Spectral Simulations of Wall-bounded Flows on Massively-parallel Computers

Qiang Li, Philipp Schlatter and Dan Henningson

1 Introduction

As the super computer develops rapidly since the last several decades, highly resolved time-dependent numerical simulation, i.e. direct numerical simulation (DNS) and large-eddy simulation (LES), has become an important tool for transition and turbulence research (Moin and Mahesh, 1998).

The super computers can be classified into two groups with respect to the architecture of the processor, i.e. vector processor and scalar processor. A vector processor, or array processor, is designed that it has registers as a vector quantity and is able to operate on multiple data elements simultaneously, thus the operations are done in parallel. In contrast, a scalar processor has registers for data as a scalar quantity and 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 nearly disappear in the super computers we have nowadays. Another classification of the super computers is according to the memory configuration. The shared and distributed memory are the two memory configurations. In the former case, all the processors share the same memory while in the latter case, 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 as the types of the super computers as shown in Table 1. Throughout this report, only the first combination, i.e. scalar processors with distributed memory, is focused on and discussed. The term "processor" means central processing unit (CPU) and 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 super computers.

A typical distributed memory computer system, e.g. a computer cluster, consists of a group of computers which are connected by network so that they can work together and can be viewed as a single computer. A computer cluster has usually about 200 to 300 processors in total. However, in order to obtain higher performances and to solve larger size problems, a massively parallel computer system has to be used. A massively parallel computer

system is a single computer with a very large number of processors, usually more than 1000 processors. Each single processor has a lower performance compared to a processor of a cluster, but this is compensated by the huge amount of processors. All the processors are connected via a high speed interconnection. Therefore, the overall performance is better than a common cluster.

Once equipped with a morden super computer, to have an efficient numerical code for simulating the 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 Fourier/Chebyshev fully spectral discretization, leading to high numerical accuracy and efficiency. The nonlinear convective terms are evaluated pseudo-spectrally in physical space to avoid the evaluation of convolution sums using fast Fourier transforms (FFT) and afterwards transformed into Chebyshev space. The dealiasing errors are removed by using the 3/2-rule. The evolution 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 could be run in either temporal mode or spatial mode. It also support disturbance formulation and linearised formulation. Multiple passive scalars, e.g. temperature field, could also be solved together with the velocity field. For boundary layer flows, the fringe region technique has to be used to fulfil the periodic boundary condition in the wall parallel plane.

2 Parallelization

The numerical code is written in FORTRAN 77 and the coarse structure can be divided into four steps. In the first step, initialization 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 are executed. In step three, the whole 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.

2.1 1D parallelization

The major computational effort of the code is in the subroutines **nonlinbl** and **linearbl** where we calculate the nonlinear terms in physical space and solve the equations and evaluate the boundary conditions in Fourier/Chebyshev space. The main structure of the code is shown in Figure 1.

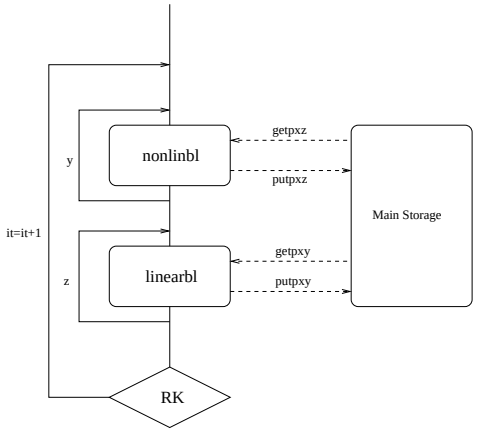

Figure 1: The main structure of the code for the 1D parallelization.

The computations are done plane by plane, i.e. in an xy-plane in **linearbl** and an xz-plane in **nonlinbl**. Outside these subroutines, we 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(on) any other one. If one processor calculates one plane at each time, we could use as many processors as there are planes running in parallel. As one can see from Figure 1, in order to get the data from the main storage onto planes, we call the subroutines **getpxz** for **nonlinbl** and **getpxy** for **linearbl**. The corresponding subroutines for putting data back to the main storage after the calculations in **nonlinbl** and **linearbl** are **putpxz** and **putpxy**, respectively.

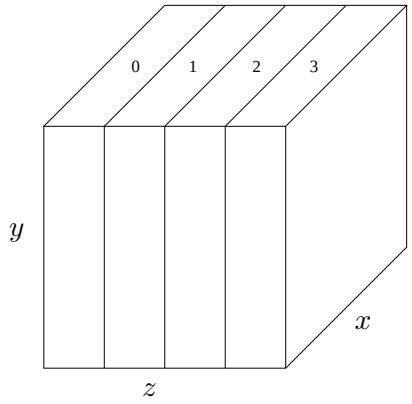

Figure 2: Data distribution among all the processors.

The code supports parallelization with both shared memory (OpenMP) and distributed memory (MPI). Here we will stick to the distributed memory machines, thus only MPI will be discussed. The main storage is distributed in the z direction only, i.e. 1D parallelization. A sketch of the distribution of the main storage is shown in Figure 2. The data in the horizontal plane, i.e. xz-plane, is saved in the Fourier space while the data in the wall-normal

plane is saved in physical space. We denote *nproc* to be the total number of the processors. In the sketch we choose $nproc = 4$ for an example. Then the amount of data saved on each processor is $memnx \times memny \times memnz$ where $memnx = \frac{nx}{2}$, $memny = nyp$ and $memnz = \frac{nz}{nproc}$. nx, nyp and nz are the number of grid points in the streamwise, wall-normal and spanwise direction, respectively. By distributing 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* consecutive wall-normal planes 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 of the data on each xz-plane, each processor has to collect data from all the other processors. The data collection is accomplished by calling subroutines **getpxz** and **putpxz**. This global data transfer gives rise to a significant amount of communication among all the processors. Actually a vast majority of the total communication among the processors happens in the subroutine **nonlinbl**, more precisely in the subroutines **getpxz** and **putpxz**. As the same for 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 data in one xz-plane and is 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 we have to perform 1D FFTs twice in the two directions on each xz-plane. Following the FFTs, all the data is stored in physical space and then the nonlinear terms can be evaluated. To remove the aliazing errors from the evaluation of the nonlinear terms, the 3/2 rule has been used (Canuto et al., 1988). Once the calculation of the nonlinear terms are finished, inverse FFTs have to be performed to transform the data back to spectral space. Last, another the global communication has to be done in **putpxz** to put 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 2D FFTs are the main computational effort of the whole code.

Considering the global communication needed in the subroutines **getpxz** and **putpxz**, two different ways are implemented. On the one hand, a hand-written version 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 and Skote (2000). On the other hand, an alternative version by adopting the standard collective communication command **MPI_ALLTOALL** can be

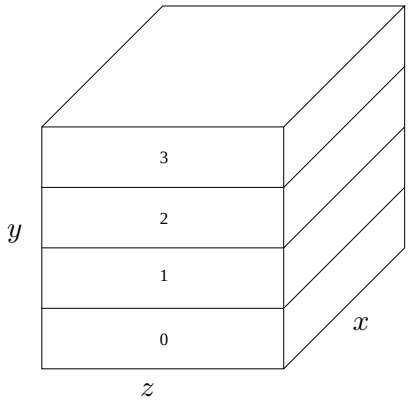

Figure 3: Data distribution after the global communication.

used. The standard MPI command MPI_ALLTOALL transfers data from

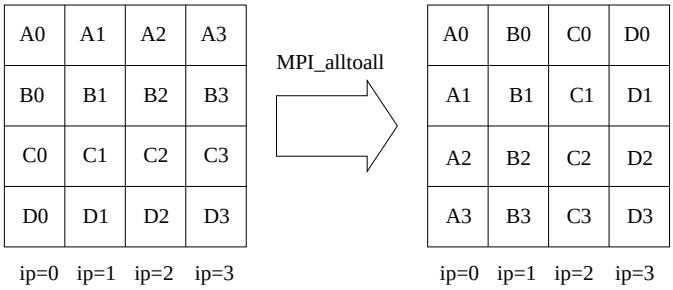

Figure 4: The MPI_ALLTOALL command illustrated for a group of four processors $ip = 0, 1, 2, 3$.

all members to 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 one can see from the figure, what the MPI_ALLTOALL does is nothing but a global transpose the data of all the members in the group.

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 saved on each processors is a slightly more and thus the amount of communication compared to the hand-written version. This is because that $memny$ has to be $(int(\frac{nyp}{nproc}) + 1) \times nproc$ 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 involving the MPI_TYPE_STRUCTURE and MPI_UB have to be used in order to fit the data structure of the code. These user defined data types used in the subroutines **getpxz** and **putpxz** are $realg1$ and $realg2$. $realg1$ has $memnz$ blocks, $memnx$ elements in each

block and $memnx \times memny$ elements between start of each block. And upper-bound of $memnx$ has to be added to the data type $realg1$ to give the correct displacement between cosecutive 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 memorys. The data type $realg2$ has also $memnz$ blocks, $memnx$ elements in each block, but $(\frac{nxp}{2}+1)$ elements between start of each block where $nxp = \frac{3}{2}nx$. And upper-bound of $(\frac{nxp}{2}+1) \times memnz$ 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, one could refer to MPI standard documentation.

2.2 2D parallelization

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

On a distributed memory machine, there are potentially three possible spatial directions which can be distributed among the different processors. The number of grid points in y, discretized by Chebyshev expansion, is often not even 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 parallelize 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 grid points in the streamwise direction is usually more than that in the spanwise direction. If we choose the streamwise direction to be the direction to parallize over, we could use more processors. However, there exists a similar problem as only parallelized in the z direction. 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 x or z direction, parallellizing in both x and z directions, i.e. a 2D parallelization 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.

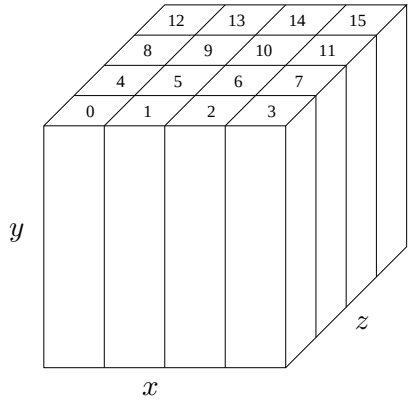

Figure 5: Initial data distribution among all the processors.

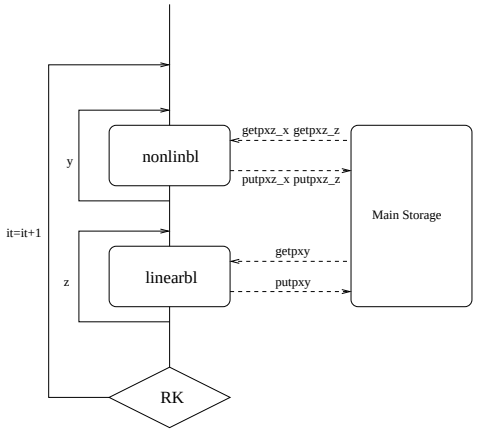

Figure 6: The main structure of the code for the 2D parallelization. Note that there exists FFTs between the subroutines **getpxz_z** and **getpxz_x** and inverse FFTs between **putpxz_z** and **putpxz_x**

The main structure of the code for the 2D parallelization is shown in Figure 6. It looks quite similar to the structure of the 1D parallelization 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**. As the same procedure in the 1D parallelization, FFTs in both directions in each xz-plane have to be performed by each processor. However, due to the data distribution on each xz-plane we have now, only the 1D FFT in the spanwise direction is

available. 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 subroutine **getpxz_x** and the corresponding one before putting data back to the tempory storage is **putpxz_x**. The details about both global data communications will be explained in later section. 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 parallelization, none of the processor has full access to the data on any whole xz-plane. When carrying out the calculations, the processors are divided into different groups. These groups are different from the default MPI communication group, i.e. 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, similarly as for the 1D parallelization, 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 parallelization, $nproc = nprocx \times nprocz$ wall-parallel planes are treated simutaneously. The subroutines **getpxy** and **putpxy** in **linearbl** are called by all the processors at the same time as well and $nprocz$ consecutive wall-normal planes are treated in each z loop. See the definitions of $nprocx$ and $nprocz$ in later section.

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 we use 16 processors as an example. We denote $nproc$, $nprocx$ and $nprocz$ to be 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 parallelization, 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 parallelization. Note that the size of the domain in the x direction is $\frac{nx}{2}$ rather than nx. This is due to the fact that we save the real and imaginary parts of the complex numbers seperately. Then the amount of data stored on each processor is $memnx \times memny \times memnz$ where $memnx = \frac{nx}{2nprocx}$, $memny = (int(\frac{nyp}{nprocz}) + 1) \times nprocz$ and $memnz = \frac{nz}{nprocz}$.

For the 2D parallelization, the global communication is implemented only using the standard collective MPI command MPI_ALLTOALL, so $memny$

needs to be $(int(\frac{nyp}{nprocz}) + 1) \times nprocz$. To make the transfer of data more efficient, again 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 start of each block. As detailed previously, an upper-bound 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 start of each block. Note that no 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 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 parallelization, $realg1$ and $realg2$ have the exactly the same size and it is also true for $realg3$ and $realg4$. nxp and nzp are defined later.

Before we calculate the nonlinear terms, we have to do the FFTs in both x and z direction to transform the data from spectral space to physical space as mentioned before. Due to the data storage after **getpxz_z**, this can be done in only z direction. Thus we have to transfer the data among different processors once more such that every processor can do the FFTs in the x direction.

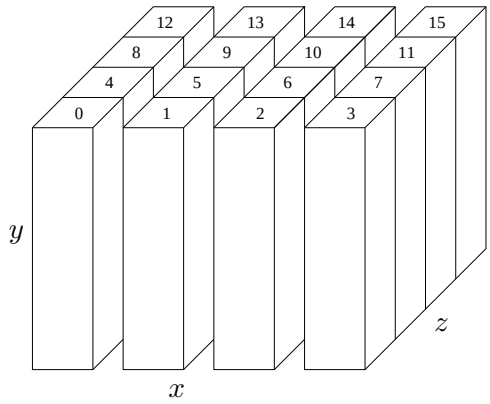

Figure 7: The configuration of the groups for the first transpose.

So first, we separate the processors into $nprocx$ groups and each group contains $nprocz$ processors. The creation of groups is shown in Figure 7. Then we use the standard MPI command MPI_ALLTOALL to transpose the data within each group. This is done by calling the subroutine **getpxz_z**. We end up with the data storage configuration shown in Figure 8. For convenience we only show the first consecutive $nprocz$ planes in wall-normal direction. The layout of the other wall-parallel planes are just a repetition of

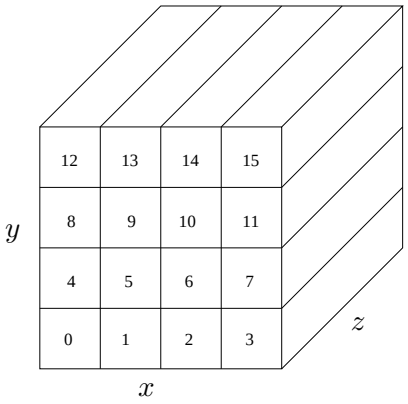

Figure 8: The data distribution after **getpxz_z**.

the first *nprocz* planes. The creation of groups and the data storage for the second global data transpose are shown in Figure 9 and Figure 10 and will be discussed later. Here we have to consider aliasing errors which are caused by representing a higher wavenumber data at a lower wavenumber. When calculating the nonlinear terms, these errors will be introduced. In order to eliminate the aliasing errors, the 3/2-rule which is to expand the original grid and pad some part with zeros has to be used. 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 divisibleb by $nprocz$, the dealizing grid needs to be $(\frac{nzp}{nprocz} + min(1, mod(nzp, nprocz))) \times nprocz$, 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" are moved to the upper part of the fine grid and the middle part are padded with zeros denoted by the $\bigcirc$. The oddball mode are also set to zero which is denoted by the gridded lines, see 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 FFT in the z direction on the fine grid.

In a second step, we create *nprocz* new groups 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 by using the MPI command MPI_ALLTOALL 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 dealising grid is $\frac{nxp}{2} + 1$ in the x direction and still nzp in z direction where $nxp = \frac{3}{2}nx$. This is shown in Figure 11 (c). The additional 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

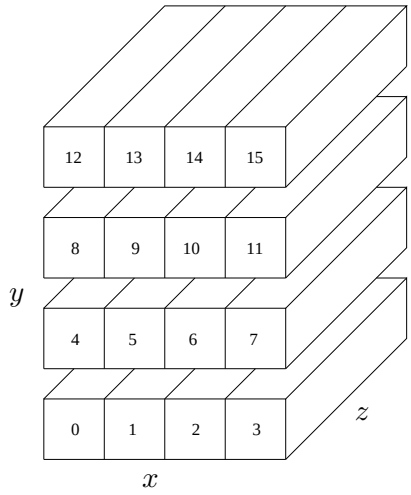

Figure 9: The configuration of the groups for the second transpose.

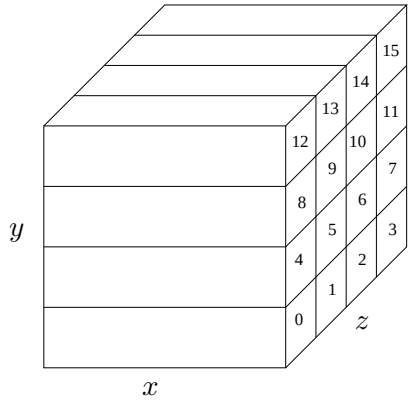

Figure 10: The data distribution after **getpxz_x**.

parts, are added. One for the zero frequency component and the other for the Nyquist frequency component. Since the real and imaginary parts are saved seperatly, only one more point is added in the x direction. The right-hand part is padded with zeros and the oddball mode is set to zero. Last, the FFT 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 . Once finishing calculating the nonlinear terms, all the data has to be transfered 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. After the calculation, the data is put back to the main memory

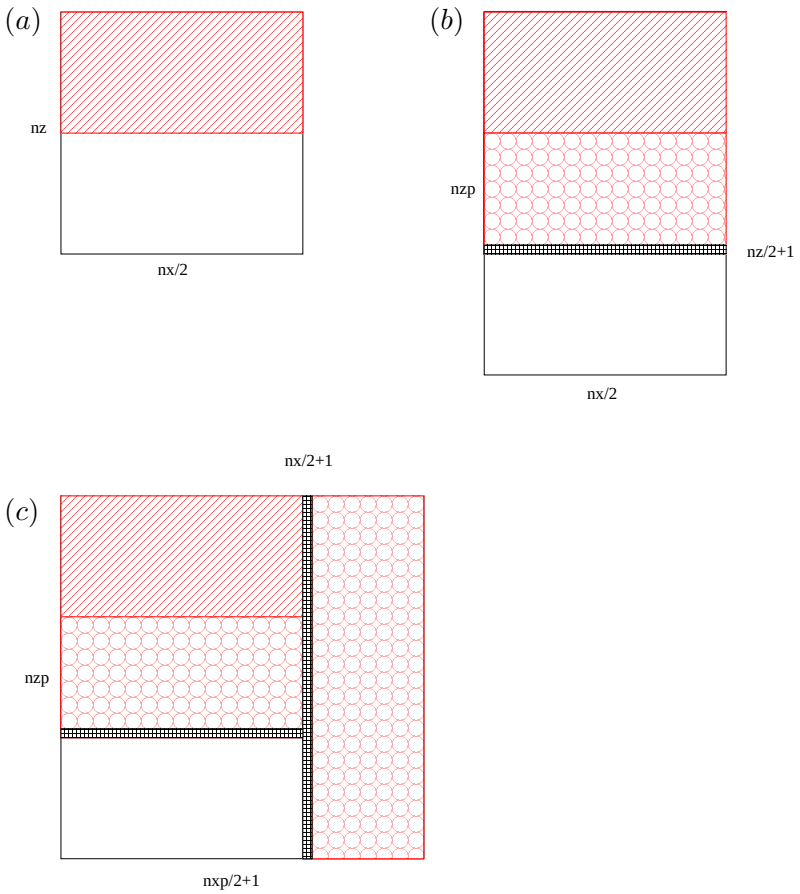

Figure 11: The dealizing grid and the odd ball location in a xz-plane.

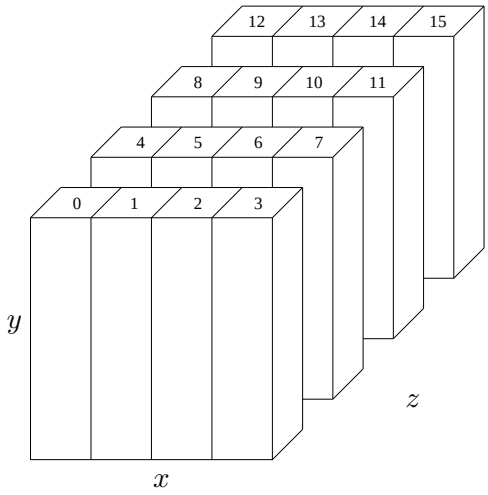

Figure 12: The data distribution in **linearbl**.

through subroutine **putpxy**.

2.3 Amount of communication

2.3.1 1D parallelization

The main communication between the processors is in the nonlinear part, i.e. subroutine **nonlinbl**. Assume that we only consider a velocity field, i.e. without pressure or scalar fields. Then for each call to **nonlinbl**, five variables (three velocity componets and two vorticity components) are collected from the main memory by calling subroutine **getpxz** and three variables (three vorticity components) are put back to the main memory by calling subroutine **putpxz** for the 1D parallelization. Thus a total of eight variables have to be communicated which is independent on the implementation of the global communication we choose. Here we only consider the version using collective communication, i.e. using standard command MPI_ALLTOALL. For the hand written version, one can find more details in Alvelius and Skote (2000).

Each processor performs calculation on approximately $(int(\frac{nyp}{nproc}) + 1)$ xz-planes. The data types of the message we use are the user defined data types, i.e. $reag1$ and $realg2$. Both of them have a size of $memnx \times memnz$ or $\frac{nx}{2} \times \frac{nz}{nproc}$, so in total the amount of data that one processor collects for one xz-plane is

$$2 \times nproc \times (\frac{nx}{2} \times \frac{nz}{nproc}) = nx \times nz \ . \tag{1}$$

Note that each processor collects data not only from the other processors but also from itself, and both the real and imaginary part gives a factor of 2 for the total amount of data which has to be communicated. This gives that for one variable each processor needs to collect

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

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

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

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 parallelization

For the new 2D parallelization, the situation is slightly different than that for the 1D case. As explained earlier, we have to transpose the data twice in order to perform the FFTs. For each call to **nonlinbl**, five variables (three velocity componets and two vorticity components) are collected from the main memory with **getpxz_z** and later three variables (three vorticity componets) are put back to the main memory with **putpxz_z** within **nonlinbl**. Six variables (three velocity componets and three vorticity 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 $(\frac{nyp}{nprocz}+1)$ 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 parallelization, the total amount of data which has to be collected for one processor for each Runge-Kutta substep and all 17 variables is then

$$8 \times (int(\frac{nyp}{nprocz})+1) \times (8 \times \frac{nx}{nprocx} \times nz + 9 \times nx \times \frac{nzp}{nprocz}) \qquad (4)$$

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

3 Performance Analysis

For the parallel computing, the maximum speed-up one can obtain of a code is

$$speed-up = \frac{1}{F+(1-F)/N} \qquad (5)$$

where F is the fraction of a calculation that is sequential, i.e. the part that cannot benefit from parallelization, and thus $(1-F)$ is the fraction that can be parallelized where N is the number of processors used for the calculation. This is usually referred as the Amdahl's law. 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 code always has some part which cannot be parallellized 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 parallel parts of a code will usually not scale linearly, but rather at a reduced rate.

A benchmark 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-a-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 manual.

3.1 Code Performance

Two cases of different sizes are tested for both 1D and 2D parallelizations. The smaller case has the resolution of $1024 \times 129 \times 128$ (run1) while the resolution of the bigger 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 subroutines **linearbl** and **nonlinbl**, and the main computational effort is the FFTs in **nonlinbl**. So the speed-up plot shown later are only based on the execution times from these two subroutines. The time we choose to calculate the speed-up is the execution time per time step, i.e. four Runge-Kutta substeps.

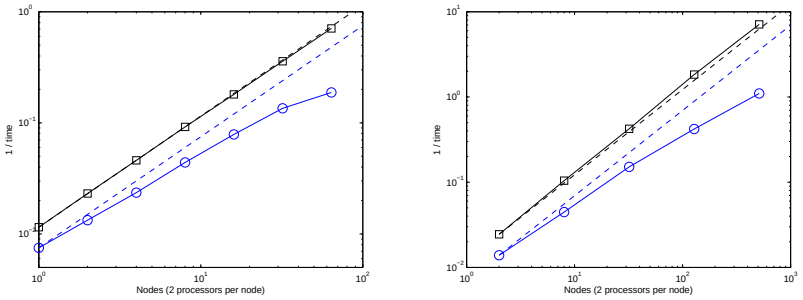

Figure 13: Performance of the two parallelizations for the small case (run1). - - -Linear speed up, □ **linearbl**, ○ **nonlinbl**. *Left*: 1D parallelization, *Right*: 2D parallelization.

In Figure 13 the two parallelizations are compared for the smaller case (run1). Due to the requirement of the 2D parallelization, i.e. the processors should be equally distributed along both directions, the first data point corresponds to 2 nodes or 4 processors. We clearly see that the linear part

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

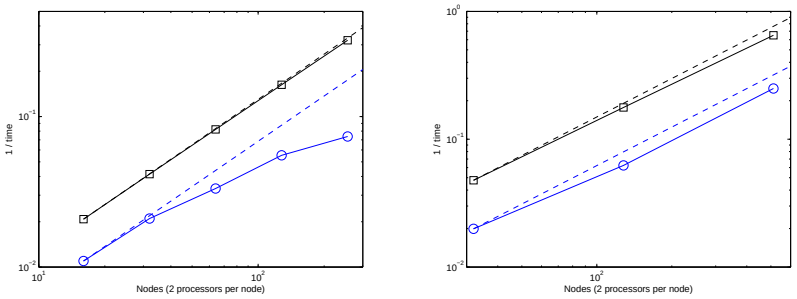

Figure 14: Performance of the two parallelizations for the large case (run2). - - -Linear speed up, □ **linearbl**, ○ **nonlinbl**. $Left$: 1D parallelization, $Right$: 2D parallelization.

In Figure 14 the large test case (run2) is shown for the two parallelizations. Note that due to the larger memory requirements, a minimum number of nodes of 16 has to be used for the 1D parallelization and 32 for the 2D parallelization. Similarly to the smaller case, the linear parts of both parallelizations still have linear performance. But for the nonlinear parts, a clear difference can be observed for the two parallelizations. As seen from the plots, the 1D parallelization shows a similar behaviour as for the smaller case reaching a saturation of the performance at 256 node, and the 2D parallelization has almost linear performance up to 512 nodes.

As we know 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 we used since we perform the global data transpose. However, there is another important fact which leads to this suboptimal behaviour, i.e. the odd number of discretization points in the wall-normal direction. Since we use the Chebyshev representation in the wall-normal direction, the spectral modes in y direction is not divisible by the number of processors, i.e. $\frac{nyp}{nproc}$ for the 1D parallelization or $\frac{nyp}{nprocz}$ for the 2D parallelization is not an integer. Therefore, for the last y loop outside subroutine **nonlinbl**, not all the processors are active. So the last loop is not completely parallelized. If $nproc$ or $nprocz$ is of the same order of magnitude as nyp, this will have very big effect on the performance. This is what we have observed from the smaller test case for both parallelizations and the bigger case for the 1D parallelization. If $nproc$ or $nprocz$ is relatively small compared to nyp, the effect form this last y loop is much smaller. This is shown by the bigger case for the new 2D parallelization. In general, for the

2D parallelization, since *nprocz* is only the square root of *nproc*, the effect of the last y loop should be smaller than the 1D parallelization. However, if we increase the number of processors for the 2D parallelization even more, at some point we will have the same problem.

To investigate the behaviour of subroutine **nonlinbl** more in detail, we split the total execution time spent in subroutine **nonlinbl** into two parts: one part only involves communication (t_{com}), the other part is the computation (t_{ser}), e.g. FFTs. Hence we have the relation $t_{nonlinbl} = t_{com} + t_{ser}$. In Figure 15 the time of communication and computation in subroutine

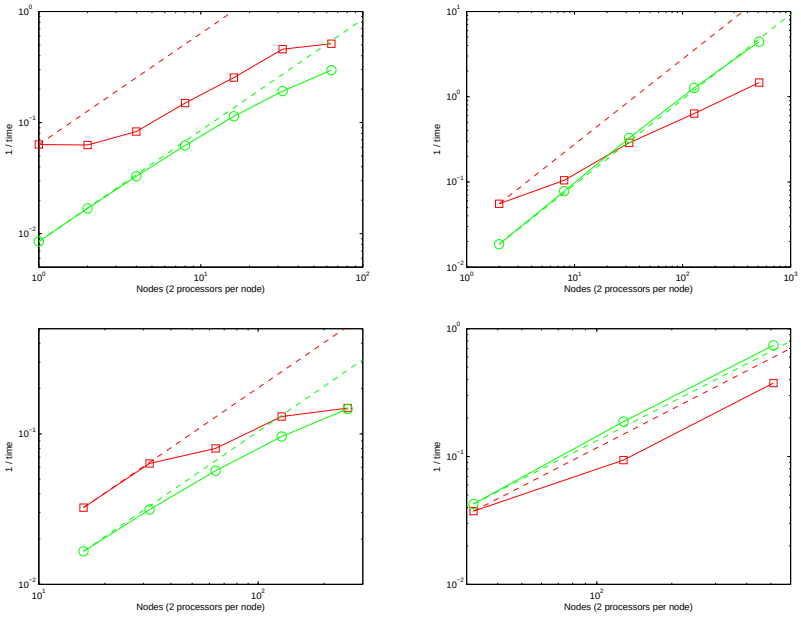

Figure 15: Performance of t_{com} and t_{ser} in **nonlinbl**. - - -Linear speed up, $\square$ t_{com}, $\circ$ t_{ser}. Top : Small case, $Bottom$: Big case. $Left$: 1D parallelization, $Right$: 2D parallelization.

nonlinbl are compared for both parallelizations. As we expected, for both parallelizations the speed up curves for the communication part do have suboptimal behaviours. It is clear by comparing the computational part for the two parallelizations that the effect of the last y loop has a smaller effect for the 2D parallelization. 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. Note that since we plot $\frac{1}{time}$, the lower the position the large the execution time is. As we increase the number of processors, more and more communication will happen which means that t_{com} will take larger and larger portion of the total time spent in **nonlinbl**. For the 2D parallelization, since we have twice(??) more communication, the growth of t_{com} is

even faster. As one can see from the plots, t_{com} could be larger than t_{ser} even at small number of processors, e.g. 32 nodes (64 processors).

To eliminate the last y loop effect on the performance, we multiply a factor k with each sampled data from **nonlinbl**. This correction is just to help us to develop the model of the whole performance of the code. Since the effect for the 2D parallelization is much smaller than that for the 1D parallelization, we only do this correction for the 1D parallelization. The correction factor k is only a function of the number of processors that has been used and is defined as

$$k = \frac{nyp}{(int(\frac{nyp}{nproc}) + 1) \times nproc} \ . \tag{6}$$

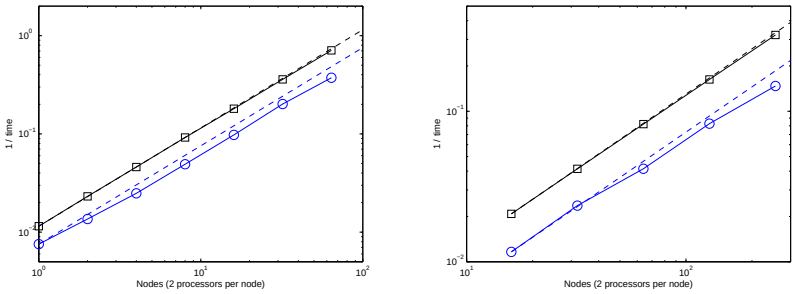

Figure 16: Performance of the 1D parallelization after correction. - - -Linear speed up, □ **linearbl**, ○ **nonlinbl**. $Left$: Small case, $Right$: Big case.

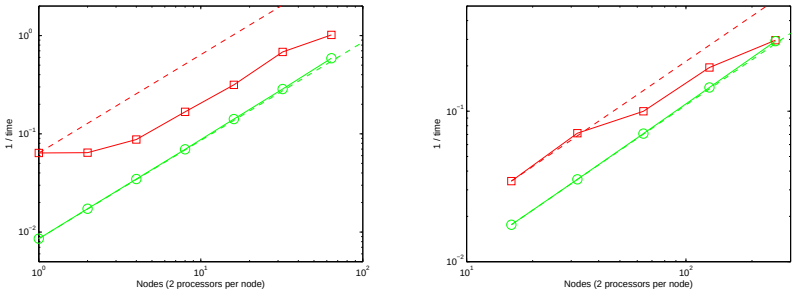

Figure 17: Performance of t_{com} and t_{ser} in **nonlinbl** after correction for the 1D parallelization. - - -Linear speed up, □ t_{com}, ○ t_{ser}. $Left$: Small case, $Right$: Big case.

After the correction, the total time spent in subroutines **linearbl** and **nonlinbl** are shown in Figure 16 and the communication and computation parts within **nonlinbl** are shown in Figure 17, respectively. Now except the

communication part, all the other parts, i.e. linear part and computation part, have linear speed up performance. And this is also true for the 2D parallelization.

In order to roughly estimate the performance on BlueGene/L without running a simulation, we have developed the performance model for the code of both 1D and 2D parallelizations. From the previous section we know that after eliminating the effects from the last y loop, only the communication part which is supposed not to scale linearly does not have a linear speed up behaviour. Therefore this part needs to be modelled. In the models, we only consider the latency and bandwidth to be the most important factors and neglect all the other influences, e.g. the distance between the two communicating processors, whether they are in the same node card or not.

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

$$2 \times (int(\tfrac{nyp}{nproc})+1) \times nproc \times \tfrac{nx}{2} \times \tfrac{nz}{nproc}$$

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

$$2 \times 2 \times (int(\tfrac{nyp}{nproc})+1) \times nproc$$

times. Note that the first factor of 2 is from the real and imaginary part of the data needs to be collected and the second one is from the fact that each processor has to send and receive data at the same time.

So the execution time of communication t_{com} of one processor for one time step, i.e. 4 Runge-Kutta iterations, all the 8 variables is approximately given by

$$t_{com} = 4 \times 8 \times 2 \times (int(\frac{nyp}{nproc})+1) \cdot nproc \cdot (2 \cdot \alpha + 8 \cdot \frac{nx}{2} \cdot \frac{nz}{nproc} \cdot \beta^{-1}) \quad (7)$$

where α is the latency $[\mu s]$ and β is the bandwidth $[Mbytes/s]$. The number of messages has been multiplied by a factor of 8 to convert unit to bytes, since we use double precision, i.e. 8 bytes for each message. The corresponding model for the hand-written version of the 1D parallelization is also given here,

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

There is only very little difference when comparing these two models given by equations (7) and (8). Following the same steps, we could develop model

of the execution time of communication t_{com} for the 2D parallelization. For one time step, i.e. 4 Runge-Kutta iterations, all the 17 variables, execution time of communication t_{com} is approximately given by

$$\begin{aligned} t_{com} &= 4 \times 17 \times 2 \times (int(\frac{nyp}{nprocz}) + 1) \\ &\cdot \quad (2 \cdot \alpha + 8 \times (\frac{8}{17} \times \frac{nx}{2 \cdot nprocx} \cdot nz + \frac{9}{17} \cdot \frac{nx}{2} \cdot \frac{nzp}{nprocz}) \cdot \beta^{-1}) \end{aligned} \quad (9)$$

for the 2D parallelization.

Once we have the model for the communication part, the model for the total execution time could be easily developed since all the other parts have a linear behaviour after some correction. We have already mentioned that most of the time of the code is spent in subroutines **linearbl** and **nonlinbl**, so we will use the sum of the execution time from these two subroutines to represent the total time for the code. Hence the total execution time (t^{1D}_{total}) for the 1D parallelization 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 the memory requirement. k is correction factor defined in equation (6) and t^*_{com} is the value calculated from equation (7) or (8) depending on which version of the global communication is used. The total execution time (t^{2D}_{total}) for the 2D parallelization can be expressed in the same way as

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

and $t_{linearbl}$, t_{ser}, t_{com} can be expressed similarly 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 (9). Remember that we do not do the correction for the last y loop of the 2D parallelization, so there is no correction factor in the model.

The predicted total execution time as well as the one from the simulation measurements for one time step of both parallelizations are shown in Figure 18 and 19. In general, both models predict the behaviours of the code for all the cases very well. The values for the latency α and bandwidth *beta* are obtained by doing a least square fit based on the measurements from the simulation. Based on these values, we then tune the values such that they fit for both the smaller and the bigger case.

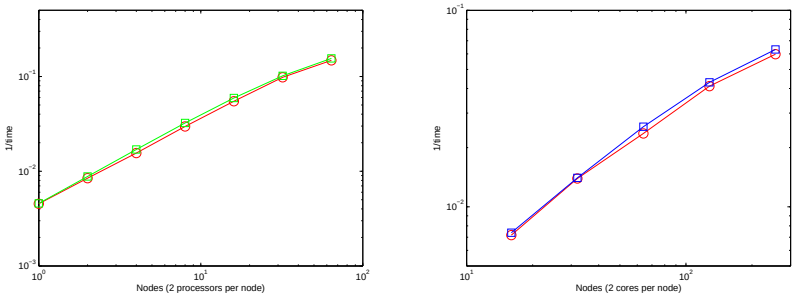

Figure 18: Model prediction versus the measurements for the 1D parallelization. □ Model, ∘ Measurements. $Left$: Small case with $\alpha = 20$ [μs] and $\beta = 85$ [Mbytes/s], $Right$: Big case with $\alpha = 20$ [μs] and $\beta = 50$ [Mbytes/s].

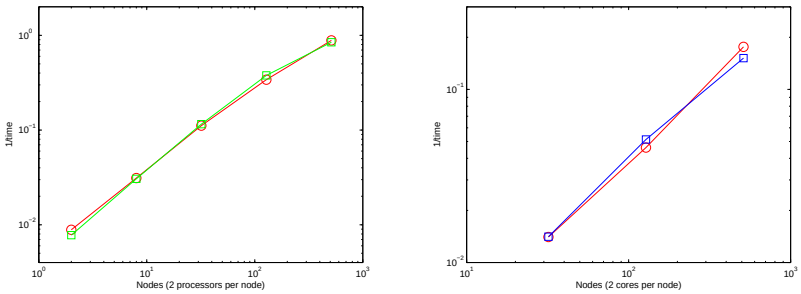

Figure 19: Model prediction versus the measurements for the 2D parallelization. □ Model, ∘ Measurements. $Left$: Small case with $\alpha = 15$ [μs] and $\beta = 175$ [Mbytes/s], $Right$: Big case with $\alpha = 15$ [μs] and $\beta = 120$ [Mbytes/s].

As one may notice that the value of the bandwidth that we use in the model is quite different from the theoretical value given by the manual of the BlueGene/L machine which are 175 [Mbytes/s] for 3D Torus and 350 [Mbytes/s] for global tree which is used for the collective communication.

Some reasons are explained below. It is a well-known fact that once the size of the data which is sent is below a certain level, the bandwidth or the data transfer speed will decrease considerablely from the maximum values. As the number of processors increases, the size of the data decreases and consequently the bandwidth for each individual processor also decreases. However, according to Alvelius and Skote (2000), although the bandwidth of each processor decreases, the effective time of the message passing should decrease since the amount of data needs to be sent becomes smaller. Another reason is that the value given by the BlueGene/L manual is for point-to-point communication, i.e. each processor is either sending data or waiting for receiving data at one time. However, in our test cases, all the processors are both sending and receiving data at the same time. So for each processor, the maximum data transfer speed of either sending or receiving data is only half of the bandwidth. Thus the theoretical value should be divided by a factor of 2 to compare with our test case. For the big case, due to more processors that we use, a factor of 3 should be used. If comparing with the half (for the small case) and one third (for the big case) of the theoretical values of the bandwidth, it agrees very well with our values predicted by the models. The latency is about twice as big as from the theoretical value which is no more than 10 μs, this might be due to the models themselves which are not correct. However, as long as we send large pieces of messages, the influences from the latency are always small enough to be neglected.

4 Conclusion

An efficient pseudo-spectral code (**SIMSON**) for solving incompressible Navier–Stokes equations of channel and boundary layer geometry has been developed during the last years. The parallelization of the code is only in spanwise direction. Due to the limitation of the parallelization, the total number of processors that can be used is at most 256. This is definitely not suitable for the massively parallelled supercomputers. Therefore, a new 2D parallelization has been implemented. This gives us the possibility to run a simulation with more than 1000 cores (processors).

By looking into the details about the performance of different parts of the code, we find that suboptimal performance is due to the fact that for the last y loop the code is not fully parallelled, i.e. some processors are doing noting but just waiting. This problem can not be eliminated due to the algorithm we use now, but the effect of the problem can be reduced to some extent. Nevertheless, the code can scales efficiently with the number of processors and therefore a high performance can be achieved. Moreover, this will greatly shorten the time for waiting in the queue.

The associated performance models for both parallelizations are developed and they predict the behaviours of the code very well. The only part

needs modelling is the communication, the other parts have linear behaviour after some simple modification. The benchmark has been done on a BlueGene/L machine. So the performance models we obtained do not imply that they will also work for another machine. They might not even work at all on another machine. Some tunings of the code is always necessary.

Acknowledgements

The computer time was provided by the Center for Parallel Computers (PDC) at the Royal Institute of Technology (KTH) and the National Supercomputer Center in Sweden (NSC) at Linköping University.

References

K. Alvelius and M. Skote. The performance of a spectral simulation code for turbulence on parallel computers with distributed memory. Technical Report TRITA-MEK 2000:17, Royal Institute of Technology, Stockholm, 2000.

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

M. Chevalier, P. Schlatter, A. Lundbladh, and D. S. Henningson. A pseudo-spectral solver for incompressible boundary layer flows. Technical Report TRITA-MEK 2007:07, Royal Institute of Technology, Stockholm, 2007.

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

A. Lundbladh, S. Berlin, M. Skote, C. Hildings, J. Choi, J. Kim, and D. S. Henningson. An efficient spectral method for simulation of incompressible flow over a flat plate. Technical Report TRITA-MEK 1999:11, Royal Institute of Technology, Stockholm, 1999.

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