Software Framework to Support Overlap of Communication and Computation in Implicit CFD Applications - a dCSE Project

Ning Li
Numerical Algorithms Group (NAG)
Wilkinson House, Jordan Hill Road,
Oxford, OX2 8DR, UK

March 11, 2013

Contents

1 Introduction

This project concerns the continuing development of an in-house CFD application Incompact3D used by the Turbulence, Mixing and Flow Control group at Imperial College. This is a Navier-Stokes solver that combines the versatility of industrial codes with the accuracy of spectral codes [7][8]. Thanks to a very successful previous dCSE, Incompact3D can now scale to more than 100,000 computational cores [9] and it is regularly being used to run 8,000-16,000 core jobs on HECToR. This enables the group to conduct cutting-edge turbulence researches productively, including their recent efforts to understand the physics of turbulence generated by multi-scale/fractal objects.

The high degree of parallelisation was achieved mainly via the development of a highly scalable 2D pencil decomposition library with a distributed FFT interface - 2DECOMP&FFT [10]. This numerical framework contains the reusable software components derived from the previous Incompact3D dCSE and it can be easily applied to other applications [4] based on similar numerical algorithms and parallel strategies. The software has subsequently become an open-source package [1] and is now part of the Open Petascale Libraries [3].

While the introduction of 2D pencil decomposition enables these applications to run at very large scale, the main impediment to scalability now becomes the high communication cost, often more than 50% of the total cost in large simulations. On production systems, the communication cost also tends to increase as the number of cores increases. To tackle this problem, one obvious solution in theory is to introduce overlap of communication and computation (OCC). Such overlap is, however, not easily achievable because of the all-to-all communication pattern used in these applications, its blocking implementations in present MPI libraries, and the general lack of mature software support. This project attempts to address the problem by introducing OCC support in the 2DECOMP&FFT framework, in order to facilitate OCC in applications like Incompact3D.

The project was granted 5 months full-time effort for one person to work on a full-time basis, which translates to roughly 1 year effort on a 40% part-time basis. The project officially started in March 2012.

Two work packages in the original proposal were supported. The work package 1 - to introduce flexible data layout support in the 2DECOMP&FFT framework, which is a prerequisite of other technical works - are discussed in section 4. The work package 2 - to create alternative FFT code for Incompact3D using non-blocking MPI collectives - are discussed in section 2 & 3.

2 Non-blocking MPI collectives

For applications using all-to-all type of communication, there are several technical approaches to realising overlap of communication and computation. For example, in a previous HECToR dCSE project, Anton *et al.*[4] has shown that OCC may be achieved using OpenMP threads. This, however, requires replacing the all-to-all calls with non-blocking MPI send/receive calls (i.e. re-implementing the high-level MPI functions using low-level operations).

Alternatively, one can implement OCC using one-sided communication. This may appear an attractive idea for Cray XE6 systems with RDMA support built in its GEMINI interconnect. However, this is generally speaking hard to implement and requires software supports in the form of low-level system libraries (libntk and libonesided on HECToR), which are, in the author's opinion, not mature enough.

This project concentrates on a third approach - to realise OCC using non-blocking MPI collectives. The idea of non-blocking MPI collectives has been around for quite a long time. It just narrowly missed the previous MPI-2.2 standard and it is possibly the most scrutinised new feature in MPI-3.0. The basic idea is straight-forward: it allows applications to issue *immediate* collective operations in the form of:

```
CALL MPI_I......(......, request, ierror)
! more computations
CALL MPI_WAIT(request, status, ierror)
```

which facilitates overlap of communication and computation, a feature already available for MPI point-to-point communication. What is directly relevant to the current project is the functions $MPI\_IALLTOALL$ and $MPI\_IALLTOALLV$ newly introduced in MPI-3.0.

2.1 LibNBC

When this project started in March 2012, the MPI-3.0 standard was not finalised. The specification was released in September 2012. Naturally it takes more time for library vendors to actually produce high-quality implementations. For these reasons, all technical works for this project was based on LibNBC, a prototype implementation of non-blocking MPI collective APIs. This unfortunately means some technical details contained in this report can become slightly outdated when it is published, although the principal ideas remain the same.

LibNBC [2][5] is a reference implementation of the non-blocking MPI collective operations. For this study, the latest version 1.1.1 (July 2012) was used. Here are technical details of building and using this library:

2.1.1 On a development system with GNU compiler and OpenMPI

Assume the library is to be installed at LIBNBC_DIR (which can be a local path), with GNU compiler and OpenMPI binaries (such as *mpicxx*) in the $PATH, simply do:

```
./configure --prefix=$LIBNBC_DIR
make
make install
```

To build applications using the library, add the following flags at linking stage:

```
$LIBNBC_DIR/lib/libnbc.a -lstdc++ -lmpi_cxx
```

2.1.2 On HECToR phase 3

On HECToR, assume the library is to be installed at LIBNBC_DIR, with the PGI compiler, do:

```
MPICXX=CC ./configure --prefix=$LIBNBC_DIR --host=x86_64-unknown-linux
make
make install
```

The *–host* part allows proper cross-compiling.[1] To build applications using the library, pass the following to the linker:

```
$LIBNBC_DIR/lib/libnbc.a
```

2.1.3 Asynchronous Progress

Ideally, MPI libraries can offload communication to interconnect hardware and have processors concentrate on computation. As reported in [6], such work in being done on some InfiniBand clusters. When such hardware and library supports are not available, one can use purely software solutions. libNBC supports the use of threads to progress communication asynchronously. To switch on such support, add either

[1] Without the flag, *configure* would fail because it would attempt to run binaries on the login node causing 'illegal instruction' errors.

```
--with-thread
```

or

```
--with-rt-thread
```

to the command line of the *configure* script. Unfortunately, applications linked to LibNBC with these options on all resulted in run-time errors on HECToR. This was not further investigated because: (1) these are quite low-level software implementation issues that are beyond the scope of this study; (2) an alternative and practical way to progress communication is available: to call MPI_TEST regularly from the main computational thread of applications.

Implementing fully asynchronous progression is often a platform-specific issue so it is better to leave the MPI library vendors to address this problem in the future.

3 APIs to support OCC in 2DECOMP&FFT

A number of new APIs have been introduced to the 2DECOMP&FFT framework, in order to help applications implement overlap of communication and computation using MPI non-blocking collectives. Depending on the numerical algorithms, there are various ways to achieve OCC. These are covered in details in this section.

3.1 Low-level API supporting non-blocking global transpositions

For large-scale applications like Incompact3D, the global transposition operations, which internally use all-to-all type of communication, are often very time-consuming. For example, assume that application data is distributed as 2D pencils[2], to transpose data from X-pencil to Y-pencil storage, applications need to use the following 2DECOMP&FFT's communication API:

```
call transpose_x_to_y(in, out)
```

where *in* contains the input data stored in X-pencil, and *out* contains the output stored in Y-pencil. This communication routine internally contains the following three steps:

1. Gathering data into the MPI send buffers from the input array.
2. Calling MPI_ALLTOALLV (or MPI_ALLTOALL, depending on user settings) to redistribute the data among MPI processes.
3. Scattering data from the MPI receive buffers to the output array in a usable form.

In particular, step 2 above is a blocking operation that will not return until the communication is completed and step 2 is often quite time-consuming in large-scale applications. In the corresponding non-blocking API, following the design of the MPI non-blocking APIs, the above operation is broken into two parts:

```
call transpose_x_to_y_start(handle, in, out, sbuf, rbuf)
call transpose_x_to_y_wait (handle, in, out, sbuf, rbuf)
```

Here *handle* is used to identify one particular set of communication. Because many non-blocking communication calls may be ongoing at the same time, applications need to supply a send buffer *sbuf* and a receive buffer *rbuf* that are associated with each communication. The *start* routine packs the MPI send buffer, starts the all-to-all communication, and returns immediately. Later, a call to the corresponding *wait* routine ensures the communication is completed and then

[2]Please refer to http://www.2decomp.org/decomp.html for an introduction of 2D pencil decomposition, in particular Fig. 2 on that web page for the exact definition of pencils.

unpacks the MPI receive buffer. Between the *start* call and the *wait* call, the content of *sbuf* should not be modified and the content of *out* should not be referenced, to avoid unpredictable results. Other unrelated computations may be carried out while the communication is ongoing.

While the principal of overlap communication and computation is straight-forward, it is up to application developers to identify suitable opportunities in their algorithms to make use of the idea. An example is given in the next section.

3.2 Application in 3D FFTs

To demonstrate the use of the non-blocking API above, an algorithm to perform multiple independent three-dimensional FFTs is presented here. Suppose that data is distributed as 2D pencils, a typical parallel FFT can be implemented using the following steps:

1. Assume the real-space data is distributed in X-pencil storage, perform 1D FFTs in X direction first.
2. Transpose from X-pencil to Y-pencil storage.
3. Perform 1D FFTs in Y direction.
4. Transpose from Y-pencil to Z-pencil storage.
5. Perform 1D FFTs in Z direction. The resulting spectral-space data is in Z-pencil storage (the different data distribution normally does not cause problem in practice. An option is always available to perform two extra transposes to bring the spectral-space data back to X-pencil storage).

When the transpositions above are realised using standard MPI_ALLTOALLV, which is blocking, only one parallel FFT can be performed at a time. With the non-blocking API, a new algorithm can be designed to overlap the communication of one FFT with the computation of another. While there are many ways to design such algorithms, one of the most straight-forward ways, proposed in [6] has been implemented in this project. This algorithm can be summarised using the following pseudo-code, assuming multiple data sets are identified by V_i:

```
1D FFTs in X for V_1
call transpose_x_to_y for V_1 (blocking)
1D FFTs in Y for V_1
call transpose_y_z_start for V_1
do k=2,N
  1D FFTs in X for V_k
  call transpose_x_to_y for V_k (blocking)
  1D FFTs in Y for V_k
  call transpose_y_to_z_start for V_k
  call transpose_y_to_z_wait for V_(k-1)
  1D FFTs in Z for V_(k-1)
end do
call transpose_y_to_z_wait for V_N
1D FFTs in Z for V_N
```

As can be seen, the Y-to-Z transpose for variable k and the computation of 1D FFT in Z for variable k-1 are overlapped.

As mentioned earlier, getting libNBC to progress the communication asynchronously turned out to be unsuccessful on HECToR. MPI_TEST calls have to be used to progress the communication. Normally, multiple 1D FFTs in the algorithm above can be performed in one go using APIs such as FFTW's advanced interface. Now they have to be implemented using simple 1D

FFTs within loops so that the MPI_TEST can be inserted and called regularly[3]. This constraint should be removed in the future when better MPI implementations become available.

The algorithm above was shown to be effective on HECToR. On phase 2b hardware, by manually optimising the number of MPI_TEST calls, up to 15% performance gain can be achieved when performing a set of independent FFTs of size 1536^3. On phase 3, without any optimising practice, a 8% performance improvement was observed with a problem size of 2048^3. Note that the problem sizes were chosen on each system for better load balance. Such algorithm was seen to be particularly efficient with large problem sizes.

3.3 High-level API supporting multiple independent FFTs with OCC

Based on the algorithms presented in the previous section, a high-level API is introduced in 2DECOMP&FFT perform multiple independent FFTs in one function call. This can be very useful in some spectral-method applications.

As the time of writing, the API is in the following format:

```
call decomp_2d_multiple_fft(in, out, opt_sbuf, opt_rbuf)
```

Here both input *in* and output *out* are four-dimensional arrays with the last dimension representing different 3D data sets. Each 3D data set has to follow the pencil-distribution storage format as defined by the 2DECOMP&FFT library. Two optional workspace arrays may be supplied by applications to be used as send/receive buffers by the MPI collective operations. They should be at least twice as large as the individual 3D data sets because two separate buffers are required when implementing communication/computation overlap. If such workspace is not supplied, the library will internally allocate sufficient scratch space.

The API may be subject to slight change after receiving feedback from key users to better accommodate data structures in use in real applications. However, the flexible design of the 2DECOMP&FFT framework, using Fortran generic interface, enables new syntax to be introduced easily.

3.4 3D FFT API using fine-grain OCC

The algorithm discussed in section 3.2 - to overlap communication of one FFT with computation of another FFT - is called in literature *coarse-grain* OCC. It is also possible to realise OCC within a single 3D FFT, which will be referred to as *fine-grain* OCC.

The idea has been tried in [11] in a distributed FFT algorithm using a similar pencil-decomposition setting, in which the author probably arbitrarily chose to transpose three quarters of the data first, start computing on the transposed data, and allow that computation to be overlapped with the transpose of remaining one quarter of data. Even with such a rudimentary approach, it is reported that up to 10% performance gain were achieved on HECToR when the problem size is sufficiently large.

The fine-grain OCC algorithm in this project is more flexible. It can be described as follows: in Figure 1, the transpose between Y-pencil (left) and Z-pencil (right) can be completed in one MPI_ALLTOALLV operation. It is also possible to redistribute the data in smaller batches, e.g. one X plane at a time using multiple MPI_ALLTOALLV operations.

Similarly, for $X \Leftrightarrow Y$ transpose, it is possible to handle data in one Z plane at a time.

There are many possibilities to implement such fine-grain OCC. Following is an algorithm applying to $X \Leftrightarrow Y$ transposes[4].

[3]The best frequency making MPI_TEST be calls is algorithm dependent, which, unfortunately, introduces one extra parameter to applications

[4]This is chosen because of the favorable memory layout associated with the operations, as will be explained in Section 4.

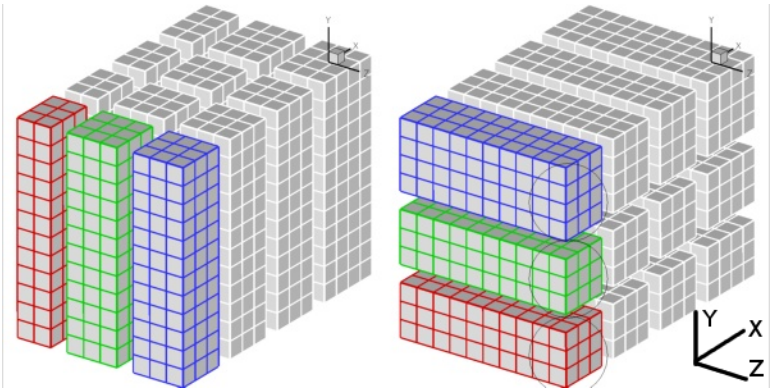

Figure 1: Transpose from Y-pencil to Z-pencil.

```
1D FFTs in X for Z_1 plane
call transpose_x_to_y_start for Z_1
do k=2,nz (where nz is the size of Z dimension)
  1D FFTs in X for Z_k plane
  call transpose_x_to_y_start for Z_k
  call transpose_x_to_y_wait for Z_(k-1)
  1D FFTs in Y for Z_(k-1) plane
end do
call transpose_x_to_y_wait for Z_nz
1D FFTs in Y for Z_nz plane
......
code for Y to Z transpose and 1D FFT in Z
```

This algorithm has similar logic to the coarse-grain OCC algorithm presented earlier - the communication for data on Z_k plane and the computation of $Z_{(k-1)}$ plane is overlapped. This algorithm was implemented in a standalone application and was shown to work well.[5]

4 Flexible data layout for 2DECOMP&FFT

This section covers the efforts under work package WP1 in the original proposal. It lays the software foundation for other work packages.

4.1 Overview

Previous versions of the 2DECOMP&FFT library only work with three-dimensional arrays in their natural storage, i.e. (i,j,k) order in Fortran where i is the fastest changing index, regardless of the pencil orientation of the distributed data. While this is convenient enough for many applications, there are situations in which more flexible data layouts are desired:

- It is a good practice to operate on the leading dimension of a multi-dimensional array in Fortran in order to improve cache efficiency, as data elements in the leading dimension are adjacent to each other in memory. For example, many FFT libraries perform best when transform stride-1 data, although most do support non-stride-1 input/output.
- Some legacy applications have dimensions of multi-dimensional arrays swapped, presumably an optimisation for old generations of hardware (for example, to increase the vector length on a vector machine).

[5]The integration of this algorithm into the 2DECOMP&FFT library was not done because of the unexpected technical difficulties to reuse 2DECOMP&FFT's communication API.

What makes the designing of flexible data layout relevant to this project is the fact that pencils, the smallest elements in terms of communication in the current 2DECOMP&FFT library, needs to be further partitioned into blocks, in order to implement communication and computation overlap (as discussed in section 3.4). While the further partitioning can be done with any data layout, only certain layouts can be implemented efficiently.

For example, in the transpose shown in Figure 1 earlier, suppose one wants to transpose one X-plane at a time, it would be desirable that i is the last dimension so that data in each X-plane is naturally contiguous in memory, making the packing/unpacking of communication buffers more convenient and efficient.

4.2 Strategy

While it is possible to store data in any i,j,k combinations, decision was made to implement only one combination other than the normal (i,j,k) format. This is described as follows:

- The X-pencil data always remains in the natural (i,j,k) order as i is already the leading dimension.
- The Y-pencil data can alternatively be stored in (j,i,k) order so that j becomes the leading dimension.
- The Z-pencil data can alternatively be stored in (k,j,i) order so that k becomes the leading dimension.

This combination ensures that in all pencil orientations the dimension that is completely local in memory is always the leading dimension, giving algorithms working on that dimension better cache efficiency.

Note that based on the reasoning in section 4.1, for Y-pencil data, (j,i,k) order is already suitable for $X \Leftrightarrow Y$ transposes; (j,k,i) order is really desirable for the $Y \Leftrightarrow Z$ transposes. Apparently, both conditions cannot be satisfied at the same time. So additional local transposes are required in the algorithms preparing the communication buffers.

4.3 Software implementation details

In 2DECOMP&FFT, the alternative data layout is implemented as a pre-processing branch of code and can be switched on with **-DSTRIDE1** flag when building the 2DECOMP&FFT library. If application chooses to use this option, its 3D arrays should be defined using the transposed data layout. A number of implementation details are noted here:

- A set of utility routines are introduced that help applications allocate distributed arrays. These routines handles the memory allocation properly according to users' storage preference so application developers do not need to intervene. Application developers still need to pay attention to the data layout when looping through the distributed arrays.
- Key I/O routines in 2DECOMP&FFT have been updated to work with the stride-1 data layout as well. Regardless of the storage in memory, data files generated by these I/O routines always store 3D arrays in their natural (i,j,k) order to simplify pre- and post-processing tasks.
- The distributed FFT interface implemented using FFTW 3 API has been updated to support stride-1 data layout. Bindings with other FFT libraries are not updated unless there are specific user requirements. Performance differences when using stride-1 data layout were not seen in small to medium size problems when running on HECToR. According to the P3DFFT User Guide, enabling stride-1 data structures may 'in some cases give some advantage in performance'. This is probably because P3DFFT has also introduced loop

blocking optimisation, which is a technique that can be useful but hard to generalise or tune automatically.

5 Conclusion and future works

A number of techniques were implemented during this project to enable 2DECOMP&FFT-based applications to have the ability to overlap communication and computation. In particular, the following goals have been achieved:

- A set of low-level API have been created in 2DECOMP&FFT to facilitate OCC in algorithms.
- The effectiveness of the API above was demonstrated in an application of multiple independent FFTs, achieving a performance gain of up to 15%.
- A high-level API has been provided to enable multiple independent distributed FFTs to run in a single subroutine call.
- By building on top of the 2DECOMP&FFT library, the CFD code Incompact3D is now MPI-3 ready and should benefit from the new MPI library development in the future.
- A sample application has been created to demonstrate the use of fine-grain OCC in 3D distributed FFTs.
- Introduced more flexible data layout in the 2DECOMP&FFT library.
- Updated 2DECOMP&FFT's I/O code to support additional data layouts.

Software developed within this project has been released wherever appropriate to benefit the wider community:

- Version 1.5 of the 2DECOMP&FFT library was released in October 2012. This release has included a number of new APIs derived from this dCSE project. The library is now used by 8 CFD codes on HECToR, and adopted by many research groups internationally.
- The CFD code Incompact3D has been open-sourced since November 2012 and has already attracted interests from the research community, leading to new collaboration opportunities.

There are two additional work packages in the original proposal which were not supported by dCSE. However, works on them outside the scope of this dCSE project have resulted the following:

- An API for Global Arrays (GA) integration was introduced in the latest version of 2DECOMP&FFT to allow applications to conveniently access GA's one-sided communication routines.
- Work has been done on using 2DECOMP&FFT with threaded versions of underlying FFT libraries. This has been shown to provide good performance on some systems.

6 Acknowledgments

This project was funded under the HECToR Distributed Computational Science and Engineering (CSE) Service operated by NAG Ltd. HECToR - A Research Councils UK High End Computing Service - is the UK's national supercomputing service, managed by EPSRC on behalf of the participating Research Councils. Its mission is to support capability science and engineering in UK academia. The HECToR supercomputers are managed by UoE HPCx Ltd and the CSE Support Service is provided by NAG Ltd. http://www.hector.ac.uk

References

[1] 2DECOMP&FFT website. http://www.2decomp.org.

[2] LibNBC website. http://www.unixer.de/research/nbcoll/libnbc/.

[3] Open Petascale Libraries website. http://www.openpetascale.org.

[4] L. Anton, N. Li, and K.H. Luo. A study of scalability performance for hybrid mode computation and asynchronous MPI transpose operation in DSTAR. In *Cray User Group 2011 conference*, May 2011.

[5] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC07)*, 2007.

[6] 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):237–246, 2011.

[7] 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.

[8] 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):471–484, 2010.

[9] S. Laizet and N. Li. Incompact3d, a powerful tool to tackle turbulence problems with up to $0(10^5)$ computational cores. *International Journal of Numerical Methods in Fluids*, 67(11):1735–1757, 2011.

[10] N. Li and S. Laizet. 2DECOMP&FFT - a highly scalable 2D decomposition library and FFT interface. In *Cray User Group 2010 conference*, May 2010.

[11] R. Saksena. On non-blocking collectives in 3D FFTs. In *Workshop on Latest Advances in Scalable Algorithms for Large-scale Systems*, 2011.