A new transpose split method for three-dimensional FFTs: performance on an Origin2000 and Alphaserver cluster

P. Wapperom [1] A.N. Beris *

Department of Chemical Engineering,
University of Delaware,
Newark, DE 19716, USA

M.A. Straka

National Center for Supercomputing Applications
4131 Beckman Institute
405 N. Mathews Ave.
Urbana, IL 61801

Abstract

We discuss a new transpose split method for parallel computations of three-dimensional Fourier transforms. By splitting the data along two dimensions over the processors it allows for a higher degree of parallelization than the original transpose split method. The traditional transpose split method involves one alltoall communication. The new method involves two communication steps in which each processor performs an "alltoall" communication in groups.

The performance of the new method has been evaluated using MPI on an Origin2000 and an Alphaserver cluster and compared with the traditional transpose split method. We found that the extra communication step introduced in the new method only slightly increases communication time. However, an efficient parallelization depends critically on how fast the communications can be performed.

Key words: Parallel 3D Fourier/Chebyshev transform; Fast Fourier transform; MPI; pseudospectral direct numerical simulation; transpose split method

* Corresponding author.
Email address: beris@che.udel.edu (A.N. Beris).
[1] Current address: Department of Mathematics, Virginia Tech, Blacksburg, VA 24061, USA

1 Introduction

An essential element of three-dimensional pseudospectral fluid dynamics calculations such as encountered in direct numerical simulations (DNS) of turbulent flows [8,7] is that those are performed alternatively in physical and spectral space. More specifically, non-linear terms in the equations are calculated at the nodal points in the physical space using the nodal values of the variables and their derivatives, whereas the derivatives themselves and the solution of Poisson-type problems are obtained in spectral space from the spectral coefficients. As such the calculations involve a large amount of Fourier/Chebyshev transforms from physical to spectral space and vice versa. Fourier transforms can be performed efficiently using the fast Fourier transform (FFT) [3] involving $O(N\log N)$ operations. Similarly, Chebyshev transforms can be reduced to Fourier transforms and carried out by FFTs as well. FFTs are usually the most time-consuming part of pseusospectral calculations, since the algebraic equations can be solved very efficiently in spectral space.

Nowadays, it is common to use parallel implementations of spectral methods to facilitate three dimensional calculations. All components of a pseudospectral DNS code parallelize trivially, as needing only local processor data, except the FFTs, which need to combine eventually all data. A commonly used algorithm to parallelize multidimensional FFTs is the transpose split method [2,5,9,1]. A three-dimensional FFT is calculated by distributing data along one direction over the processors, thus involving one communication. However, this limits the number of processors that can be used to the minimum array length in one of the dimensions. Particularly for cases involving only a small number of grid points in one of the (neutral) directions this may severely limit the efficiency of the algorithm.

In this paper we discuss an extension of the transpose method, which allows to split multiple directions over the processors. We apply the method to a turbulent straight-channel flow, which involves a 3D Fourier/Chebyshev transform. Furthermore, the performance of this transform is evaluated separately and in the full code, and compared with the traditional transpose split method on an Origin2000 and an Alphaserver cluster.

2 Method

The turbulent channel flow we discuss here, has one non-periodic (shear) y-direction and two periodic directions x (streamwise) and z (spanwise). The transformation from spectral to physical space involves in sequence a Chebyshev transform in y, a complex-to-complex Fourier transform in z, and a

complex-to-real Fourier transform in x direction. We will denote the number of nodes in the x, y, and z-direction by N_x, N_y, and N_z, respectively. The Chebyshev transform is obtained by an FFT involving $N_y - 1$ points [10], so that $N_y - 1$ should be factorizable in terms of low prime numbers for an efficient FFT.

For the parallelization of the three-dimensional Fourier/Chebyshev transform, we have chosen to use non-overlapping communications and computations. Recently, Dubey and Tessera [6] reported that overlapping gave very little benefit and that it even increases the overall execution time because of the extra overhead. Additionally, we use alltoall type of communications which were found to perform better than communications using separate send and receives [6].

In [1] the parallelization of the transform was performed by the traditional transpose split method. At the start of the transform, each processor contains all data in the y and z direction and $(N_x/2)/N_{\text{proc}}$ data of the x direction, see Fig.1. Schematically, this method involves the following steps

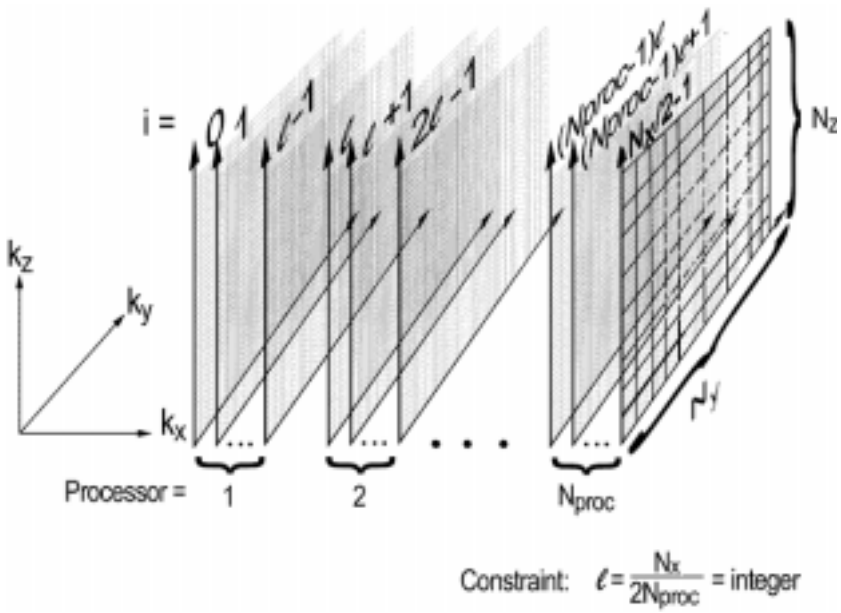

Fig. 1. Data allocation for the original transpose split method in spectral space. The k_x, k_y, and k_z denote the mode numbers in x, y, and z direction, respectively, and i indicates which mode numbers k_x each processor contains.

(1) Each processor performs $N_z \times (N_x/2)/N_{\text{proc}}$ one-dimensional Chebyshev transforms in y-direction and reorders data for z transform.
(2) Each processor performs $N_y \times (N_x/2)/N_{\text{proc}}$ one-dimensional complex-to-complex FFTs in z-direction and reorders data array for communication.
(3) All processors are synchronized using mpi_barrier and data communications are performed to effectively transpose the data using mpi_alltoall

on all processors.

(4) Each processor reorders data for FFTs and performs $N_y \times N_z/N_{\text{proc}}$ one-dimensional complex-to-real FFTs in x-direction.

At the end of a three-dimensional Fourier/Chebyshev transform, every processor contains all data in the x and y direction and N_z/N_{proc} data in the z direction, see Fig. 2. The maximum number of processors is limited to

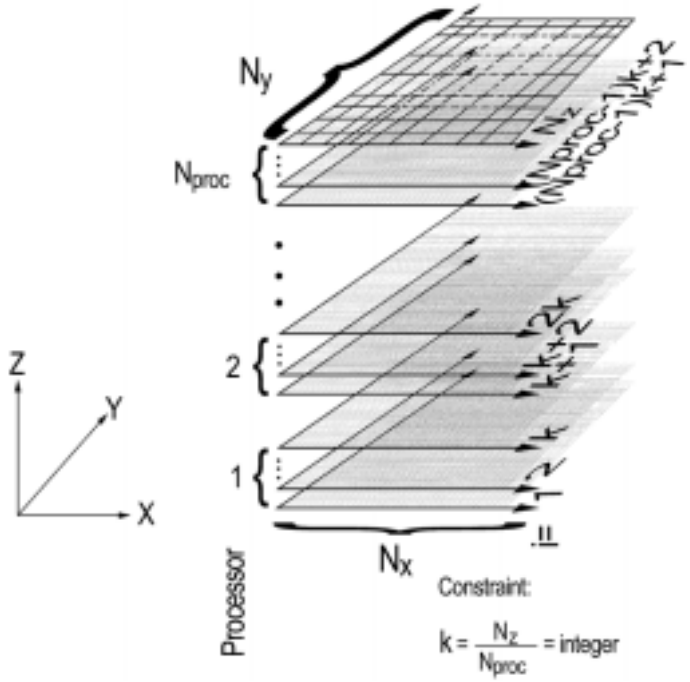

Fig. 2. Data allocation for the original transpose split method in physical space; i indicates which nodal point numbers in the z direction each processor contains.

$\min(N_x/2, N_z)$. Particularly for cases where either N_x or N_z is much smaller than N_y, this severely limits the number of processors that can be used.

To overcome this limitation, we have extended the transpose split method to allow for the distribution of a second direction over the processors. The extra cost is, of course, an additional communication. Schematically, we have the following procedure for a 3D transform from spectral to physical space:

(1) Each processor performs K_{zx} one-dimensional Chebyshev transforms in y-direction and reorders data for communication.
(2) The processors are synchronized using mpi_barrier and data communications are performed to effectively transpose the data using mpi_alltoall in groups.
(3) Each processor reorders data for FFTs, performs K_{yx} complex-to-complex FFTs in z-direction, and reorders data for communication.
(4) The processors are synchronized using mpi_barrier and data communications are performed to effectively transpose the data using mpi_alltoall in

groups.

(5) Each processor reorders data for FFTs and perform K_{yz} complex-to-real FFTs in x-direction.

In the above scheme, K_{yz}, K_{zx}, and K_{yx} denote the number of one-dimensional transforms that have to be performed in the x, y, and z direction, respectively. The data allocation and the one-dimensional FFTs at step 1, 3, and 5 is illustrated in Fig. 3. The transformation from physical to spectral space is

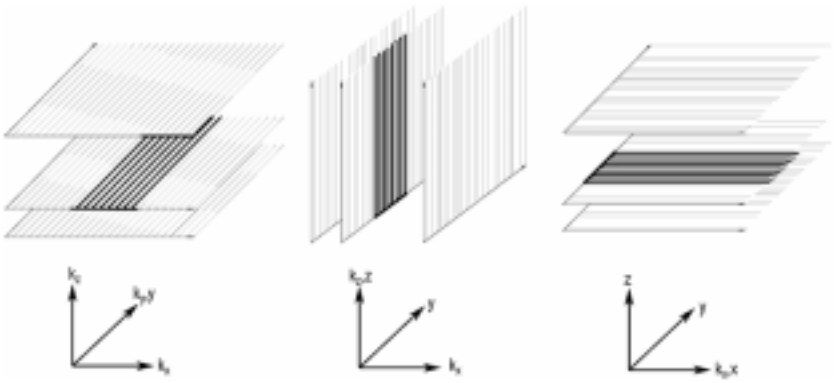

Fig. 3. Data allocation for the new transpose split method during a FFT transform from spectral to physical space. Indicated with black lines are the data contained in one of the processors during the three stages of the transformation. At the coordinate axes, k_y, y, k_z, z, and k_x, x denote the transformations from spectral to physical space in the respective coordinate directions.

simply the mirror of this procedure. For further reference, we will denote the group alltoall in step 2 by yz alltoall and in step 4 by zx alltoall. Note that for the traditional transpose method we only have a zx alltoall (step 3) that involves all processors.

For the extended transpose method, we restrict ourselves to cases where the following criteria are met

$$\begin{aligned} \frac{N_{\text{proc}}}{N_x/2} &= \text{integer} \\ K_{zx} = \frac{N_x/2 \times N_z}{N_{\text{proc}}} &= \text{integer} \\ K_{yx} = \frac{N_x/2 \times N_y^+}{N_{\text{proc}}} &= \text{integer} \\ K_{yz} = \frac{N_y^+ \times N_z}{N_{\text{proc}}} &= \text{integer} \end{aligned}$$

where, to facilitate data communication, $N_y^+ \geq N_y$ has to be chosen such that the above requirements are fulfilled. The elements $N_y + 1$ to N_y^+ are filled with zeroes.

In step 1 we have chosen to first distribute the data along the x-direction over the processors. Thus if $N_{\rm proc} > N_x/2$, processor 1 holds all y data for z data 1 till K_{zx} and x data 1. Processor 2 has all y data for z data $K_{zx}+1$ till $2K_{zx}$ and x data 1 etc. After a transform from spectral to physical space, each processor holds all x data for a selected number of y and z data.

Also note that in the YZ and ZX alltoall, it is not necessary for a processor to communicate with all other processors. For this we have divided the processors into groups using mpi_comm_split. In the YZ alltoall, $N_x/2$ transpose communications, each of which involves $N_{\rm proc}/(N_x/2)$ processors. In the ZX alltoall, $N_{\rm proc}/(N_x/2)$ transpose communications, each of which involves $N_x/2$ processors. For the example of $N_x/2 = 32$ and 64 processors, the YZ alltoall involves communication between processor 1 and 2, 3 and 4, ..., 63 and 64, forming 32 groups of 2 processors each. The ZX alltoall involves communication of all even processors mutually and of all odd processors mutually, thus forming two communication groups of 32 processors each. Note that each processor communicates with a total of 32 other processors in the two communication steps together, while for the traditional transpose split method every processor communicates with all, 63, other processors.

3 Results

The most time-consuming part of pseudo-spectral calculations is the three-dimensional Fourier/Chebyshev transform. For this we have investigated the scalability of this transform separately in a test code that performs a "do loop" of 1050 transforms from spectral to physical space and vice versa. The total number of transforms in the test code, 2100, was chosen such that it corresponds to 100 time steps of a straight channel calculation (where, per time step, there exist 12 transforms from spectral to physical space and 9 from physical to spectral). The performance of the straight-channel code is discussed at the end of this section. All calculations have been performed on an Origin2000 with 128 processors and an Alphaserver cluster with 256 processors.

Optimizing on a single processor has been performed on the Origin2000. It has been found crucial to perform cache optimization to attain best performance, particularly when each processor has a large amount of data. The key is to perform as many operations as possible on an array while it is still in cache, since bringing data into cache takes a multiple of the actual computation time. The advantage of using the cache has most convincingly been demonstrated in computer runs reported in [4]. If we take the Chebyshev transform in the spectral-to-physical transform in appendix A as an example, this means that both the Chebyshev call and the reordering of the array are inside the loop

over the K_{zx} arrays. First performing all K_{zx} Chebyshev transforms and then reordering the large array in a separate loop resulted in a considerable increase of CPU time due to cache misses. On the Alphaserver cluster we used the publicly available FFTW routines (http://www.fftw.org) in the Fourier and Chebyshev transforms. On the Origin, we found the FFT routines of the SCS library faster and we have used these instead. Since the MPI communication time far outweighs the FFT time, we have not felt it necessary at this point to attempt further FFT optimization on the Alphaserver, such as using the native Compaq math library routines. Similarly, we have not exhausted the available compiler options which might yet yield some further optimization.

A highly optimized code with relatively few cache misses is essential to obtain fair performance ratios. Programs with bad cache usage profit more in the computational part due to smaller array sizes and therefore fewer cache misses. This may compensate for a possible bad communication performance for large values of N_{proc}. Therefore, we have timed every FFT and Chebyshev transform, barrier, and alltoall communication separately (see appendix A), to check the performance of both the computational and communication parts of the code. The timings of the FFT routines also include the loops to copy to and extract from the communication arrays.

The computations on an SGI Origin2000 have been performed at the National Center for Supercomputing Applications (NCSA). This Origin2000 consists of 64 nodes with two 250 MHz processors each. For the computations on the Alphaserver cluster, we used the Terascale Compaq Alphaserver Cluster at the Pittsburgh Supercomputing Center which comprises 64 ES40 nodes. Each computational node contains four 667 MHz processors and a Quadrics interconnection network connects the nodes. Henceforth, we refer to this system as Terascale Computing System (TCS). All parallel jobs have been run in dedicated mode to avoid large fluctuations in communication time which are present when the system is shared with other users. Furthermore, the distribution of the processors over the computational nodes is important to obtain best communication performance. Assigning only one processor per memory outperformed the multiple processors per memory runs. Taking two processors per memory on the TSC, resulted in a doubling of the communication time. This is a result of the hardware implementation which allows, at a time, only one of the processors to communicate with a processor that does not belong to that memory. Taking two processors per node on the Origin, also resulted in an increased communication time. Its magnitude, however, strongly depends on the number of data and number of processors and varied between a marginal increase and a doubling of the communication time.

All timings reported below are obtained with mpi_wtime, which measures wall clock times. To check mpi_wtime we compared with CPU and system time obtained from etime on the Origin2000 and found good agreement. The

exact location of the timing calls in the program can be found in the Fortran program in appendix A.

3.1 xyzfft test runs

For the test code which only times the three-dimensional transforms from spectral to physical space and vice versa, we have performed two series of calculations for various values of $N_{\rm proc}$. For the first one (case a) we used $N_x = N_z = N_{\rm proc}$ which involves two groups of $N_{\rm proc}/2$ processors in the zx communication and $N_{\rm proc}/2$ groups of two processors in the yz communication. For the second case (b) we employed $N_x/2 = N_z = N_{\rm proc}$ which involves only one traditional alltoall communication. For case a and b, N_y^+ equals $N_y + 1$ and N_y, respectively. To increase the data size we have increased the value of N_y. These three-dimensional Fourier transform calculations have a ratio of computations and communications between 20 and 40.

For the yz alltoall communication, the total amount of data that one processor sends to all $N_{\rm proc}/(N_x/2)$ processors belonging to the same group is

$$N_{\rm data} = K_{zx} \times K_{yx} \times \frac{N_{\rm proc}}{N_x/2} = \frac{N_y^+ \times N_z \times N_x/2}{N_{\rm proc}},$$

where one data (one complex number) consists of 16 bytes. In a zx alltoall communication, the total amount of data that one processor communicates to all $N_x/2$ processors belonging to the same group is

$$N_{\rm data} = K_{yz} \times N_x/2 = \frac{N_y^+ \times N_z \times N_x/2}{N_{\rm proc}}.$$

Note that the same amount of data is involved, but compared to the yz alltoall we now have a different number of groups of a different number of processors. Henceforth, the quantity $N_{\rm data}$ is used as a measure for the amount of transmitted data.

The performance ratio E is defined by

$$E = \frac{T(1)}{N_{\rm proc} \times T(N_{\rm proc})} \tag{1}$$

where $T(N_{\rm proc})$ denotes the computational time per processor for a computation involving $N_{\rm proc}$ processors and $T(1)$ the computational time involving a single processor. The values of $T(1)$ have been estimated by extrapolation

of the multi-processor results, since the computational part scales practically linearly.

Performance ratios for various N_{proc} and N_{data} are displayed in Fig. 4 for the Origin2000 and TCS. For small N_{data}, we found very poor performance ratios

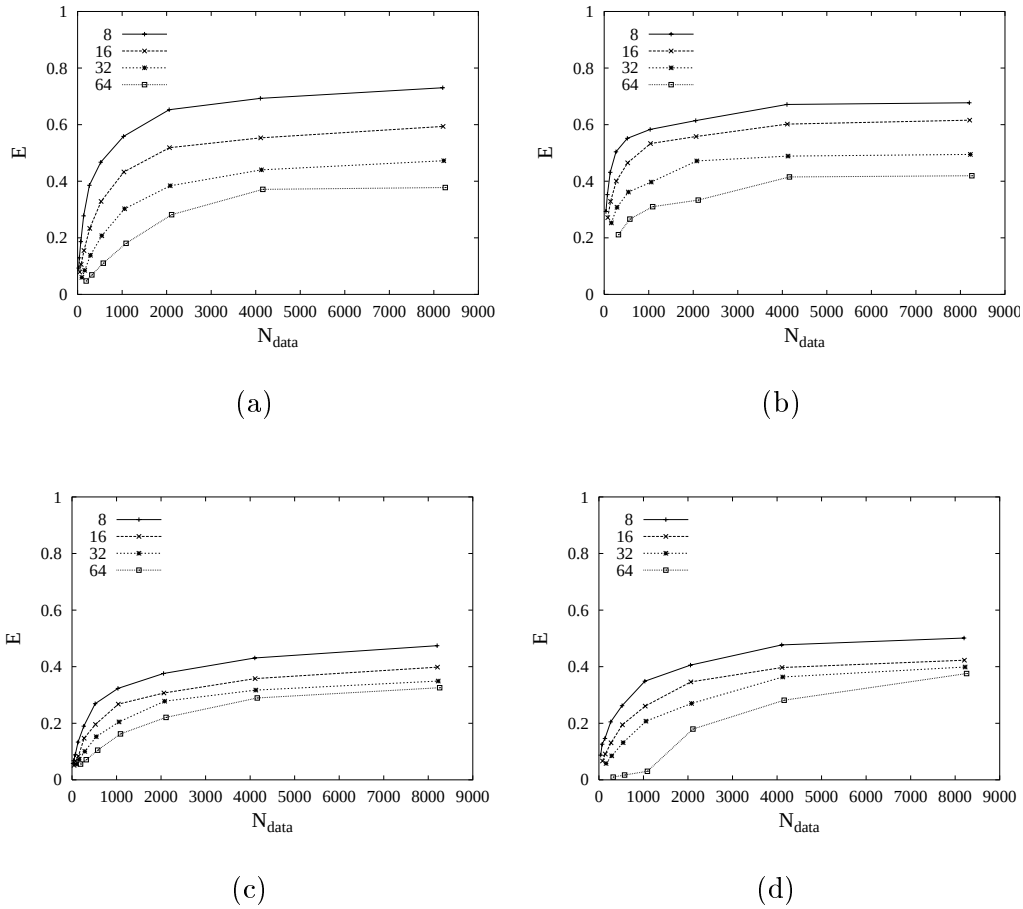

Fig. 4. Performance ratios as function of data size for various N_{proc}; (a) Origin case a, (b) Origin case b, (c) TCS case a, (d) TCS case b.

of the order of 10%. The initial increase is mainly caused by the fact that less data than the maximum bandwidth are transmitted. For large N_{data} a good performance of about 70% is obtained on the Origin for the 8 processor case. For the 64 processor case, however, this drops to 40%. Also note that for small N_{data} on the Origin, surprisingly, the performance ratios for case a are much worse than for case b. On the TCS this is absent. What is remarkable there is the poor performance ratios for all cases. This is due to the faster processors, while the data transfer rate is of the same order. For all programs using a substantial data communication, the faster and faster processors make parallelization less and less efficient when data transfer rates remain relatively low.

The barriers only take a substantial time for small data sizes as can be observed

in from Fig. 5. On the Origin, the new transpose split method spends con-

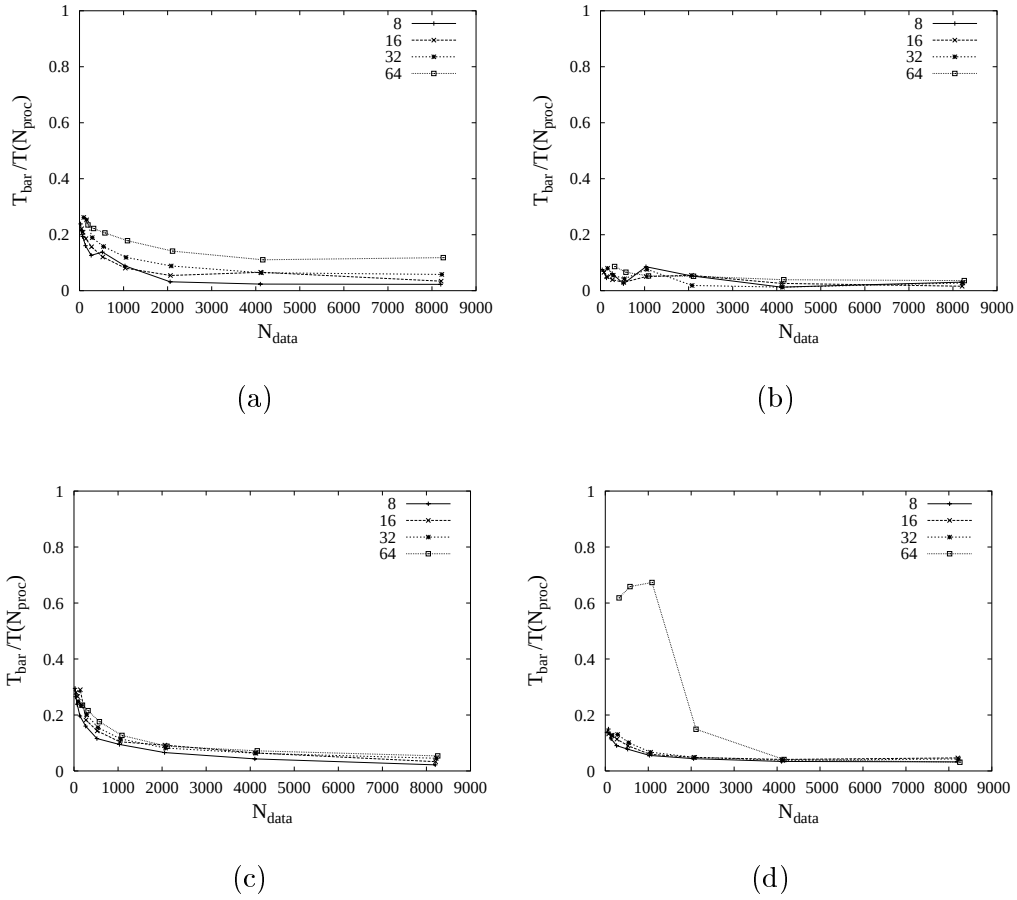

Fig. 5. Relative barrier times as function of data size for various N_{proc}; (a) Origin case a, (b) Origin case b, (c) TCS case a, (d) TCS case b.

siderably more time in barriers than the traditional transpose split method, particularly for large N_{proc}. On the TCS this behavior is absent. There the extremely poor performance for case b with $N_{\mathrm{proc}} = 64$ and low data sizes is remarkable. For some unknown reason, the alltoall communication and barrier for small data sizes fluctuate heavily per processor and take a multiple of the time needed for the large data sizes. Recalculation of the small data size cases showed the same trends. We conclude from Fig. 5 that lumping can decrease the barrier times significantly for small N_{data}.

For a communication step, we define a corresponding data transfer rate R_{data} in Mb/s as

$$R_{\mathrm{data}} = 16 \times 10^{-6} \frac{N_{\mathrm{fft}} \times N_{\mathrm{data}}}{T_{\mathrm{c}}},$$

where N_{fft} is the number of 3D transforms, T_{c} the time per processor spent in an alltoall communication. The decrease of the data transfer rate for larger

$N_{\rm proc}$ can be clearly observed in Fig. 6, where we plotted the data transfer rate

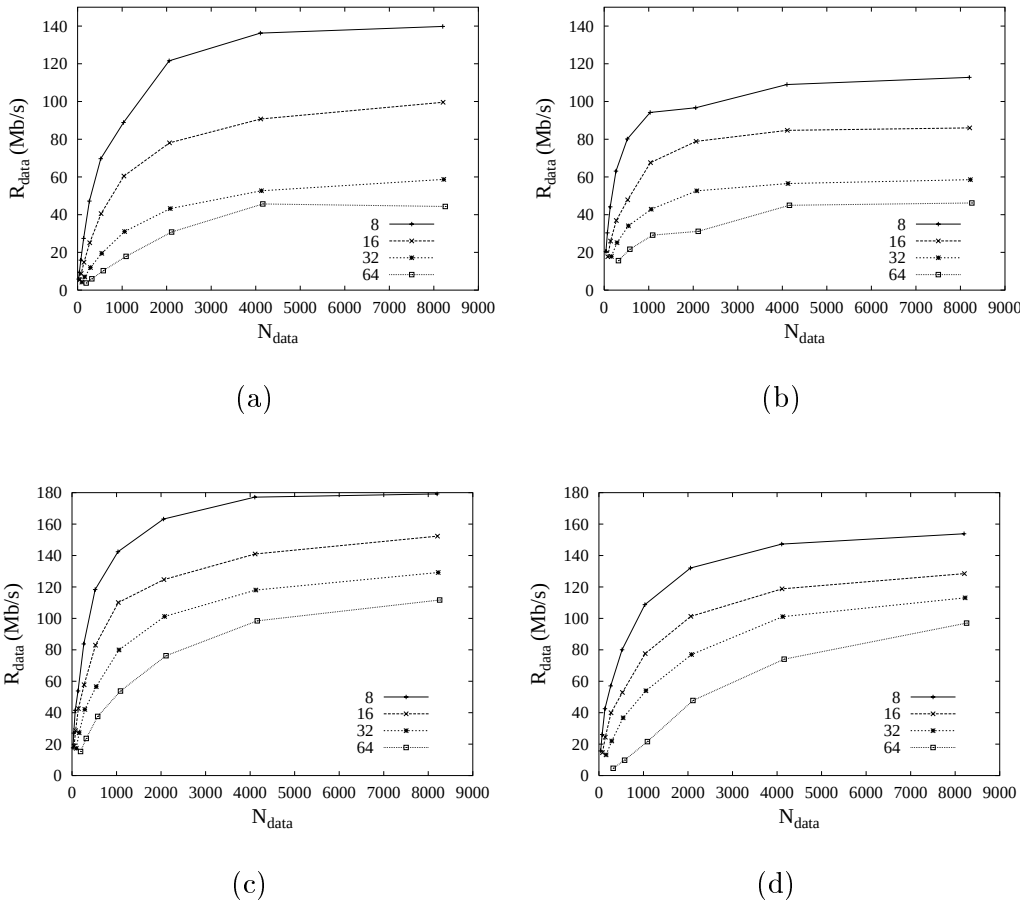

Fig. 6. Data transfer rates for zx communication as function of data size for various $N_{\rm proc}$; (a) Origin case a, (b) Origin case b, (c) TCS case a, (d) TCS case b.

(in Mb/s) of the zx alltoall, i.e. counter 25 in the Fortran program in appendix A. (Remember that for case a, two groups of $N_{\rm proc}/2$ are communicating K_{yz} data to all processors in the group, while for case b, where N_x is doubled, K_{yz} data are sent to all $N_{\rm proc}$ processors.) However, for the TCS the decrease is much less than for the Origin. We remark in passing that when we look at the data transfer rate against the amount of data that one processor sends to one other processor, (increasing $N_{\rm data}$ by a factor two when $N_{\rm proc}$ is increased by two in Fig. 6), the transfer rate only slightly decreases for the TCS, particularly for case b. On the Origin2000, for small data sizes, the data transfer rates for case a are very low compared to case b. This reverses for large data sizes. For the TCS, this behavior is absent, and the data transfer rates for case a always outperform the traditional transpose split method. Lumping may considerably improve the data transfer rates for small data size. For large data size, however, the data transfer rate reaches a plateau value and any speed-up achieved through lumping will only be marginal.

In Fig. 7 we compare for 64 processors the total communication time for case a and b. For small data sizes we observe anomalous behavior on both systems

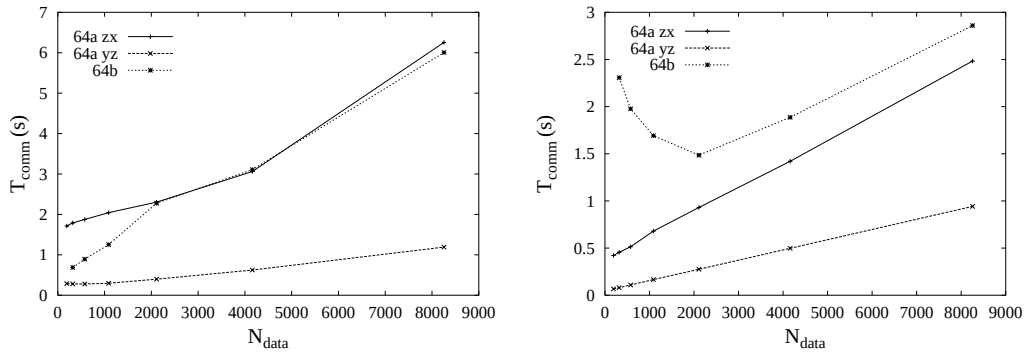

Fig. 7. Communication time for zx and yz communication as function of data size; (a) Origin, (b) TCS.

so that we have to remain inconclusive. For large enough data size we find, on the Origin, that the traditional and new alltoall perform very similar. Then the increase in communication time is the yz communication only, which is relatively small. For the TCS, the zx communication of case a is considerably faster than for case b. The overall communication time however remains larger than for the traditional transpose split method.

3.2 DNS runs

To validate the new transpose split method in a real flow calculation, we have timed a direct numerical simulation of turbulent flow in a straight channel using a pseudospectral code based on that developed by Sureshkumar et al. [9] and Beris and Dimitropoulos [1]. As discretization we took a standard test case as used in these references: $N_x \times N_y \times N_z = 64 \times 65 \times 64$ ($N_x/2 \times N_y^+ \times N_z = 32 \times 66 \times 64$ complex data in the transpose). The traditional transpose split method allows a maximum number of 32 processors where as the new method allows a maximum of $32 \times 64 = 2048$ processors. The DNS calculations are different from the transform timings in the sense that we keep the total amount of data fixed, so that the data size N_{data} decreases with increasing number of processors. Timings are given for 100 time steps on the Origin2000 and TCS. Per time step, 21 transforms (12 from spectral to physical space and 9 from physical to spectral) have to be computed, so that this is equivalent to the timings for the test code described above. We distinguish between 8, 16, 32, and 64 processors, for which $N_{\mathrm{data}} = 16640$, 8320, 4160, and 2112, respectively. The traditional transpose split method has been used for up to 32 processors, while the new transpose split method

needs to be used for the 64 processor case. Figure 8 displays the timings of the total calculation (excluding I/O and initialization), the three-dimensional transforms, the communications, and the time spent in FFT routines. The

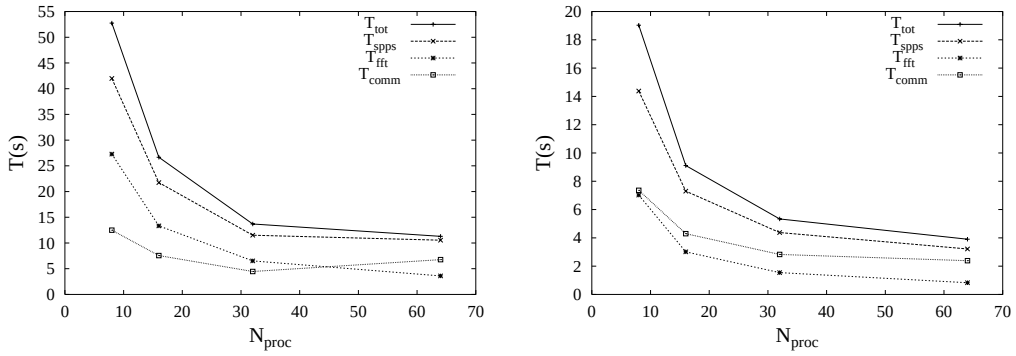

Fig. 8. DNS of turbulent flow: total time, 3D transform time, time in FFT/Chebyshev routines, communication time (including barriers); (a) Origin, (b) TCS.

difference between the total time and the time spent in the three-dimensional transforms represents the time for the additional computations compared to the test program (building right-hand side vectors and solve matrix-vector equations). The time for the three dimensional transforms consists of a computational part T_{fft} and a communication part T_{comm}. We observe that the computational part on both the Origin and TCS practically scale as $1/N_{\text{proc}}$. The scaling of the communication part, however, is much less, particularly for the higher values of N_{proc} as was already observed for the test code. Additionally, we now have a decrease in data size with increasing number of processors, resulting in decreased data transfer rates (see Fig. 6, for 8, 16, 32, and 64 processors, we have $N_{\text{data}} = 16640$, 8320, 4160, and 2112, respectively). The latter effect, however, can be compensated for by lumping of various arrays into one bigger array or when finer discretizations are used. As a result, the overall performance, scales well up to 32 processors, i.e. for the original transpose method. However, for the 64 processor case (with the new transpose split method), the scaling deteriorates, particularly for the Origin. Because of this reason, we have not timed the code with N_{proc} larger than 64, although the method allows up to $N_z \times N_x/2 = 64 \times 32$ processors.

As remarked earlier, the performance of the computations depends crucially on the number of processors that are allocated on each computational node. In Fig. 9 we have plotted the timings for the 64 processor calculation on the TCS with 1, 2, or 4 processors allocated on each node. Doubling the number of processors allocated on each node practically doubles the communication time, because at each time only one of the processors can communicate with a processor on another node. The doubling of the communication time also

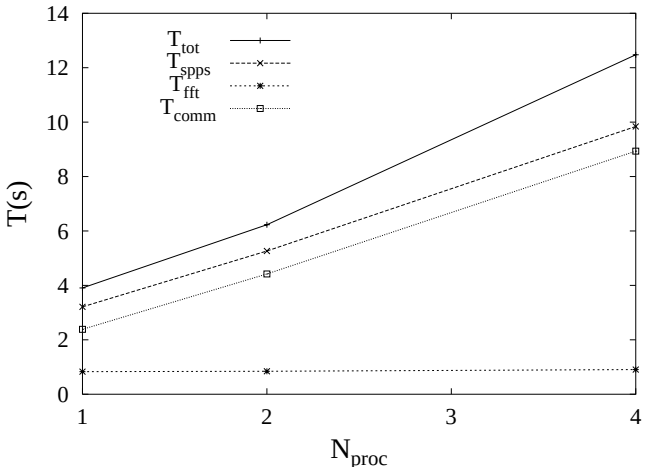

Fig. 9. DNS of turbulent flow; dependence on number of processors per node on TCS: total time, 3D transform time, communication time (including barriers), time in FFT/Chebyshev routines.

results in a doubling of the total time since the computation time, which remains constant, is only a fraction of it. For computations with a considerable alltoall data communication it is thus essential that multiple processors do not share one communication line.

4 Concluding remarks

We have discussed a new transpose split method for three-dimensional Fourier transforms. The traditional transpose split method splits data in one of the dimensions over the processors and involves a single alltoall communication performed between all processors. By splitting data along two dimensions over the processors, the new method allows for the use of more processors even when one of the periodic directions involves few Fourier modes. In terms of implementation, the new method involves an additional communication step and alltoall communications are no longer to all processors but are localized in groups. The performance has been evaluated for a test code comprising the three-dimensional Fourier transform only and in a full DNS of turbulent flow. To detect system peculiarities, all calculations have been performed on an Origin2000 and Alphaserver cluster, which have two and four processors per computational node, respectively.

Performances of the new method have been compared with the traditional transpose split method for various numbers of processors and data sizes. To obtain a better communication performance is was essential to allocate only one processor per node; using n processors per node results in n times the communication time of a single processor per node. For the test code we found

a considerable decrease in performance with increasing number of processors. This was caused by the relatively poor scaling of the communication time. Except for low data sizes on the Origin2000, the new transpose communication was faster than the traditional one with the same data size. Nevertheless, performance ratios are not high (similar conclusions of course can be drawn using the traditional transpose split method), particularly when the number of processors increases beyond 32 or 64. The main cause is a relatively slow communication compared to computation. Additionally, the data transfer rate decreases with increasing number of processors so that the performance decreases accordingly. Overlapping of communications and computations will not improve things since the communication time is dominant for a large number of processors.

Applications of the new transpose split method lie in the area of direct numerical simulations of turbulent flows, where three-dimensional fast Fourier transforms are used to transform from physical to spectral space. As an example we have discussed direct numerical simulation of turbulent flow in a straight channel. The transpose split method becomes more powerful when the number of data points in one of the directions is especially short compared to the other two directions. This is typically the case for simulations in wavy channels, where the number of nodes in the spanwise direction is much lower, thus limiting the number of processors that can be used with the traditional transpose split method. Another application is weather forecasting where finite difference schemes are used at the moment for reasons of efficiency. To use a much better spectral approximation, one needs both fast communications and a scheme that allows to take full advantage of large processor arrays containing thousands of processors. The latter is exactly what the new transpose split method can achieve.

In a real flow simulation with fixed data size, the performance decreases with increasing number of processors due to a smaller data array per communication. In direct numerical simulations in straight channels, the latter effect can be compensated for by lumping of arrays of different variables for the communication transfer. Most important, however, for a good scalability of parallel codes with a reasonable amount of data transfer (here a ratio of computations and communications between 20 and 40), are faster communications. As the processor speed increases, it makes good sense to also increase the sustainable communication rate of the network.

Acknowledgements

This work is supported by the NSF grant CTS-9981388. We are grateful to the National Center for Supercomputing Applications (NCSA) and the Pitts-

burgh Supercomputing Center for providing the computational resources on the Origin2000 and National Science Foundation Terascale Computing System, respectively. The authors would also like to acknowledge the help of Randolf Oswald in preparing Figures 1-3.

A Fortran code of three-dimensional Fourier/Chebyshev transform

Below we have included the relevant Fortran code for the three-dimensional Chebyshev/FFT spectral-to-physical transform. The various parts are timed with the calls to the subroutines perfon and perfoff. The physical-to-spectral transform is the mirror, except for a scaling factor.

The communicators yz_comm and zx_comm for the yz and zx communication involving the grouping are easily obtained with mpi_comm_split as follows

icol = (myproc*N_x/2) / $N_{\rm proc}$
call mpi_comm_split(mpi_comm_world, icol, myproc, yz_comm, ierr)

ngroup_zx = $N_{\rm proc}/(N_x/2)$
ih = myproc / ngroup_zx
icol = myproc - ih*ngroup_zx
call mpi_comm_split(mpi_comm_world, icol, myproc, zx_comm, ierr)

where myproc denotes the rank of the processor in mpi_comm_world. The Fortran code for the transform reads

```
subroutine xyzfftsp( as, ap )
complex*16 as(N_y, K_zx)
real*8 ap(N_x, K_yz)
complex*16 a1s(N_y), a1(N_y^+), a2(N_z), a3(N_x/2 + 1), a2s(N_z)
integer yz_comm, zx_comm
common/cbpar1/ yz_comm, zx_comm
integer kproc_yz, kproc_zx
parameter ( kproc_yz = N_proc/(N_x/2), kproc_zx = N_x/2 )
complex*16 catyz(K_zx, K_yx,kproc_yz), cat2yz(K_zx, K_yx,kproc_yz)
complex*16 catzx(K_yz,kproc_zx), cat2zx(K_yz,kproc_zx)

C++++++++++++++++++++++++++++++++++++++++++++++++++++++
C chebyshev back-transform in y-direction and reordering of arrays
C add zeros for proper data communication
C++++++++++++++++++++++++++++++++++++++++++++++++++++++
call perfon(41)
isign = 1
```

```
do k = 1, K_zx
  do iy = 1, N_y
    a1s(iy) = as(iy,k)
  enddo
  call cheb( a1s, a1, N_y, isign )
  do iy = N_y + 1, N_y^+
    a1(iy) = dcmplx(0.0d0,0.0d0)
  enddo
  do iproc = 1, kproc_yz
    do iyeff = 1, K_yx
      iy = iyeff + (iproc-1)*K_yx
      catyz(k,iyeff,iproc) = a1(iy)
    enddo
  enddo
enddo
call perfoff(41)

C++++++++++++++++++++++++++++++++++++++++++++++++++
C alltoall between y and z transform
C++++++++++++++++++++++++++++++++++++++++++++++++++
nt = K_yx * K_zx
call perfon(16)
call mpi_barrier( yz_comm, ierr )
call perfoff(16)
call perfon(6)
call mpi_alltoall( catyz, nt, mpi_double_complex, cat2yz, nt,
                   mpi_double_complex, yz_comm, ierr )
call perfoff(6)

C++++++++++++++++++++++++++++++++++++++++++++++++++
C FFT back-transform in z-direction and reordering of arrays
C++++++++++++++++++++++++++++++++++++++++++++++++++
call perfon(42)
isign = 1
do k = 1, K_yx
  do iproc = 1, kproc_yz
    do izeff = 1, K_zx
      iz = izeff + (iproc-1)*K_zx
      a2(iz) = cat2yz(izeff,k,iproc)
    enddo
  enddo
  call myzzfft( isign, N_z, a2, a2s )
  do iz = 1, N_z
    iproc = ( (iz-1) + (k-1)*N_z ) / K_yz
```

```
      izy = iz + (k-1)*N_z - iproc*K_yz
      catzx(izy,1+iproc) = a2(iz)
   enddo
enddo
call perfoff(42)

C++++++++++++++++++++++++++++++++++++++++++++++++++
C alltoall between z and x transform
C++++++++++++++++++++++++++++++++++++++++++++++++++
nt = K_yz
call perfon(15)
call mpi_barrier( zx_comm, ierr )
call perfoff(15)
call perfon(25)
call mpi_alltoall( catzx, nt, mpi_double_complex, cat2zx, nt,
                   mpi_double_complex, zx_comm, ierr )
call perfoff(25)

C++++++++++++++++++++++++++++++++++++++++++++++++++
C FFT transform in x direction and reordering of arrays
C++++++++++++++++++++++++++++++++++++++++++++++++++
call perfon(43)
isign = 1
do k = 1, K_yz
   do ix = 1, N_x/2
      a3(ix) = cat2zx(k,ix)
   enddo
   call myzrfft( isign, N_x, a3, ap(1,k) )
enddo
call perfoff(43)

return
end subroutine xyzfftsp
```

References

[1] A. N. Beris, C. D. Dimitropoulos, Pseudospectral simulation of turbulent viscoelastic channel flow, Comp. Meth. Appl. Mech. Eng. 180 (1999) 365–392.

[2] C. Calvin, Implementation of parallel FFT algorithms on distributed memory machines with a minimum overhead of communication, Parallel Computing 22 (1996) 1255–1279.

[3] J. W. Cooley, J. W. Tukey, An algorithm for the machine calculation of complex

Fourier series, Mathematics of Computation 19 (1965) 297–301.

[4] C. H. Crawford, C. Evangelinos, D. J. Newman, G. E. Karniadakis, Parallel benchmarks of turbulence in complex geometries, Comp. Fluids. 25 (1996) 677–698.

[5] P. Dmitruk, L. P. Wang, W. H. Matthaeus, R. Zhang, D. Seckel, Scalable parallel FFT for spectral simulations on a Beowulf cluster, Parallel Computing 27 (2000) 1921–1936.

[6] A. Dubey, D. Tessera, Redistribution strategies for portable parallel fft: a case study, Concurrency Computat.: Pract. Exper. 13 (2001) 209–220.

[7] P. Moin, J. Kim, On the numerical simulation of time-dependent viscous incompressible fluid flows involving solid boundaries, J. Comput. Phys. 35 (1980) 381–392.

[8] S. A. Orszag, L. C. Kells, Transition to turbulence in plane Poiseuille and plane Couette flow, J. Fluid Mech. 96 (1980) 159–205.

[9] R. Sureshkumar, A. N. Beris, R. A. Handler, Direct numerical simulation of turbulent channel flow of a polymer solution, Phys. Fluids 9 (1997) 743–755.

[10] P. N. Swarztrauber, Symmetric FFTs, Mathematics of Computation 47 (1986) 323–346.