1

Paper 1

Spectral simulations of wall-bounded flows on massively-parallel computers

By Qiang Li, Philipp Schlatter & Dan S. Henningson

Linné Flow Centre, KTH Mechanics
SE-100 44 Stockholm, Sweden

Technical report

1. Introduction

Due to the rapid development of high-performance super-computers for the last several decades, highly resolved time-dependent numerical simulations, i.e. direct numerical simulation (DNS) and large-eddy simulation (LES), have become an important tool for transition and turbulence research (Moin & Mahesh 1998; Sagaut 2005).

Such super-computers can be primarily classified into two groups with respect to the architecture of the processor: vector processor and scalar processor. A vector processor, or array processor, is designed with the so-called vector registers which are able to operate on multiple data elements simultaneously, thus the operations are done in parallel at least to some extent. In contrast, a scalar processor has registers for data as a scalar quantity and in principle only processes one data element at a time. Vector processors were widely used to form the basis of most super computers in the 1980s and 1990s, but have nearly disappeared in super-computers nowadays. Another classification of super-computers is according to the memory configuration. Shared and distributed memory are the two main memory configurations. In the former case, all the processors share the same memory while in the latter one, each processor has its own memory so that data has to be sent and received if used by another processor. According to the categories above, there are four different combinations of the types of the computers as shown in Table 1. Throughout this report, only the first combination, i.e. scalar processors with distributed memory, is considered and discussed since this is the most dominant architecture in use with most Linux clusters at the moment. The term "processor" means central processing unit (CPU) and will alternatively be termed "core".

	distributed memory	shared memory
scalar processor	1	2
vector processor	3	4

TABLE 1. The four combinations of the types for the parallel computers.

The simplest parallel computing architecture might be the multi-core architecture usually consisting of two or more independent cores. The cores may or may not share the memory and message passing or shared memory inter-core communication methods may be implemented. Another type is a distributed memory computer system, e.g. a computer cluster, consisting of a group of independent computers which are connected by network such that they can work together and can be viewed as a single parallel machine. A cluster has usually about 200 to 300 processors in total. However, in order to obtain higher performance to solve large-size problems, a massively parallel computer system has to be used. A massively parallel computer system is a single machine with a very large number of processors, usually more than 1000 processors. In such a system, a single processor may have a lower performance compared to a processor of a regular cluster, but this is compensated by the larger amount of processors. All the processors are connected via a high speed interconnection. Therefore, the overall performance might be better than a common cluster depending on the application.

Once equipped with a modern parallel computer, the availability of an efficient numerical code for simulating turbulent flow becomes more important. An efficient numerical code (**SIMSON**) to solve the Navier–Stokes equations for incompressible channel and boundary-layer flows has been developed at KTH Mechanics for the last years, see the reports by Lundbladh *et al.* (1992), Lundbladh *et al.* (1999) and Chevalier *et al.* (2007). The numerical method is based on a standard fully spectral Fourier/Chebyshev discretization, leading to high numerical accuracy and efficiency. The nonlinear convective terms are evaluated pseudo-spectrally in physical space using fast Fourier transforms (FFT) to avoid the evaluation of convolution sums. Aliasing errors are removed by using the 3/2-rule (Canuto *et al.* 1988) in the wall-parallel directions. The governing equations are then solved and the prescribed boundary conditions are applied in Fourier/Chebyshev space. Time integration is done by a mixed third order Runge–Kutta/Crank–Nicolson method. The code can be run in either temporal mode or spatial mode. It also supports disturbance formulation and linearised formulation. Multiple passive scalars, e.g. a temperature field, can be solved together with the velocity field. For boundary-layer flows, the "fringe region" is added to fulfil the periodic boundary condition in the streamwise direction (Nordström et al. 1999).

2. Parallelisation

2.1. *Code structure*

The numerical code **SIMSON** is written in FORTRAN 77 and the coarse structure can be divided into four steps (Chevalier *et al.* 2007). In the first step, initialisation of the flow solver is done, e.g. reading in the input files, setting the initial parameter values, opening the output files, etc. In the second step, the time integration loop is started and the computations in physical space, i.e. the evaluation of non-linear terms, are executed. In the third step, the linear

part of the evolution equations are solved in Fourier/Chebyshev space and the time-stepping parameters are recalculated for the next time step. In the last step, the output files are written after the time integration loop is finished. The major computational effort of the code is in the subroutines `nonlinbl` (step 2) in which the nonlinear terms are calculated in physical space and `linearbl` (step 3) in which the linear part of the governing equations are solved and the boundary conditions are evaluated in Fourier/Chebyshev space, see also Figure 1. For a typical run in serial mode, it turns out that about 55% of the total execution time is spent in subroutine `nonlinbl` and 40% in subroutine `linearbl`.

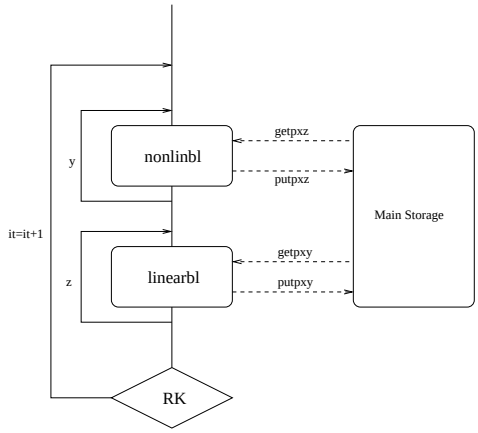

FIGURE 1. The main structure of the code for the 1D parallelisation.

2.2. *Domain decomposition*

The most straightforward parallelisation method is to use data parallelisation or domain decomposition through which the work has been divided among the processors so that each processor has approximately the same amount of work to do. Thus the efficiency is maximised and therefore the overall performance is greatly increased.

2.2.1. *1D parallelisation*

The main structure of the code for the 1D parallelisation is shown in Alvelius & Skote (2000). The computations are done plane by plane, i.e. in an xy-plane in `linearbl` and an xz-plane in `nonlinbl` where x, y and z are the spatial coordinates representing the streamwise, wall-normal and spanwise direction, respectively. Outside these subroutines, there is a loop over the third direction, i.e. z for `linearbl` and y for `nonlinbl`. This means that the calculation in each plane is independent of any other one. If one processor calculates one plane at each time, as many processors as there are planes can be used running in parallel. As seen from Figure 1, in order to get the data from the main

storage onto planes, the subroutines `getpxz` for `nonlinbl` and `getpxy` for `linearbl` are called. The corresponding subroutines for putting data back to the main storage after the calculations in `nonlinbl` and `linearbl` are `putpxz` and `putpxy`, respectively.

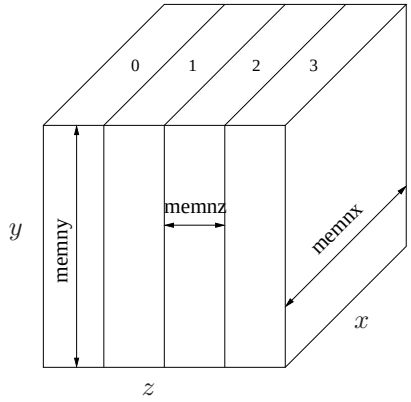

FIGURE 2. Data distribution among all the processors.

The code supports parallelisation with both shared memory (OpenMP) and distributed memory (MPI). Here only the distributed memory machines based on MPI communication are discussed in detail. The main storage is distributed in the z-direction only; a sketch of that distribution is shown in Figure 2. The data in the horizontal plane, i.e. xz-plane, is stored in Fourier space while the data in the wall-normal direction is in physical space. The total number of the processors is denoted by $nproc$. In the sketch $nproc = 4$ is chosen as an example. Then the amount of data stored on each processor is $2 \times memnx \times memny \times memnz$ where $memnx = \frac{nx}{2}$, $memny = nyp$ and $memnz = \frac{nz}{nproc}$. nx, nyp and nz are the number of collocation points in the streamwise, wall-normal and spanwise direction, respectively. $memnx$ is $\frac{nx}{2}$ becuase complex Fourier modes are considered; the factor 2 represents the real and imaginary parts. nyp has to be an odd number since it represents the total number of Chebyshev modes. By distributing the data in this way, the calculations in the subroutine `linearbl` are easily done in parallel since each processor has full access to the data on one xy-plane in each z-loop. Note that the subroutines `getpxy` and `putpxy` are called by all the processors at the same time and they operate on $nproc$ wall-normal planes seperated in z in each z-loop without involving communication between any two processors. However, when doing the calculations in the subroutine `nonlinbl`, in order to obtain full access to the data on an xz-plane, each processor has to collect data from all the other processors. The data collection is accomplished by calling the subroutines `getpxz` and `putpxz`. This global data transfer gives rise to a significant amount of communication among all the processors. In fact, the majority of the total communication among the processors happens in the

subroutine `nonlinbl`, more precisely in the subroutines `getpxz` and `putpxz`. As similarly for the subroutines `getpxy` and `putpxy`, the subroutines `getpxz` and `putpxz` are also called by all the processors at the same time but they operate on *nproc* consecutive wall-parallel planes in each y-loop.

The data distribution after the global communication is shown in Figure 3. After the global communication in `getpxz`, each processor has full access to the data in one xz-plane and is thus able to perform the 2D FFTs on this xz-plane to transform the data from spectral space to physical space. Since the cost of the communication is relatively high for parallel computers with distributed memory, the best way to implement the 2D FFTs is to perform the FFTs by each processor. Thus 1D FFTs have to be performed twice in both directions on each xz-plane. Following the FFTs, all the data is stored in physical space and then the nonlinear terms can be evaluated pseudo-spectrally. To remove the aliasing errors from the evaluation of the nonlinear terms, the 3/2-rule has been used. Once the calculation of the nonlinear terms is finished, inverse FFTs have to be performed to transform the data back to spectral space. Last, another global communication has to be done in `putpxz` to write the data back to the storage location as shown in Figure 2. Then the calculations in `linearbl` can be easily done. It turns out that the evaluation of FFTs are the main computational cost of the whole code which may take up to 55% of the total execution time for a serial run, i.e. without communication.

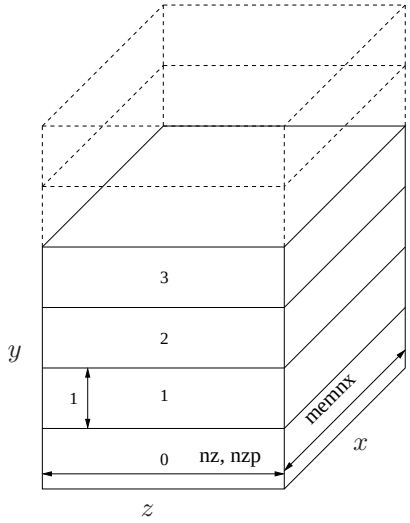

FIGURE 3. Data distribution after the global communication.

For the global communication needed in the subroutines `getpxz` and `putpxz`, two different ways are currently implemented. On the one hand, a self-written version of the global transpose which is based on the explicit point to point communication using MPI commands **MPI_ISEND**, **MPI_WAIT**

and **MPI_RECV** is available. For more details about this implementation, see Alvelius & Skote (2000). On the other hand, an alternative version is to adopt the standard collective communication command **MPI_ALLTOALL**. This standard MPI command transfers a subset of data from all members to

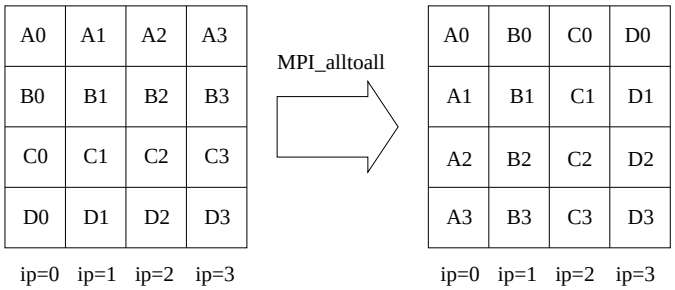

FIGURE 4. The **MPI_ALLTOALL** command illustrated for a group of four processors $ip = 0, 1, 2, 3$.

all members within a group. Each processor sends distinct data to each of the other processor. How the data is transfered is illustrated in Figure 4. As seen from the figure, the **MPI_ALLTOALL** actually does a global transpose of the data among all the members in the group. A performance model for the **MPI_ALLTOALL** is developed, see the later chapter for more detail.

Both versions of implementation for the global communication essentially perform approximately the same in terms of speed and memory requirement. However, if the collective communication version is used, the amount of data stored on each processors is slightly larger and thus the amount of communication compared to the hand-written version is marginally increased. This is because *memny* has to be $(\mathrm{int}(\frac{nyp}{nproc}) + 1) \times nproc$ instead of *nyp* to fulfil the requirement of the **MPI_ALLTOALL** command. The operation "int" means taking only the integer part. Another issue concerning the collective communication version is that user-defined data types including the **MPI_TYPE_STRUCTURE** and **MPI_UB** commands have to be used in order to fit the data structure of the code. These user-defined data types in the subroutines `getpxz` and `putpxz` are *realg*1 and *realg*2. *realg*1, representing the data type as in the Figure 2, has *memnz* blocks, *memnx* elements in each block and $memnx \times memny$ elements between the start of each block. An upper-bound of size *memnx* has to be added to the data type *realg*1 to give the correct displacement between consecutive elements. The upper-bound data type can be considered as a "pseudo-data type" which extends the upper bound of a data type. It does not have any effects on the content or the size of the data type and does not influence the message defined by the data type. What it changes is the spatial extent of the data type in the memory. The data type *realg*2, representing the data type as in the Figure 3,

has also $memnz$ blocks, $memnx$ elements in each block, but $(\frac{nxp}{2}+1)$ elements between the start of each block where $nxp = \frac{3}{2}nx$. An upper-bound of size $(\frac{nxp}{2}+1) \times memnz$ has also to be added to the data type $realg2$. Note that the two data types $realg1$ and $realg2$ have exactly the same size, i.e. they contain the same amount of real numbers. Different blocks of the same data type or different data types can be constructed as a general data type by using MPI command **MPI_TYPE_STRUCTURE**. For more about the user-defined data type, refer to MPI standard documentation.

2.2.2. *2D parallelisation*

The distribution of the main storage in only the spanwise direction introduced in the previous section naturally imposes a restriction to the code, i.e. the upper limit of the maximum possible number of processors to be nz. For typical flow cases, nz is not larger than 256. Nowadays, as the computer clusters become more powerful with more and more processors and cores available, a new way to parallelise the code is strongly needed such that more processors than just nz processors can be used.

There are potentially three possible spatial directions along which the data can be distributed among the different processors. The number of grid points in y, discretized by Chebyshev expansion, is required to be odd and thus not evenly divisible by the number of processors. So choosing y (the wall-normal) direction is out of consideration. Either x (the streamwise) or z (the spanwise) direction can be easily chosen as the direction to parallelise over since the number of the grid points in these two directions are naturally even and can be chosen to be divisible by the number of processors. The number of grid points in the streamwise direction is usually larger than that in the spanwise direction. If the streamwise direction is chosen to be the direction to parallelise over, it is possible to use more processors, however again limited by the number of modes in x. In addition, there might be a local load-levelling when treating the y and z directions with much less collocation points. Therefore, in order to fully exploit massively parallel computer systems with large amounts of processors and without being limited by the number of grid points in either the x or z direction, parallelising in both the x and z directions, i.e. a 2D parallelisation operating on a number of pencils, becomes a natural choice.

Hence the whole field (velocity, pressure or scalar) is distributed in both x and z direction among the different processors. A sketch of the data distribution is shown in Figure 5.

The main structure of the code for the 2D parallelisation is shown in Figure 6. It looks quite similar to the structure of the 1D parallelisation given in Figure 1. As seen from Figure 6, in order to get the data from the main storage onto part of the planes, subroutines `getpxz_z` in `nonlinbl` and `getpxy` in `linearbl` have to be called. The corresponding subroutines for putting data back to the main storage after calculations are `putpxz_z` for `nonlinbl` and `putpxy` for `linearbl`. In a similar way as for the 1D parallelisation, FFTs in both

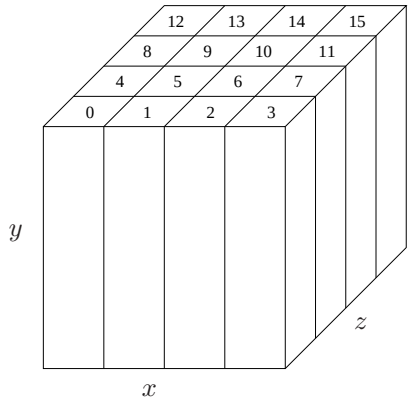

FIGURE 5. Initial data distribution among all the processors.

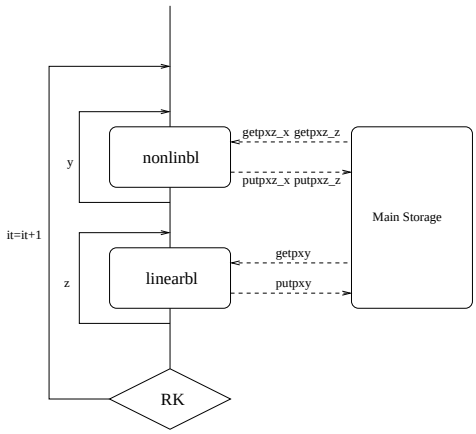

FIGURE 6. The main structure of the code for the 2D parallelisation. Note that there exist FFTs between calls to the subroutines `getpxz_z` and `getpxz_x` and inverse FFTs between calls to `putpxz_z` and `putpxz_x`.

directions in each xz-plane have to be performed by each processor. However, due to the data distribution on each xz-plane now, only the 1D FFT in the spanwise direction is possible to do without further communication. Thus in order to perform the 1D FFT in the streamwise direction, another global data communication has to be performed. This is done by calling the subroutine `getpxz_x`. The corresponding subroutine for putting data back to the temporary storage is `putpxz_x`. The details about both global data communications will be explained in later sections. Note that the global communication will inherently not scale linearly with the number of processors that are involved in the communication. This will pose a challenge for using large number of

processors. Again, most of the total communication of the code is in `nonlinbl` and the main computational effort is the FFTs.

Due to the 2D parallelisation, none of the processors has full access to the data of any whole xz-plane. When carrying out the communication and calculations, the processors are divided into different groups. These groups are different from the default MPI communication group, **MPI_COMM_WORLD**, which contains all the processors. At each time, only a certain fraction of the total number of the processors will be involved in each group to do the communication in each plane and different groups take care of the different planes. In this way all the processors then run in parallel. Again, similar to the 1D parallelisation, the subroutines `getpxz_z`, `getpxz_x`, `putpxz_z` and `putpxz_x` in `nonlinbl` are called by all the processors at the same time and $nprocz$ consecutive wall-parallel planes are treated in each y-loop. Remember that in the 1D parallelisation, the subroutines `getpxy` and `putpxy` in `linearbl` are called by all the processors at the same time and $nproc$ wall-parallel planes are treated simultaneously.

A short description of how the data is transferred in `nonlinbl` is given now. As depicted in Figure 5, initially the field is equally distributed among all the processors, here 16 processors are used as an example. $nproc$, $nprocx$ and $nprocz$ represent the number of the total processors, the processors in the x and z direction, respectively. Thus in this case $nproc = 16$ and $nprocx = nprocz = 4$. Note that with the present implementation of the 2D parallelisation, it is required that the number of processors should be equally distributed along both the x and z directions, i.e. $nprocx = nprocz = \sqrt{nproc}$. The size of the domain is $\frac{nx}{2}$ in x-direction, nz in z-direction and nyp in y-direction where nx, nyp and nz are the same as defined in the 1D parallelisation. Note that the size of the domain in the x-direction is $\frac{nx}{2}$ rather than nx which is due to the fact that the real and imaginary parts of the complex numbers are stored in separate arrays. The amount of data stored on each processor is $memnx \times memny \times memnz$ where $memnx = \frac{nx}{2nprocx}$, $memny = (\mathrm{int}(\frac{nyp}{nprocz}) + 1) \times nprocz$ and $memnz = \frac{nz}{nprocz}$.

For the 2D parallelisation, the global communication is implemented only using the standard collective MPI command **MPI_ALLTOALL**, so $memny$ needs to be $(\mathrm{int}(\frac{nyp}{nprocz}) + 1) \times nprocz$. To make the transfer of data more efficient, user-defined data type rather than the default ones provided by the MPI library have to be used. The data types used in the subroutines `getpxz_z` and `putpxz_z` are $realg1$ and $realg2$. $realg1$ has $memnz$ blocks, $memnx$ elements in each block and $memnx \times memny$ elements between the start of each block. As detailed previously, an upper-bound (**MPI_UB**) of $memnx$ has to be added to the data type $realg1$. The data type $realg2$ has $memnz$ blocks, $memnx$ elements in each block and $memnx$ elements between the start of each block. Note that no explicit upper-bound element is needed here. In the subroutines `getpxz_x` and `putpxz_x`, the data types used are $realg3$ and $realg4$. The data

type $realg3$ has $\frac{nzp}{nprocz}$ blocks, $memnx$ elements in each block and $memnx$ elements between the start of each block while $realg4$ has $\frac{nzp}{nprocz}$ blocks, $memnx$ elements in each block and $(\frac{nxp}{2}+1)$ elements between the beginning of each block, again a upper-bound of size $memnx$ has to be used for $realg4$. Similar to the 1D parallelisation, $realg1$ and $realg2$ have the exact same size and represent the data types before and after the transpose. This is also true for $realg3$ and $realg4$. nxp and nzp are related to the dealiasing and will be defined later.

Before calculating the nonlinear terms, the FFTs in both x and z direction have to be performed as mentioned before. Due to the data storage after `getpxz_z`, the FFTs can be performed in only z-direction. Thus it is necessary to transfer the data among different processors once more such that every processor can perform the FFTs in the x-direction.

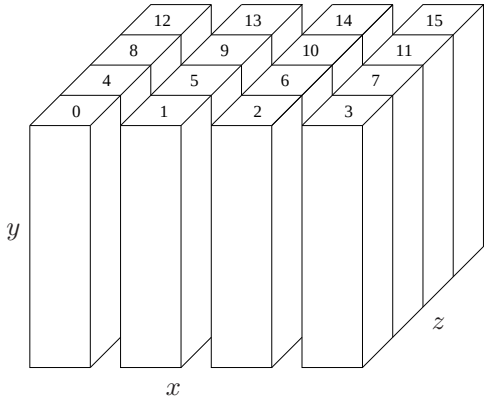

FIGURE 7. The configuration of the groups for the first transpose.

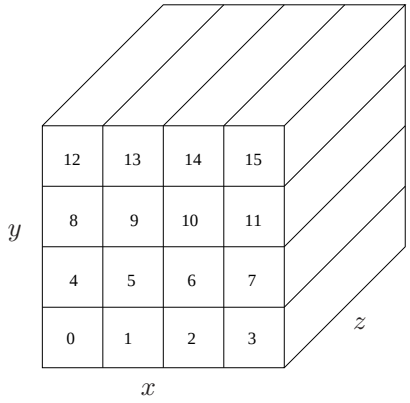

FIGURE 8. The data distribution after `getpxz_z`.

First, the processors are separated into *nprocx* groups and each group contains *nprocz* processors. The creation of groups is shown in Figure 7. Then the standard MPI command **MPI_ALLTOALL** is used to transpose the data within each group. This is done by calling the subroutine `getpxz_z`. The data storage configuration after the transpose is shown in Figure 8. For convenience only the first consecutive *nprocz* planes in the wall-normal direction are plotted. The layout of the other wall-parallel planes is just a repetition of the first *nprocz* planes. The creation of groups and the data storage for the second global data transpose are shown in Figures 9 and 10 and will be discussed later. Now, aliasing errors which are caused by representing higher wavenumber data at lower wavenumbers have to be considered. A detailed description can be found in e.g. Canuto *et al.* (1988). In order to eliminate the aliasing errors, the standard 3/2-rule is employed which means to expand the original grid and pad high-wavenumber part with zeros. The original grid, as shown in Figure 11 (a), is therefore expanded to a finer grid which has a dimension of $\frac{nx}{2}$ in the x-direction and nzp in the z-direction where $nzp = \frac{3}{2}nz$, see Figure 11 (b). If nzp is not divisible by $nprocz$, the dealiasing grid needs to be $(\frac{nzp}{nprocz} + \min(1, \mathrm{mod}(nzp, nprocz))) \times nprocz$, where the operation "min" is to take the minimum and "mod" is the modulo operation. Then part of the data denoted by the "slashed area" is moved to the upper part of the fine grid and the middle part is padded with zeros denoted by the $\bigcirc$. The so-called "oddball mode" is also set to zero which is denoted by the grided lines, in Figure 11. For more details about the oddball mode, refer to Chevalier et al. (2007). At the end of this step, each processor can perform the backward FFTs in the z-direction on the fine grid.

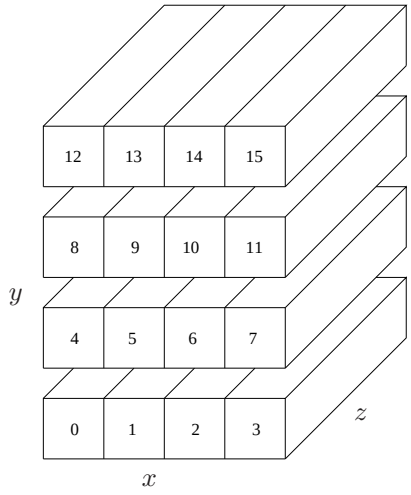

FIGURE 9. The configuration of the groups for the second transpose.

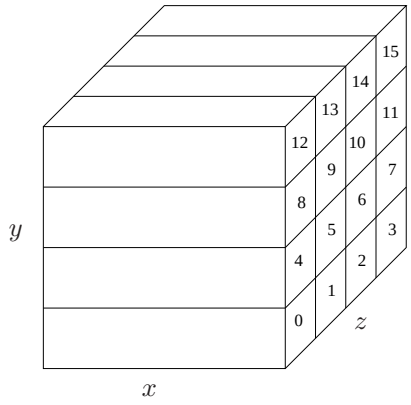

FIGURE 10. The data distribution after `getpxz_x`.

In a second step, *nprocz* new groups are created and each group contains *nprocx* processors. This is to prepare for the FFTs in the x-direction to be performed. The newly created groups are sketched in Figure 9. Then another transpose of the data in each group can be executed. This second data transpose is done by calling subroutine `getpxz_x`. After the data transpose, the data storage configuration on each processor is shown in Figure 10. The grid is expanded in the x-direction this time in order to account for the 3/2-rule and the dimension of the dealiasing grid is $\frac{nxp}{2}+1$ in the x-direction where $nxp = \frac{3}{2}nx$. In the spanwise direction, nzp are continued to be used. This grid is shown in Figure 11 (c). The additional grid point in streamwise direction is due to requirement of the FFT. For a complex to real FFT, two more points, i.e. two zeros for the imaginary parts, are added. One is for the zero frequency component and the other for the Nyquist frequency component. Since the real and imaginary parts are stored separately, this leads to the additional point in the x-direction. The right-hand part is padded with zeros and the oddball mode is also set to zero. Last, the FFTs in the x-direction on each processor are performed. After the FFTs, all the data is in physical space and the nonlinear terms can be calculated. Afterwards, all the data has to be transformed back to the spectral space to match the configuration shown in Figure 5. This will be exactly a reverse process of the steps mentioned above.

In the subroutine `linearbl`, the data is put onto xy-planes by calling the subroutine `getpxy`. As shown in Figure 12, each processor only calculates a portion of one xy-plane at each time, therefore the computation is carried out in parallel and there is no communication involved between any two processors as the same for 1D parallelisation. After the calculation, the data is put back to the main memory through subroutine `putpxy`, again without requiring any MPI communication.

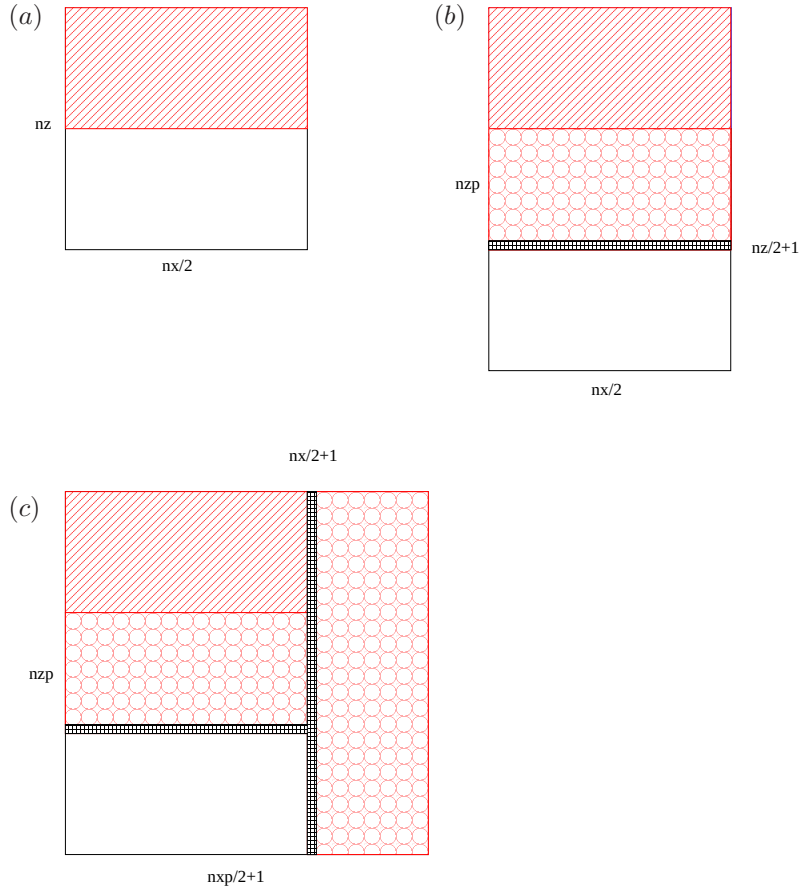

FIGURE 11. The dealiasing grid and the oddball location in a xz-plane. (a) Grid before dealiasing, (b) Grid after dealiasing in z, (c) Grid after dealiasing in z and x.

2.3. *Amount of communication*

2.3.1. *1D parallelisation*

The main communication between the processors is in the nonlinear part, i.e. subroutine `nonlinbl`. Consider in the following only the velocity field, i.e. without pressure nor scalar fields. Then for each call to `nonlinbl`, five variables (three velocity components and two components of the vorticity) are collected from the main memory by calling subroutine `getpxz` and three variables (three vorticity components) are put back to the main memory via the subroutine `putpxz` for the 1D parallelisation. Thus a total of eight variables have to be communicated which is independent of the implementation of the global communication. Here only the version of collective communication using standard

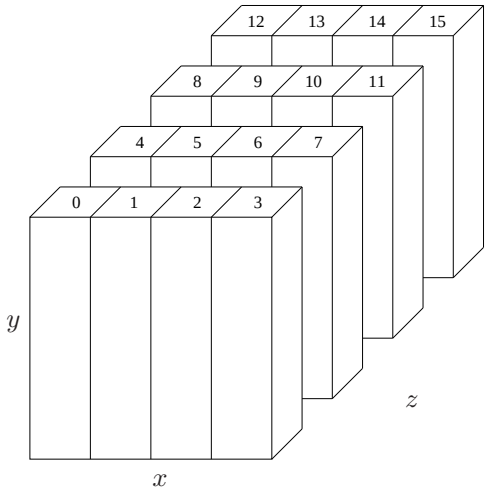

FIGURE 12. The data distribution in `linearbl`.

command **MPI_ALLTOALL** is discussed. For the hand-written version based on explicit point-to-point communication, more details can be found in Alvelius & Skote (2000). However, the amount of data to be communicated is roughly the same.

Each processor performs calculation on approximately $(\mathrm{int}(\frac{nyp}{nproc})+1)$ xz-planes. The data types of the message are the user-defined data types, i.e. $realg1$ and $realg2$. Both of them have a size of $memnx \times memnz$ or $\frac{nx}{2} \times \frac{nz}{nproc}$. The number of messages that one processor collects is

$$2 \times (nproc-1) \times (\mathrm{int}(\frac{nyp}{nproc})+1) \; . \tag{1}$$

Note that each processor collects data from all the other processors and both the real and imaginary part gives a factor of 2 for the total number of messages which has to be communicated. This gives that for one flow variable each processor needs to collect

$$(nproc-1) \times (\mathrm{int}(\frac{nyp}{nproc})+1) \times nx \times \frac{nz}{nproc} \tag{2}$$

real numbers from all the processors at each Runge-Kutta substep. Since each number has double precision, i.e. 8 bytes for each real number, the total amount of data that needs to be communicated for a single processor for all 8 variables and one iteration (Runge-Kutta substep) is

$$8 \times 8 \times (nproc-1) \times (\mathrm{int}(\frac{nyp}{nproc})+1) \times nx \times \frac{nz}{nproc} \tag{3}$$

bytes. The amount of data that each processor has to send at the same time is as large as the amount that has to be received.

2.3.2. *2D parallelisation*

For the 2D parallelisation, the situation is slightly different than that for the 1D case. As explained earlier, the data has to be transposed twice in order to perform the FFTs. For each call to `nonlinbl`, five variables (three velocity components and two vorticity components) are collected from the main memory with `getpxz_z` and later three variables (the evaluated non-linear terms) are put back to the main memory with `putpxz_z` within `nonlinbl`. Six variables (three velocity and three right-hand side components) are collected from the temporary storage via `getpxz_x` and only three variables (three vorticity components) are put back with `putpxz_x` to the temporary storage. In the 2D case, each processor performs calculations on approximately $(\mathrm{int}(\frac{nyp}{nprocz}) + 1)$ xz-planes. The data types used for the message are $realg1$ and $realg2$ in `getpxz_z` and `putpxz_z` while $realg3$ and $realg4$ in `getpxz_x` and `putpxz_x`. Both $realg1$ and $realg2$ have a size of $\frac{nx}{2nprocx} \times \frac{nz}{nprocz}$ and the size of $realg3$ and $realg4$ is $\frac{nx}{2nprocx} \times \frac{nzp}{nprocz}$. Following the same steps as for the 1D parallelisation, the number of messages needs to be communicated for the first and the second transpose are

$$2 \times (nprocz - 1) \times (\mathrm{int}(\frac{nyp}{nprocz}) + 1) \tag{4}$$

and

$$2 \times (nprocx - 1) \times (\mathrm{int}(\frac{nyp}{nprocz}) + 1) \tag{5}$$

respectively. The total amount of data which has to be collected for one processor for each Runge-Kutta substep and all 17 variables is

$$\begin{aligned} &8 \times (nprocz - 1) \times (\mathrm{int}(\frac{nyp}{nprocz}) + 1) \times (8 \times \frac{nx}{nprocx} \times \frac{nz}{nprocz}) + \\ &8 \times (nprocx - 1) \times (\mathrm{int}(\frac{nyp}{nprocz}) + 1) \times (9 \times \frac{nx}{nprocx} \times \frac{nzp}{nprocz}) \end{aligned} \tag{6}$$

bytes. Again the same amount of data has to be sent by each processor at the same time.

2.3.3. *Communication comparison*

In this section, the total amount of messages sent by each processor (including the messages it sends to itself) at each Runge-Kutta substep are compared for both 1D and 2D parallelisations with a fixed size for nx, nyp and nz. The results are plotted in Figure 13. For a fixed resolution, it is clear that the 2D parallelisation does involve more communication than the 1D parallelisation. Nevertheless, the overall performance of the 2D parallelisation is better. The main reason is that the processors are divided into groups and each processor only communicates within its own group and not globally. All the groups run in parallel and the influences due to the congestions from the one group to the other are small. Therefore, the total performance is better for the 2D parallelisation. The ratio between the messages sent by the 1D and 2D parallelisation is

also plotted in Figure 13. As the number of processors increases, the ratio decreases slowly for small number of processors. However, for a typical simulation case, this ratio is always larger than unity.

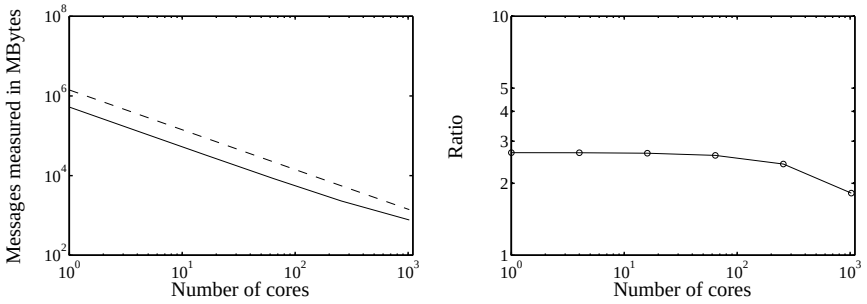

FIGURE 13. A comparison of the total amount of communication for the 1D and 2D parallelisation for a size of $2048 \times 2049 \times 2048$. *Left*: Messages that being sent for the two parallelisations, ——1D parallelisation, - - -2D parallelisation, *Right*: The ratio of the messages being sent for the two parallelisations.

3. Performance analysis

For the parallel computing, the maximum speed-up of a code can be obtained is

$$\text{speed-up} = \frac{1}{F + (1-F)/N} \tag{7}$$

where F is the fraction of a calculation that is sequential, i.e. the part that cannot benefit from parallelisation, and thus $(1-F)$ is the fraction that can be parallelised where N is the number of processors used for the calculation. This is usually referred as the Amdahl's law (Amdahl 1967). Ideally, if the whole code can run in parallel, a maximum of speed-up of N can be obtained, i.e. the so-called linear speed-up. However, a real application code always has some part which cannot be parallelised in practice. This will lead to a maximum speed-up by $\frac{1}{F}$ for large N. Therefore, a lot of effort is devoted to reducing F to a value as small as possible. Additionally, even the parallelisible parts of a code will usually not scale linearly, but rather at a reduced rate.

A benchmark for the parallelisation of **SIMSON** has been performed on a BlueGene/L machine manufactured by IBM. The building block of the BlueGene/L system is the compute card consisting of 1 node. On each node there are two processors (cores) with up to 1024 Mbytes of memory. Each processor is an embedded 32-bit PowerPC 440 with a clock frequency of 700 MHz. The system-on-chip design contains three interconnections: Gbit Ethernet, global tree (collective communication) and 3D Torus. Two execution modes are available for each node. One is the co-processor mode (CO), i.e. only one MPI

process per node with maximum 1024 Mbytes of memory per process. In this mode, one processor is dedicated to communication and the other to general processing. The other mode is the virtual-node mode (VN), i.e. two MPI processes per node with maximum 512 Mbytes of memory per process. In this case, each processors uses half of the resources and works as an independent processor. A detailed description about the BlueGene/L machine can be found in the online manual (Mullen-Schultz & Sosa 2007).

3.1. *Performance of the* ***MPI_ALLTOALL***

As the main communication pattern for the communication in the present code and also one of the most important collective communication used in scientific computing, it is necessary and crucial to understand how the **MPI_ALLTOALL** collective operation performs. Therefore, a performance test has been carried out and a performance model for **MPI_ALLTOALL** is developed.

During an **MPI_ALLTOALL** operation, each process holds $m \times n$ data, where n is the number of processes involved in the communication and m is the message size (measured in bytes) sent to or received from one other process. Following the linear point-to-point model, the communication time spent on one process is

$$T = (n-1) \times (\alpha + m \times \beta^{-1}) \tag{8}$$

where α is the start-up time, i.e. the latency between two processes, and β is the bandwidth of the link. Unfortunately, this model does not work so well when intensive communication happens as pointed out by Steffenel *et al.* (2007). Thus, network contention-aware communication models are suggested. However, due to the non-deterministic behaviour of the network contention, some authors suggested a few techniques to adapt the existing models, e.g. Clement *et al.* (1996) introduced the contention factor to correct the performance model. Bruck *et al.* (1997) came up with the similar ideas to use the slowdown factor. Recently, Steffenel *et al.* (2007) proposed a new model consisting of determining a contention ratio. For simplicity, the contention ratio is considered to be a constant and only depends on the network characteristics. Following the model by Steffenel *et al.* (2007), the performance model which is an extension of the linear point-to-point model reads

$$T = (n-1) \times (\alpha + \gamma \times m \times \beta^{-1}) \tag{9}$$

where γ is the contention ratio. If N is chosen to represent the total message that has to be sent by all the processes, m would be $\frac{N}{n^2}$. Then the equation (9) will read

$$T = (n-1) \times (\alpha + \gamma \times \frac{N}{n^2} \times \beta^{-1}) \ . \tag{10}$$

Using this performance model of **MPI_ALLTOALL**, a series of tests has been run on a BlueGene/L machine using both VN and CO mode. The total amount of the message to be sent is fixed, i.e. $N = 3072 \times 3072$, with the number of cores involved in the communication varying from 2 to 1024.

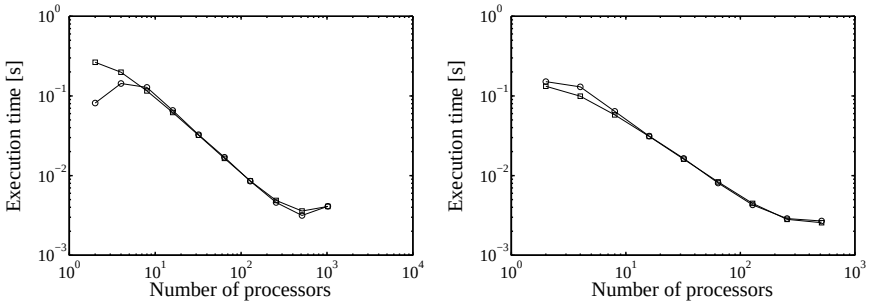

FIGURE 14. Performance of the **MPI_ALLTOALL** for VN and CO mode. ∘ Measurements, □ Model prediction. *Left*: VN mode with $\alpha = 3$ [μs], $\beta = 300$ [Mbytes/s] and $\gamma = 4.29$, *Right*: CO mode with $\alpha = 3$ [μs], $\beta = 300$ [Mbytes/s] and $\gamma = 2.14$.

The results of the execution time spent on one core are shown in Figure 14. As seen from the figure, the model predictions compare well with the measurements for number of cores larger than 4. The discrepancies for the measurements at low number of cores are partially due to the not fully contentioned network and also due to the non-linear affects when the messages sent by each process is sufficiently large. The latency α for both cases are on the order of 1 μs which is consistent with the theoretical value from the BlueGene/L manual. On the other hand, the bandwidth used for both cases are obtained from measurement of a point-to-point communication. The contention ratio for the CO mode is about twice as for the VN mode. The reason is due to the fact that for the VN mode case, both processors (cores) share one link (bus, port) within each node which leads to twice the contention ratio for each core.

3.2. *Code performance*

Two cases of different sizes are tested for both 1D and 2D parallelisations. The smaller case has the resolution of $1024 \times 129 \times 128$ (run1) while the resolution of the larger one is $512 \times 513 \times 512$ (run2). Only the velocity field, i.e. without pressure and scalar fields, is simulated for periodic channel flow, and the code has been run in virtual-node mode (VN). As mentioned before, most of the execution time is spent in the linear and nonlinear parts, i.e. in the subroutines `linearbl` and `nonlinbl`, and the main computational effort is the FFTs in `nonlinbl`. The speed-up plots shown later are thus only based on the execution times from these two subroutines. The execution time per time step, i.e. four Runge-Kutta substeps, is chosen to calculate the speed-up.

In Figure 15 the two parallelisations are compared for the smaller case (run1). Due to the requirement of the 2D parallelisation, i.e. that the processors need to be equally distributed along both wall-parallel directions, the first data point corresponds to 2 nodes or 4 processors. Clearly seen from the figure, the

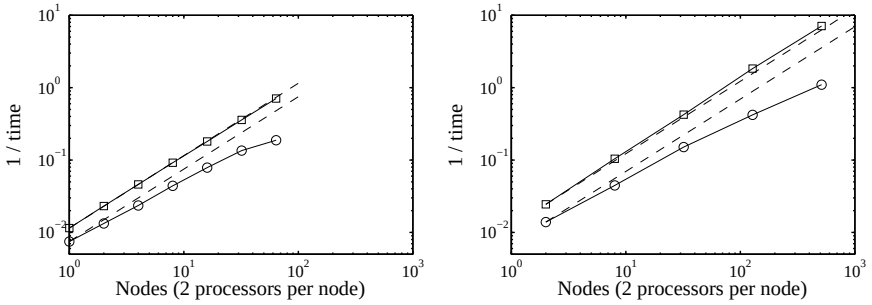

FIGURE 15. Performance of the two parallelisations for the Run1 (run1). - - -Linear speed-up, □ `linearbl`, ○ `nonlinbl`. *Left*: 1D parallelisation, *Right*: 2D parallelisation.

linear part has a linear behaviour, even a slightly super-linear behaviour for the 2D parallelisation. However, the speed-up curves of the nonlinear part for both parallelisations deviate from the linear speed-up curve further and further as more and more processors are used.

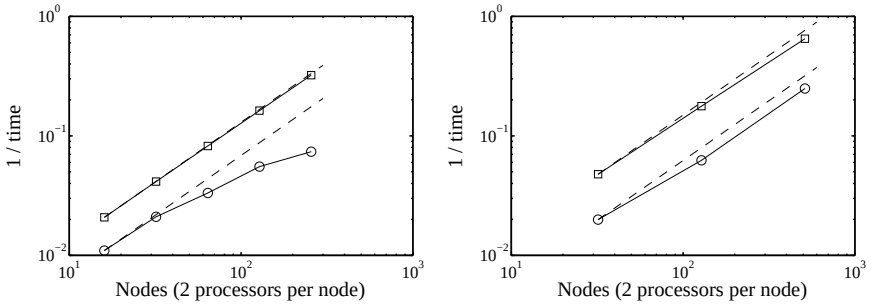

FIGURE 16. Performance of the two parallelisations for the Run2 (run2). - - -Linear speed-up, □ `linearbl`, ○ `nonlinbl`. *Left*: 1D parallelisation, *Right*: 2D parallelisation.

In Figure 16 the larger test case (run2) is shown for the two parallelisations. Note that due to the larger memory requirements, a minimum number of nodes of 16 has to be used for the 1D parallelisation and 32 for the 2D parallelisation. Similarly to the smaller case, the linear parts of both parallelisations still have linear performance. But for the nonlinear parts, a clear difference can be observed for the two parallelisations. As seen from the plots, the 1D parallelisation shows a similar behaviour as for the smaller case reaching a saturation of the performance at 256 node, and the 2D parallelisation has almost linear performance up to 512 nodes.

As known from the previous section, in the nonlinear part, i.e. subroutine `nonlinbl`, the speed-up curve will inherently not scale linearly with the

number of processors used since a global data transpose has to be performed. However, there is another important fact which leads to this suboptimal behaviour, i.e. the particular number of discretization points in the wall-normal direction. Since a Chebyshev representation in the wall-normal direction is used, the number of grid points in the y-direction is not divisible by the number of processors, i.e. $\frac{nyp}{nproc}$ for the 1D parallelisation or $\frac{nyp}{nprocz}$ for the 2D parallelisation is not an integer number. Therefore, for the last y-loop outside the subroutine `nonlinbl`, not all the processors are active. The last instance of the loop is not completely parallelised. If *nproc* or *nprocz* is of the same order of magnitude as *nyp*, this will have very large effect on the performance. This is what are observed from the 1D parallelisation. If *nproc* or *nprocz* is relatively small compared to *nyp*, the effect form this last y-loop is much smaller. This is clearly demonstrated by the 2D parallelisation. In general, for the 2D parallelisation, since *nprocz* is only the square root of *nproc*, the effect of the last y-loop should be smaller than the 1D parallelisation. However, if increasing the number of processors for the 2D parallelisation even more, the same problem will appear eventually.

To investigate the behaviour of subroutine `nonlinbl` in more detail, the total execution time of `nonlinbl` is split into two parts: one part only involves communication (t_{com}), the other part is the computation (t_{ser}), e.g. the FFTs. Hence the relation $t_{nonlinbl} = t_{com} + t_{ser}$ is obtained. Note that these times correspond to the wall-clock time on one processor, and that all processors are synchronised using appropriate barriers.

In Figure 17, the time for communication and computation in subroutine `nonlinbl` are compared for both parallelisations. As expected, for both parallelisations the speed-up curves for the communication part do have suboptimal behaviours, i.e. not scaling linearly. It is clear by comparing the computational part for the two parallelisations that the effect of the last y-loop has a smaller effect for the 2D parallelisation. Remember that the main computational effort is the FFTs which indicates that the time spent by the computation should be larger than that by the communication and this is also observable from the plots for the 1D parallelisation. Note that since $\frac{1}{time}$ is plotted, the lower the data point the large the execution time is. As the number of processors is increased, more and more communication will happen which means that t_{com} will take a larger and larger portion of the total time spent in `nonlinbl`. For the 2D parallelisation, there is even more communication happening, so the growth of t_{com} is even faster. As seen from the plots, t_{com} can become larger than t_{ser} even at small number of processors, e.g. 32 nodes (64 cores).

A summary of the execution time (measured in wall-clock time [s]) of one time step on one processor for both the small and the large runs is shown in Tables 3.2–3.2. Note that the numbers in the bracket are the corresponding percentage of the total execution time (t_{sum}).

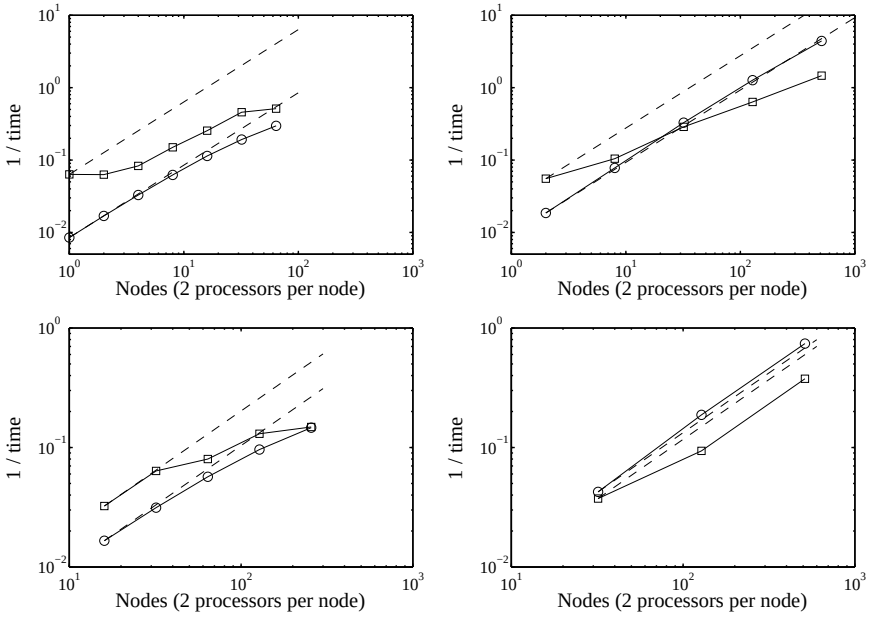

FIGURE 17. Performance of t_{com} and t_{ser} in `nonlinbl`. - - -Linear speed-up, □ t_{com}, ∘ t_{ser}. *Top*: Run1, *Bottom*: Run2. $Left$: 1D parallelisation, *Right*: 2D parallelisation.

	`linearbl`	`nonlinbl`			
nproc	$t_{linearbl}$	t_{ser}	t_{com}	$t_{nonlinbl}$	t_{sum}
1	147.5 (42.3%)	186.0 (53.4%)	15.0 (4.3%)	201.1 (57.7%)	348.6
2	87.0 (39.5%)	117.6 (53.4%)	15.8 (7.2%)	133.3 (60.5%)	220.4
4	43.2 (36.5%)	59.2 (50.0%)	15.9 (13.5%)	75.1 (63.5%)	118.3
8	21.7 (33.8%)	30.4 (47.4%)	12.0 (18.7%)	42.5 (66.1%)	64.2
16	10.9 (32.3%)	16.0 (47.7%)	6.7 (19.8%)	22.7 (67.5%)	33.7
32	5.5 (30.4%)	8.8 (48.1%)	3.9 (21.5%)	12.7 (69.6%)	18.3
64	2.8 (27.4%)	5.2 (51.1%)	2.2 (21.4%)	7.4 (72.5%)	10.2
128	1.4 (20.9%)	3.4 (50.0%)	2.0 (28.9%)	5.3 (78.9%)	6.8

TABLE 2. Execution time of the Run1 using the 1D parallelisation.

3.3. *Modelling*

To eliminate the effect of last y-loop on the performance, a factor k is multiplied with each sampled data from `nonlinbl`. This correction is just to help to develop the model of the whole performance of the code. Since the effect for the 2D parallelisation is much smaller than that for the 1D parallelisation, this correction is only done for the 1D parallelisation. The correction factor k is a

	`linearbl`	`nonlinbl`			
nproc	$t_{linearbl}$	t_{ser}	t_{com}	$t_{nonlinbl}$	t_{sum}
4	40.9 (36.2%)	53.8 (47.7%)	18.1 (16.0%)	71.9 (63.7%)	112.8
16	9.6 (29.9%)	12.8 (40.0%)	9.6 (29.8%)	22.4 (69.8%)	32.1
64	2.4 (26.3%)	3.0 (33.9%)	3.5 (38.5%)	6.6 (73.6%)	9.0
256	0.5 (18.6%)	0.8 (26.9%)	1.6 (53.7%)	2.4 (80.9%)	2.9
1024	0.1 (12.5%)	0.2 (20.0%)	0.7 (60.4%)	0.9 (80.5%)	1.1

TABLE 3. Execution time of the Run1 using the 2D parallelisation.

	`linearbl`	`nonlinbl`			
nproc	$t_{linearbl}$	t_{ser}	t_{com}	$t_{nonlinbl}$	t_{sum}
32	48.1 (34.5%)	60.3 (43.2%)	30.1 (22.2%)	91.2 (65.5%)	139.4
64	24.1 (33.6%)	31.8 (44.4%)	15.7 (21.9%)	47.6 (66.3%)	71.8
128	12.2 (28.8%)	17.6 (41.5%)	12.5 (29.5%)	30.1 (71.1%)	42.3
256	6.1 (25.3%)	10.4 (42.8%)	7.7 (31.5%)	18.1 (74.3%)	24.3
1024	3.1 (18.6%)	6.8 (40.8%)	6.7 (40.3%)	13.6 (81.1%)	16.7

TABLE 4. Execution time of the Run2 using the 1D parallelisation.

	`linearbl`	`nonlinbl`			
nproc	$t_{linearbl}$	t_{ser}	t_{com}	$t_{nonlinbl}$	t_{sum}
64	20.9 (29.4%)	23.5 (33.0%)	26.7 (37.5%)	50.2 (70.5%)	71.3
256	5.6 (26.0%)	5.3 (24.6%)	10.6 (49.2%)	16.0 (73.9%)	21.7
1024	1.5 (27.1%)	1.4 (23.8%)	2.7 (46.9%)	4.0 (70.7%)	5.7

TABLE 5. Execution time of the Run2 using the 2D parallelisation.

function of the number of processors and is defined as

$$k = \frac{nyp}{(\mathrm{int}(\frac{nyp}{nproc}) + 1) \times nproc} \ . \tag{11}$$

After the correction, the total time spent in subroutines `linearbl` and `nonlinbl` are shown in Figure 18 and the communication and computation parts within `nonlinbl` are shown in Figure 19, respectively. Except the communication part, all the other parts, i.e. `linearbl` and the computation part within `nonlinbl`, have linear speed-up performance. And this is also true for the 2D parallelisation, however without requiring a correction for the last y-loop.

In order to roughly estimate the performance on BlueGene/L without running a simulation, a performance model has been developed for the code of both 1D and 2D parallelisations. Known from the previous section that after eliminating the effects from the last y-loop, only the communication part does not have a linear speed-up behaviour. Therefore this part needs to be modelled. In the model, only the latency, bandwidth and contention ratio are considered to be the most important factors and all the other influences are neglected,

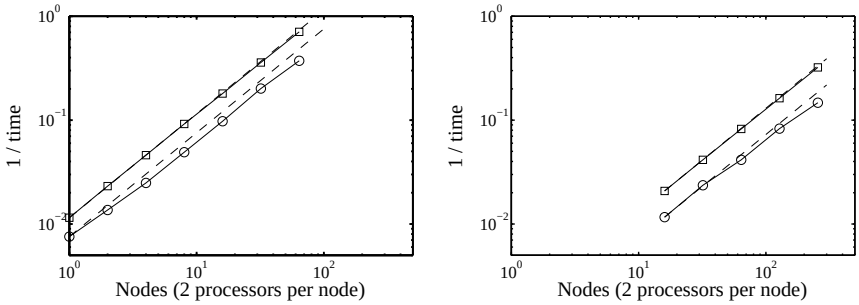

FIGURE 18. Performance of the 1D parallelisation after correction for the 1D parallelisation. - - -Linear speed-up, □ `linearbl`, ∘ `nonlinbl`. *Left*: Run1, *Right*: Run2.

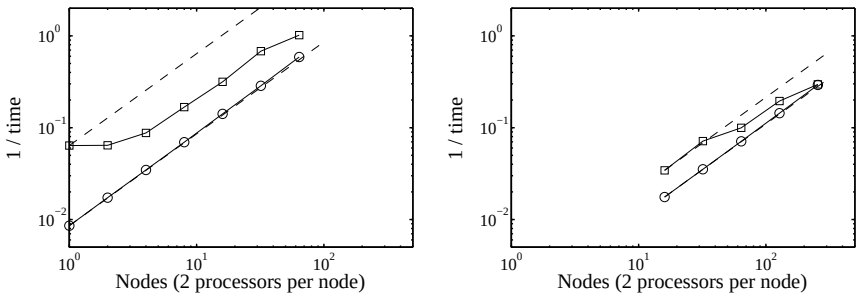

FIGURE 19. Performance of t_{com} and t_{ser} in `nonlinbl` after correction for the 1D parallelisation. - - -Linear speed-up, □ t_{com}, ∘ t_{ser}. *Left*: Run1, *Right*: Run2.

e.g. the topology of the system, the distance between the two communicating processors, etc.

For the **MPI_ALLTOALL** version of the 1D parallelisation, the model of the execution time of communication t_{com} for one Runge-Kutta substep and one variable can be built up as the following steps: First, each processor performs calculation on approximately $(\mathrm{int}(\frac{nyp}{nproc}) + 1)$ xz-planes. Second, for each xz-plane each processor needs to collect

$$2 \times (nproc - 1) \times \frac{nx}{2} \times \frac{nz}{nproc}$$

messages (real numbers) from all the other processors for one variable and Runge-Kutta substep. The associated count for latency of one processor for collecting all these data is

$$2 \times (nproc - 1)$$

times. Note that the factor of 2 is from the real and imaginary part of the data needs to be collected. The execution time of communication t_{com} of one

processor for one time step, i.e. 4 Runge-Kutta substeps, and all the 8 variables is approximately given by

$$t_{com} = 4\times 8\times 2\times(\mathrm{int}(\frac{nyp}{nproc})+1)\cdot(nproc-1)\cdot(\alpha+8\cdot\gamma\cdot\frac{nx}{2}\cdot\frac{nz}{nproc}\cdot\beta^{-1}) \quad (12)$$

where α is the latency $[s]$, β is the bandwidth $[bytes/s]$ and γ is the contention ratio. The number of messages has been multiplied by a factor of 8 to convert to bytes, since double precision is used, i.e. 8 bytes for each real number. The corresponding model for the hand-written version of the 1D parallelisation is also given here as,

$$t_{com} = 4\times 8\times 2\times\frac{nyp}{nproc}\cdot(nproc-1)\cdot(\alpha+8\cdot\gamma\cdot\frac{nx}{2}\cdot\frac{nz}{nproc}\cdot\beta^{-1})\ . \quad (13)$$

There is only very little difference when comparing these two models given by equations (12) and (13). Following the same steps, a model of the execution time of the communication t_{com} for the 2D parallelisation can be developed. For one time step and all variables, the execution time of communication t_{com} is approximately given by

$$t_{com} = 4\times 17\times 2\times(\mathrm{int}(\frac{nyp}{nprocz})+1)\cdot(nprocz-1)\cdot(\alpha+8\cdot\gamma$$
$$\cdot(\frac{8}{17}\cdot\frac{nx}{2\cdot nprocx}\cdot\frac{nz}{nprocz}+\frac{9}{17}\cdot\frac{nx}{2\cdot nprocx}\cdot\frac{nzp}{nprocz})\cdot\beta^{-1})\ . \quad (14)$$

Once having the model for the communication part, the model for the total execution time can easily be developed since all the other parts have a linear behaviour after some correction. As already mentioned before, the sum of the execution time from subroutines `linearbl` and `nonlinbl` is used to represent the total time for the code. Hence the total execution time (t^{1D}_{total}) for the 1D parallelisation can be expressed as

$$t^{1D}_{total} = t_{linearbl} + t_{nonlinbl} = t_{linearbl} + t_{com} + t_{ser}$$

and $t_{linearbl}$, t_{ser}, t_{com} are calculated as

$$\begin{aligned} t_{linearbl} &= t^1_{linearbl}\cdot\frac{nproc^1}{nproc} \\ t_{ser} &= \frac{t^1_{ser}}{k}\cdot\frac{nproc^1}{nproc} \\ t_{com} &= \frac{t^*_{com}}{k} \end{aligned}$$

where all the terms with a superscript 1 are the first available measurement data from a simulation, because it is not always possible to get the data from a serial version of the code, e.g. due to memory requirements. k is the correction factor defined in equation (11) and t^*_{com} is the value calculated from equation (12) or (13) depending on which version of the global communication is used. The total execution time (t^{2D}_{total}) for the 2D parallelisation can be expressed in a similar way as

$$t^{2D}_{total} = t_{linearbl} + t_{nonlinbl} = t_{linearbl} + t_{com} + t_{ser}$$

and $t_{linearbl}$, t_{ser}, t_{com} are given as

$$\begin{aligned} t_{linearbl} &= t^1_{linearbl} \cdot \frac{nproc^1}{nproc} \\ t_{ser} &= t^1_{ser} \cdot \frac{nproc^1}{nproc} \\ t_{com} &= t^*_{com} \end{aligned}$$

where again all the terms with a superscript 1 are the first available measurement data from a simulation. t^*_{com} is the value calculated from equation (14). Remember that the correction for the last y-loop in case of the 2D parallelisation is not applied, so there is no correction factor in the model.

3.4. *Comparison to the measurements*

The predicted total execution time as well as the one from the simulation measurements for one time step of both parallelisations are shown in Figures 20 and 21. In general, both models predict the behaviours of the code for all the cases very well.

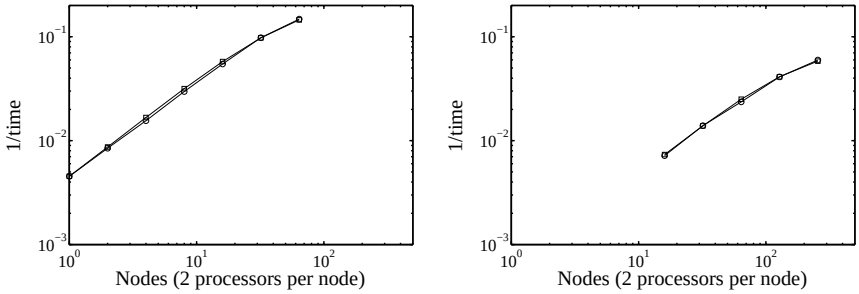

FIGURE 20. Model prediction versus the measurements for the 1D parallelisation. □ Model prediction, ∘ Measurements. *Left*: Run1 with $\alpha = 40$ [μs], $\beta = 300$ [Mbytes/s] and $\gamma = 4.29$, *Right*: Run2 with $\alpha = 80$ [μs], $\beta = 300$ [Mbytes/s] and $\gamma = 4.29$.

Note that the values of the bandwidth used in the model is obtained from a point-to-point communication test and is 300 [Mbytes/s]. The theoretical value given by the specification of the BlueGene/L machine for global tree interconnection (used for the collective communication) is about 350 [Mbytes/s]. The latency used in the model is much larger than the theoretical value which is usually less than 10 μs, this might be due to the models themselves which do not capture some non-linear effects. However, as long as large messages are sent, the influence of the latency is always small enough to be neglected.

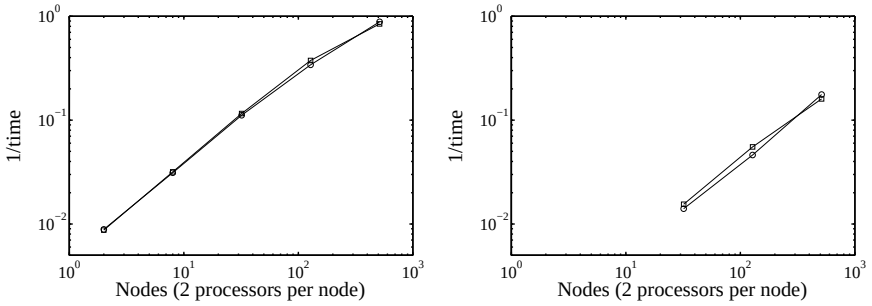

FIGURE 21. Model prediction versus the measurements for the 2D parallelisation. □ Model prediction, ∘ Measurements. *Left*: Run1 with $\alpha = 30$ [μs], $\beta = 300$ [Mbytes/s] and $\gamma = 4.29$, *Right*: Run2 with $\alpha = 30$ [μs], $\beta = 300$ [Mbytes/s] and $\gamma = 4.29$.

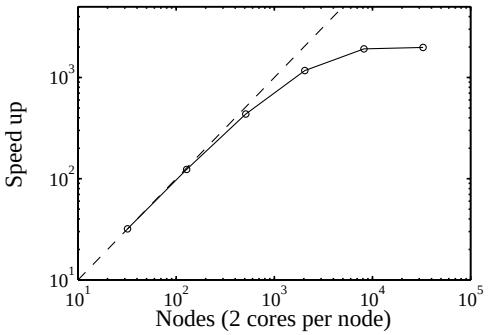

FIGURE 22. Model prediction of the Run2 (run2) for the 2D parallelisation. - - - Linear speed-up, ∘ Model prediction with $\alpha = 5$ [μs], $\beta = 300$ [Mbytes/s] and $\gamma = 4.29$.

3.5. *Limiting behaviour*

The model for the 2D parallelisation is used to predict the limiting performance for large processor counts and the plot is shown in Figure 22. If a large number of processors is used, the message size becomes comparably small. The influence of the latency is then much more important. Therefore, in the model, the value for the latency is chosen from the machine specifications. As mentioned before, the effect from the last y-loop will become significant once the number of processors is large enough. This is clearly seen in the figure. The predicted maximum speed-up is about 1000 (on 2048 nodes), 2000 (on 8192 nodes) and 2000 (on 32768 nodes). In practice, 8192 nodes might be the upper limit.

4. Conclusions

An efficient pseudo-spectral code (**SIMSON**) for solving the incompressible Navier–Stokes equations in channel and boundary-layer geometry has been developed during the last years at KTH Mechanics with the parallelisation only in the spanwise direction. Due to the limitation of this parallelisation approach, the total number of processors that can be used is at most the number of spanwise grid points, i.e. on the order of 256. This is definitely not suitable for the massively parallel super-computers which are getting more and more common at computer centres. Therefore, a 2D parallelisation has been implemented. This gives us the possibility to run simulations with more than 1000 cores (processors).

By looking into the details of the performance pertaining to different parts of the code, the suboptimal performance is due to the fact that for the last y-loop the code is not fully parallelised, i.e. some processors are idling. This problem cannot be eliminated completely due to the algorithm used now, but the effect of the problem can be reduced to some extent. Nevertheless, the code scales efficiently with the number of processors and therefore a high performance can be achieved. For the 2D parallelisation, the total amount of communication is larger than that in the 1D case, but due to parallel disjunct groups among processors, the overall performance is even better. For a large test case, a speed-up of about 130 (on 256 nodes) and 400 (on 512 nodes) for the 1D and 2D parallelisation respectively can be obtained.

Associated performance models for both parallelisations are developed and they predict the behaviours of the code reasonably well. The only part which needs modelling is the communication part, all the other parts have inherent linear behaviour after some simple modification. Benchmarks have been performed on a BlueGene/L machine with up to 1024 nodes.

Acknowledgements

The computer time was provided by the Centre for Parallel Computers (PDC) at the Royal Institute of Technology (KTH) and the National Supercomputer Centre in Sweden (NSC) at Linköping University. Discussions with Nils Smeds (IBM), Ulf Andersson (PDC) and Torgny Faxén (NSC) are gratefully acknowledged.

References

Alvelius, K. & Skote, M. 2000 The performance of a spectral simulation code for turbulence on parallel computers with distributed memory. *Tech. Rep* TRITA-MEK 2000:17. KTH Mechanics, Stockholm, Sweden.

Amdahl, G. 1967 Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS spring joint computer conference, Atlantic City, N.J., U.S.A.*, pp. 483–485. AFIPS Press, Reston, V.A., U.S.A.

Bruck, J., Ho, C.-T., Upfal, E., Kipnis, S. & Weathersby, D. 1997 Efficient algorithms for All_to_All communications in multiport message–passing systems. *IEEE Trans. Parallel Distrib. Syst.* **8** (11), 1143–1156.

Canuto, C., Hussaini, M. Y., Quarteroni, A. & Zang, T. A. 1988 *Spectral Methods in Fluid Dynamics*. Springer, Berlin, Germany.

Chevalier, M., Schlatter, P., Lundbladh, A. & Henningson, D. S. 2007*a* A pseudo-spectral solver for incompressible boundary layer flows. *Tech. Rep* TRITA-MEK 2007:07. KTH Mechanics, Stockholm, Sweden.

Chevalier, M., Schlatter, P., Lundbladh, A. & Henningson, D. S. 2007*b* A pseudo-spectral solver for incompressible boundary layer flows. *Tech. Rep* TRITA-MEK 2007:07. Royal Institute of Technology, Stockholm.

Clement, M. J., Steed, M. R. & Crandall, P. E. 1996 Network performance modelling for PVM clusters. In *Proceedings of the 1996 ACM/IEEE conference on supercomputing*. IEEE Computer Society, Washington, D.C., U.S.A.

Lundbladh, A., Berlin, S., Skote, M., Hildings, C., Choi, J., Kim, J. & Henningson, D. S. 1999 An efficient spectral method for simulation of incompressible flow over a flat plate. *Tech. Rep* TRITA-MEK 1999:11. KTH Mechanics, Stockholm, Sweden.

Lundbladh, A., Henningson, D. S. & Johansson, A. V. 1992 An efficient spectral integration method for the solution of the navier–stokes equations. *Tech. Rep* FFA-TN 1992-28. Aeronautical Research Institute of Sweden, Bromma, Sweden.

Moin, P. & Mahesh, K. 1998 Direct numerical simulation: A tool in turbulence research. *Ann. Rev. Fluid Mech.* **30**, 539–578.

Mullen-Schultz, G. L. & Sosa, C. 2007 *IBM System Blue Gene Solution: Application Development*. `http://www.redbooks.ibm.com`.

Nordström, J., Nordin, N. & Henningson, D. S. 1999 The fringe region technique and the Fourier method used in the Direct Numerical Simulation of spatially evolving viscous flows. *SIAM J. Sci. Comp.* **20** (4), 1365–1393.

Sagaut, P. 2005 *Large Eddy Simulation for Incompressible Flows, An Introduction*, 3rd edn. Springer, Berlin, Germany.

Steffenel, L. A., Martinasso, M. & Trystram, D. 2007 Assessing contention effects on mpi_alltoall communications. In *Advances in Grid and Pervasive Computing* (ed. C. Cérin & K.-C. Li), *Lecture Notes in Computer Science*, vol. 4459, pp. 424–435. Springer-Verlag, Berlin, Germany.