certain that large applications can run faster on multi-core chips if not using all the cores from each physical node. This may also improve the computational efficiency due to improved memory bandwidth. Unfortunately this is rarely practical on most supercomputers where the resource charging is on per-node basis. Shared-memory programming, as discussed in Section 5, may improve the situation.

Application users need to be aware that they have the freedom to choose the shape of the 2D processor grid $P_{row} \times P_{col}$ when using this library. Depending on the hardware, in particular the network layout, some processor grid option deliver much better performance than others. Application users are highly recommended to test this issue before running large simulations.

Fig. 6 shows the performance of a test application using 256 MPI ranks on HECToR. It can be seen that subject to constraint $\max(P_{row}, P_{col}) < \min(nx, ny, nz)$, $P_{row} << P_{col}$ is the best possible combination for this particular hardware (using Torus interconnection network). There are several technical reasons for such behaviour. First of all the hardware is equipped with quad-core processors (4-way SMP). When P_{row} is smaller than or equal to 4, half of the MPI_ALLTOALLV communications are done entirely within physical nodes which can be very fast. Second, as the communication library handles ijk-ordered arrays, larger nx/P_{row} leads to better use of system cache. Similar results were also reported by P3DFFT authors[7].

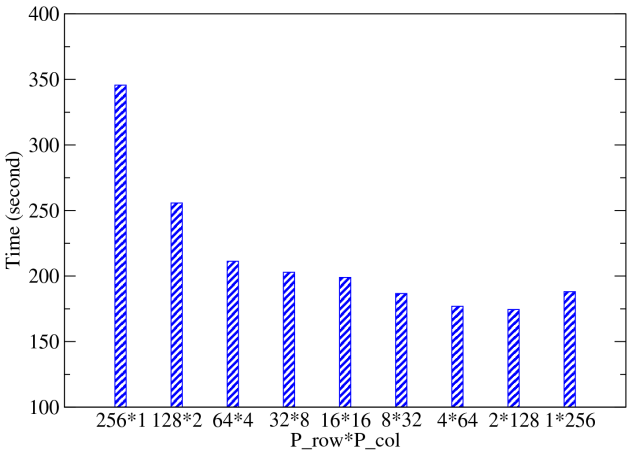

Figure 6: Communication library performance dependency on the shape of the processor grid.

Please note however, that this result is by no mean representative. In fact the behaviour is also highly dependent on the time-varying system workload and the size and shape of the global mesh, among other factors. An auto-tuning algorithm may be included in a future release to allow the best processor grid to be determined at runtime.

9. Application – Incompact3D

This project concerns the development of a unique research code **Incompact3D** to make the best use of the recent unprecedented developments in HPC technology, and to improve our understanding of fluid turbulence. The Turbulence, Mixing and Flow Control group at Imperial College London has been working on cutting-edge energy problems for nearly a decade. One very recent example of a new flow concept originating from this group concerns turbulence generated by multiscale/fractal objects (as shown in Fig. 7). This class of new flow concepts is offering possibilities for brand-new flow solutions useful in industrial mixers, silent air-brakes, new ventilation and combustion devices. Many wind tunnel measurements have been performed with impressive results[10][12]. To complement these experimental results, high-resolution simulations of multiscale generated flows are required in order to understand the underlying physics. These are very large-scale billion-mesh simulations that demand significant software development in order to use the supercomputing facilities available.

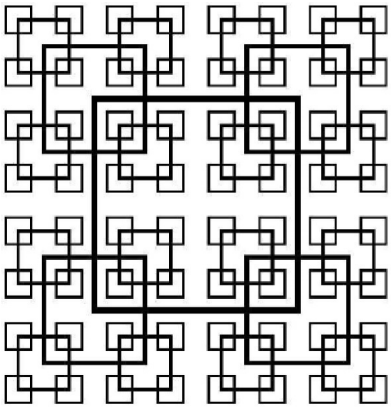

Figure 7: square-based fractal grid.

The old Incompact3D was parallelised using a 1D slab decomposition. This seriously limited its applications to large-scale simulations. For example, a typical simulation using a 2048*515*512 mesh can only use up to 512 cores. On HECToR, this translates to a runtime of 25 days (wall-clock time) or 50 12-hour sessions, excluding queueing time. Obviously at this rate it is impossible to conduct productive scientific studies.

The new Incompact3D has been completely rewritten using the 2DECOMP&FFT library, making it scalable to tens of thousands of cores. The finite difference part of the code evaluates the convection and diffusion terms of the Navier-Stokes equation using 6-order compact schemes. This results in solving tridiagonal systems along global mesh lines when evaluating spatial derivatives, applying spatial filters and doing spatial interpolations. The base decomposition library is used to move the data, while the advanced user interface is used extensively to transpose quantities of slightly different global sizes, a consequence of using a staggered mesh for the pressure field. On the other hand, the pressure-Poisson equation is solved using a spectral solver similar to the Poisson solver discussed in Section 7 but with customised formulation for modified wave numbers in order to have the equivalence between physical and spectral operators which are suitable for the numerical framework and mesh configurations[11].

To examine the performance of the new Incompact3D, several simulations were set up on meshes up to 4096^3 points and ran on HECToR using up to 16384 cores (72% of the full capacity). The strong scaling results are very satisfactory for all three problem sizes. As the computational part of Incompact3D are all based on tightly nested 3D loops which are easily vectorisable, there is no surprise that all performance results inherit the good scaling from the underlying communication library.

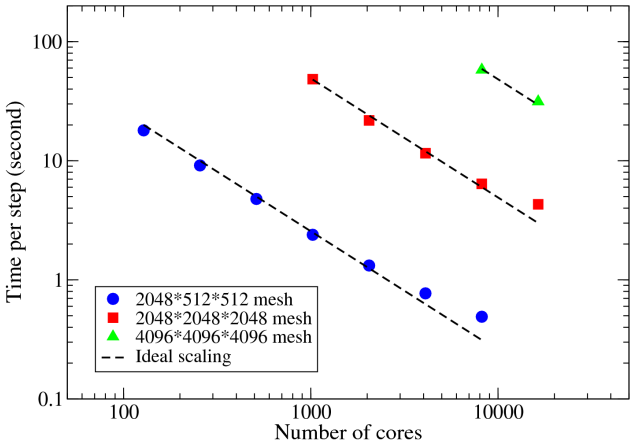

Figure 8: Strong scaling of Incompact3D on HECToR.

For comparison purpose, the scaling of the spectral DNS code by P.K. Yeung[8], which uses P3DFFT internally, is reproduced in Fig. 9. What is directly comparable is the data set taken from supercomputer Franklin (was also a Cray XT4) shown as green triangles, exhibiting very similar scaling behaviour as Incompact3D on HECToR.

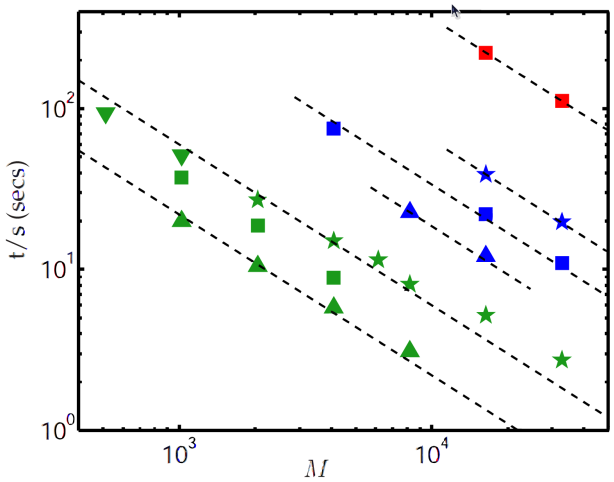

Figure 9: Scaling of spectral DNS code by P. K. Yeung[8] using P3DFFT.

Fig. 10 shows the weak scaling of Incompact3D at a workload of 4191304 mesh points per MPI rank. A performance comparison between the old and the new code is given. The old Incompact3D, implemented using a simple 1D decomposition, is faster on smaller core counts. But the new Incompact3D outperforms the old code for larger cases, partly because the communications are only among subsets of MPI ranks which is more efficient.

Also the constraint applied by the decomposition method on the old code (1024 cores) does not apply to the new code. Fig. 10 also presents the new code performance using only 2 cores per quad-core node, which is consistently 30% faster due to improved network bandwidth.

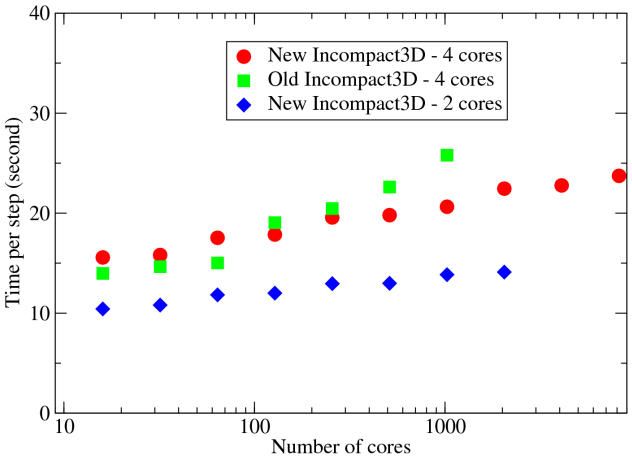

Figure 10: Weak scaling of Incompact3D on HECToR.

Finally, some scientific results are given in Fig. 11 showing the simulated vorticity field downstream of a square-shaped fractal grid in a wind tunnel. Comparing to the conventional grid, the fractal grid can generate much more turbulence while introducing much smaller pressure drop to the flow – a more energy efficient configuration that can be explored in applications such as combustion devices.

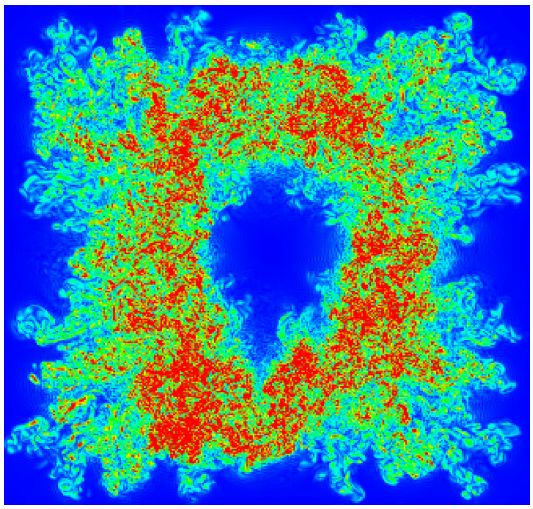
Figure 11: Vorticify field of flow after a fractal square grid.

Conclusion

Through a dedicated software-development project sponsored by the HECToR dCSE programme, the CFD application Incompact3D has been transformed into a software package which can make good use of the modern HPC resources. High scalability has been demonstrated on HECToR and additional benchmarks are being carried out on several other supercomputing sites. This work will enable cutting-edge turbulence research to be conducted.

The reusable software components, in particular a 2D decomposition module and a distributed FFT module, have been packed into a library, which will hopefully help scientists to develop other applications which are highly efficient, scalable and portable.

References

1. http://www.ffte.jp/ - D. Takahashi's FFTE Fast Fourier Transform package.
2. http://www.fftw.org/ - FFTW official website.
3. http://www.jjj.de/fft/glassman-fft.f - Source code of Glassman's algorithm in Fortran.
4. http://www.netlib.org/fftpack/ - FFTPACK home page at the Netlib Repository.
5. http://www.netlib.org/fishpack/ - FISHPACK home page at the Netlib Repository.
6. http://www.sandia.gov/~sjplimp/docs/fft/README.html - Parallel FFT Package by S. Plimpton.
7. http://www.sdsc.edu/us/resources/p3dfft/ - SDSC P3DFFT website.
8. D. A. Donzis, P. K. Yeung, and D. Pekurovsky, “Turbulence simulation on o(10^4) processors”, In *TeraGrid'08*, June, 2008.
9. W. E. Ferguson Jr., “Simple derivation of Glassman's general N fast Fourier transform”, *Computers & Mathematics with Applications*, 8(6):401-411, 1982.
10. D. Hurst and J.C. Vassilicos, “Scalings and decay of fractal-generated turbulence”, P*hysics of Fluids*, 19(3), 2007.
11. S. Laizet and E. Lamballais, “High-order compact schemes for incompressible flows: A simple and efficient method with quasi-spectral accuracy”, *Journal of Computational Physics*, 228(16):5989-6015, 2009.
12. R.E. Seoud and J.C. Vassilicos, “Dissipation and decay of fractal-generated turbulence”, P*hysics of Fluids*, 19(10), 2007.

13. P.N. Swarztrauber, “The methods of cyclic reduction, Fourier analysis and the FACR algorithm for the discrete solution of Poisson's equation on a rectangle”, *SIAM Review*, 19(3):490-501, 1977.
14. R.B. Wilhelmson and J.H. Erickson, “Direct solutions for Poisson's equation in three dimensions”, *Journal of Computational Physics*, 25(4):319-331, 1977.

Acknowledgements

Ning Li would like to thank colleagues Chris Armstrong and Ian Bush at NAG for their significant help in developing this library. Most benchmarks presented in this paper were carried out on HECToR - a Research Councils UK High End Computing Service.

Sylvain Laizet acknowledges support from the EPSRC grant EP/E029515/1 and the UK Turbulence Consortium (EP/G069581/1) for the CPU time made available on HECToR without which this work would not have been possible. He also thanks Prof. Eric Lamballais for very useful discussions.

This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725.

About the Authors

Ning Li is an HPC software developer at the Numerical Algorithms Group (NAG). He is a member of the HECToR CSE team and he is the leading author of the 2DECOMP&FFT library. He can be reached at NAG Ltd, Wilkinson House, Jordan Hill Road, Oxford, OX2 8DR, United Kingdom, E-mail: ning.li@nag.co.uk.

Sylvain Laizet is a research associate at Imperial College London. He is an expert in Computational Fluid Dynamics and he is the main author of Incompact3D. He can be reached at the Department of Aeronautics, Imperial College London, South Kensington Campus, London, SW7 2AZ, United Kingdom**,** E-mail: s.laizet@imperial.ac.uk.