A Spectral Finite Element Model for Shallow Water Equations: Implementation Experiences on the Cray T3D

Giovanni Erbacci, CINECA - Interuniversity Computing Center, Bologna, Italy, *Valeria Simoncini*, MGA - CNR, Modena, Italy, *Giorgio Bertero*, Computer Sciences Institute, University of Bologna, Italy, *Roberto Ansaloni*, Cray Research S.r.l, Milano, Italy, and *Antonio Navarra*, IMGA - CNR, Modena, Italy

ABSTRACT: *Finite element methods have recently gained great interest for solving large-scale ocean circulation models, which are based on 2-3D coupled Navier-Stokes equations. The corresponding discretized system can be treated by using implicit or explicit iterative methods. In this work we are interested in the effective implementation of a recent semi-implicit method on CINECA Cray-T3D, for integrating in time the corresponding velocity-pressure equations. The original scalar code was initially implemented at the Institute of Marine and Coastal Sciences, Rutgers University (NJ). Of particular interest is the fact that the elemental structure of the discretized problem may yield effective load distribution among the nodes. A complete data layout analysis is proposed, and first experimental results are provided in order to investigate the efficiency of the code.*

1 Introduction

The analysis of global atmospheric circulation models is based on different and appropriate discretization techniques. While finite differences methods are characterized by a simple implementation on regular domains, the more recently employed finite element and spectral element methods are more flexible when dealing with complex geometries (e.g. polar zones) and provide better accuracy.

The problem then becomes that of efficiently implementing the discretized model on the prescribed domain. If high accuracy is required, the number of grid nodes representing the domain may be extremely large so that the integration of the model equations for long periods of time becomes computationally very expensive and CPU time consuming. This motivates the use of high speed computers that can also exploit the structure of the numerical model.

In this work we are interested in a parallel implementation of a spectral finite element method. The subdivision of the domain in elements justifies the use of massively parallel machines which can locally handle the computation on each element at low communication cost.

2 Model Description

Shallow water equations represent a convenient formulation for describing an atmospheric circulation model for appropriate parameters. Such formulation in 2D consists of the following Navier-Stokes equations

$$u_t + u \cdot \nabla u + \mathrm{f}\, k \times u + \mathrm{g}\, \nabla \zeta + \gamma u - \nu \nabla^2 u = \frac{\tau}{\rho(\mathrm{h} + \zeta)} \quad (1)$$

$$\zeta_t + \nabla \cdot \left[(\mathrm{h} + \zeta) u \right] \cong 0 \quad (2)$$

where $\mathbf{u} = (u, v)$ is the horizontal velocity vector, ζ is the surface elevation, f the Coriolis parameter, g the gravitation acceleration, γ the friction coefficient, ν the diffusion coefficient, ρ the density of the fluid and finally τ the forcing term. The boundary conditions are of Dirichlet type. The equations above are discretized by means of a mixed method that exploits both finite elements and basis functions proper of spectral methods [Ma-93]. The domain is subdivided in quadrangular conformal elements in which a weak solution is sought, belonging to a conveniently chosen Sobolev space. A spectral method is then applied on each element. Denoting by N^v the number of nodes per direction in each element of the velocity grid, each variable in (1)-(2) is interpolated

by $u(x, h) = \sum_{i=1}^{N^v} \sum_{j=1}^{N^v} u_{i,j} h_i^v(x) h_j^v(h)$ where $u_{i,j}$ is the first component of the velocity vector at the node $(\xi_{i,j}^v, \eta_{i,j}^v), i, j = 1, \ldots, N^v$. Analogous formulations hold for $v_{i,j}$ and $\xi_{i,j}$, where for the pressure a staggered grid has been used with $N^p = N^v - 2$ nodes on each direction. The interpolating spectral functions h_i^v are Legendre Cardinal functions, associated to Legendre polynomials of degree $N^v - 1$, and, if x_i are the Gauss-Lobatto nodes for each element, it holds $h_i^v(x_i^v) = \delta_{i,j}$ where $\delta_{i,j}$ is the Kronecker delta (analogously for h_i^p). Problem (1)-(2) then transforms to the ordinary differential equations

$$M^v \frac{du}{dt} + (\gamma M^v + \nu D)u - Fv + gP^x\zeta + A^x = r^x \quad (3)$$

$$M^v \frac{dv}{dt} + (\gamma M^v + \nu D)v - Fu + gP^y\zeta + A^y = r^y \quad (4)$$

$$M^p \frac{d\zeta}{dt} + C^x u + C^y v + A^z = 0 \quad (5)$$

where capital letters stand for matrices corresponding to integrals on the domain [IHB-93]. It is worth to remark that $A^i, (i = x, y, z)$, include non linear terms and that M^v, M^p are diagonal matrices.

Equations (3)-(5) are numerically solved by means of a semi-implicit method which explicitly evaluates the non linear terms by means of a third order Adams-Bashforth method and implicitly determinates the new velocity iterates with the Crank-Nicolson recursion [Sew-88]. More precisely, with obvious notation the following recursion is iterated in time

$$\frac{u^{n+1}}{\Delta t} - \frac{1}{2}B^{n+1} = \frac{u^n}{\Delta t} - \frac{1}{2}B^n + \alpha_3 A^n + \alpha_2 A^{(n-1)} + \alpha_1 A^{(n-2)} \quad (6)$$

where, for algorithmic convenience, the Coriolis parameter has also been treated explicitly. An analogous relation holds for the continuity equation. The implementation of a semi-implicit method partially relieves from the constrain on the timestep that often penalizes an explicit less expensive approach. The additional cost of the implicit step is mainly due to the solution of the linear system in (6)

$$\left[\begin{array}{cc|c} \alpha M^v + \nu D & 0 & gP^x \\ 0 & \alpha M^v + \nu D & gP^y \\ \hline C^x & C^y & \alpha M^p \end{array}\right] \begin{pmatrix} q \\ p \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}$$

where the coefficients matrix corresponds to the discretized derivative and to $B^{(n+1)}$, and $q = (u, v)$. Denote by A_i the four blocks of the coefficients matrix. The inexact Uzawa algorithm can be employed for the solution of this system. Letting $d = A_1^{-1} f_1$ and $T = (-A_3 A_1^{-1} A_2 + A_4)$, the algorithm consists of explicitly determining the two unknowns p and q by means of (cf. [MPa-89]).

$$p = T^{-1}(f_2 - A_3 d) \ , \quad q = d - A_1^{-1} A_2 p \ .$$

The (approximate) solution of two small systems with A_1 are required, where the size of the matrix is usually about one third of that of the original one. The iterative nonsymmetric solver TFQMR is used for solving the system with A_1 [Fre-93]. A major advantage in using this technique is that assembling the coefficients matrix is not required, and that short term recurrence makes it computationally very competitive. For a faster convergence, diagonal preconditioning is applied with mass matrix.

3 Parallel Implementation

The model described in the previous section was originally implemented at the Institute of Marine and Coastal Sciences of Rutgers University, New Jersey, and the implementation was done in Fortran for scalar systems [IHB-93]. This scalar implementation constitues the basis for the *implicit* parallel version we present in this work; a parallel version of the *explicit* method was recentely proposed in [HCI-94].

The model requires time integration of differential equations and the number of time steps remarkably increases when it is adopted for the simulation of large scale ocean circulation. In this way, for real applications it becomes time consuming so it is important to provide a parallel version. The algorithm is outlined in Fig. 1. Seven routines are called inside the do loops consuming more than 90% of the algorithm total time. Therefore, in order to parallelize the algorithm, we can focus on the computational kernel of these routines.

The spectral finite element technique is particolarly appropriate for implementation on parallel systems and the basic idea for the parallelization simply comes from the data structure used by the original algorithm.

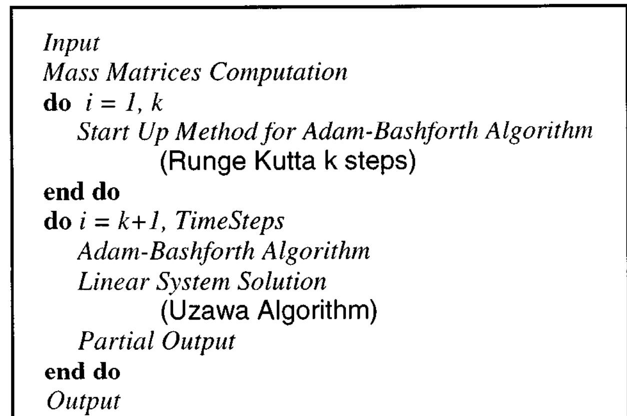

Figure 1: Flow Structure for the Shallow Water Algorithm

In this section we first introduce the data structure used by the original algorithm to solve the shallow water equations applied to the propagation of the equatorial Rossby soliton of Boyd [Boy-80], then we describe how to modify this data structure to realize an efficient parallel implementation of the original algorithm.

3.1 Original Data Structure

The computational domain is divided into blocks, each block is then mapped into a rectangle before its subdivision into spectral elements.

The original algorithm uses two grids, a *velocity grid* and a *pressure grid*, and, depending on physical parameters, (such as diffusion coefficient, gravity coefficient, etc.,) computes the propagation of the equatorial Rossby soliton.

The boundary nodes that are shared by more elements are replicated in every local structure.

The grids introduced are described by the coordinates of the nodes that compose the grids themselves. Such nodes (*global nodes*) are numbered on the grids from bottom to up and from left to right, as we can see in Fig. 2.

The algorithm independently works on each grid element, so it is necessary to know which are the global nodes held in each element. Inside each element, the grids are described using a local numbering scheme (*local nodes*). The following data structure describes the velocity grid:

nodeglob: total number of velocity nodes in the spectral grid.

nelem: total number of elements in the spectral grid.

npts: number of nodes per element per direction in the velocity grid.

node: number of nodes per element in the velocity grid, ($node = npts \times npts$).

XGLOB[*nodeglob*]: contains the x coordinates of the nodal grid points on the velocity grid.

YGLOB[*nodeglob*]: contains the y coordinates of the nodal grid points on the voelcity grid.

IGLOB[*node*,*nelem*]: describes the grid element connectivity; iglob[i, n] gives the global node number of node i ($1 \le i \le node$) in element number n ($1 \le n \le nelem$).

ndiric: total number of Dirichlet nodes in each direction.

ndiricu: number of u-Dirichlet nodes.

ndiricv: number of v-Dirichlet nodes.

ndiricz: number of ζ-Dirichlet nodes.

IDIRIC[*ndiric*]: contains the global node numbers with Dirichlet conditions (u-Dirichlet+ v-Dirichlet + ζ-Dirichlet).

As an example, Fig. 2 shows a velocity grid with eight elements ($nelem = 8$), 16 points in each direction per element ($npts = 16$), so that the total number of grid nodes is 1891.

The pressure grid is similar to the velocity grid (the same number of elements) but differs for the number of points in each direction per element: $nptp = npts - 2$ in this case. The total number of nodes in each element becomes $nodeglbp = nptp^2$.

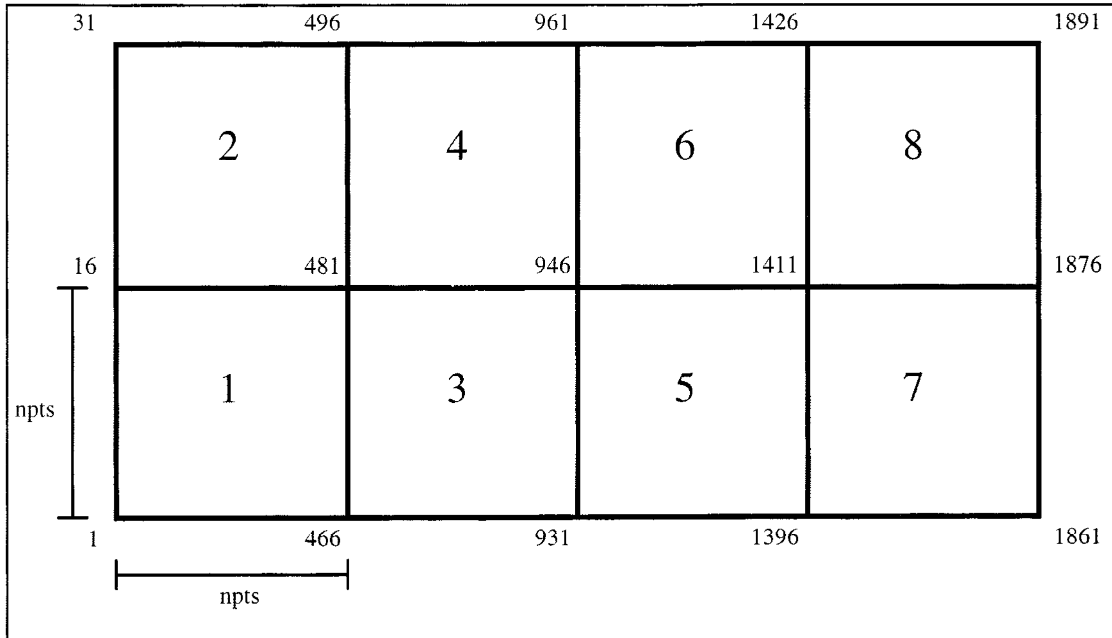

Figure 2: Velocity grid: nelem = 8, npts = 16, nodeglob = 1891

Besides the global data structure that represents the problem parameters, as for the velocity grid, two arrays *XGLBP*[*nodeglbp*] and *YGLBP*[*nodeglbp*] contain the x and y coordinates of global nodes and *IGLBP*[*nodep*, *nelem*] describes the element connectivity.

3.2 New Data Structure

Adam-Bashforth and Uzawa algorithms for the linear system solution operate on each grid element, so that for an efficient implementation we need to completely decouple the elements of the grid. In order to do this we need to modify the data structure of the original algorithm, essentially acting on the boundary conditions.

We can observe that the *element* is the basic unit for the spatial discretization of the spectral finite element scheme. Therefore it is not convenient to distribute each element among the processors but, on the contrary, to act in order that each processor can work independently on one or more elements.

In this way the communication overhead is very low also if the level of parallelism is limited to the number of elements in the grid. Moreover, in order to have a good *load balance*, we need a grid with a number of elements that is multiple of the number of available processors.

To assure a true independence for each element, we transformed all the global arrays (*nodeglob* elements for the velocity grid, and *nodeglbp* elements for the pressure grid) in bidimensional arrays of $node \times nelem$ elements ($nodep \times nelem$ for the pressure grid), containing also the replicated nodes corresponding to the boundary nodes between the elements.

In this way each element does not share points with other elements, so each processor holds one or more grid elements, i.e. the columns of these new arrays, and can work independently on the given elements.

The parallel algorithm has been implemented on the Cray T3D using the *work sharing* paradigm allowed with the Cray Research Fortran Programming Model (CRAFT). Work sharing supports SPMD (Single Program Multiple Data) programming style and contains directives to distinguish between data objects that are shared among all processors and those that are private to a processor [PMD-94], [T3D-94]. Using the work sharing syntax we can distribute the arrays in the following way:

```
cdir$ shared array (:,  :block(1))
```

This distribution assignes a column of *array* per processor. In this way only the columns af *array* are forced to be a power of two.

The new arrays contain replicated values in correspondence of the boundary points between two contiguous elements because they contain values corresponding to nodes that are local to an element while the original arrays describe global nodes (i.e. defined on the whole grid).

This new data structure permits an optimal work distribution among the processors but introduces the problem of the boundary points between two elements. The algorithm updates such points considering the contributions coming from the elements that share the boundary nodes in the grid, as we can see in Fig. 3.

Each processor computes the values for its own local nodes, then updates the values for the boundary nodes, using the results produced from the contiguous processor.

This work can be done in parallel for all the grid elements. To do this we need to define four work arrays of *nelem* x *npts*

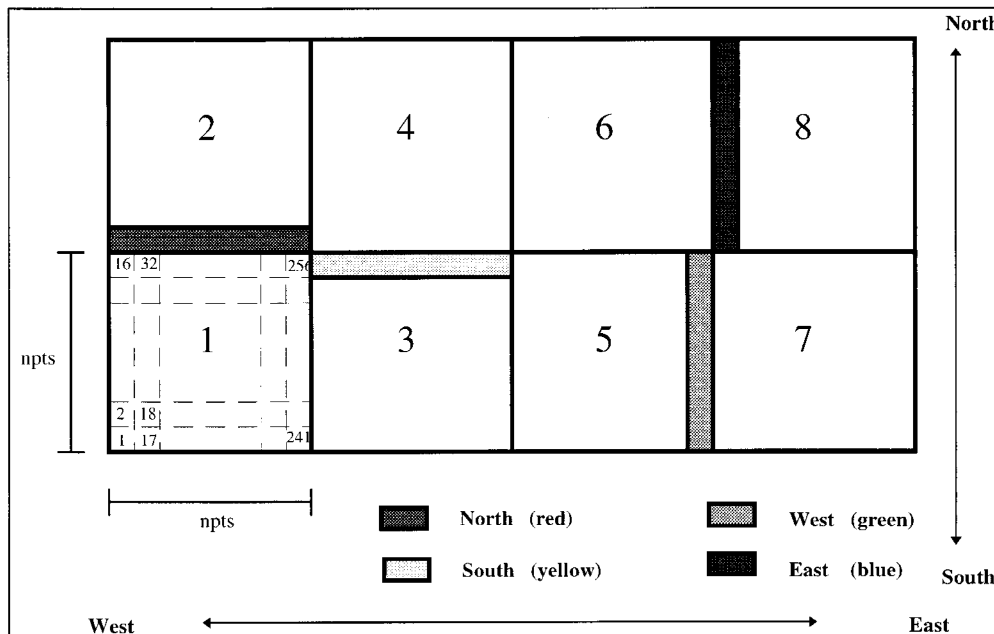

Figure 3: Neighbour points for each element: red = element 1 North points, yellow = element 4 South points, green = element 7 West points, blue = element 6 East points.

elements shared between the processors (*nelem* x *nptp* elements for the pressure grid) to retain the values of the boundary elements. For example, for each boundary point *i*:

T_EAST[*n*, *i*] will contain the copies of the boundary points of the element at the East (if it exists) of the element *n*.

T_WEST[*n*, *i*] will contain the copies of the boundary points of the element at the West (if it exists) of the element *n*.

Using the *work sharing* syntax, the arrays can be distributed by row among the processors. For example for array T_EAST, we can use the directive:

```
cdir$ shared T_EAST(:block(1),  :)
```

The work array we introduced are necessary for *mutual exclusion* problems. In fact, the Cray T3D is a distributed memory MIMD parallel computer system characterized by Non Uniform Memory Access (NUMA) and from a single processor it is possible to access directly the memory of every remote processor.

We can observe that the copy from the East side can be done concurrently with the copy from the West and analogously the copy from the South with that from the North. But a mutual exclusion problem arises if we try to do the copy from the East in parallel with the copy from the South or from the North. The same considerations hold for the copy from the West. The reason for this constraint are the cross points shared by four elements.The Fortran code that implements the updating algorithm for the velocity grid is shown in Fig. 4.

For the full parallelization of the algorithm another data structure has to be modified: the array *IDIRIC*[*ndiric*] containing the Dirichlet points. This array contains the global points of the velocity grid on which the Dirichlet conditions are imposed. Such conditions are set on the global boundary of the velocity grid, in both x and y directions.

Using a global structure such as *IDIRIC* would cause a loss of locality for the references (remote accesses, and computation for the two structure address mapping) and therefore a decreasing in performance. Then a new structure has been introduced for the Dirichlet boundary conditions to allow a good reference locality.

A new array for each of the three components *U*, *V*, *Z* of the velocity has been introduced for each element of the grid: *IDIRICU*, *IDIRICV*, *IDIRICZ.*

IDIRICU (*V* and *Z*) contains the node numbers on which the Dirichlet conditions are imposed. Observing that the Dirichlet conditions are imposed only on the velocity grid global boundaries, we deduce that it is not necessary to define arrays with *node* rows but only with *npts* rows. Each element has *npts* points in each direction, at the maximum, in which the conditions are imposed. In this way, IDIRICU contains the numbers of local *node* in which the condition is imposed.

Using the *work sharing* syntax, we can distribute the arrays by column among the processors; for example array IDIRICU is distributed as follows:

```
cdir$ shared IDIRICU(:, :block(1))
```

When it is necessary to establish the Dirichlet conditions, each processor for their own elements, controls if the first element of the local IDIRICU column is greater than zero and in this case runs through all the column and sets the conditions on the nodes contained in IDIRICU, as we can see from the example showed in Fig 5.

In this way only one test is needed to know if the Dirichlet conditions are imposed in the element. Moreover, the data locality is preserved, allowing to achieve good performance because remote data access is more time consuming than local references.

4 Results

In this section we describe the results obtained running the algorithm on the CINECA Cray T3D using the following parameters: $\tau = 0$, $\nu = 0$, $\gamma = 0$, f=y h= 1 (unviscous case); all parameters are in non dimensional units [IHB-93]. The application ran for 200 time steps using two different grids of 8 or 32 elements; each element contains 16 *nodes* on each direction (256 total points per element).

In the following we report on timings for a very early Cray T3D implementation and for the current version (the work is presently in progress). We report the global time ($\mathbf{T_{tot}}$) including I/O operations, and the computation time ($\mathbf{T_{comp}}$). All timings are expressed in seconds.

As a reference, the performance of the serial implementation of the same algorithm on the CINECA Cray C-90 are reported in Table 1 and in Table 2. We note that no specific optimization effort has been applied for the serial version: in particular the short vector length of the computational kernel (npts = 16) severely limits the possibility to achieve a significant portion of C-90 peak performance.

In Table 3 and 4 we report the Cray T3D timings for the two versions of the program on the 8 element grid.

From the previous tables and from Fig. 6, we can see that the current version shows very good performances compared to the original code and almost linear scalability. We can see also that on this algorithm one C-90 CPU is equivalent to about four Cray T3D processors.

Due to the data distribution method the number of processors is limited by the number of grid elements.

In Table 5 and in Fig. 7 the performance figures for the 32 element grid computation are shown (optimized version only).

As we can see from Table 5 and Fig. 7 the performance is again very satisfactory. The scalability is almost linear and this trend is maintained when the number of processors increases from 8 up to 32.

5 Conclusions

In this work we have presented the parallel implementation of a spectral finite element method for the Cray T3D. The good

```
        subroutine set_boundary_velocity(var)
        implicit  none
        include   'Shallow.h'
        real*8    var(node,nelem)
        real*8    t_est(nelem,npts),t_west(nelem,npts)
        real*8    t_north(nelem,npts),t_south(nelem,npts)
        integer i,n,nest,nwest,nnorth,nsouth,ik
cdir$ shared var(:, :block(1))
cdir$ shared t_est(:block(1), :),t_west(:block(1), :)
cdir$ shared t_north(:block(1), :),t_south(:block(1), :)
        ik = npts*(npts-1)
cdir$ doshared (n) on var(1,n)
        do n=1,nelem
c.....east neighbour copy in t_est
          nest = n + nx
          if (nest.le.nelem) then
            do i-1,npts
              t_est(n,i) = var(i,nest)
            end do
          end if
c..... west neighbour copy in t_west
          nwest = n - nx
          if (nwest.gt.0) then
            do i=1,npts
              t_west(n,i) = var(ik+i,nwest)
            end do
          end if
        end do
c..... east and west boundary element update
cdir$ doshared (n) on var(1,n)
        do n=1,nelem
          nest = n + nx
          if (nest.le.nelem) then
            do i=1,npts
              var(ik+i,n) = var(ik+i,n) + t_est(n,i)
            end do
          end if
          nwest = n - nx
          if (nwest.gt.0) then
            do i=1,npts
              var(i,n) = var(i,n) + t_west(n,i)
            end do
          end if
        end do
c..... north neighbour copy in t_north
cdir$ doshared (n) on var(1,n)
        do n=1,nelem
          nnorth = n + 1
          if (mod(n,nx).ne.0) then
            do i=1,npts
              t_north(n,i) = var((i-1)*npts+1,nnorth)
            end do
          end if
c..... south neighbour copy in t_south
          nsouth = n - 1
          if (mod(nsouth,nx).ne.0) then
            do i=1,npts
              t_south(n,i) = var(npts*i,nsouth)
            end do
          end if
        end do
c.....north and south boundary element update
cdir$ doshared (n) on var(1,n)
        do n=1,nelem
          nnorth = n + 1
          if (mod(n,nx).ne.0) then
            do i=1,npts
              var(npts*i,n) = var(npts*i,n) + t_north(n,i)
            end do
          end if
          nsouth = n - 1
          if (mod(nsouth,nx).ne.0) then
            do i=1,npts
              var((i-1)*npts+1,n) = var((i-1)*npts+1,n) + t_south(n,i)
            end do
          end if
        end do
        return
        end
```

Figure 4: Neighbour update for the velocity grid elements

```
c.......Restablish the Dirichlet conditions
cdir$ doshared (n) on rhsug(1,n)
        do n=1,nelem
c.....test for the Dirichlet conditions on U
c.....for the element.    (one test!)
          if (idiricu(1,n).ne.0) then
            do i=1,npts
              k=idiricu(i,n)
              rhsug(k,n) = uold(k,n)
            end do
          end if
c.....test for the Dirichlet conditions on V
c.....for the element.    (one test!)
          if (idiricv(1,n).ne.0) then
            do i=1,npts
              k=idiricv(i,n)
              rhsvg(k,n) = vold(k,n)
            end do
          end if
        end do
```

Figure 5: Dirichlet conditions

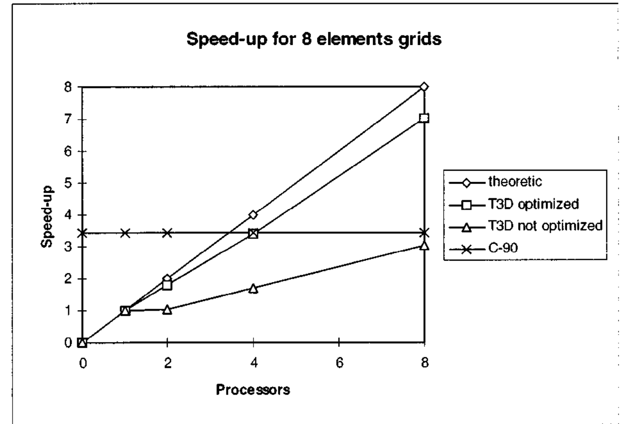

Figure 6: Speed-up for 8 element grid

Table 1. C-90 timing: 8 element grid.

Version	T_{comp}	T_{tot}
Original	77.05	78.79
Modified	75.36	77.27

Table 2. C-90 timings: 32 element grid.

Version	T_{comp}	T_{tot}
Original	349.54	356.34
Modified	342.47	350.17

Table 3. Non optimized version timings.

Processors	T_{comp}	Speed-up	Efficiecy
1	387.92		
2	371.59	1.04	0.52
4	227.48	1.70	0.42
8	127.62	3.03	0.37

Table 4. Optimized version timings.

Processors	T_{comp}	Speed-up	Efficiecy
1	259.01		
2	143.81	1.80	0.90
4	76.34	3.39	0.84
8	36.84	7.03	0.87

Table 5. T3D timing: 32 element grid, optimized version.

Processors	T_{comp}	Speed-up	Efficiency
1	1377.71		
2	812.94	1.69	0.84
4	404.15	3.40	0.85
8	202.58	6.80	0.85
16	106.89	12.88	0.80
32	52.76	26.11	0.81

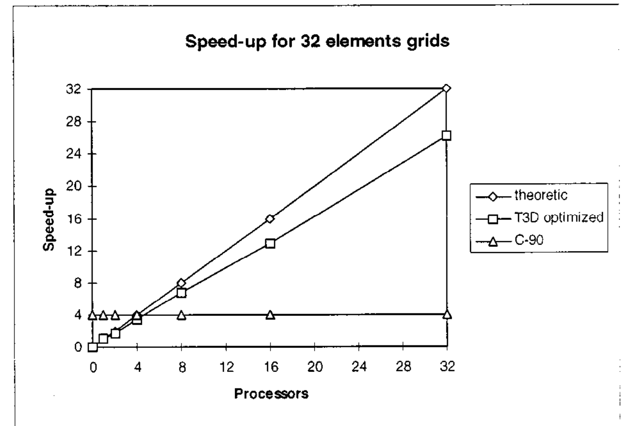

Figure 7: Speed-up for 32 element grid

performance obtained is mainly due to the new data structure introduced that allows a large computational load within each PE as compared to the low cost of inter-processor communications.

As we can see from these preliminary results by increasing the problem size (grids with a high number of elements) the use of a massively parallel processor becomes very favourable, due to an observable almost linear scalability.

Some preliminary single processor optimization techniques have been applied on the current working version but some more detailed optimization analysis are still in progress (single PE optimization, cache alignment, use of **libsci** routines). Furthermore we are working on a new grid with 256 elements that will fully utilize the 64 PEs of CINECA Cray T3D.

Acknowledgments

We thank Dr. M. Iskandarani and Prof. D. Haidvogel for providing us with the original implementation of the scalar code.

References

[Boy-80] Boyd J.P., 'Equatorial solitary waves Part I: Rossby solitons', Journal of Physical Oceanography, Vol 10, N. 11, pp. 1699-1717, (1980).

[Fre-93] Freund R., ' A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems, SIAM J. Sci. Comput. 14, pp. 470-482, (Mar. 1993).

[HCI-94] Haidvogel D., Curchitser E., Iskandarani M., Hughes R., 'Global Modeling of the Ocean and Atmosphere Using the Spectral Element Method', Submitted to Atmosphere-Ocean, (1994).

[IHB-93] Iskandarani M., Haidvogel D., Boyd J. P., 'A Staggered Spectral Finite Element Model With Application to the Oceanic Shallow Water Equations', Technical Report, Institute of Marine and Coastal Sciences, Rutgers University, N.J., (1993).

[Ma-93] Ma H., 'A Spectral Element Basin Model for the Shallow Water Equations', J. of Comp. Phys., 109, pp. 133-149, (1993).

[Mpa-89] Maday Y., Patera A., 'Spectral Element Methods for the Incompressible Navier-Stokes Equations', A. K. Noor and J. T. Oden Eds, (1989).

[PDM-94] Pase D. M., Mac Donald T. Meltzer A., 'MPP Fortran Programming Model', Cray Research Technical Report, Cray Research Inc., Eagan,MN (1994).

[Sew-88] Sewell G., 'The Numerical Solution of Ordinary and Partial Differential Equations, Academic Press, Inc., (1988).

[T3D-94] 'Cray MPP Fortran Reference Manual', SR-2504 6.2 , Cray Research Inc., (Nov. 1994).