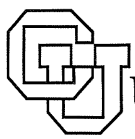
Implementation and Performance of a Grand Challenge 3d Quasi-Geostrophic Multi-Grid code on the Cray T3D and IBM SP2 *

Clive F. Baillie
James C. McWilliams
Jeffrey B. Weiss
Irad Yavneh

CU-CS-771-95



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

*Technical paper for Supercomputing '95, December 4-8, 1995

Implementation and Performance of a Grand Challenge 3d Quasi-Geostrophic Multi-Grid code on the Cray T3D and IBM SP2 *

Clive F. Baillie
Department of Computer Science
University of Colorado, Boulder, CO 80309
email: clive@cs.colorado.edu
phone: (303) 492-7852

James C. McWilliams
Department of Atmospheric Science
University of California, Los Angeles, CA 90095-1565
email: jcm@arapahoe.atmos.ucla.edu

Jeffrey B. Weiss
Program in Atmospheric and Oceanic Sciences
Department of Astrophysical, Planetary, and Atmospheric Sciences
University of Colorado, Boulder, CO 80309
email: jweiss@colorado.edu

Irad Yavneh
Department of Computer Science
Technion - Israel Institute of Technology, Haifa, Israel
email: irad@csa.cs.technion.ac.il

15 March 1995

Keywords: Grand Challenge, Quasi-Geostrophic, Multi-Grid, Turbulence, Rotating Stably Stratified Fluids, Massively Parallel Processors, Cray T3D, IBM SP2

Abstract

We have taken our existing auto-tasked vector Cray C-90 3d Quasi-Geostrophic Multi-Grid (QGMG) code and implemented it in a portable way on most of todays MPPs. Here we report on its performance for the Cray T3D and IBM SP2. On all 16 processors of the C-90 the code achieved 6 Gflops; currently on 256 processors of the T3D we obtain almost 4 Gflops and on 256 processors of the SP2 we obtain 5 Gflops. We find almost perfect scaling of performance with processor number for the T3D all the way up to 256 processors, and good scaling for the SP2 up to 128 processors. In this paper we describe how we parallelized the QGMG code and report performance measurements of it.

*Technical paper for Supercomputing '95, December 4-8, 1995

1 Introduction

One of the most important computational problems today is the numerical simulation of high Reynolds number fluid turbulence. As members of the NSF HPCC Grand Challenge Applications Group (GCAG), "Coupled Fields and Geophysical and Astrophysical Fluid Dynamical Turbulence", we are studying incompressible fluid dynamics in several regimes involving environmental rotation and/or stable or unstable density stratification, all of which are motivated by geophysical phenomena. The common theme in this GCAG research is that significant new insights into the dynamics of turbulence can be obtained from high resolution, high Reynolds number computational solutions obtained with efficient algorithms on Massively Parallel Processors (MPPs). Three equation sets are used which are applicable to different physical regimes: quasi-geostrophic, balanced and Boussinesq flows.

Here we concentrate on the quasi-geostrophic equations which describe the nonlinear dynamics of rotating, stably stratified fluids; this is the relevant regime for most planetary-scale motions in the Earth's atmosphere and ocean. The computational methods used to solve these equations are explicit and implicit multigrid (MG) solvers. We have developed an efficient implicit Quasi-Geostrophic Multi-Grid (QGMG) solver and used it to investigate fluids with periodic horizontal boundary conditions and various vertical boundary conditions: periodic, solid-boundary and Ekman drag [1].

The Cray C-90 and current MPPs (Massively Parallel Processors) are very different machines: the former is a shared memory vector supercomputer and the latter are distributed memory computers whose nodes contain superscalar RISC processors. The Cray C-90 is a physically shared memory machine, which means that there is only one memory which all the processors can access. One typically writes a data parallel program and the compiler distributes the loop iterations amoung processors (auto-tasking).

MPPs, on the other hand, have physically distributed memories, which means that each processor has its own local memory; in order to share data the processors must send messages. One therefore writes a message passing program with each processor responsible for part of the problem domain. The message passing system used on the T3D is Cray's version of the Parallel Virtual Machine (PVM) software from Oak Ridge National Laboratory [2]. The IBM SP2 also has PVM but it is not as efficiently implemented as the Message Passing Interface (MPI) software, a new message passing standard which has been widely adopted and implemented by vendors [3]. Therefore we wrote our parallel Quasi-Geostrophic Multi-Grid (QGMG) code with two interfaces, one for PVM and one for MPI, thereby ensuring a completely portable code across all MPPs as well as workstations. In fact we initially tested and debugged the code on a workstation running PVM.

This paper is organized as follows. In section 2 we explain the quasi-geostrophic equations. In section 3 we outline parallel multigrid algorithms and in section 4 describe the implementation of the QGMG code in detail. The performance of the code on the Cray T3D and IBM SP2 is given in section 5, and we finish with some conclusions.

2 The Quasi-Geostrophic equations

Planetary-scale fluid motions in the Earth's atmosphere and oceans are influenced by strong stable stratification and rapid planetary rotation. The appropriate equations of motion for this asymptotic regime are the Quasi-Geostrophic (QG) equations [4]. The extremely turbulent nature of planetary flows leads us to perform high-resolution numerical simulations of QG turbulence in an effort to better understand the large-scale flows which are so important to the Earth's climate.

Due to stratification and rotation, QG flow is nearly incompressible in horizontal planes, and can be described in terms of horizontal velocities u and v, and an associated streamfunction $\psi(x, y, z)$: $u = -\partial\psi/\partial y$, $v = \partial\psi/\partial x$. The resulting small vertical velocities, together with the thinness of the Earth's atmosphere and oceans, make it appropriate to use a vertical coordinate stretched by N/f. Here, f is the Coriolis frequency, equal to twice the vertical component of the planetary rotation rate, and N is the vertical average of $N(z)$, the Brunt-Väisälä frequency, which is related to gradients in the mean density profile $\rho(z)$ and the acceleration due to gravity g by $N^2(z) = g\partial \ln \rho/\partial z$. On the Earth, N/f is typically of order 100. In this stretched coordinate system the QG equations of motion are

$$\frac{\partial q}{\partial t} + \frac{\partial \psi}{\partial x}\frac{\partial q}{\partial y} - \frac{\partial \psi}{\partial y}\frac{\partial q}{\partial x} + \beta\frac{\partial \psi}{\partial x} = -\mathcal{D}, \tag{1}$$

where the potential vorticity q is

$$q = \frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} + \frac{\partial}{\partial z}\left(\frac{1}{S(z)}\frac{\partial \psi}{\partial z}\right). \tag{2}$$

Vertical inhomogeneity in the stratification is represented by $S(z) = N^2(z)/N^2$, and the so-called β-plane approximation is used to include the effect of the variation in f with latitude, $\beta = \partial f/\partial y$. The dissipation operator $\mathcal{D}$ represents the effects of all scales of motion smaller than those explicitly resolved in the numerical calculation; we typically use hyperviscous diffusion, $\mathcal{D} = \nu\nabla^4 q$, where ν is a small hyperviscosity [5]. Thus there are only two main variables in the QG code: ψ and q.

Over the last few years considerable effort has been invested into adapting and developing multigrid techniques for non-elliptic and singular perturbation problems, such as the flows found at high Reynolds number in stably stratified fluids. Using tools developed by Yavneh [6] we now have multilevel algorithms for time-dependent systems that describe geophysical flows, such as QG. Integration of the QG equations requires solving an elliptic boundary-value problem in three dimensions even if the nonlinear advection equation for the potential vorticity is integrated explicitly. Moreover, the vertical-derivative term in this equation generally varies along the vertical coordinate z. We use a multigrid algorithm, which is one of the best known methods for this problem. Furthermore, we discretize the nonlinear advection equation implicitly in time and solve the entire system simultaneously, employing the so-called Full Approximation Storage (FAS) version of the multigrid algorithm. We employ grid-coarsening only in the horizontal directions, using line Gauss-Seidel relaxation in the vertical. The efficiency of the resulting solver is then insensitive to the vertical variation of $S(z)$. Also, the timestep is unrestricted by CFL stability constraints and can be determined by accuracy criteria.

Our implementation of the above equations of motion allow us to study QG turbulence at unprecedented resolutions. Previous computations on the Cray C90 focused on the maximally symmetric case, $S = 1, \beta = 0$ [1, 7, 8], where we found significant discrepancies from a long-standing theoretical prediction of isotropy [9, 10]. Associated with this anisotropy is the self-organization of the potential vorticity field into a large population of roughly spherical coherent vortices, which then align in the vertical. Currently we are performing several computations on both the Cray T3D and the IBM SP2 – investigating the effects of including nonconstant S and nonzero β, in both decaying and equilibrium turbulence.

3 Parallel Multi-Grid algorithms

Multi-grid algorithms are used to accelerate the convergence of relaxation methods like Gauss-Seidel for the numerical solution of partial differential equations. They achieve this by using a

hierarchy of coarser grids with larger spacings to provide corrections to the approximate solution obtained on the finest grid. Thus there are three parts to any multigrid algorithm: relaxation on a given grid (also called level), restriction from a fine to a coarser grid and interpolation back from a coarse to a finer grid. The main problem associated with implementing multigrid on a MPP is the low processor utilization for the coarser grids – in the extreme case of the coarsest grid only one processor is actually doing anything. The first systematic description of the standard parallel multigrid algorithm was that of Brandt [11]. Since then three improved algorithms have been invented: Concurrent Iteration MG (CIMG) [12], Parallel Superconvergent MG (PSMG) [13], and Chopped Parallel MG (CPMG) [14]. The improved algorithms result in better parallel efficiency in terms of the computer time but do not necessarily improve the numerical efficiency of the overall algorithm, which is problem specific. Therefore as a first step we have implemented only the standard algorithm and are employing it in our current production runs while we evaluate the other algorithms for our quasi-geostrophic problem.

4 Implementation of QGMG

The QGMG code employs the typical V-cycle multigrid, restricting all the way down to a 4×4 coarsest grid. As explained above, coarsening is performed only for the two horizontal (out of the three) dimensions so in what follows we ignore the third dimension for the sake of clarity. We discuss each of its three parts – relaxation, restriction and interpolation – in turn, first sequentially and then for the parallel version.

4.1 Sequential

Relaxation of both ψ and q is performed using the Gauss-Seidel algorithm. For q four-color ordering is necessary since we use the Arakawa nine-point discretization stencil for the Jacobian. However for ψ red-black ordering is sufficient (due to five-point stencil for Laplacian). We shall discuss only the simpler code for the red-black checkerboard case i.e. all the even points are updated first, then all the odd points. The sequential relaxation code for ψ looks like:

```
      do ij = 0,1

        do j = 1, bny
          do i = 1+mod(j+ij,2), bnx, 2
            psi(i,j,lv) = 0.25*(psi(i-1,j,lv) + psi(i+1,j,lv) +
     >      psi(i,j-1,lv) + psi(i,j+1,lv) - rhs(i,j,lv))
          end do
        end do

        call update(psi,bnx,bny,lv)

      end do
```

where `ij=0` is red and `ij=1` is black. Note the call to `update`, which exchanges the information on the periodic boundaries.

The restriction operation averages each 2×2 block of points on the fine grid into one point of the coarse grid (`X` denotes point which remains, `0` is point which is destroyed):

```
X0X0X0X0   ->   X X X X
00000000
X0X0X0X0        X X X X
00000000
```

First the residual is calculated on the fine grid and then used on the coarse grid to calculate the right-hand-side. The sequential code for this is straight-forward:

```
      do j = 1, bny
        do i = 1, bnx
          res(i,j,lv) = -((psi(i-1,j,lv) + psi(i+1,j,lv) +
     >      psi(i,j-1,lv) + psi(i,j+1,lv) - rhs(i,j,lv))
     >      - 4.0 * psi(i,j,lv))
        end do
      end do

      call update(res,bnx,bny,lv)

      do bj = 1, bny/2
        do bi = 1, bnx/2
          i = bi*2-1
          j = bj*2-1
          rhs(bi,bj,lv+1) = res(i,j,lv) +
     >      0.5 * (res(i-1,j,lv) + res(i+1,j,lv) +
     >             res(i,j-1,lv) + res(i,j+1,lv)) +
     >      0.25 * (res(i-1,j+1,lv) + res(i+1,j-1,lv) +
     >              res(i-1,j-1,lv) + res(i+1,j+1,lv))
        end do
      end do
```

where `bnx` $\times$ `bny` is size of grid points on fine level; `bnx/2` $\times$ `bny/2` is size of grid points on coarse level; `i, j` are indices of fine grid points on level `lv`; `bi, bj` are indices of coarse grid points on level `lv+1`. `res` is the residual and so-called full-weighting (in which diagonal points are included in the average) is used to average it to obtain the right-hand-side `rhs`.

The most complicated part of the multigrid algorithm is the interpolation from the coarse to the fine grid. The picture is the opposite of the restriction operation (now `0` is a point which is created):

```
X X X X   ->   X0X0X0X0
               00000000
X X X X        X0X0X0X0
               00000000
```

There are actually two ways to do this. The most obvious way is to just send the "top-left-corner" point to the other three points: "top-right-corner", "bottom-left-corner" and "bottom-right-corner", where it is then averaged. However this method is only simple for linear interpolation; for cubic and higher-order interpolation schemes it rapidly becomes complicated due to the number of points required to calculate the average for the "bottom-right-corner" point. Therefore we use a two-step implementation: during the first step the points in the i-direction are interpolated, then in the second step the points in the j-direction. This looks like:

```
X X X X   ->   X0X0X0X0  ->   X0X0X0X0
                              00000000
X X X X        X0X0X0X0       X0X0X0X0
                              00000000
```

and is coded sequentially as follows:

```
          do bj = 1, bny
            do bi = 1, bnx
              i = bi*2-1
              j = bj*2-1
C intermediate top-left-corner
              tpsi(i,j,lv-1) = psi(bi,bj,lv)
C intermediate top-right-corner
              tpsi(i+1,j,lv-1) =
     >          0.5 * (psi(bi,bj,lv) + psi(bi+1,bj,lv))
            end do
          end do

          call update(tpsi,bnx*2,bny*2,lv-1)

          do bj = 1, bny
            do i = 1, bnx*2
              j = bj*2-1
C final top-left-corner and final top-right-corner
              psi(i,j,lv-1) = psi(i,j,lv-1) + tpsi(i,j,lv-1)
C bottom-left-corner and bottom-right-corner
              psi(i,j+1,lv-1) = psi(i,j+1,lv-1)
     >        + 0.5 * (tpsi(i,j,lv-1) + tpsi(i,j+2,lv-1))
            end do
          end do
```

where `tpsi` is used as a temporary variable to store the partially updated points in the i-direction. Note the call to update between the two steps which is necessary to take care of the periodic boundaries for `tpsi` on the fine grid.

4.2 Parallel

It is relatively straight-forward to parallelize both of the sequential implementations detailed above assuming a square grid, and square processor mesh, and that the number of points per processor is always at least one. However, in practice *none* of these assumptions are valid! We can of course choose to use a square rather than a rectangular grid, but only if the physical system being simulated is in a horizontally 1:1 ratio box. We cannot choose the processor mesh to be square because it unduly limits the number of processors that can be used. Finally, the multigrid algorithm restricts all the way down to a 4×4 (or sometimes even a 2×2) grid, therefore having at least one point per processor would restrict simulations to a maximum of 16 (or 4) processors.

Thus, we must develop a general parallel multigrid code which works on non-square girds, on non-square processor meshes and where the number of points can be less than the number of processors. It is this last condition which causes the parallel multigrid code to be more complicated

than one would initially think. We discuss only the two-step implementation of the interpolation scheme because is easier to parallelize. First the easy case where each processor has at least one point, and then the harder case when some processors do not have any points (which only happens on the more restricted levels).

Let us assume that the finest grid we are using has `nx` $\times$ `ny` points and the processor mesh consists of `n` $\times$ `m` = `np` processors. We also assume `n` divides `nx` and `m` divides `ny`. We arrange the processors, numbered `me` = `0,1,...,np-1`, as in the following example for a 32 processor 8$\times$4 mesh:

```
 0  1  2  3  4  5  6  7
 8  9 10 11 12 13 14 15
16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31
```

where we have drawn the first index `i` increasing from left to right and the second index `j` increasing downwards. Using domain decomposition, each processor gets a part of the grid of size `(nx/n)` $\times$ `(ny/m)` starting and ending as follows:

```
start(lv,1) = 1 + mei * (nx/n)
end(lv,1)   = (nx/n) + mei * (nx/n)
start(lv,2) = 1 + mej * (ny/m)
end(lv,2)   = (ny/m) + mej * (ny/m)
```

where `mei` and `mej` are the i, j positions of processor `me` in the grid (e.g. processor `me`=17 has `mei` = 1 and `mej` = 2) and `lv` is the multigrid level. Of course, each processor has its own local indices for these global pieces of the grid, these are simply

```
loops(lv,1) = 1
loope(lv,1)   = (nx/n)
loops(lv,2) = 1
loope(lv,2)   = (ny/m)
```

Hence loops in the original code looking like:

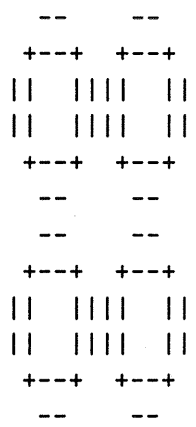
```
do bj = 1, bny
  do bi = 1, bnx
```

become

```
do bj = loops(lv,2), loope(lv,2)
  do bi = loops(lv,1), loope(lv,1)
```

In addition, this method of domain decomposition takes care of internal boundaries between the processors as follows. In the original code, the 2d grid was surrounded by a layer of "ghost points" which store copies of the points on the opposite sides of the grid in order to take care of the periodic boundary conditions. This looks like:

Therefore when we perform the domain decomposition on, say, 4 processors, each processor will get its own ghost points as follows:



The "outside" layer of ghost points are the original ones for the periodic boundaries. The ones on the "inside", i.e. between the processors, are called "internal processor boundaries" and are an artifact of the domain decomposition. However they are an extremely useful artifact: for in them we shall store copies of the points from the neighboring processors which we shall need while executing the multigrid algorithm. Then we need only exchange the internal boundary data once per call of restrict, interpolate or relax, rather than every time we see an `i+1`, `i-1`, `j+1` or `j-1` index in the code.

Again we discuss relaxation, restriction and interpolation. As said above the red-black Gauss-Seidel relaxation is easily parallelized. The only complication is figuring out if a given point `i,j` local to a processor is red or black i.e. even or odd. We do this by having a flag `first_even` telling us whether the first point `i=1`, `j=1` in the processor is even. Thus the parallel relaxation code looks like:

```
      do idum = 0,1

        if (first_even(lv)) then
          ij = idum
        else
          ij = 1-idum
        endif

        do j = loops(lv,2), loope(lv,2)
          do i = loops(lv,1)+mod(j+ij,2), loope(lv,1), 2
            psi(i,j) = 0.25 * (psi(i-1,j) + psi(i+1,j) +
     >      psi(i,j-1) + psi(i,j+1) - rhs(i,j))
          end do
        end do

        call exchange_i(psi,bnx,bny,lv)
        call exchange_j(psi,bnx,bny,lv)

      end do
```

Next the parallel restriction code is obvious:

```
      do j = loops(lv,2), loope(lv,2)
        do i = loops(lv,1), loope(lv,1)
          res(i,j,lv) = -((psi(i-1,j,lv) + psi(i+1,j,lv) +
     >    psi(i,j-1,lv) + psi(i,j+1,lv) - rhs(i,j,lv))
     >    - 4.0 * psi(i,j,lv))
        end do
      end do

      call exchange_i(res,1,lv)
      call exchange_j(res,2,lv)

      do bj = loops(lv+1,2), loope(lv+1,2)
        do bi = loops(lv+1,1), loope(lv+1,1)
          i = bi*2-1
          j = bj*2-1
          rhs(bi,bj,lv+1) = res(i,j,lv) +
     >    0.5 * (res(i-1,j,lv) + res(i+1,j,lv) +
     >           res(i,j-1,lv) + res(i,j+1,lv)) +
     >    0.25 * (res(i-1,j+1,lv) + res(i+1,j-1,lv) +
     >            res(i-1,j-1,lv) + res(i+1,j+1,lv))
        end do
      end do
```

Lastly the parallel interpolate code, which only works for bnx >= n and bny >= m, looks like:

```
      do bj = loops(lv,2), loope(lv,2)
        do bi = loops(lv,1), loope(lv,1)
          i = bi*2-1
          j = bj*2-1
          tpsi(i,j,lv-1) = psi(bi,bj,lv)
          tpsi(i+1,j,lv-1) =
     >    0.5 * (psi(bi,bj,lv) + psi(bi+1,bj,lv))
        end do
      end do

      call exchange_j(tpsi,lv-1)

      do bj = loops(lv,2), loope(lv,2)
        do i = loops(lv-1,1), loope(lv-1,1)
          j = bj*2-1
          psi(i,j,lv-1) = psi(i,j,lv-1) + tpsi(i,j,lv-1)
          psi(i,j+1,lv-1) = psi(i,j+1,lv-1)
     >    + 0.5 * (tpsi(i,j,lv-1) + tpsi(i,j+2,lv-1))
        end do
      end do
```

Now for the complicated case, where we have restricted the grid to bnx $\times$ bny points where bnx $<$ n and bny $<$ m. There are actually two possible approaches when the number of points

is less than the number of processors. First, we can "stay parallel" and have only the processors with points compute and communicate, while the rest do nothing. Or, second, we can "go serial" and send all of the remaining points to all of the processors and have them all continue doing the restricted levels; then on the "way back up" we "go parallel" when we get back to at least one point per processor. So far, we have implemented only the first of these alternatives, which is easier to code.

To explain this, suppose the grid is 2×2 so the only processors in our 8×4 mesh which have points are

```
 0          4

16         20
```

and when we interpolate to 4×4 we get

```
 0    2    4    6
 8   10   12   14
16   18   20   22
24   26   28   30
```

Therefore the processors no longer communicate with their nearest neighboring processors, as they did when `bnx` >= `n` and `bny` >= `m`, now they communicate with processors `skipi` and `skipj` away in the i and j directions respectively, with

```
skipi = n/bnx
skipj = m/bny
```

For both the relaxation and the restriction parts of the multigrid algorithm use of `skipi` and `skipj` in the exchange routines works perfectly well no matter how many points per processor there are. It is only in the interpolate phase that there is any difference. This difference can be reduced to two extra function calls `send_i` and `send_j` invoked only when `skipi` and `skipj` are respectively greater than 1. Thus the final parallel interpolate code looks like:

```
      do bj = loops(lv,2), loope(lv,2)
        do bi = loops(lv,1), loope(lv,1)
         i = bi*2-1
         j = bj*2-1
         tpsi(i,bj,lv-1) = psi(bi,bj,lv)
         tpsi(i+1,bj,lv-1) =
     >     0.5 * (psi(bi,bj,lv) + psi(bi+1,bj,lv))
        end do
      end do

      if (skipi(lv) .gt. 1) call send_i(tpsi,lv-1)

      call asym_exchange_j(tpsi,lv-1)

       do bj = loops(lv,2), loope(lv,2)
```

```
          do i = loops(lv-1,1), loope(lv-1,1)
           j = bj*2-1
           psi(i,j,lv-1) = psi(i,j,lv-1) + tpsi(i,bj,lv-1)
           psi(i,j+1,lv-1) = psi(i,j+1,lv-1)
     >      + 0.5 * (tpsi(i,bj,lv-1) + tpsi(i,bj+1,lv-1))
          end do
        end do

      if (skipj(lv) .gt. 1) call send_j(psi,lv-1)
```

Note that `update` has been replaced with a call to `send_i` if `skipi` > 1 and a call to `asym_exchange_j`. `send_i` sends the values of `tpsi` from the processors which are still "alive" to their "dead" neighbors `+skipi` away in the i direction. `asym_exchange_j` sends back values of `tpsi` from processors `+skipj*2` away in the j direction, to be computed locally. Then finally we also need to call `send_j` which sends the correct interpolated values of `psi` from the alive processors to their `+skipj` dead neighbors. Note that for the sends/receives in the j direction *all* the points in the i direction are involved – this is because we are using the two-step implementation of the multigrid interpolation algorithm.

5 Performance of QGMG

We parallelized the QGMG code in the summer of 1994 when only PVM was widely available. Therefore we had to create and manage the processor topology ourselves as PVM contains no functions for this. Around the end of 1994 it became generally known that IBM's version of MPI performed much better than their version of PVM (called PVMe) on the SP2 and so we wrote an MPI version of QGMG. This involved changing only our exchange functions plus minor changes to start-up and shut-down function calls. The MPI code still uses our processor topology despite the fact that MPI provides similar functionality. More recently we discovered that the MPI code also performs significantly better than the PVM version on the Intel Paragon.

We have been running in production on the T3D for some time and are just starting to do so on the SP2. It turns out that the T3D has much more reliable I/O (using its host Cray to do it) and therefore production running there is significantly easier. Of course, even today, no MPP yet provides decent user-level parallel I/O so we had to write our own package. We shall leave discussion of this for another time.

5.1 Cray T3D

In Table 1 we present times in seconds of a ten multigrid cycle run of QGMG using a grid of size 256^3, on various numbers of processors (and for different processor topologies) for several different versions of the code. Note that this size problem does not fit into the memory of less than 32 T3D processors. When we first implemented QGMG on the T3D, the Cray PVM available was version 3.2 which did not have global reduction operations, that is functions like SUM, MAX, MIN, so we had to write our own. Unfortunately they were very inefficient and did not scale properly, as can be seen from the times in the second column of Table 1. Recently, Cray released version 3.3 of PVM which does have global reduction functions and these gave dramatic improvements in performance (column 3). For a further smaller improvement we also tried the so-called shared memory (SHMEM) global reduction functions (column 4). For the PVM 3.3 and SHMEM cases

Table 1: Total times in seconds for 10 multigrid cycles of QGMG on various numbers of processors of the T3D.

Processors	PVM 3.2	PVM 3.3	SHMEM
32 (1x32)	321.9	239.8	244.8
32 (4x8)	-	209.1	211.7
64 (1x64)	477.6	144.3	144.6
64 (8x8)	-	120.8	116.2
128 (1x128)	-	99.5	99.9
128 (8x16)	-	63.2	60.0
256 (1x256)	-	95.1	87.2
256 (16x16)	-	42.3	31.7

Table 2: Total times in seconds for 10 multigrid cycles of QGMG on various numbers of processors of the SP2.

Processors	PVMe	MPI
4 (2x2)	454.3	474.6
16 (4x4)	238.1	151.4
32 (4x8)	376.4	94.1
64 (8x8)	-	59.5
128 (8x16)	-	29.4
256 (16x16)	-	22.7

the processor topology makes a significant difference: choosing a square configuration (e.g. 16x16) yields more than twice the performance of a linear one (1x256).

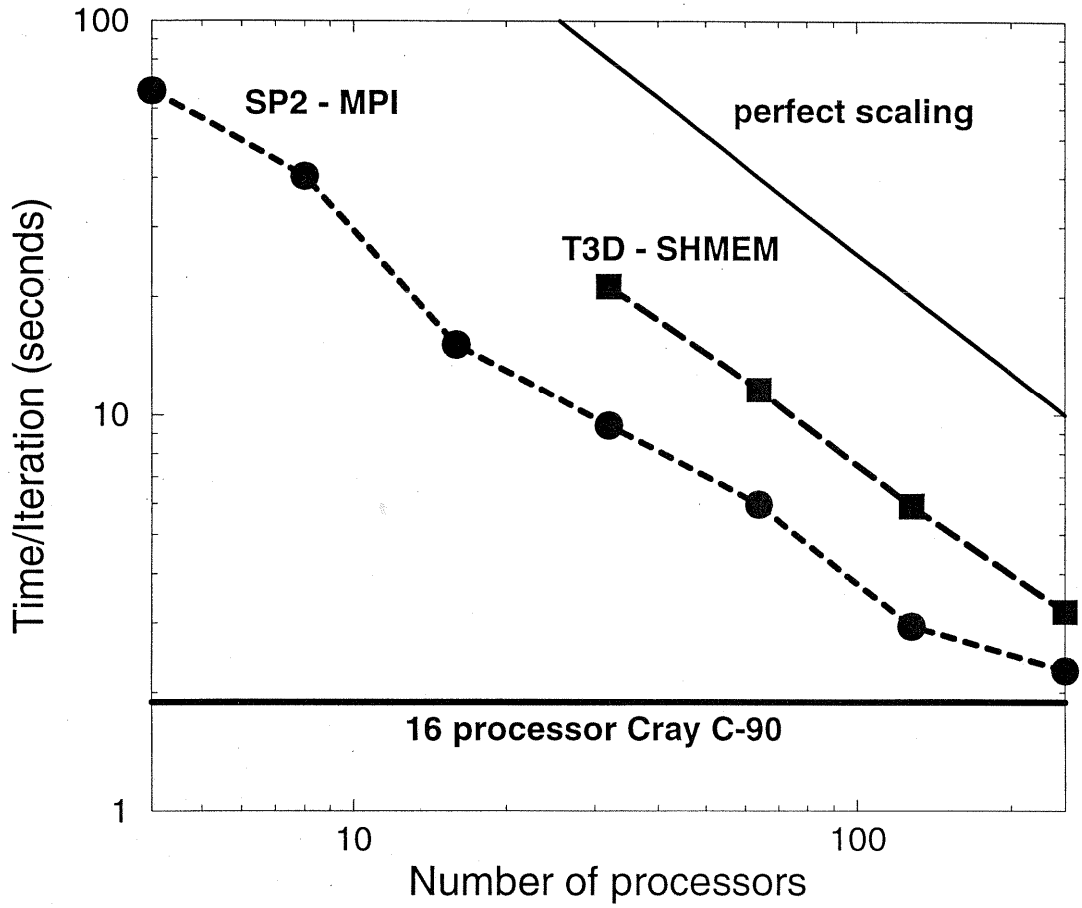
The original QGMG code running on all 16 processors of the C-90 achieved 6.0 Gflops and took 18.9 seconds for this benchmark run. Thus the fastest time we obtained on the T3D, 31.7 seconds for 256 processors, corresponds to 3.6 Gflops.

5.2 IBM SP2

In Table 2 we present times in seconds of a ten multigrid cycle run of QGMG using a grid of size 256^3, on various numbers of processors for several different versions of the code. We use the most square processor topology possible. Note that this size problem fits into the memory of as little as 4 SP2 processors since they each have eight times as much memory as the T3D processors (512 MB compared with 64 MB). Initially we tried our PVM version of QGMG with IBM's PVMe software but the performance was not impressive – again due to lack of global reduction functions – see column 2 of Table 2. Now we are using MPI which has its own global reduction functions to obtain very good performance (column 3). However the scaling is not a good for the SP2 as it was for the T3D - 256 processors is only about 20% faster than 128. This is because the communication network in the SP2 is not as good. Overall the performance on the SP2 is still faster than the T3D because the SP2 processors are between two and three times faster for this code.

The fastest time we obtained on the SP2 was 22.7 seconds for 256 processors which corresponds to 5.0 Gflops.

Figure 1: QGMG Performance



6 Conclusions

We have parallelized a 3d quasi-geostrophic multigrid code originally designed for the Cray C-90 in a portable fashion and are running it in production on the Cray T3D and IBM SP2. We have done this via domain decomposition and message passing using PVM and MPI. On all 16 processors of the C-90 the original code achieved 6 Gflops; currently on 256 processors of the T3D we obtain almost 4 Gflops and on 256 processors of the SP2 we obtain 5 Gflops, with the parallel code. In Figure 1 we summarize the best performance results on each machine with log-log axes in order to display the scaling. The slope of the line for the T3D is very close to the perfect scaling slope of -1. As mentioned above the performances of both the T3D and the SP2 have been improving steadily as new versions of software are released. Therefore we expect that ultimately both of these machines will surpass the C-90 in overall performance.

To date we have implemented only the standard multigrid algorithm, in the future we will try out some of the improved algorithms we mentioned in section 3 (CIMG, PSMG and CPMG).

Currently we are performing large-scale computations of quasi-geostrophic turbulence on both the Cray T3D and IBM SP2 with system sizes of 256^3 and higher.

Acknowledgements

This work is supported by NSF Grand Challenge Applications Group Grant ASC-9217394. CFB is also partially supported by DOE contract DE-FG02-91ER40672 and by NASA HPCC Group Grant NAG5-2218. JBW is also partially supported by NOAA grant DOC-NA-26-GPO-12201. IY and JCM were also partially supported by NSF through the National Center for Atmospheric Research. The Cray T3D runs were performed at the Pittsburgh Supercomputing Center under grant MCA93AS010P through funding from the National Science Foundation. We would like to especially thank Raghurama Reddy for his help. The IBM SP2 results were obtained on the machine at the Cornell Theory Center through its early user program. We would like to particularly thank John Zollweg and Robert Feldman for their help.

References

[1] I. Yavneh and J.C. McWilliams, "Multigrid solution of stably stratified flows: the quasi-geostrophic equations", submitted to *J. Sci. Comp.* (1995).

[2] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, "PVM: Parallel Virtual Machine – A Users' Guide and Turorial for Networked Parallel Computing" (The MIT Press, Cambridge, MA, 1994).

[3] W. Gropp, E. Lusk, A. Skjellum, "Using MPI: Portable Parallel Programming with the Message-Passing Interface" (The MIT Press, Cambridge, MA, 1994).

[4] P.B. Rhines, *Annu. Rev. Fluid Mech.*, **11**, 401 (1979).

[5] R. Sadourny and C. Basdevant, *C.R. Acad. Sci.*, **39**, 2138 (1981).

[6] I. Yavneh, *SIAM J. Sci. Comput.*, **14**, 1437-1463 (1993).

[7] J.C. McWilliams, J.B. Weiss, and I. Yavneh, *Science*, **264**, 410 (1994).

[8] J.C. McWilliams and J.B. Weiss, *CHAOS*, **4**, 305 (1994).

[9] J.G. Charney, *J. Atmos. Sci.*, **28**, 1087 (1971).

[10] J. Herring, *J. Atmos. Sci.*, **37**, 969 (1980).

[11] A. Brandt, "Multigrid solvers on parallel computers", in: *Elliptic problem solvers*, ed. M. Schultz (Academic Press, New York, NY, 1981).

[12] D. Gannon and J. van Rosendale, *J. Parallel Distributed Comput.*, **3**, 106-135 (1986).

[13] P. Frederickson and O. McBryan, "Parallel Superconvergent Multigrid", in: *Proceedings of the Third Copper Mountain Conference on Multigrid Methods*, ed. S. McCormick (Marcel Dekker, New York, NY, 1989).

[14] S.N. Gupta, M. Zubair and C.E. Grosch, *J. of Scientific Comput.*, **7**, 263-279 (1992).