TASK QUARTERLY 10 No 2, 125–138

EFFICIENT IMPLEMENTATION OF A COMPACT-PSEUDOSPECTRAL METHOD FOR TURBULENCE MODELING

ARTUR TYLISZCZAK

Institute of Thermal Machinery,
Czestochowa University of Technology,
Al. Armii Krajowej 21, 42-200 Czestochowa, Poland
atyl@imc.pcz.czest.pl

(Received 3 March 2006)

Abstract: The paper is devoted to parallel implementation of a compact discretization scheme combined with the Fourier pseudospectral method. The idle time of processors resulting from the method of computating derivatives using compact schemes is eliminated by proper ordering of subtasks and by performing useful computations when processors are waiting for data from their neighbors. The correctnes of the algorithm is confirmed by comparison of results of LES simulations with DNS data for flow in a 3D channel with periodic non-slip wall boundary conditions.

Keywords: large eddy simulation, compact scheme, channel flow

1. Introduction

Although computers' performance is continually enhanced, the modeling of turbulent flows remains one of the most difficult tasks from the point of view of computational costs. Using the classical approach based on the Reynolds-Averaged Navier-Stokes (RANS) method, one may obtain a converged solution of a 2D or 3D problem within a few hours or days, respectively. Obviously, this is only possible when the numerical mesh used to discretize the computational domain is relatively coarse, resulting in small numbers of grid nodes. However, there is no need to use very fine meshes in the RANS approach as the final solution is much more dependent on the dissipative nature of the closure model applied than by the numerical dissipation due to the mesh size. In RANS simulations, the time evolution of the flow field is either lost by definition (steady RANS) or considerably suppressed (unsteady RANS); thus all small-scale spatial phenomena related to variations of the time flow field are absent from the simulations and mesh refinement will not improve the solution's accuracy in capturing "more physics". The only improvement achieved by mesh refinment is in accuracy of the discretization method. This property of the RANS approach results from the Reynolds-averaging assumption which is true only when the averaging time is sufficiently long. In unsteady RANS simulations there is also no

need for very small numerical time steps: by analogy to spatial discretization errors they can only decrease errors resulting from time integration methods but not those resulting from the RANS assumptions. Contrary to RANS modeling, Direct Numerical Simulation (DNS) or Large Eddy Simulation (LES) allow to accurately predict the flow evolution in time and small scale spatial phenomena related to flow fields varying in time. Small-scale phenomena varying in time and space require accurate solutions. Therefore, the numerical meshes applied in DNS or LES are considerably finer compared to those used in RANS modeling. This leads to much longer computational times needed to obtain the solution; even for simple flow cases, the statistically converged solution using LES or DNS requires days or weeks of computations. At this point the efficiency of the applied numerical algorithm and its implementation in the numerical code become very important. In the LES or DNS method, the accuracy of the solution is improved by applying high-order discretization methods reducing the number of mesh points while preserving the accuracy of results. Although the use of high-order discretization methods is very limited in complicated geometries, their applications in simple domains allow us to analyze phenomena which require very precise solutions. In this paper, a combination of the compact discretization method [1] with the pseudospectral method [2] based on the Fourier series is used to solve the Navier-Stokes equation in a 3D domain. These methods are characterized by very good accuracy partly attributable wide computational stencil used to calculate derivatives. Therefore, a parallel implementation of the compact and pseudospectral methods is not trivial and it may happen that the parallel code will be less efficient than the serial one. This may be due to the large amount of data which has to be sent to or received from particular processors. When the ratio of the amount of data sent/received by a given processor to the effective work (computations) performed by this processor is high, the overall performance of the parallel code is reduced. This happens when the calculation of particular quantities (*e.g.* a derivative) requires non-local information – in our case this problem arises due to the wide computational stencil. This paper is mainly devoted to parallel implementation of compact discretization methods where calculation of derivatives requires solving a linear system of equations. In our case, the system is three-diagonal and thus the computationally less expensive way to solve it is to use the Thomas algorithm, sometimes referred to as the Three-Diagonal Matrix Algorithm (TDMA). However, this algorithm is purely serial in nature and cannot be parallelized effectively. The proposed method of parallelization is an attempt to use the time when processors are waiting for the data from neighboring processors (idle time) for useful work – intuitively, the simplest possible solution as long as there is "useful work" which can be performed during idle time.

2. The governing equations

Although this paper is focused on the parallel efficiency of the numerical code, the correctness of the obtained results had to be confirmed by comparison with the available numerical and experimental data. The flow in a periodic channel was chosen as a test case, but the proposed algorithm may be applied to many other flow problems. The Fourier pseudospectral method was applied for periodic directions (stream-wise and span-wise) while the compact scheme was used in the

wall-normal direction. The time integration was performed with the low-storage III step Runge-Kutta method. The pressure field computed according to the projection method with Neumann boundary conditions on the walls. The geometry of the flow is shown in Figure 1. The dimensions of the channel are $4\pi h \times 2\pi h \times 2h$ stream-wise, span-wise and in the wall-normal direction, respectively. The incompressible flow is governed by the continuity equation and the Navier-Stokes equations, in context of LES given as:

$$\frac{\partial \bar{u}_j}{\partial x_j} = 0, \tag{1}$$

$$\frac{\partial \bar{u}_i}{\partial t} + \frac{\partial \bar{u}_i \bar{u}_j}{\partial x_j} = -\frac{\partial \bar{p}}{\partial x_i} + \frac{\partial}{\partial x_j}\left((\nu + \nu_T)\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right)\right) + F_i. \tag{2}$$

Variables u_i and p are dimensionless velocity and pressure. Symbols ν and ν_T respectively denote the kinematic and subgrid viscosities. The last term of Equation (2) is a source term forcing the flow and plaing the role of a constant pressure gradient.

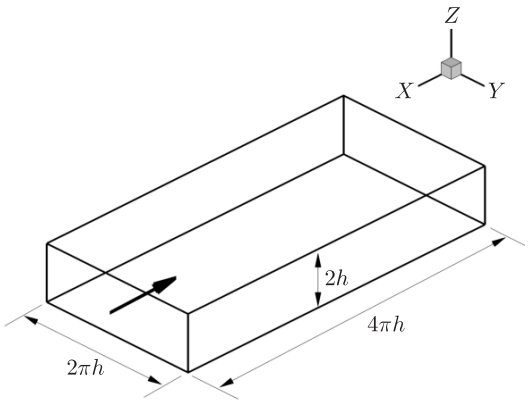

Figure 1. 3D schematic view of the computational domain

The results presented in this paper were obtained with the Smagorinsky model [3] and the dynamic Germano model [4], which can be written as:

$$\nu_T = (\Delta C)^2 \sqrt{2\bar{S}_{ij}\bar{S}_{ij}} * \mathscr{D}, \tag{3}$$

where $\Delta = (\Delta x \Delta y \Delta z)^{1/3}$ is the LES filter width and C is the Smagorinsky constant assumed equal 0.1 in the case of the Smagorinsky model, computed depending on the flow according to the dynamic Germano procedure. Symbol S_{ij} denotes the deformation tensor of the filtered field,

$$\bar{S}_{ij} = \frac{1}{2}\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right). \tag{4}$$

The last term, $\mathscr{D}$, in Equation (3) is a damping function defined similarly to the van Driest damping formula:

$$\mathscr{D} = \left(1 - \exp\left(\frac{y+}{26}\right)\right)^n, \tag{5}$$

where $y+ = u_\tau y/\nu$. The damping function is used only with the Smagorinsky model. For $n = 1$ the damping function is reduced to the van Driest formula. However, we ob-

served that when computations are atarted from a randomly disturbed initial solution with $n=1$ the flow became laminar. Therefore, in order to dampen subgrid viscosity close to the walls more intensely, the computations were performed with $n=2$.

3. Discretization scheme

In case of the Fourier pseudospectral method, the calculation of a derivative at a particular grid point may be performed either by Fast Fourier Transform (FFT) or by matrix-vector multiplication [2]. The former requires information from all mesh points along a given grid line and when the grid line is split between processors all the necessary data have to be transferred between them. If parallel computations are performed on clusters built from separate computers connected through a network, sending and receiving so large amounts of data decreases the efficiency of the code considerably. The situation is better when computations are performed on shared memory parallel (SMP) computers, where all processors share the same physical memory and there is no need to send or receive data via a network. However, due to relatively high costs of such computers, the cluster architecture is the most popular nowadays and, consequently, it has been assumed in the following that communication between processors takes place via a network using Message Passing Interface (MPI) libraries. Obviously, the code that using MPI may also be run on SMP computers.

When the derivatives are computed by matrix-vector multiplications, the amount of data that has to be transferred between processors is reduced but still each processor must communicate with all the remaining processors. The application of the so-called collective communications (MPI_REDUCE, MPI_SCATTER) seems to be the most efficient in such cases, rather than direct communication among particular processors. The overhead due to sending and receiving data is still relatively high, but compared with computation of derivatives by FFT, parallel matrix-vector multiplication is more efficient as the ratio of transferred data to effective computations is smaller in this case. However, we are mainly interested in the real time required for computations and not in the above mentioned parallel efficiency. Therefore, it is necessary to realize that the number of operations (multiplications, additions) for matrix-vector multiplication increases considerably compared with the number of operations performed during the FFT procedure: from $\mathcal{O}(N\log N)$ in the case of FFT to N^2 for matrix-vector multiplication (where N is the number of nodes along a grid line). This is the main reason of "increased" efficiency of the matrix-vector multiplication method, which in fact leads to worse overall performance of the parallel code. Therefore, in the proposed parallelization method, we do not divide the computational domain in directions where the pseudospectral method is applied; thus all data needed to compute derivatives in those directions are directly available. Instead, we divide the domain in the direction where the compact discretization method is applied. A disadvantage of this approach is that each processor always sends/receives the same number of data to its neighbors, while in the same time the amount of computations on a given processor decreases in proportion to the total the number of processors used in computations. However, judging from the author's experience, this approach is more efficient than parallelization of FFT on cluster-architecture computers.

In the case of compact schemes, there are also two approaches to compute derivatives, both of them relying on solutions of linear systems of equations. In one approach, we may compute matrix inversion and then solve the system of equations by matrix-vector multiplication. The other approch benefits from the properties of the resulting system of equations to be solved, *viz.* that this system is tridiagonal or at most pentadiagonal and may thus be treated by efficient methods of linear algebra. For example, the cost of solving tridiagonal systems using the Thomas algorithm is very low (of the order of $\mathcal{O}(N)$) and hence the most efficient. However, as mentioned in previous section, this algorithm cannot be efficiently parallelized. In the following considerations, the idea of the compact discretization method is presented and parallelization issues are discussed.

3.1. The compact discretization scheme

A compact scheme for the first derivative is only presented, as an approximation for the second derivative may be obtained by two consequtive applications of the first derivative approximation. For a node i, the relation between the values of a function, f, and its derivative, f', is defined [1] by linear combination of function f and f':

$$\frac{1}{3}f'_{i-1}+f'_i+\frac{1}{3}f'_{i+1}=\frac{1}{9}\frac{f_{i+2}-f_{i-2}}{4\Delta x}+\frac{14}{9}\frac{f_{i+1}-f_{i-1}}{2\Delta x}, \tag{6}$$

where Δx is the mesh size. The values of the coefficients have been obtained by expanding function f and its derivative f' into a Taylor series around node i and by matching various orders of these expansions. The presented scheme is formally of sixth-order accuracy; readers interested in derivations of compact schemes are refered to Lele's seminar paper [1]. The formula of Equation (6) can be used for inner nodes starting from node $i=3$ up to $i=N-2$, while approximations on $i=1$, 2 and $i=N$, $N-1$ require different treatment. The equations for derivatives at the boundary nodes are given as:

$$\begin{aligned} f'_1+2f'_2&=\frac{1}{h}\left(-\frac{15}{6}f_1+2f_2+\frac{1}{2}f_3\right),\\ f'_N+2f'_{N-1}&=\frac{1}{h}\left(\frac{15}{6}f_N-2f_{N-1}-\frac{1}{2}f_{N-2}\right), \end{aligned} \tag{7}$$

while the equation for the second and the penultimate node is as follows:

$$f'_{i-1}+4f'_i+f'_{i+1}=3\frac{f_{i+1}-f_{i-1}}{h}. \tag{8}$$

The values of derivatives are obtained by solving a linear system of equations given as:

$$\mathbf{A}\mathbf{f}'=\mathbf{B}\mathbf{f} \tag{9}$$

where matrices $\mathbf{A}$ and $\mathbf{B}$ consist of the coefficients defined on the left- and right-hand sides of Equations (6)–(8).

3.2. Parallel implementation of the compact scheme

A tridiagonal system of equations presented above may be written as:

$$a_i f'_{i-1} + b_i f'_i + c_i f'_{i+1} = RHS_i, \tag{10}$$

where coefficients a_i, b_i and c_i correspond to coefficients of matrix $\mathbf{A}$, and RHS_i represents the product of multiplication $\mathbf{Bf}$. The first part of the Thomas algorithm is factorization:

$$d_1 = b_1, \quad d_i = b_i - a_i \frac{c_{i-1}}{d_{i-1}}, \quad \text{for} \quad i = 2,\ldots,N. \tag{11}$$

The remainder of the algorithm may be divided into two steps. One, called the forward step, is defined as:

$$g_1 = \frac{RHS_1}{d_1}, \quad g_i = \frac{RHS_i - a_i g_{i-1}}{d_i}, \quad \text{for} \quad i = 2,\ldots,N. \tag{12}$$

The other, called backward substitution, is defined as:

$$f'_N = g_N, \quad f'_i = g_i - f'_{i+1}\frac{c_i}{d_i}, \quad \text{for} \quad i = N-1,\ldots,1. \tag{13}$$

Let us now assume that a single grid line which consist of N nodes is mapped onto P processors, so that each processor has access to the values at $i = 1,\ldots,N/P$ nodes. There are no common or overlapping nodes shared by processors and thus each node has a unique location in physical space. In parallel implementation of the Thomas algorithm, coefficients a_i, b_i, c_i, d_i and coefficients of matrix $\mathbf{B}$ are also mapped onto P processors. The solution starts from computations of RHS_i^k on every k^{th} processor. According to formula (6), computation of RHS_i^k requires data from nodes $i-2$, $i-1$, i, $i+1$, $i+2$, which means that for the $i = 1$ and $i = N$ values at nodes $i-2$, $i-1$ and $i+1$, $i+2$, respectively, are inaccessible for the k^{th} processor. Similarly, for $i = 2$ and $i = N-1$ the values at $i-2$ and $i+2$ are "invisible" for a given processor. Except for processor number 1, where values from $i-2$, $i-1$ are not required, and processor number P, where values from $i+1$, $i+2$ are not necessary (see Equations (7) and (8)), the data for the remaining processors for nodes $i-2$, $i-1$ and $i+1$, $i+2$ have to be received from their neighbors. Therefore, each processor has the so-called dummy nodes where it stores received data, *i.e.* the local numbering of nodes is $i = -1$, 0, $1,\ldots,N/P$ $N/P+1$, $N/P+2$. RHS_i^k can be computed when the values in nodes $i = -1$, 0 and $N/P+1$, $N/P+2$ are received. The next step of the Thomas algorithm is the forward step defined by Equation (12). This is a recurrence formula where value g_i is computed based on the value of g_{i-1}. In a serial code, such procedure does not decrease the efficiency of the algorithm, whereas in the case of parallel computations starting from the left most processor (*i.e.* $k = 1$) this procedure blocks the computations on all the successive processors (*i.e.* $k = 2$, $k = 3$, *etc.*). For example, let us consider a case when a domain (grid line) is mapped onto 3 processors:

1. processor 1 starts the forward step;
2. processor 2 will not start the forward step until the value of $g_{N/P}$ has been computed at processor 1; when it has been computed at processor 1, it is sent to processor 2 and which can start the forward step;
3 during this time, processor 3 waits until processors 1 and 2 have performed the forward step.

It is obvious that the overall performance of the above procedure cannot be higher than that of computations performed with a serial code. In this case, the only benefit from parallelization is that RHS^k has been computed on separate processors. When the forward step has been completed at all processors then:

1. processor 3 starts the backward substitution step, which is also the recurrence formula;
2. processor 2 will not start the backward step until the value of f_1' has not been computed at processor 3 (it starts the backward step when it has received f_1' from processor 3);
3. processor 1 waits until processor 2 and 3 have performed the backward step.

Similar to the forward step, the efficiency of backward substitution is worse than using a serial code. The time of processors' waiting for data from their precursors or successors is called idle time. In our simple case, it is possible to compute this time directly as function of the forward and backward steps. We assume that the procedure is finished when processor 1 has finished the backward step. We denote forward and backward steps at particular processors as: FS on P1, FS on P2, BS on P1, BS on P2 and so on. Hence,

- processor 1: after sending g_N/P to processor 2 waits FS on P2, FS on P3, BS on P3 and BS on P2, resulting in two FS and two BS;
- processor 2: before starting the forward step waits FS on P1, before starting the backward step waits FS on P3 and BS on P3, after that waits BS on P1, resulting in two FS and two BS;
- processor 3: before starting the forward step waits FS on P1 and FS on P2, after performing the forward and backward steps waits BS on P2 and BS on P1, resulting in two FS and two BS.

The idle time is the same for all processors and equals the real time required to perform two forward and two backward steps on $N/3$ number of nodes. If we add the real time when the processors compute the forward and backward step, it becomes clear that each processor effectively works only $1/3^{\text{rd}}$ of the total time, the remaining $2/3^{\text{rds}}$ being idle time. Hence, when the Thomas algorithm in the presented form is mapped onto P processors, useful work is performed only during $1/P$ of the total time required to finish forward and backward steps on all processors, the remaining $(P-1)/P$ of the total time being idle time. Fortunately, in CFD computations we need to compute derivatives of different variables, for instance we need derivatives of 2 components of velocity. The simplest approch may be schematically presented as the following succesive operations:

```
CALL TDMA_RHS! to compute RHS u-component
CALL TDMA_FORWARD! for u-component
CALL TDMA_BACKWARD! for u-component

CALL TDMA_RHS! to compute RHS v-component
CALL TDMA_FORWARD! for v-component
CALL TDMA_BACKWARD! for v-component
```

The calls to TDMA_RHS, TDMA_FORWARD and TDMA_BACKWARD denote particular steps of the Thomas algorithm described above. It is assumed that at

the beginning of each call to TDMA_FORWARD there is also a call to the MPI subroutine to receive data (*i.e.* g_N) from processor $k-1$ and that at the end of each TDMA_FORWARD there is a call to the MPI subroutine to send data (*i.e.* g_N) currently computed to processor $k+1$. By analogy, at the begining and at the end of TDMA_BACKWARD there are calls to the MPI to send and receive $u'(1)$ and $v'(1)$. With the procedure presented above, the ratio of the idle time of each processor to the time needed to compute 2 derivatives is exactly the same as previously in the case of computing a derivative of a single variable and, hence, there is no benefit from the presented approach.

Let us now rearrange the particular steps of this procedure to decrease idle time. We would then have:

```
CALL TDMA_RHS! to compute RHS u-component
CALL TDMA_RHS! to compute RHS v-component

CALL TDMA_FORWARD! for u-component
CALL TDMA_FORWARD! for v-component

CALL TDMA_BACKWARD! for u-component
CALL TDMA_BACKWARD! for v-component
```

The only modification is that all steps, *i.e.* the computations of RHS_k, forward and backward steps, are executed separately. However, thus has crucial consequences as the ratio of idle time of each processor is considerably reduced in comparison with the previous procedure. Assuming that the domain is divided into three subdomains, when all processors have computed RHS_k, then:

1. processor 1 computes TDMA_FORWARD for the u-component and sends $g_N(u)$ to processor 2; then it computes forward steps for the v-component and sends $g_N(v)$;
2. processor 2 is idle when processor 1 computes TDMA_FORWARD for the u-component; right after processor 2 has received $g_N(u)$, it starts to compute the forward step; when it has finished, there is $g_N(v)$ already available from processor 1, so there is no more idle time for processor 2 during the forward step;
3. processor 3 waits until processor 2 completes the forward step for $g_N(u)$.

When processor 3 receives $g_N(u)$ from processor 2 it continues the forward step for the u-component, after which $g_N(v)$ is already available as it was computed on processor 2 when processor 3 computed the forward step for the u-component. So, as processor 3 finishes the forward step for the v-component:

1. processor 3 starts to compute TDMA_BACKWARD for the u-component; then it sends u'_1 to processor 2 and starts to compute TDMA_BACKWARD for the v-component;
2. processor 2 waits for u'_1 from processor 3; then it computes TDMA_BACKWARD for the u-component and sends u'_1 to processor 1; after that there is already v'_1 available from processor 3, so processor 2 may continue computations and there is no more idle time for processor 2 during the backward step;
3. processor 1 waits until processor 2 has finished the backward step for the u-component; there is no more idle time for processor 1 as the backward step for

the v-component is computed at processor 2 simultaneously with the backward step for the u-component at processor 1.

If we add waiting time during forward and backward steps, we can see that each processor waits two FS and two BS and hence the idle time while computing two derivatives is the same as during computations of one derivative. The ratio of idle time to the total time needed to compute derivatives of u and v components equals twice 1/2. Extending this to P processors, we may show that the ratio of idle time to the total time needed to compute D derivatives equals $(P-1)/(P-1+D)$, whereas the ratio of time of processors' effective work to the total time equals $D/(P-1+D)$. This is considerable improvement compared with the computations of either a single derivative or a sequence of derivatives. However, some idle time and we can try to reduce it further. The simplest solution is to use processors to perform useful work not related to backward and forward steps of the Thomas algorithm. In other words, all computations which are not related to the derivatives in directions in which the compact scheme is applied should be performed during idle time. In the case of 2D or 3D flow problems it is not difficult to define such tasks: these could be computations of derivatives in the remaining directions, explicit filtering of the LES method, computations of subgrid viscosity (computationally very expensive with some subgrid models), *etc.* Indeed, as the complexity of tasks unrelated to exchange of data increases, the algorithm becomes more efficient. Ideally, the time needed to perform these computations should exceed the idle time of every processor. However, it must be noted that the ratio of these independent tasks to idle time decreases as the number of processors increases. For example, let us assume a 3D domain discretized by an $M \times K \times N$ computational mesh and divide it in one direction into P subdomains such that the mesh resulting in each of them consists of $M \times K \times N/P$ nodes. The independent tasks in subdomains are directly proportional to the number of nodes, while idle time is proportional to $M \times K(P-1)/(P-1+D)$. When the time needed to perform the tasks exceds the idle time of the processors, one would expect that they will perform computations continuously and the overall performance of the code will be good. The proposed algorithm can be schematically presented for the k^{th} processor as follows:

```
CALL TDMA_RHS! this is performed by processors
! in the same time for u,v,w,etc.

CALL MPI_IPROBE! this is to check whether there are incoming
! data (i.e. g_N) from preceding processor (k-1)

IF(MPI_IPROBE return FALSE) THEN
! there are not data available yet so let us do useful work
! but check "from time to time" by MPI_IPROBE whether there
! are incoming data; if they are then immediately stop
! these computations and go to the forward step (TDMA_FORWARD)
 ...
 ...
 ...
END IF
!
 receive g_N from processor (k-1)

CALL TDMA_FORWARD! for u,v,w,etc.
!
```

```
 send necessary data (i.e. q_N) computed at current k-th!
 processor to successive processor (k+1)

CALL MPI_IPROBE! this is to check whether there are incoming
! data (i.e. f'_1) from processor (k+1)

IF(MPI_IPROBE return FALSE) THEN
! continue works which are independent of TDMA
! but check "from time to time" by MPI_IPROBE whether there
! are incoming data; if they are then immediately stop
! these computations and go to the backward step (TDMA_BACKWARD)
 ...
 ...
 ...
END IF
!
 receive f'_1 from processor (k+1)

CALL TDMA_BACKWARD! for u,v,w,etc.
!
 send f'_1 computed at current k-th processor!
 to processor (k-1) and then continue all remaining works!
 if there are any left; at this point f' is already available.
```

In the above procedure one may find calls to non-blocking MPI subroutine MPI_IPROBE, used to check whether there are incoming data from neighboring processors. Compared with other MPI subroutines (*e.g.* MPI_SEND, MPI_RECV) or the idle time of processors, calls to MPI_IPROBE return very quickly, but they still require some time. Therefore, the frequency of these calls should be minimized and is in practice adjusted depending on the number of mesh points and subdomains.

It should be remembered that in a parallel CFD code it is not only derivatives that require parallel communications. For example, solution of Poisson's equations in the projection method (parallel matrix multiplication), evaluation of the subgrid viscosity near subdomain boundaries, convergence analysis, data saving, *etc.*, all require communication. However, from the algorithmic point of view there is little hope to find "places" which could be modified (improved) for better performace. In this work, attention has been paid to using effective MPI subroutines and so, wherever possible, we applied the so-called non-blocking communication between processors, which allows to post send/receive messages (MPI_ISEND, MPI_IRECV) and subsequently perform other tasks with non-blocking messages being completed by a call to MPI_WAIT. The advantage of non-blocking communication over the blocking one is that in the latter calls to blocking subroutines (MPI_SEND, MPI_RECV) will not return until data are sent or received.

Additionally, as a majority of the sendings and receipts are performed repeatedly in a time integration loop, we have used the so-called persistent communication pattern, which reduces the time needed to "agree" readiness for communication between processors. However, it must be stressed that neither non-blocking send/receive nor persistent communications will reduce the time needed to transfer data and therefore one should never expect linear speed up, *i.e.* doubling the number of processors will not increase the speed twice. Here, the measure of speed-up corresponds to the ratio of time needed to perform computations on P processors to the time of computations performed on $2P$ processors. In the following section we report speed-ups obtained using P processors relative to computations on a single processor.

4. Results

The results presented in this work concern computations performed for flow in a channel at a Reynolds number of $\mathrm{Re}_\tau = 180$. The meshes used in the computations were $32 \times 64 \times 32$ nodes respectively in the stream-wise, wall-normal and span-wise directions, and $64 \times 128 \times 64$ nodes. The domain was divided in the wall-normal direction into 2, 4 and 8 subdomains. First, the results of code validation are presented, followed by the efficiency of the parallel algorithm. The results presented in the following comparison were obtained using 4 subdomains.

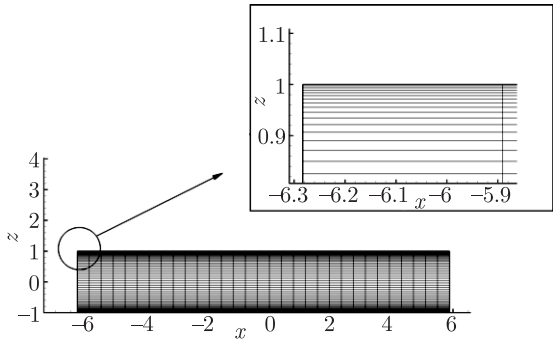

Figure 2. View of the numerical mesh $32 \times 64 \times 32$ nodes

The computational meshes were refined near the walls by the tangent hyperbolic function and adjusted so that there were 10 nodes from the wall to $y+ = 10$ and the first node was located at $y+ < 1$ (enlarged mesh close to the wall is shown in Figure 2, top right). The computations were started with the mean velocity approximated by analytical formula $u+ = y+$ for $y+ = 0 \to 10$ and $u+ = 2.5 \cdot \ln(y+) + 5$ from $y+ = 10$ to the channel's axis. The non-dimensional mean stream-wise velocity resulting from such initialization was around 15.5. The initial turbulent disturbances were imposed randomly for stream-wise velocity at the level of 30%. After the initial stage, which took approximately 10 non-dimensional time units, we started the averaging procedure preformed over 20 time units (more than 20 flow-throughs). The profiles of mean stream-wise velocity obtained in computations using $32 \times 64 \times 32$ nodes are presented in Figure 3, where the results of LES are compared with the DNS data [5]. The results obtained with the Germano subgrid model are in very good agreement with those from DNS, both in linear and logarithmic regions. Indeed, it is difficult to distinguish between them and this is why enlarged parts of Figure 3 are presented in Figure 4. The plots shown in this figure confirm previous observations. The results obtained with the Smagorinsky model are also in acceptable agreement with the DNS data, although in this case agreement is considerably worse.

The fluctuating components of stream-wise and normal velocities are shown in Figure 5, and also here the results obtained with the Germano procedure are in much better agreement with the DNS data than those obtained using the Smagorinsky model. The disagreements of the data obtained with Smagorinsky model for the mean and fluctuating velocity components are similar to these observed when applying low-order discretization methods. In low-order methods, the high level of numerical

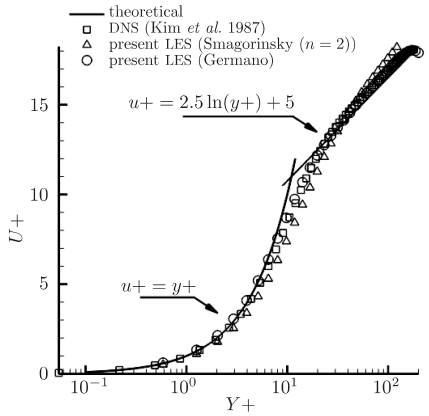

Figure 3. Normalized stream-wise velocity across the channel from the wall to the channel's axis

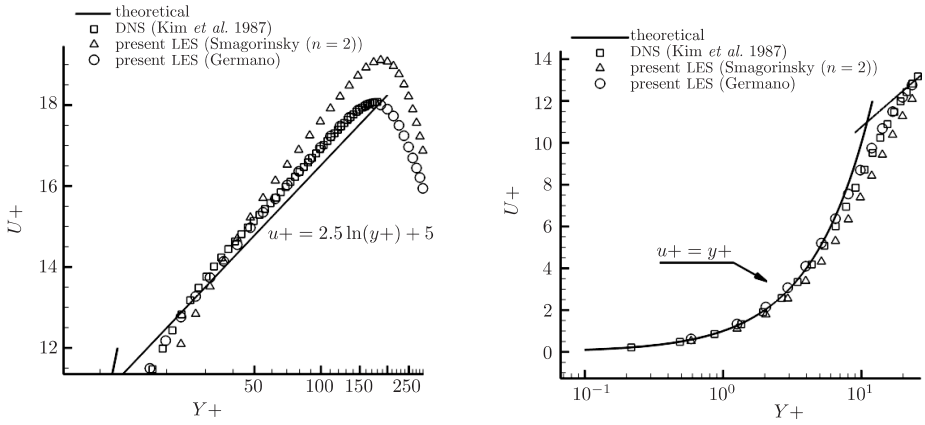

Figure 4. Normalized stream-wise velocity across the channel: enlargements of the linear region (left) and the logarithmic layer (right)

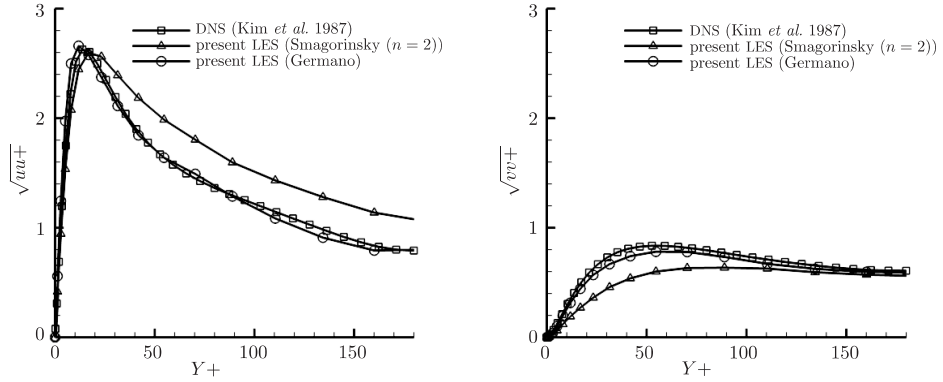

Figure 5. Normalized stream-wise and wall-normal velocity fluctuations

dissipation play a role similar to that played in subgrid models. In our case, the excess of dissipation caused by the Smagorinsky model is manifested in the same way. Refining the mesh improves the accuracy of computations with the Smagorinsky

Table 1. Speed-up obtained for computations performed on $P4\times1$

Method	$1P$	$2P$	$4P$	$8P$
No-optim.	1.00 (1.00)	1.28 (1.28)	1.44 (1.84)	– –
Optimized	1.00 (1.00)	1.71 (1.71)	1.80 (3.07)	– –

Table 2. Speed-up obtained for computations performed on $P4\times2$

Method	$1P$	$2P$	$4P$	$8P$
No-optim.	1.00 (1.00)	1.42 (1.42)	1.56 (2.21)	1.40 (3.10)
Optimized	1.00 (1.00)	1.74 (1.74)	1.90 (3.31)	1.86 (6.14)

model considerably and agreement of the obtained results (not shown) with DNS data is practically the same as for the Germano model. The presented results confirm correctness of the numerical algorithm from the point of view of the discretization method, subgrid modelling, time integration, *etc.*, but also from the point of view parallel implementation of the code. In order to test parallel efficiency, we performed computations on two different PC clusters: one consisted of 4 nodes with two IA-64 Itanium 2 1.3GHz processors on each main board (results of which are denoted as $P4\times2$), the other – of 4 nodes with an Athlon 2.0GHz processor on each main board (results denoted as $P4\times1$). The results are reported in Tables 1 and 2 for the cases when derivatives were computed successively, referred as *No-optim.*, and according to the proposed method, denoted as *Optimized*. In this case, we attempted to eliminate idle time by performing computations of derivatives in the remaining directions, stress tensor components and the subgrid viscosity: the procedure has increased efficiency by about 5–7%, depending on the number of processors used. The presented comparisons of the code's efficiency concern computations performed with the Smagorinsky model. Application of the Germano model was computationally more expensive, as this model requires considerably more computations, but the speed-up was approximately the same as per the Smagorinsky model. Hence, it can be assumed that the idle time was eliminated and there was no more "free time" between forward and backward steps to perform more computations.

The columns in Tables 1 and 2 show normalized speed-up when the number of processors has been doubled and relative to computations on a single processor (numbers in brackets). There is great difference between not optimized and optimized computations for both clusters. When we compare computations using four processors ($4P$), the differences in speed-up relative to computations using one processor are approximately 50% for both clusters. Better performance obtained on $P4\times2$ (3.31 against 3.07) is due to a different type of the network connecting the computers. In the case of computations on $P4\times2$ with 2 or 4 processors, they were always selected from different computers. Comparing computations of the not optimized and optimized algorithm using eight processors ($8P$) on $P4\times2$, the difference is almost 100%. This clearly shows the benefits resulting from proper ordering of parallel subtasks. It is also interesting that worse speed-up considerably between computations on one and two processors was obtained when doubling the number of processors. This is to some extent in contradiction with previous considerations, where it was said that upon increasing the number of processors the ratio of the number of computations to the amount of sent/received data decreases and this negatively influences parallel

performance. Obviously, the ratio is higher for two processors than for four. Therefore, one should expect that seed-up between 2 processors → 1 processor will be greater than between 4 processors → 2 processors. However, the presented tables suggest the opposite conclusion. The reason for these discrepancies is that in parallel computations data must be prepared to be sent (*i.e.* copied to a sending buffer) and recovered (*i.e.* copied from a receiving buffer) and this consumes time. This is not the case in one-processor computations where these operations are not executed and some of the subroutines in the code are omitted. Therefore, while analyzing parallel efficiency, one should perhaps make computations on two processors the basis of comparison. Then, the relative speed-up decreases as one could expect: 1.90 on four processors and 1.86 on eight processors (see Table 2).

5. Conclusions

Technical issues of parallel implementation of the compact-pseudospectral code used in computations of turbulent flows have been presented. Verification of the algorithm, numerical discretization and correctness of implementation of the Smagorinsky and Germano subgrid models has been confirmed by computations of flow in a channel with periodic and non-slip boundary conditions. Thw proposed algorithm of parallel implementation of the compact scheme has been shown to reduce the idle time of processors and improve parallel performance considerably. It has also been shown that a proper schedule of forward and backward steps of the Thomas algorithm has considerable impact on idle time reduction. Additionally, code's efficiency can be increased by performing useful computations during the remaining idle time of processors.

Acknowledgements

The Autor is grateful to Prof. Piotr Doerffer for strong motivation which was helpful in preparing the presented work. Autor is also grateful to the TASK Computing Center in Gdansk for granting access to the computing resources on their Holk PC Cluster.

This work was performed as part of statutory research BS-1-103-301/2004/P and EU WALLTURB Project No. AST4-CT-2005-516008.

References

[1] Lele S K 1992 *J. Comp. Phys.* **103** 16
[2] Canuto C, Hussaini M Y, Quarteroni A and Zang T A 1988 *Spectral Methods in Fluid Dynamics*, Springer-Verlag
[3] Smagorinsky J 1963 *Month. Weather Rev.* **91** 99
[4] Germano M, Piomelli U, Moin P and Cabot W 1991 *Physics of Fluids* **A 3** 1760
[5] Moin P, Kim J and Moser R 1987 *J. Fluid Mech.* **177** 133