The performance of a spectral simulation code for turbulence on parallel computers with distributed memory

By Krister Alvelius and Martin Skote

Department of Mechanics, KTH, SE-100 44 Stockholm, Sweden

The performance of a pseudo spectral turbulence simulation code on various supercomputers, with either shared memory or distributed memory, is presented. The communication with the memory is intense, and careful consideration of the two memory configurations is needed to obtain high performance. The investigations of the performance show that the scaling with the number of processors is excellent for both memory systems. Also, vector processors are compared with super scalar processors, and the performance is generally higher for the vector processor since the code vectorizes well. However, the computers with the scalar processors, and distributed memory, have a much larger number of processors which gives an overall better performance for the total machine. The numerical code, with e.g. $24 \cdot 10^6$ degrees of freedom, was run at 3.5 Gflop/s on 64 processors on an IBM SP2 machine.

1. Introduction

Direct numerical simulation (DNS) of turbulent flows is a major field for the use of super computers world-wide, and has throughout the years been an important factor in driving the process of developing super computers. This is an area which efficiently makes use of all available memory of the computer, and the scaling with the number of processors is excellent. Also the communication between the processors is relatively intense. The performance of DNS codes is hence an important measure of a computers ability to handle real problems of physical interest.

1.1. *Scientific background*

The nature of turbulence is very complex. A turbulent flow consists of both large and small scale motions that fluctuate in time and space. The incompressible flow of e.g. air can be described by the Navier-Stokes (NS) equations together with suitable boundary conditions. They are non-linear equations, which generally need to be discretized in order to yield a solution. In these equations the non-linear term generates turbulent motions and the viscous term dissipates flow fluctuations. The viscous dissipation of turbulence adjusts

itself to the production, which determines the smallest scales in the flow, η, usually referred to as the Kolmogorov micro length scale. The viscous term becomes large in the presence of large velocity gradients associated with small scale motions. It does, however, only scale linearly with the magnitude of the velocity field, while the scaling of the non-linear term is quadratic. Therefore, an increased magnitude in the velocity causes the smallest scales to become even smaller (through the cascading action of the non-linear terms) to yield increased velocity gradients and a larger dissipation which balances the production by the non-linear terms.

The size of the largest scales, l, is usually determined by the geometry of the flow domain. The size of the range of scales in the flow is measured by the Reynolds number ($Re \sim (l/\eta)^{4/3}$) which depends on the type of flow, the magnitude of the flow velocities, the domain size and the kinematic viscosity.

In a DNS of a turbulent flow all scales (l – η) need to be captured by the numerical method. Since turbulent flows are always three-dimensional, even moderate Reynolds numbers give a significant degree of freedom for the resulting discrete dynamic system that needs to be solved. In addition there is a span of timescales that needs to be resolved. The time step is determined by the smallest turbulent timescales and stability requirements of the numerical method. Typically a large number of discrete time integrations needs to be performed in order to include one large time scale in a simulation. In order to obtain statistically converged results it is also necessary to integrate the solution over many large time scales.

DNS have, until recently with the development of modern computers, been an impossible task even for small Reynolds numbers. Therefore, researchers have been led to study the averaged equations instead, which give much smoother solutions and significantly reduces the computational effort. This approach introduces an unknown quantity, the Reynolds stress tensor, which needs to be modelled. Development of such models, with different degrees of complexity, has been an important task for turbulence researchers. Calibrations of models are essential and can be performed in e.g. windtunnel experiments. It shows that the modelled quantities behaves differently in different flow situations. Although it is relatively easy to obtain high Reynolds numbers in the windtunnel experiments, they sometimes fail to give accurate descriptions of complex quantities in the flow. Also, they cannot give a total description of the flow situation since the complete velocity field is not available.

The DNS of turbulent flows gives the time development of the complete velocity field and allows the study of any particular flow phenomenon you choose in detail. This can be used to increase the understanding of the underlying mechanism, resulting in better turbulence models. Also a new method, large eddy simulation (LES), similar to that of DNS, have been developed in recent years where only the smallest scales are modelled in the flow and the large scales are resolved. This method has been found to be successful in computing real engineering flows with complicated geometries, using only simple models

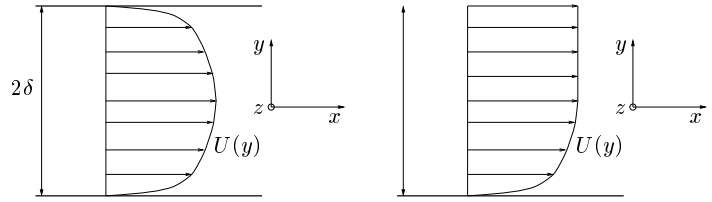

FIGURE 1. The flow configurations of plane channel flow and boundary layer flow.

for the unknown subgrid-scale stresses, since the main effect of the domain geometry on the flow enters through the large scales which are resolved. The numerical implementation of this method is similar to that of DNS. In particular, when developing models for the subgrid-scale quantities, DNS data is essential since the modelled quantities fluctuate in time and space, and the whole velocity field has to be available at a single instant.

Both in DNS and in experimental investigations it is important to know what effects that are important in the flow in order to be able to make correct conclusions. Therefore the flow should be constructed so that effects that you are not interested in are negligible or controllable. One such important test case is the plane channel flow (figure 1) where only effects of plane mean shear and of the solid walls are present. In DNS it is also important to have a simple geometry which simplifies the numerical implementation and improves the accuracy in the numerical discretization. This is true in the plane channel flow which has no curvature and only needs grid stretching in the non-homogeneous wall normal direction. In addition to the shear and wall effects in the plane channel flow, the effect of curvature can be studied by adding system rotation to the governing equations.

The boundary layer flow (figure 1) on a flat plate is another example of a simple flow with a solid wall and plane shear. In this case a free boundary gives a more complicated flow, e.g. the boundary layer grows downstream. The flow can be studied in various aspects. The fully turbulent flow, as well as transitional flow, where a breakdown from laminar to turbulent flow occurs, is of great importance in many industrial applications. There is still no complete picture of the mechanisms behind this breakdown, and further investigations are needed. Both the transitional and turbulent flows can be studied with additional complications such as external pressure gradients or three-dimensional mean flow.

Both the turbulent channel flow and the fully turbulent boundary layers can be studied for gaining data used for calibration and development of turbulence models. But the data are not only used for modeling purposes, the instantaneous turbulent structures can be thoroughly studied since the whole flow field is accessible at each time step.

1.2. *The numerical discretization*

Ideally, the plane channel is considered to be infinitely long and infinitely wide, with the flow driven by a mean pressure gradient in the streamwise direction. In the numerical simulation periodic boundary conditions is imposed in the streamwise and spanwise directions. In the wall normal direction a non-slip condition is applied at the solid walls.

For the boundary layer flow the downstream direction must be treated in a different way. The boundary layer is increasing (getting thicker) downstream, and that direction can thus not be considered periodic. It is possible to create a periodic flow if a so called fringe region is added downstream of the physical domain, figure 2. In the fringe region the flow is forced from the outflow of the physical domain to the inflow. In this way the physical domain and the fringe region together satisfy periodic boundary conditions. The fringe region is implemented by the addition of a volume force which form is designed to minimize the upstream influence.

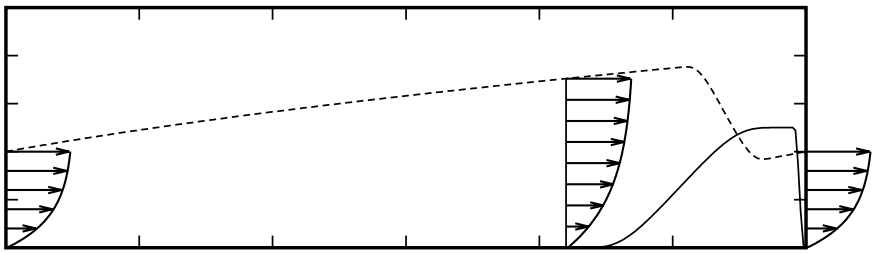

FIGURE 2. Computational box with the fringe region. — the strength of the volume force in the fringe region. - - the boundary layer thickness.

The periodic boundary conditions make it possible to use Fourier representation of the velocity field, which gives an accurate description of the spatial derivatives in the discretization direction. Compared to a finite difference method, which typically gives only a second-order approximation of the spatial derivatives, the numerical accuracy is significantly increased. The wall normal direction is discretized using Chebyshev polynomials. Hence, spectral methods are used in all spatial directions, which gives an overall highly accurate discretization of the governing equations.

The time integration is discretized with a second order Crank-Nicolson scheme for the linear terms and a four stage third-order Runge-Kutta scheme for the non-linear terms. The Crank-Nicolson method is implicit and hence absolutely stable, whereas the Runge-Kutta method is explicit, which imposes a restriction on the time step to yield stable solutions. The time step is determined by the CFL number and adjusts itself automatically to the actual flow situation.

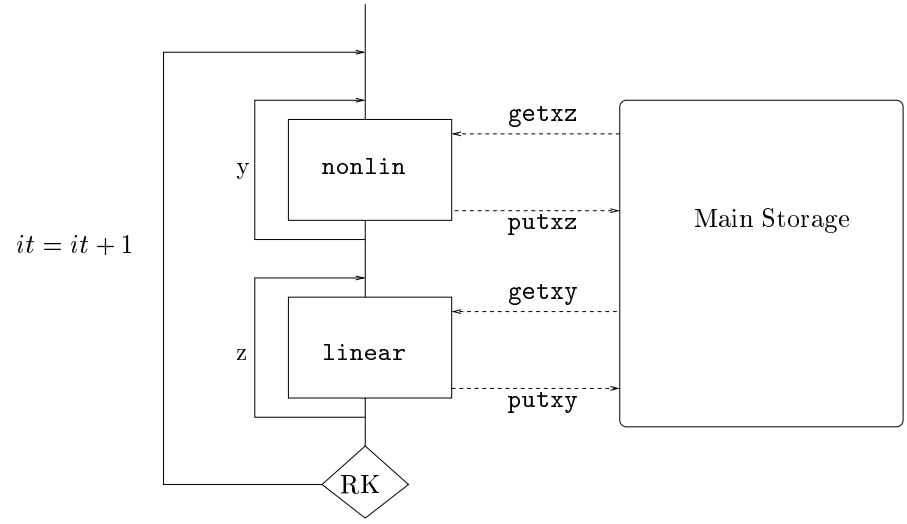

FIGURE 3. The main structure of the program.

Instead of solving for all three velocity components and the pressure, a vorticity-velocity formulation is used, in which the fluctuating pressure is eliminated. In this formulation only two equations need to be solved instead of the original four equations (the NS equations and the continuity equation).

The discretization results in a tri-diagonal equation system in spectral space for each of the two variables, which needs to be solved at each Runge-Kutta step. The non-linear terms in the equations are calculated in physical space, using fast Fourier transforms (FFT) in the transformations between spectral and physical space. The velocity field is represented on a 3/2 times finer mesh in the streamwise and spanwise directions in physical space compared to spectral space. This results in a 3/2-dealiasing method which is energy conserving.

The numerical code is written in FORTRAN and consists of two major parts (figure 3), one linear part (`linear`) where the actual equations are solved in spectral space, and one non-linear part (`nonlin`) where the non-linear terms in the equations are computed in physical space. The actual flow variables are stored at an intermediate level with spectral representation in the streamwise (x) and spanwise (z) directions and physical representation in the wall normal (y) direction. All spatial derivatives are calculated in the spectral formulation. The main computational effort in these two parts is in the FFT.

In the linear part one xy-plane is treated separately for each z variable. The field is transformed in the y direction to spectral space, a solution is obtained and then transformed to physical space in the y direction. This is performed

with an loop over all z values where the subroutine `linear` is called for each z. The xy-planes are transferred from the main storage with the routine `getxy` to the memory where the actual computations are performed. The corresponding storing of data is performed with `putxy`.

In the non-linear part the treatment of the data is similar to that in the linear part. One xz-plane is treated separately for each y variable. The field is transformed in both the x and z directions to physical space where the non-linear terms are computed. Then the field is transformed in the x and z directions to spectral space. This is performed with a loop over all y values where the subroutine `nonlin` is called to at each y. The xz-planes are transferred from the main storage with the routine `getxz` to the memory where the actual computations are performed. The corresponding storing of data is performed with `putxz`.

1.3. *Computer background*

The super computers used for large computations can be divided in two major groups with respect to the memory configuration. The computers can also be divided in two groups when considering the architecture of the processor. Thus there are four different combinations that constitute the type of computer.

The two memory configurations are shared and distributed memory. In the former a common memory is used by all the processors. In the latter case, every processor has its own memory and data must be sent and received if used by another processor. The two types of processors are scalar and vector. The scalar (or super-scalar) processor has registers for data as a scalar quantity and can perform operations on this data fast. There is a small memory set, the cache, in the processor to keep easy access to the data being processed. The data transfer between main memory and the cache is slow, therefore optimized usage of the cache is important. The vector processor has registers for data as a vector quantity and perform operations on the scalar elements of the vector, all at the same time. The processor itself thus operates in parallel. The transfer of data from the main memory is fast if there is no memory contention.

The distributed memory computers typically have many processors (≈ 200) with e.g. 256 Mbyte of memory each, resulting in a larger total memory than a vector computer with typically 4 to 8 Gbyte memory. The forthcoming computers often have a shared memory for a small number of processors but with a (from necessity) overall distributed memory.

The code used for the computations has to be adjusted when ported from one group to another. The four groups are listed in table 1, together with the computers that have been used in the present study. At the time of the present investigation the Cray J90, IBM SP2 and Fujitsu VPP300 have 32, 152 and 3 processors respectively and are located at PDC, KTH in Stockholm. The Cray C90 and T3E have 7 and 232 processors respectively and are located at NSC in Linköping. The Cray T90 has 14 processors and is located at SDSC in

San Diego. The SGI Origin 200 has 4 processors and is the property of Joseph Haritonidis at OSU in Columbus.

	shared memory	distributed memory
scalar processor	SGI Origin 200	Cray T3E, IBM SP2
vector processor	Cray J90, C90, T90	Fujitsu VPP300

TABLE 1. The four categories of super-computers

The optimization and tuning of the code have different features for the different groups. They can be opposed to each other, e.g. tuning for a vector processor will make the code unsuitable for a scalar processor and vice versa.

The code used for the numerical simulation of turbulence was earlier optimized for vector processors and shared memory computers. In resent years distributed memory and scalar processors have become a common architecture for super computers. Therefore a lot of effort has been put into the redesign of the code to perform well on such computers.

Most of the time in the code is spent in the FFT. The vector and scalar optimization is therefore concentrated to this part of the code. There are two different versions of the FFT to be used on the two types of processors.

The parallelization on a shared memory system is fairly straightforward and is very efficient. The MPI (Message-Passing Interface) has been used to parallelize the code on the distributed memory systems. A lot of effort has been put into keeping the memory requirement low as to make it possible to perform large simulations.

1.4. *Examples of simulations*

In order to illustrate the complexity of the flow and give examples of the computational effort two examples are given, one for the turbulent boundary layer flow and one for the rotating channel flow.

1.4.1. *Turbulent boundary layer*

As an example of a flow field from a simulation of a turbulent boundary layer, see figure 4 where contour lines of the downstream velocity component are shown in a plane perpendicular to the wall. The downstream direction is denoted x and the wall normal direction y (observe the different scaling in the two directions in the figure). The simulation starts with a laminar boundary layer and is then tripped to turbulence by a random volume force near the wall. All the quantities are non-dimensionalized by the freestream velocity (U) and the displacement thickness (δ^*) at the starting position of the simulation ($x = 0$) where the flow is laminar and $Re_{\delta^*} \equiv U\delta^*/\nu = 400$. The length (including the fringe), height and width of the computation box were $450 \times 24 \times 24$ in these units. The number of modes was $480 \times 161 \times 96$.

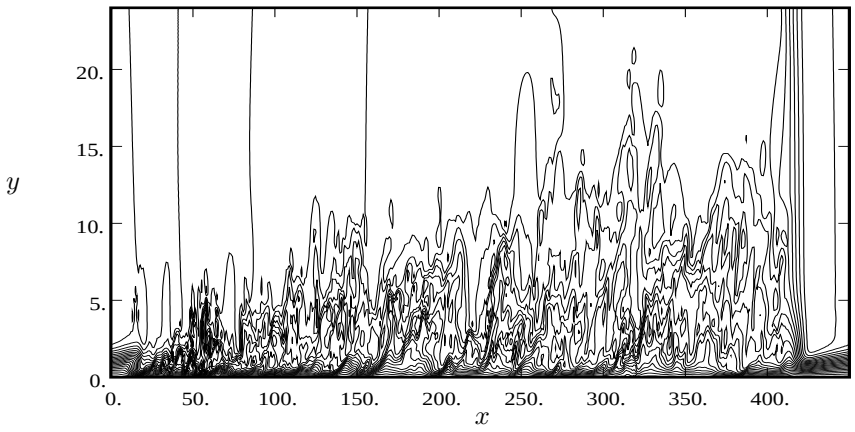

FIGURE 4. Contour lines of the velocity in the downstream direction (x).

To obtain a flow field as the one in figure 4, the simulation need to run for 2500 time units (δ^*/U), starting from a laminar field. The flow traveled through the box approximately 5 times during this period of time, and is sufficient for the turbulence to be adjusted to the imposed boundary conditions such as a pressure gradient. To obtain sufficiently smooth statistical data, the simulation runs for another 2000 time units. Thus the simulation was run for a total of 4500 time units. The time step determined by the CFL number is 0.02, and the CPU-time for one processor on the Cray J90 is 280 seconds, and 66 seconds on the Cray C90. Thus a simulation of this kind needs $63 \cdot 10^6$ CPU seconds on the J90 which is equivalent to 24 CPU-months. If the simulation is run on eight processors on the J90, or two on the C90, the simulation takes three months on either machine. If the time spent in the queuing system is included, at least half a year must be expected before the simulation is completed.

In figure 4 the laminar flow is visible at the beginning of the box, then a rapid transition to turbulence occurs and the turbulence is fully developed at $x = 150$. At $x = 400$ the fringe region starts, the turbulence is suppressed and the flow is forced back to its initial laminar profile. The velocity at the upper boundary, the freestream velocity, is not constant due to an adverse pressure gradient applied through the boundary conditions. The boundary layer is increasing in thickness rapidly in the downstream direction due to the decrease of the freestream velocity. If a strong enough adverse pressure gradient is applied, the boundary layer would separate from the wall. This is what happens on the wing of an aircraft when the stalling angle of attack is approached. Separation also occurs on the rear window of a car and increases the drag. The evaluation of the turbulent statistics from this and similar simulations are presented in Skote *et al.* (1998).

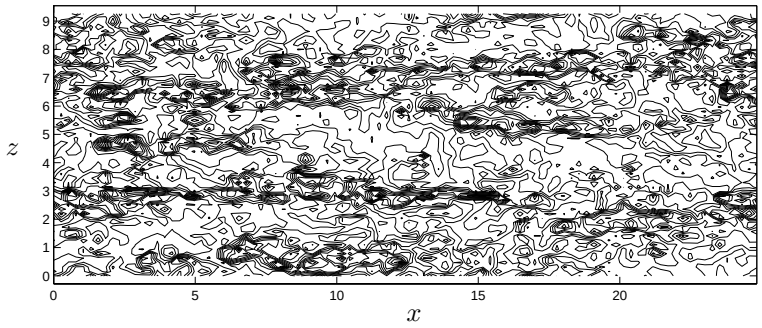

FIGURE 5. Contours of constant streamwise velocity in a rotating channel flow for a plane in the streamwise and spanwise directions at the distance $y = \delta/2$ from unstable wall, where 2δ is the channel width.

1.4.2. *Rotating channel flow*

In rotating channel flow, the extra Coriolis term in the governing equations has a strong effect on the development of the large structures. It has either a stabilizing or a destabilizing effect depending on the sign of the mean velocity gradient. On the unstable side, the Coriolis force makes negative streamwise fluctuating velocities and positive wall-normal fluctuating velocities enforce each other. This effect, which actually is strongest for relatively low rotation rates, produces long structures in the flow (figure 5). The periodic boundary condition requires that the computational domain needs to substantially longer than the largest flow structure if the results is to be unaffected by the domain size. If the domain is too short the numerical artifact of the periodicity condition will, through resonant effects, enforce the long structures and the true behavior of the flow is not captured. Typically, for this case, the computational domain needs to be approximately five times as long compared to the non-rotating flow case, which significantly increases the computational effort.

The present simulation was performed with $384 \times 129 \times 240$ number of modes, which gives a computational effort of the same order as for the boundary layer example above. The length of the largest scales (figure 5) actually suggests that the computational domain should be at least twice as long for this case, which implies that $768 \times 129 \times 240$ grid points have to be used in the simulation.

2. Results

The main aim of optimizing a code is to obtain an overall high performance. This is a complicated matter and different parts of the code might need different treatment. This suggests that the different parts should first be optimized separately. In the present paper the various choices of the compiler options are omitted, and results are only presented for the optimal choice.

The investigations presented here is divided in four sections. In section 2.1 the vectorization is discussed. The FFT is investigated in section 2.2 for the two types of processors through a model problem using the different FFT routines. Section 2.3 is devoted to the parallelization on the two memory configurations. Also here a model problem was created to try out the different techniques for MPI. And finally in section 2.4, the performance of the MPI and the overall performance of the code is presented for the two distributed memory computers IBM SP2 and Cray T3E.

For the shared memory computers with vector processors, the vectorization and parallelization have been tested only for the complete code. The code was already parallelized and optimized for vector processors (Lundbladh *et al.* 1992, 1994), and the results are only included for comparison with the main result concerning the parallelization on distributed memory computers.

Two different sizes of the problem has been tested, table 2. Test case two is too large to be run on the shared memory computers used in this investigation.

case	size	number of points
one	$128 \times 97 \times 128$	$1.6 \cdot 10^6$
two	$512 \times 193 \times 256$	$25.3 \cdot 10^6$

TABLE 2. Numerical mesh for the two test cases

The goal when optimizing a numerical code is to minimize the computational (CPU) time. An usual measure of the performance of a code is the number of floating point operations that is performed per CPU second (flop/s). If two codes do a different number of floating point operations for the same computation it is not really relevant to directly compare the respective performance in flop/s but rather in the computational time. The different versions of the code presented here have a negligible difference in computational work (flop) for the same tasks. The main differences involve moving data in the memory and changing the order of computation which give no contribution to the computational work. Therefore, either the computational speed or the computational time may be compared whichever is found most suitable.

2.1. *Vectorization*

Due to the spectral representation in Fourier series the degree of vectorization is very high. The vectorization is dependent on the size of the problem, a large problem will in a natural way contain long vectors. For small problems the size of the vectors can be increased by arranging the arrays in the code in a proper way. This is done by letting the vectors contain more than one plane. The arrays are collected from the main memory by the routines `getxy` and `getxz` as shown in figure 3. In figure 6 the vectors are shown in the y- or z-direction. In the linear part the xy-planes are treated, thus the length of the vectors is determined by how many z-positions are gathered at the same time by the routine `getxy`. In the non-linear part the length is determined by the number

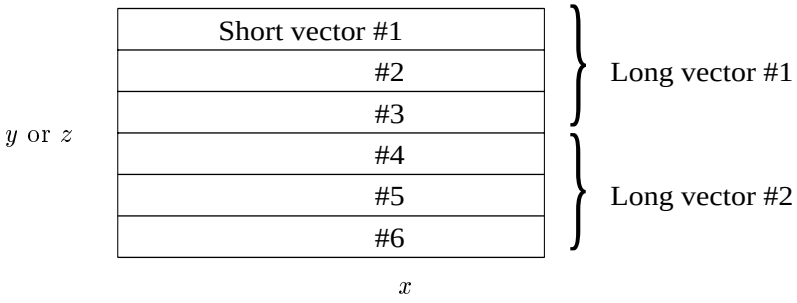

FIGURE 6. The short vectors contain one $xz-$ or xy-plane. The vector length is increased by treating more than one plane at a time. The long vector is constructed by letting the short vectors be stored concurrently in the x-direction. The third direction (z or y) is omitted in the figure.

of y-positions gathered at the same time by `getxz`. The degree of vectorization is crucial for some computers to perform well. The FFT is written in such a way that the vector registers are used to a full extent. The memory used by the code is increased significantly when using longer vector lengths due to the increased size of the two-dimensional working area.

We have tested vector processors from Cray and Fujitsu (table 3). For the specific problem size (case one in table 2) the optimal vector length was obtained when six planes were used at the same time on J90, eight on C90, sixteen on T90 and four on VPP300. When running a smaller problem, the optimal number of planes is higher and when running a larger, it is less. It should be mentioned that the performance is highly dependent on the problem size. The size used here gives a very high efficiency on the C90 (over 50 % of peak performance) while the VPP300 efficiency is only 24 %. However, the code performance was 800 Mflop/s (36 %) on the VPP300 for a different problem size. The J90 is not so dependent on the problem size and performs at 100 Mflop/s for different problem sizes if the tuning parameters are set to obtain the optimal vector length.

	J90	C90	T90	VPP300
peak processor performance	220	952	1700	2200
one plane	93	361	501	303
optimized vector length	100	522	710	525

TABLE 3. The speed on different vector machines for one processor for case one.

2.2. *The FFT*

The FFT package is an extended version of the `FFTPACK` from netlib. The FFT is applied in all three physical directions in the transforms between physical space and spectral space. The transforms are performed on the two-dimensional matrix, where the main difference between the treatment in the three directions is with respect to the memory. The x-transform is performed on the first index, which yields unit stride memory access, while the y- and z-transforms are performed on the second index. The treatment of the y- and z-transforms are hence similar and we shall therefore only investigate the behavior of the x- and z-transforms.

2.2.1. *The original version for vector processors*

The FFT that is used for the transform in the x- and z-directions consists of two one-dimensional FFT:s. In the forward transform they are a real to half-complex in x followed by a half-complex to complex in z. The real data in physical space is stored in two matrices with odd points in x in the first matrix and the even points in x in the second, see figure 7. After transforming in the x-direction, the real and imaginary Fourier components are stored in the two matrices respectively. Then the z-transform is applied to get the full two-dimensional transform. The data is stored all the time with the x-direction column-wise and the z-direction row-wise. The access to data in FORTRAN is row-wise, thus the elements in the x-direction are positioned one after another. However, when performing the z-transform the elements are separated by $nx/2$, i.e. the vector stride is $nx/2$. This means that different memory locations are accessed all the time. For vector processors the access to memory is fast and the most important issue is that no bank-conflicts occur.

2.2.2. *Scalar processor*

As described above, the FFT was originally written for a vector processor. This feature of the FFT is a disadvantage when running on scalar processors. The changes needed in the code for scalar processing was to a great extent already accomplished by the original author, Anders Lundbladh. The alternative FFT was however only fast in the x-direction, where the changes consisted of reducing the amount of data used at the same time. An outer loop in x was introduced as to process data from one row at a time instead of the whole plane (or several planes if the vectorization tuning parameters are used.) To accomplish a faster transform in the z-direction, the z-transform was rewritten to process data for one line at a time, in much the same way as was already done in the x-direction. To perform the transform in this manner, the data need to be transposed before transformed, see figure 8, otherwise the vector stride will prevent the data from lying concurrently.

In the table 4 the original (modified in the x-direction), modified (in the z-direction) and library FFT are compared. Both the x- and z-direction are transformed for a 512×256 grid corresponding to case two in table 2. The

library FFT is faster than the in-house FFT. But the data transfer due to the library FFT usage of complex variables decreases the performance considerably. Also the modified approach with transposing before the transform in the z-direction gives a faster FFT but the transposing itself decreases the performance. The three different transforms performs approximately the same number of operations (flop), thus the difference in performance (Mflop/s) is due to better efficiency. The number of flops correspond to the formula in the book by Canuto *et al.* (1988), flop$\approx$5Nlog$_2$N for one one-dimensional FFT. The decrease of the performance when including the transpose and data transfer respectively, is due to the fact that these redistributions of data do not include any operations (flop). Since the original version do not require any additional rearranging of the data, it is the fastest method for the FFT. Thus, the only difference between the FFT for scalar and vector processors is the treatment in the x-direction.

The routines written by the authors for the transpose are substantially faster than the library routines on the T3E, due to the E-registers which can be used explicitly in the code. Also on the SP2 the library transpose is slower than the ones written in FORTRAN, probably due to the generality of the library routines.

	T3E	SP2
peak processor performance	600	640
original	59	177
modified	64	180
modified incl. transpose	55	100
library	66	209
library incl. transfer	54	82

TABLE 4. The performance of the FFT given in Mflop/s averaged in both the x- and z-directions.

2.3. *Parallelization*

The computation of each xy-plane (in `linear`) and xz-plane (in `nonlin`) is independent of the other planes. Therefore, parallelization of the code is performed by distributing different planes in the loops to the different processors, which then runs in parallel. Since the major part of the computation is spent in the `linear` and `nonlin` subroutines, the code should parallelize efficiently. In the following, n_{proc} denotes the number of processors.

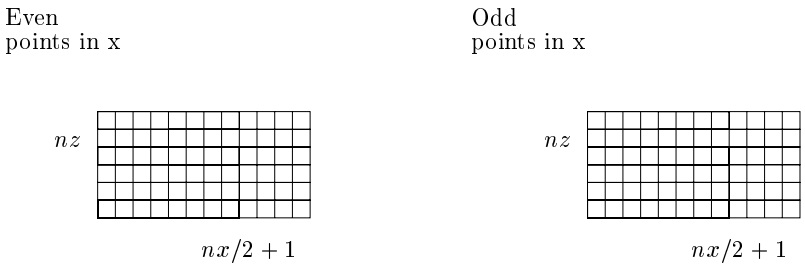

FIGURE 7. The structure of the data in physical space. On this data the transform in the x-direction is performed.

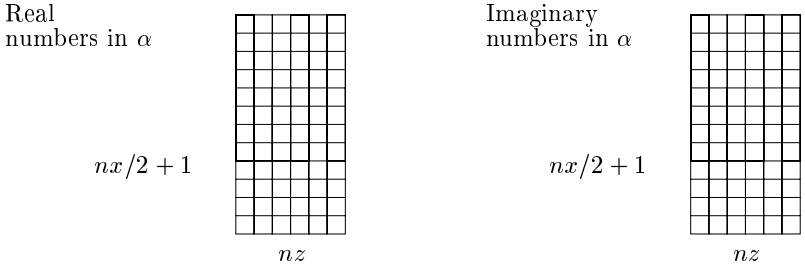

FIGURE 8. The structure of the data in half-complex form after transposing the data. On this data the transform in the z-direction is performed in the modified method.

2.3.1. *The maximum speed up*

The maximum possible speed up of the code when using several processors is determined by the part of the program that do not allow for parallel computing. Ideally, if the the whole program runs in parallel, the maximum speed up equals the number of processors. Some parts of a code are always impossible to divide between the processors. Let a denote the portion of the code that is parallelizable, and b the portion that only can be run on one processor. The maximum possible speed up of the code is then

$$\frac{a+b}{a/n_{proc}+b} = \{\frac{a}{a+b} = q\} = \frac{1}{q/n_{proc}+(1-q)}. \tag{1}$$

This is usually referred to as Amdahls law. As q approaches unity the maximum speed up goes to n_{proc}. It is desirable to have the portion b of the code that only runs on one processor small.

Figure 9 shows that the speed up is far from linear for large values of n_{proc} if q is not very close to unity. In the limit of infinite number of processors the computational time is completely determined by the part of the code that do not run in parallel and the maximum speed up approaches $1/(1-q)$.

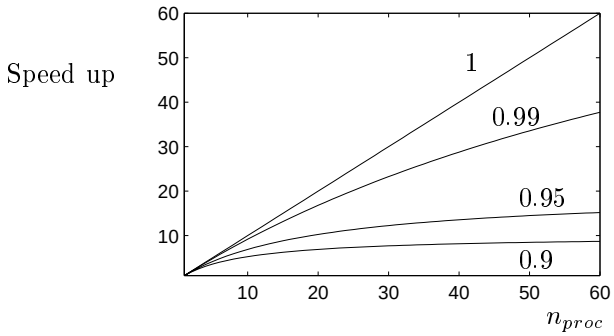

FIGURE 9. The maximum speed up for $q = 1,\ 0.99,\ 0.95,\ 0.9$.

2.3.2. *Shared memory*

The parallelization and optimization of the code for shared memory systems was done in the original version of the code. The two main loops in the code, over the linear and non-linear parts, were splitted as to account for the parallel computing. The parallel processing requires extra two-dimensional local variables, one for each processor in the computation, in order for the different processors not to use the same memory position. A compiler directive must be included in the code because within the loops there are subroutine calls. As can be seen from figure 10 the scaling is excellent on the Cray C90 and J90. When q in Amdahls law (1) is calculated from the measured performance, it gets a value of 0.99. On the Origin 200, with a scalar processor, the measured performance was 53 Mflop/s on one processor and 181 on four, which corresponds to a value of q as low as 0.94.

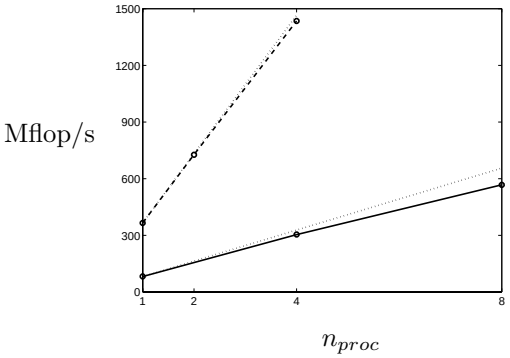

FIGURE 10. Mflop/s rates for different number of processors for case one. — J90; - - C90; $\cdots$ maximum speed up with $q = 1$ in Amdahls law (1).

2.3.3. *Distributed memory*

On distributed memory machines, e.g. IBM SP2 and Cray T3E, the whole field needs to be divided among the different processors. There are three possible spatial directions which can be distributed among the different processors. The x-direction is not suitable to divide since it would give significant communication between the processors in both the linear and nonlinear part of the code. The number of y-discretization points in Chebyshev polynomials is often not even divisible with the number of processors. The discretization in the z-direction by the Fourier series can easily be chosen to obtain a number of discretization modes which is divisible with the number of processors. Also the communication with the main storage is more frequent in the linear part. The whole field is hence divided in the z-direction between the different processors (figure 11a), which yields easy access to the complete field in the linear part since the two dimensional field is available locally on the processor. In the nonlinear part, however, the local two dimensional storage needs to collect data from all the other processors. This is performed with the MPI standard.

The message passing

Two different methods of moving the data with MPI between the processors have been investigated. The first method transposes the whole field at once before the non-linear calculations, so that it becomes divided in the y-direction between the different processors. This makes the two dimensional field available locally at each processor. Before the linear calculation, the data is transposed back to its original position. The transposing of the data requires additional memory, since the whole field is stored twice, and additional moving of data from one field to another. In the second method the main storage is kept at its original position on the different processors. In the non-linear part each processor collects the two dimensional data from the other processors, on which it performs the computations, and then redistributes it back to the main storage. In this way no extra memory is needed. Figure 11b shows an example of the data gathering for one processor. The two methods are described more thoroughly in Appendix A

The amount of communication

The main communication between the processors is in the non-linear part, where five variables are collected from the main memory and three variables are stored to the main memory. Thus a total of eight variables have to be sent. Each processor performs calculations on approximately ny/n_{proc} xz-planes. The amount of data that needs to be collected from the other processors is for each plane $nx(nz - nz/n_{proc})$. This gives that for all variables each processor

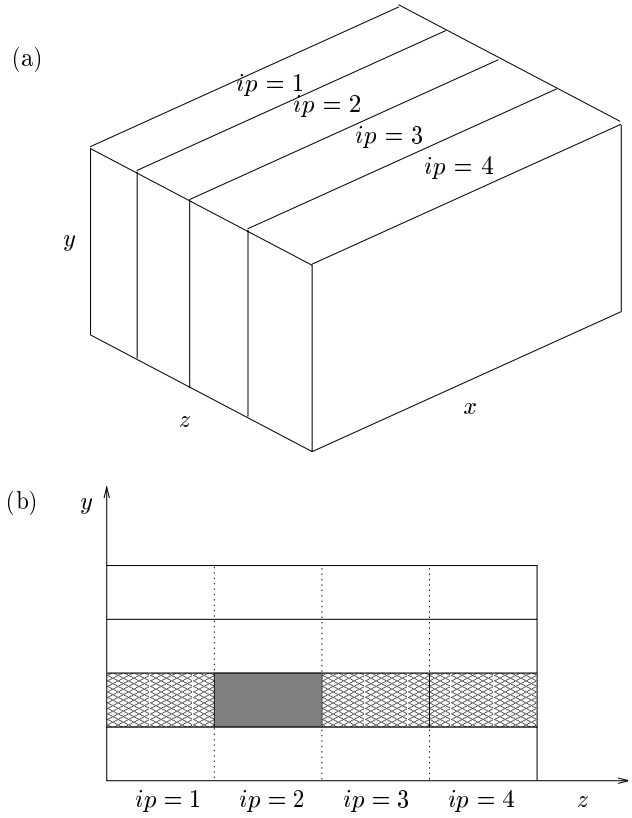

FIGURE 11. a) The distribution of the main storage on four processors $ip = 1, ..., 4$. b) The gathering of data in the nonlinear part (`nonlin`) of the code for processor number two. The completely shaded area is local on the processor and need not to be received from the others, and the half-shaded area is sent to processor number two. The x-direction is omitted for clarity.

needs to collect

$$N_{tot} = \frac{ny\ nx\ nz}{n_{proc}} \left(1 - \frac{1}{n_{proc}}\right) \tag{2}$$

real numbers from the other processors at each Runge-Kutta iteration. The amount of data that each processor needs to send is equally large.

The IBM SP2 communicates with a high performance switch which has a bandwidth of $110 MByte/s$ and a latency of $25 - 30 \mu s$. The MPI on the Cray T3E has a bandwidth of $320 MByte/s$ and a latency of $12.8 \mu s$. In order

to reduce the effect of the latency and use the bandwidth optimally, the data should be arranged in large groups when sent.

2.4. *Performance on the SP2 and T3E*

The main computational work in the code is in the FFT which was treated in the previous chapter. The performance of the two methods of redistributing data described in section 2.3.3 is investigated by using a model problem of the same structure as the code. The performance of the code can then be measured, using the most efficient FFT and message passing method.

The y-direction is not generally even divisible with the number of processors, i.e. ny/n_{proc} is not an integer. Therefore, in the last y-loop count not all processors are active and the loop is not completely parallelizable. If n_{proc} is relatively small compared to ny (as is the case with e.g. the Cray C90 and J90) the effect from this on the performance is small. When n_{proc} is of the same order as ny (as may be the case for e.g. the IBM SP2 and the Cray T3E) this effect may be of importance.

2.4.1. *Performance of the MPI*

The most important feature of the message passing is that it should be fast, i.e. the time spent in moving data between processors should be small so that the speed up of the code is not limited by the message passing. It is also interesting to study how the efficiency of the data transfer depends on the number of processors. The speed up is of course dependent on how much work is performed between the data redistributions. The test problem was set up so that a clear difference could be seen between the two methods.

In figure 12 the two methods are compared for the two cases, table 4. They show that for case two method two is faster than method one for both computers (method one was not possible to run with less than 4 processors due to the large memory requirement). However, for case one method one is faster on the T3E while method two still is faster on the SP2. Hence, method one is better than method two only on the T3E for a small problem (case one) on many processors. However, the size corresponding to case one would not need to run on more than 16 processors in a real application, thus method two is the one to prefer in the code. It is also observed that the SP2 performs better for the small case while the T3E performs better for the larger case.

In the following only results using method two of the data communication will be presented since method one was found not to be useful due to the lower performance, especially on the SP2, and the larger amount of required memory.

If we take into account that the y-loop is not even divisible with the number of processors, the maximum possible speed up of the test problem code is

$$\frac{ny}{\left(\mathrm{Int}\left(\frac{ny}{n_{proc}}\right)+1\right)}. \tag{3}$$

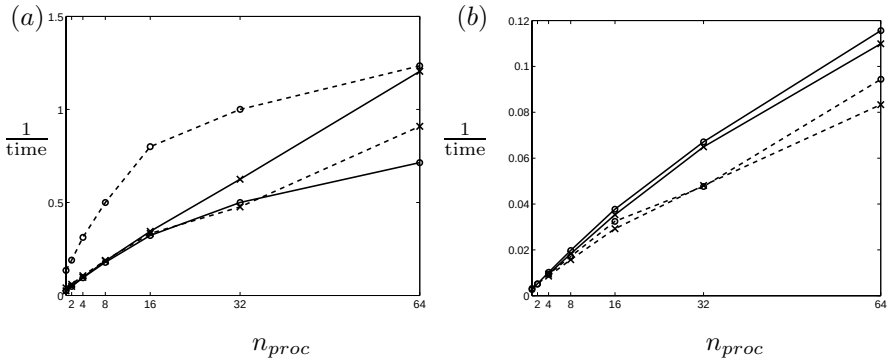

FIGURE 12. Performance on — T3E - - SP2; x Method 1 o Method 2. a) Case one. b) Case two.

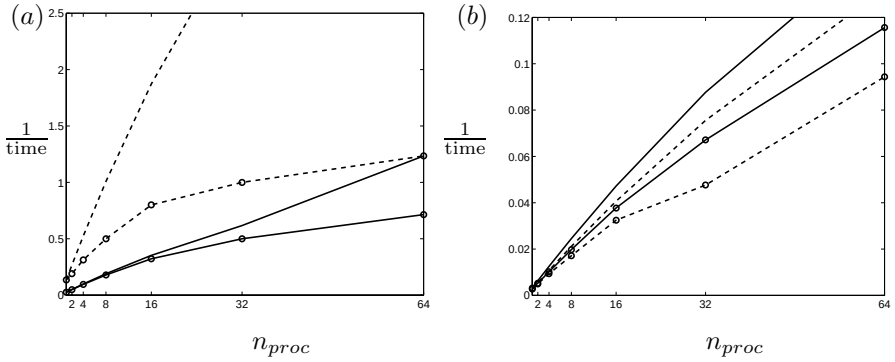

FIGURE 13. Performance on — T3E - - SP2; o Method 2. Curves with no circles are the optimal speed up. a) Case one. b) Case two.

In figure 13 the performance of the test problem is shown together with the optimal speed up from (3). The T3E is slightly closer to the optimal speed up performance than the SP2, indicating that the message passing performs better. We also have a closer adherence to the optimal curves for the larger case (for both T3E and SP2) which is associated with larger data sets which increases the performance of the message passing.

Bandwidth

It is a well known fact that the bandwidth, or the data transfer rate, decreases considerably from the maximum values quoted in section 2.3.3 if the amount of data which is sent decreases below a certain level, e.g. one Mbyte on the T3E.

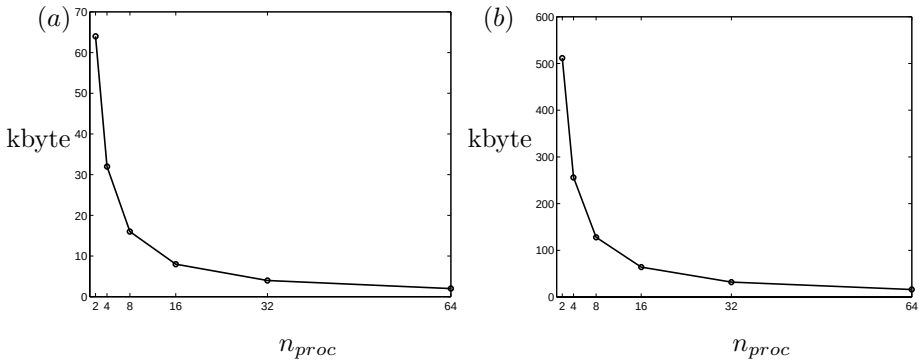

FIGURE 14. Size of data package. a) Case one b) Case two.

The size of the individual data package sent from one processor to another is

$$\text{Size} = \frac{nx\ nz}{n_{proc}}. \tag{4}$$

This quantity is plotted for different number of processors in figure 14. For both cases the size is well below one Mbyte for all values of n_{proc} which would affect the bandwidth. The size decreases with increasing number of processors and consequently the data transfer rate for each individual processor should decrease. Although the performance of each processor decreases, the effective time of the message passing should actually decrease since the amount of data that is sent becomes smaller.

If it is assumed that the parallelization is optimal, i.e. $q = 1$ in Amdahls law (1), the time spent on data transfer, t_{MPI}, is derived as the difference between total time and the total time for one processor divided by the number of processors,

$$t_{\text{MPI}} = t_{n_{proc}} - \frac{t_{n_{proc}=1}}{n_{proc}}, \tag{5}$$

where $t_{n_{proc}}$ is the program CPU time for one processor when a total of n_{proc} processors is used. As already noted in (3) the speed up is not linear in n_{proc} but depends on both n_{proc} and ny. Using this approach a more accurate estimate is obtained,

$$t_{\text{MPI}} = t_{n_{proc}} - \frac{\text{Int}\left(\frac{ny}{n_{proc}}\right) + 1}{ny} t_{n_{proc}=1}. \tag{6}$$

In figure 15 the performance of the test cases is plotted for the two computers together with inverse of the time t_{MPI}. Due to memory limitations on the T3E case two was not possible to run on a single processor and the time t_{MPI} is not available. From figure 15 it is clear that the time of the message passing indeed decreases with increasing values of n_{proc}, except for the small case on the T3E which actually obtains relatively high performance at few processors.

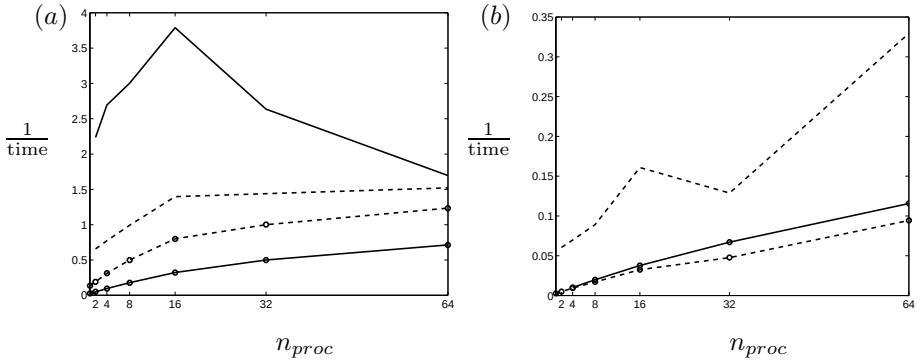

FIGURE 15. Performance on — T3E - - SP2; corresponding curves with no rings show the MPI performance. a) Case one. b) Case two.

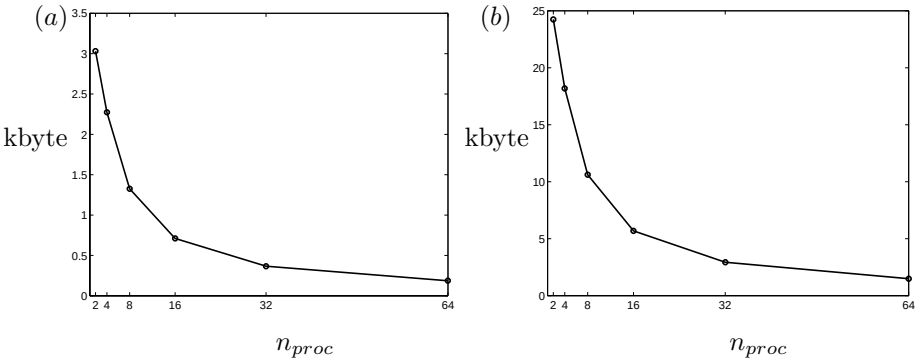

FIGURE 16. Total amount of data sent. a) Case one b) Case two.

For the larger case the message passing takes a smaller amount of time compared to the total time and the increase in performance of the message passing with n_{proc} seems to be stronger.

The amount of data that is being sent by each processor is given by equation (2), and is shown in figure 16. To get the data transfer rate, the amount of data is divided by the time spent on the transfer t_{MPI}. As the communication is very intense between the processors, i.e. each processor needs to both send and receive data from all other, the maximum possible transfer rate of receiving data for each processor is only half of the bandwidth.

The rate is plotted in figure 17 for both cases. For case one the T3E performs better with regards to the bandwidth for a small number of processors. Also, the rate decreases with increasing number of processors, due to the smaller package size observed in figure 14, and the difference between the two computers disappear. In case two the rate is overall higher due to larger data packages and it also shows a slight increase at the highest values of n_{proc}.

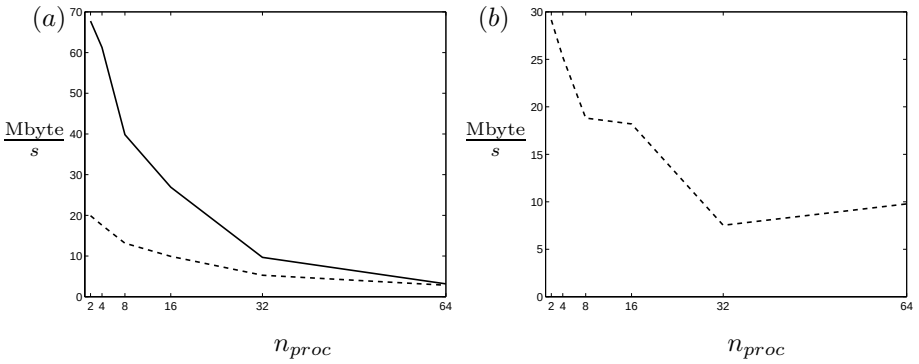

FIGURE 17. Data transfer rate per processor. — T3E - - SP2.
a) Case one. b) Case two.

The theoretical data transfer rate as a function of message buffer size can be found in a Cray manual. The low data transfer rate is surprising since the theoretical values for the T3E are much higher. If case one is considered for two processors, the theoretical value is 230 Mbyte/s. Since this value is for point-to-point communication, the value has to be divided by two to be compared with our case where all the processors are both sending and receiving. The measured rate for our test case one is 70 Mbyte/s on two processors. If the MPI derived data type used in the communication is changed to a standard vector type, and sent to another vector of the same type, the performance increases to 114 Mbyte/s, which is very close to the expected theoretical value. If now the case one on 64 processors is considered the theoretical value is 25 Mbyte/s, whereas the measured value is 3 Mbyte/s. If the same number of bytes is sent between only two processors, the rate increases to 12 Mbyte/s, which again is half of the theoretical value. In this case there is no difference in the performance if a standard vector is used instead of a MPI derived data type. The low bandwidth is thus explained by two different effects. For a small number of processors, when the data packages still are relatively large, the low bandwidth is due to the MPI derived data type which slows down the communication. For a large number of processors, the explanation is the decrease of communication efficiency (network contention) when many processors are being used.

2.4.2. *Performance of the code*

The complete code has been run with the optimal FFT and method two for the message passing for the two cases. Since the non-linear part is not completely parallelizable the factor q in Amdahls law depends on n_{proc} and is difficult to give directly. Most time of the code is spent in the linear and non-linear parts. If it is assumed that they take the same computational time on one processor and that the remaining computational time is negligible the following

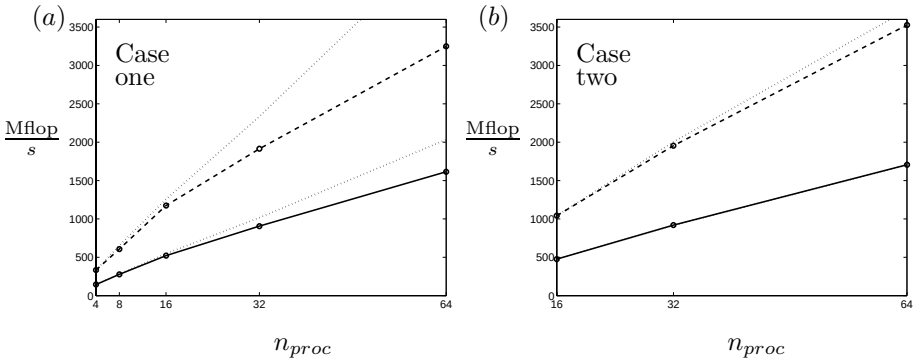

FIGURE 18. Mflop/s rates for different number of processors.
— T3E - - SP2 $\cdots$ maximum speed up.

expression can be derived for the optimal speed up,

$$\frac{2}{\frac{1}{n_{proc}} + \frac{1}{ny}\left(\mathrm{Int}\left(\frac{ny}{n_{proc}}\right) + 1\right)}. \tag{7}$$

This formula, with $n_{proc} = 8$ and $ny = 97$, gives when using Amdahls law (1), a q of value 0.99, corresponding to the measured values in figure 10. In figure 18 the performance of the code is shown as Mflop/s together with the optimal speed up given by (7). The speed up is better for case two than case one. The scaling is better on the T3E for both cases, which was noticed also for the model problem. We actually obtain an optimal performance for the larger case on the T3E. However, the overall performance is better on the SP2, which is approximately twice as fast as the T3E. This was also observed in the model problem for the FFT (see table 4).

Hence, for both the large and small cases, the scaling is better for the T3E associated with the higher performance of the message passing, and the overall performance is better for the SP2, mainly due to the higher performance of the FFT.

For case one it is possible to compare the performance with the vector computers (see figure 10). One processor on the C90 corresponds approximately to four processors on the J90, four processors on the SP2 and eight on the T3E.

3. Conclusions

To be able to implement a code on a parallel computer with distributed memory, a lot of effort must be put into programming the communication between the processors efficiently. The most sensible approach seems to be through a model of the code with the same structure of the variables. By using a test problem, the different options of communication can be tested and evaluated before implementing the communications in the code.

The MPI routines might be working in different ways on different computers, depending on the implementation. The code might be working perfectly on one machine, but this does not imply that the code can be expected to run efficiently, or be working at all, on another. Portability of codes seems to have been lost.

It is concluded that it is possible to achieve high performance on super scalar machines, with computational speeds comparable and higher than those of the traditional vector machines. The code seems to scale efficiently with the number of processors and therefore a high performance might be obtained by using many processors in the simulation. The lower scaling on the smaller case is not critical since it does not need to be run on many processors in a real computation.

Another important issue is the availability of the computer, i.e. how long time the user have to spend queuing before a job begins and how many processors you may use. In some cases the job may be subjected to timesharing with other jobs which reduces the performance.

Acknowledgments

Computer time was provided by the Center for Parallel Computers (PDC) at the Royal Institute of Technology (KTH), the The National Supercomputer Center in Sweden (NSC) at Linköping University, and The San Diego Supercomputer Center (SDSC) at the University of California. In addition we thank Prof. Joseph Haritonidis at the Ohio State University (OSU) for letting us use his Origin 200.

Appendix A.

The first method

The first method was developed by Högberg (1998) and is based on the MPI_ALLTOALLV command. When the problem size and the number of processors have been determined, vectors describing where to send data, and from where to receive data, are set up for each processor individually. When these vectors are available MPI_ALLTOALLV is the only command needed for all communications. Every processor has two data vectors, one for xz-planes and one for xy-planes. The size of these vectors depends on the problem size and the number of processors. Other vectors contain information about how many planes each processor handles, and information about where to take data for sending and where to put received data. These vectors are used with MPI_ALLTOALLV to transpose the whole field that has been distributed among the processors. The main drawback of this method is that data is stored twice, which will increase the memory requirements compared to the second method.

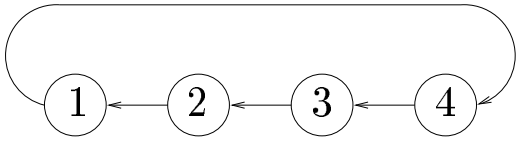

FIGURE 19. The first left neighbor in the four processor case.

The second method

The sending and receiving data is performed with the commands MPI_SEND and MPI_RECV, which are used in the modified `getxz` and `putxz` routines. When parallelized, the original y-loop, which goes from 1 to ny, is changed to a loop that goes from 1 to $\mathrm{Int}[(ny-1)/n_{proc}]+1$. In this modified loop, the number of y-planes treated at the same time is n_{proc}, except for the last loop count where the remaining y-planes are treated. At each loop count each processor sends and receives data to and from all other processors. This is achieved by an inner loop, from 1 to $n_{proc}-1$, in which all processors sends data to another (all different) processor.

Let ip be the number of a processor. Processors ip and $ip-1$ are said to be neighbors (figure 19). In particular processor 1 is a neighbor to processor n_{proc}. At each loop count ii, which goes from 1 to $n_{proc}-1$, each processor sends data to its neighbor number ii to the left, i.e. processor ip sends data to processor $ip-ii$.

At each MPI_SEND command the data that is sent is defined through a special vector in the MPI standard, describing the number of elements in the data set, the length of one element and the distance (stride) between two elements. A corresponding vector is constructed to define where the data is to be stored with the MPI_RECV command. These vectors are determined by the number of discretization modes in each direction and the number of processors.

On the IBM SP2 the MPI_SEND is implemented as a standard send that is not able to complete until the receive has started. Since all processors start with sending and none is receiving a deadlock occurs. In this case a non-blocking standard send, MPI_ISEND, together with a test, MPI_WAIT, are used. If this method is used on the Cray T3E instead of the standard send no change in the performance is detected.

References

CANUTO, C., HUSSAINI, M. Y., QUARTERONI, A. & ZANG, T. A. 1988 *Spectral Methods in Fluids Dynamics*. Springer.

HÖGBERG, M. 1998 Private communication.

Lundbladh, A., Henningson, D. S. & Johansson, A. V. 1992 An efficient spectral integration method for the solution of the Navier-Stokes equations. FFA-TN 1992-28, Aeronautical Research Institute of Sweden, Bromma.

Lundbladh, A., Schmid, P. J., Berlin, S. & Henningson, D. S. 1994 Simulation of bypass transition in spatially evolving flows. Proceedings of the AGARD Symposium on Application of Direct and Large Eddy Simulation to Transition and Turbulence, AGARD-CP-551.

Skote, M., Henkes, R. A. W. M. & Henningson, D. S. 1998 Direct numerical simulation of self-similar turbulent boundary layers in adverse pressure gradients. *Flow, Turbulence and Combustion* **60**, 47–85.