CONCURRENCY: PRACTICE & EXPERIENCE, 4(7), 509-531 (OCT. 1992)

PARALLELIZING THE SPECTRAL TRANSFORM METHOD. PART II

DAVID W. WALKER*, PATRICK H. WORLEY* AND JOHN B. DRAKE*

Abstract. The spectral transform method is a widely-used numerical technique for solving partial differential equations on the sphere in global climate modeling. This paper describes the parallelization and performance of the spectral method for solving the nonlinear shallow water equations on the surface of a sphere using a 128-node Intel iPSC/860 hypercube. Solving the shallow water equations represents a computational kernel of more complex climate models. This work is part of a research program to develop climate models that are capable of much longer simulations at a significantly finer resolution than current models. Such models are important in understanding the effects of the increasing atmospheric concentrations of greenhouse gases, and the computational requirements are so large that massively parallel multiprocessors will be necessary to run climate models simulations in a reasonable amount of time.

The spectral method involves the transformation of data between the physical, Fourier, and spectral domains. Each of these domains is two-dimensional. The spectral method performs Fourier transforms in the longitude direction followed by summation in the latitude direction to evaluate the discrete spectral transform. A simple way of parallelizing the spectral code is to decompose the physical problem domain in just the latitude direction. This allows an optimized sequential FFT algorithm to be used in the longitude direction. However, this approach limits the number of processors that can be brought to bear on the problem. Decomposing the problem over both directions allows the parallelism inherent in the problem to be exploited more effectively – the grain size is reduced, so that more processors can be used.

Results are presented that show that decomposing over both directions does result in a more rapid solution of the problem. The results show that for a given problem and number of processors, the optimum decomposition has approximately equal numbers of processors in each direction. Load imbalance also has an impact on the performance of the method. The importance of minimizing communication latency and overlapping communication with calculation is stressed. General methods for doing this, that may be applied to many other problems, are discussed.

Key words. hypercube multiprocessors, parallel Fourier transform, parallel Legendre transform, parallel spectral transform method shallow water equations,

AMS(MOS) subject classifications. primary 65W05; secondary 65M60, 68M20

1. Introduction.

In order to understand the effects of the increasing atmospheric concentrations of greenhouse gases, climate models are needed that are capable of much longer and more numerous simulations at a significantly finer resolution than are currently available. Developing such an advanced climate model will require advances in hardware, numerical algorithms, and model physics. In particular, it is clear that massively parallel multiprocessors will be necessary to run such simulations in a reasonable amount of time. As part of this research effort, we are investigating whether current numerical techniques are suitable for use in an advanced climate model.

The spectral transform method [5] is a widely-used numerical technique for solving

* Mathematical Sciences Section, Oak Ridge National Laboratory, P.O. Box 2008, Oak Ridge, Tennessee 37831-6367 (walkerdw@ornl.gov, worleyph@ornl.gov, or bbd@ornl.gov). This research was supported by the Atmospheric and Climate Research Division, Office of Energy Research, U.S. Department of Energy under contract DE-AC05-84OR21400 with Martin Marietta Energy Systems Inc.

partial differential equations on the sphere in global climate modeling (see, for example, [1]). For example, it is used in CCM1 [10] (the Community Climate Model 1), and its successor CCM2. There are both numerical and algorithmic issues to be considered before using the spectral transform method for climate models with much finer resolutions. In the work described here, we restrict ourselves to investigating how efficiently the spectral transform method can be parallelized on distributed memory multiprocessors, and how this performance is likely to scale as both the problem size and the number of processors increase.

In this paper, which follows on from the preliminary work described in [11], results are presented for a parallel FORTRAN program that uses the spectral transform method to solve the nonlinear shallow water equations on the sphere. These results show that an efficient implementation is possible on a 128-node Intel iPSC/860 hypercube, and that the high-resolution cases of interest are expected to run efficiently on larger, more powerful, distributed memory machines, such as the Intel Delta and Sigma multiprocessors. The results also highlight the need for specialized programming techniques on machines for which computation is fast compared with the asymptotic communication speed and message latency. Except for embarassingly parallel problems, such machines can be exploited efficiently only if steps are taken to reduce and/or mask the effects of communication overhead. In this work we use large granularity pipelining and task interleaving to reduce the impact of communication overhead to an acceptably low level. These techniques, discussed in more detail in Section 5.4, are applicable to a large class of scientific and engineering problems.

2. The Problem. The shallow water equations constitute a simplified weather prediction model that has been used to investigate numerical methods, and benchmark a number of machines. The sequential code, SSWMSB, from which the parallel version described in this work was derived, was originally written by Dr. J. J. Hack at NCAR. This particular code is a good approximation to a computational kernel of CCM2, which is currently being developed at the National Center for Atmospheric Research (NCAR). We are currently developing a parallel version of CCM2 that will run on the Intel iPSC/860 and similar multiprocessors, and will present our initial results in parallelizing CCM2 in a subsequent paper.

The SSWMSB code uses the spectral transform method to solve the shallow water equations on the surface of a sphere. In each timestep the state variables of the problem are transformed between the physical domain, where most of the physical forces are calculated, and the spectral domain, where the terms of the differential equation are evaluated. A more complete description of the method is given in [11]. The spectral representation of a state variable, ξ, on the surface of a sphere is defined by an approximation to the variable by a truncated series of spherical harmonic functions,

$$\xi(\lambda, \mu) = \sum_{m=-M}^{M} \sum_{n=|m|}^{N(m)} \xi_n^m P_n^m(\mu) e^{im\lambda} \tag{1}$$

where $\mu = \sin\theta$, θ is latitude, λ is longitude, and $P_n^m(\mu)$ is the associated Legendre function. In the physical domain, state variables are approximated on an $I \times J$ longitude-latitude grid. Exact, unaliased transforms of quadratic terms are obtained if M is chosen to satisfy $J \geq (3M+1)/2$, and if $I = 2J$, and $N(m) = M$. In this work we use a fast Fourier transform (FFT) algorithm that requires I to be a power of 2, and M is chosen to be the minimum value satisfying the preceding conditions. Thus, the value of M characterizes the grid resolution, and the term "TM" is used to denote a particular discretization. For example, the T85 case has $M = 85$, $J = 128$, and $I = 256$.

Transforming from physical coordinates to spectral coordinates involves performing an FFT for each line of constant latitude, followed by integration over latitude using Gaussian quadrature to obtain the spectral coefficients,

$$\xi_n^m = \sum_{j=0}^{J-1} \xi^m(\mu_j)\, P_n^m(\mu_j)\, w_j \tag{2}$$

where ξ^m is the mth Fourier coefficient, and w_j is the Gaussian quadrature weight corresponding to latitude μ_j. In the parallel implementation, the fast Fourier transform and the integration over latitude, that together give the spectral transform, define the problem. All other calculations, such as advancing the spectral coefficients by one time step, and optionally filtering the solution to improve stability, are perfectly parallel, requiring no interprocessor communication.

3. Data Distribution. In general, a one-dimensional array of data can be distributed (or *decomposed*) among a set of processors by first arranging the data into non-intersecting subsets, and then uniquely assigning one subset to each processor. In many cases, the decomposition of arrays over more than one dimension can be expressed as the Cartesian product of one-dimensional decompositions over each array dimension.

The indices of a one-dimensional array of N items can be partitioned into N_p subsets, $\mathcal{J}_i$, as follows,

$$\mathcal{J}_i = \{\, j \,:\, j \in Z^+ \quad \& \quad k_{min}(i) \leq \Pi(j) < k_{max}(i) \quad \& \quad 0 \leq j < N \,\} \tag{3}$$

for $i = 0, 1, \ldots, N_p - 1$, where it has been assumed that array indices are non-negative and start at 0. Here Z^+ is the set of non-negative integers, and k_{min} and k_{max} are integer-valued functions satisfying,

$$\begin{aligned} k_{min}(0) &= 0 \\ k_{min}(i) &= k_{max}(i-1) \qquad (i = 1, 2, \ldots, N_p - 1) \\ k_{max}(N_p - 1) &= N_c, \end{aligned} \tag{4}$$

where $N_c = \lceil N/N_p \rceil \cdot N_p$, and $\Pi(\cdot)$ is the *partitioning function* that reorders the index set $\{0, 1, \ldots, N-1\}$. Thus, the partitioning in Eq. (3) can be regarded as taking place in two phases. First the index set is reordered so that the integers $\Pi(j)$ form a sequence running from 0 up to $N-1$. This ordering of the indices is then divided into blocks of contiguous items. The functions k_{min} and k_{max} should be chosen to ensure good load balance.

Two common examples of data distributions are the linear and scattered (or wrap) partitionings. In a linear partition the index set is simply divided into contiguous blocks.

Thus, no reordering is required, and $\Pi(j) = j$. In a scattered partitioning the data are reordered according to the function,

$$\Pi(j) = \lfloor j/N_p \rfloor + (j \bmod N_p) \cdot \lceil N/N_p \rceil \tag{5}$$

with $k_{min}(i) = i \cdot \lceil N/N_p \rceil$. This groups together data items whose indices differ by multiples of N_p.

Having partitioned the data each subset must next be assigned to one of the N_p processors. This can be expressed as $A(i) = p$, where $A(\cdot)$ is the *assignment function*, indicating that the ith subset is assigned to processor p. Examples of assignment functions are the identity mapping, $A(i) = i$, and the binary-reflected Gray code mapping, $A(i) = G(i)$. If $c_k(i)$ denotes the kth most significant bit of i, then the bitwise definition of $G(\cdot)$ is

$$c_k(G(i)) = \mathrm{XOR}\,(c_{k+1}(i), c_k(i)), \qquad k = 0, 1, \ldots \tag{6}$$

where XOR denotes the bitwise exclusive OR of its arguments.

In solving the shallow water equations, computations are performed in both the physical and the spectral domains, and transforming from one domain to the other involves passing through the Fourier domain. Thus, we must be concerned with the distribution of data in three domains. In all three domains the basic data structures are two-dimensional, and the decompositions in the physical and Fourier domains (but not the spectral domain) can be expressed as Cartesian products of two one-dimensional decompositions, as explained in more detail below. In all three domains the data are distributed across a (logical) $N_x \times N_y$ processor grid. In the next section we describe the data decomposition in each domain, giving a specific example for the T10 case decomposed onto a 4×4 processor grid.

3.1. Decomposition of the Physical Domain. The physical domain of the problem is two-dimensional, with longitude being one dimension, and latitude the other. Longitude and latitude are discretized to form an $I \times J$ grid. In the evaluation of the spectral coefficients, FFTs are performed over the longitude direction, and integration (i.e., weighted summation) is performed over the latitude direction. Since there is no coupling between different data points in either the physical or spectral domain, the data partitioning can be optimized in the longitude direction for the evaluation of the FFTs, and in the latitude direction for the integration.

In the FFT algorithm data items interact in a pairwise fashion, and the array indices of each interacting pair of data items differ in exactly one bit. For this reason, on hypercube multiprocessors non-local communication in the concurrent FFT algorithm is minimized if the input data are decomposed in "natural" order. Thus, in the physical domain a linear partitioning is used in the longitude direction, and the assignment function in the longitude direction is the identity function.

A linear partitioning could also be used in the latitude direction, but it is computationally more efficient to process the corresponding latitude lines in the north and south hemispheres in pairs. Thus, if $i = J/N_y$ is the number of latitudes per processor, and we define

$$b_i(j) = i \cdot \left\lfloor \frac{j}{i/2} \right\rfloor + j \bmod (i/2) \tag{7}$$

then in the physical domain the partitioning function in the latitude direction is,

$$\Pi(j) = \begin{cases} b_i(j) & \text{if } j < I/4 \\ I - 1 - b_i(j) & \text{if } j \geq I/4 \end{cases} \tag{8}$$

The k_{min} and k_{max} in Eqs. (3) and (4) are chosen so that, as nearly as possible, each processor contains the same number of data points.

A ring algorithm is used to perform the integration over latitude, and a Gray code assignment function is used in this direction, since this ensures that neighbors in the ring are directly connected by a communication channel. It should be noted that this assignment function is appropriate only for a hypercube topology. For a mesh topology, for example, the identity function should be used in both the longitude and latitude directions.

The partitioning of the longitude-latitude grid in the physical domain can be expressed as the Cartesian product of the one dimensional partitionings in each direction. Thus, each subset of the data is labeled by two indices, (i, j). The assignment function can be written as,

$$A(i,j) = i + G(j) \cdot N_x, \qquad (i = 0, 1, \ldots, N_x - 1, \quad j = 0, 1, \ldots, N_y - 1) \tag{9}$$

where $G(j)$ denotes the binary-reflected Gray code of j (see Eq. (6)). Note that here it has been assumed that blocks of contiguous bits in the binary representation of the processor number have been assigned to each dimension. That is, the least significant $\log_2 N_x$ bits of the processor number correspond to the longitude dimension, and the most significant $\log_2 N_y$ bits to the latitude dimension. In general, the partitioning of bits over dimensions can be done in any unique way.

3.2. Decomposition of the Fourier Domain. The Fourier domain can be regarded as a wavenumber-latitude grid, so like the physical domain, the Fourier domain is two-dimensional. However, a different decomposition is used. The differences arise because of the way in which the FFT algorithm permutes the ordering of the output Fourier coefficients. The one-dimensional FFT produces Fourier coefficients in "bit-reversed" order. That is, if the number of data points to be transformed is $I = 2^k$, then the array index of the mth Fourier coefficient is given by the k bits of m written in reverse order. Denoting this quantity by $B_k(m)$ then, for example, $B_4(12) = 3$ since if the 4 bits in the binary representation of 12 are written in reverse order we get 0011, the decimal representation of which is 3. A second factor also influences the partitioning in the Fourier domain. As described in Section 4.1, a version of the FFT algorithm designed for transforming real functions is used. Having performed the FFT on the half-length complex array, the Fourier coefficients of the original real data are found by combining data for indices m and $I/2 - m$. In general, communication is needed to bring these points into the same processor. This is done by reordering the data so that data points to be combined differ by 1 in their array indices. Thus, in the Fourier domain the partitioning function, $\Pi(\cdot)$, in the wavenumber direction includes the combined effects of bit reversal and the reordering needed to extract the Fourier coefficients, and is given by,

$$\Pi(j) = \begin{cases} B_k(j) & \text{if } j \leq I/4 \\ B_k(3I/4 - j) & \text{if } j > I/4 \end{cases} \tag{10}$$

The partitioning function in the latitude direction is the same as in the physical domain.

The assignment function in the Fourier domain also differs from that in the physical domain. The difference arises because of the way that data are communicated in the outer loop of the FFT algorithm, which is explained in more detail in Section 4.2. In the Fourier domain the ith set of data in the wavenumber direction is assigned to the processor obtained by cyclically shifting the d bits of the processor number one bit to the right (where the number of processors is 2^d). Denoting this quantity by $R_d(i)$ then, for example, $R_3(1) = 4$, since if the 3 bits, 001, of the binary representation of 1 are cyclically shifted one bit to the right we get 100, the decimal representation of which is 4. A binary-reflected Gray code assignment function is still used in the latitude direction, so in the Fourier domain the assignment function is

$$A(i,j) = R_{d_x}(i) + G(j) \cdot N_x, \qquad (i = 0, 1, \ldots, N_x - 1, \quad j = 0, 1, \ldots, N_y - 1) \tag{11}$$

where $d_x = \log_2 N_x$.

3.3. Decomposition of the Spectral Domain. In decomposing the spectral coefficients, ξ_n^m, a two-dimensional processor grid is again used. In the Fourier domain, the ith subset of wavenumbers is assigned to column number $R_{d_x}(i)$ in the processor grid. A similar partitioning over wavenumber is used in the spectral domain. However, wavenumbers for which $m > M$ are not used, so in the spectral domain we have the following partitioning over wavenumber;

$$\mathcal{S}_i = \{ j : j \in Z^+ \quad \& \quad k_{min}(i) \leq \Pi(j) < k_{max}(i) \quad \& \quad 0 \leq j \leq M \} \tag{12}$$

Each column of the processor grid contains N_y processors, and column number $R_{d_x}(i)$ is responsible for the spectral coefficients ξ_n^m, where $m \in \mathcal{S}_i$ and $n = m, m+1, \ldots, M$. Within each column of processors these coefficients can be ordered as a linear array by running first over n and then over m. This array is then divided into subsets using a linear partitioning, and is assigned to the processors in the column using a binary-reflected Gray code assignment function. It should be noted that, since the number of spectral coefficients assigned to each column of processors will differ slightly, the partitioning of the spectral coefficients cannot be expressed as the Cartesian product of two one-dimensional partitionings. Some degree of load imbalance will also arise from this partitioning, and this topic is discussed in Section 5.3.

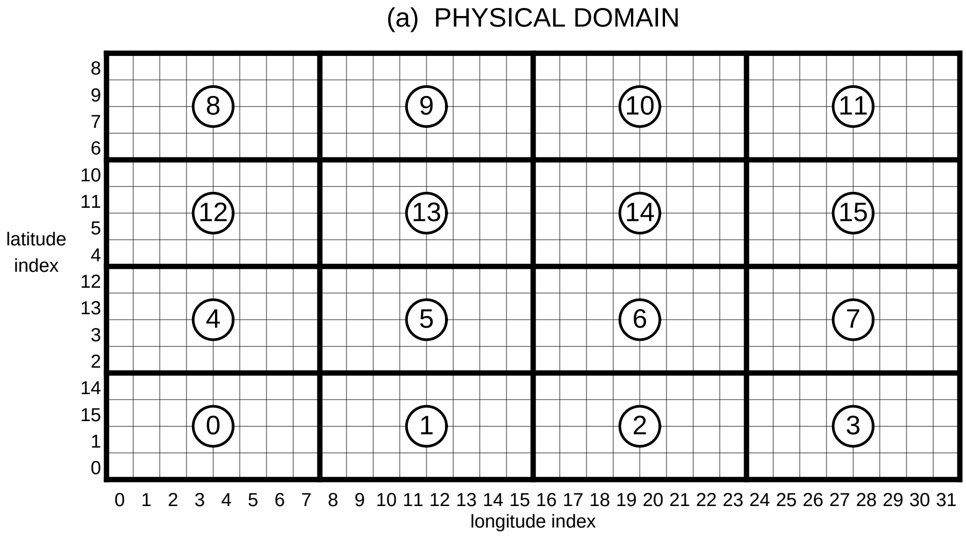
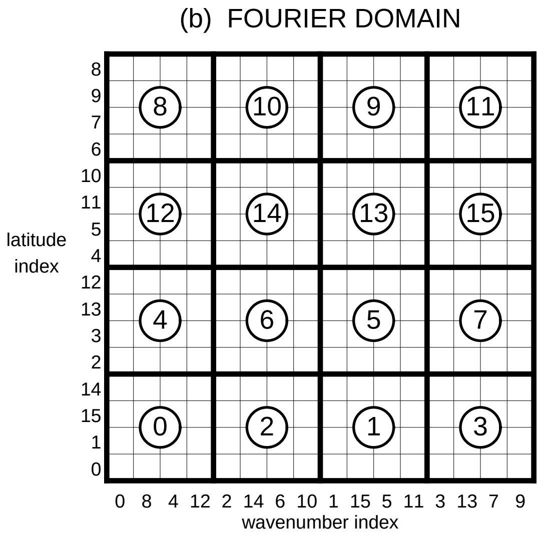


Figure 1. The decomposition of (a) the physical, and (b) the Fourier domains over a 4×4 grid of processors. The assignment of processors to data subblocks is appropriate for a hypercube topology. Each small cell represents a data item. The thicker lines show the boundaries between processors. The numbered circles show the processor numbers.



3.4. Data Decomposition for an Example Problem. To demonstrate how the physical, Fourier, and spectral domains are decomposed for a specific problem, we consider the T10 case for a 4×4 processor grid. The T10 case does not have sufficient resolution to be of practical interest, however, it is useful for illustrative purposes. For the T10 problem there are 32 data points in the longitude direction and 16 data points in the latitude direction.

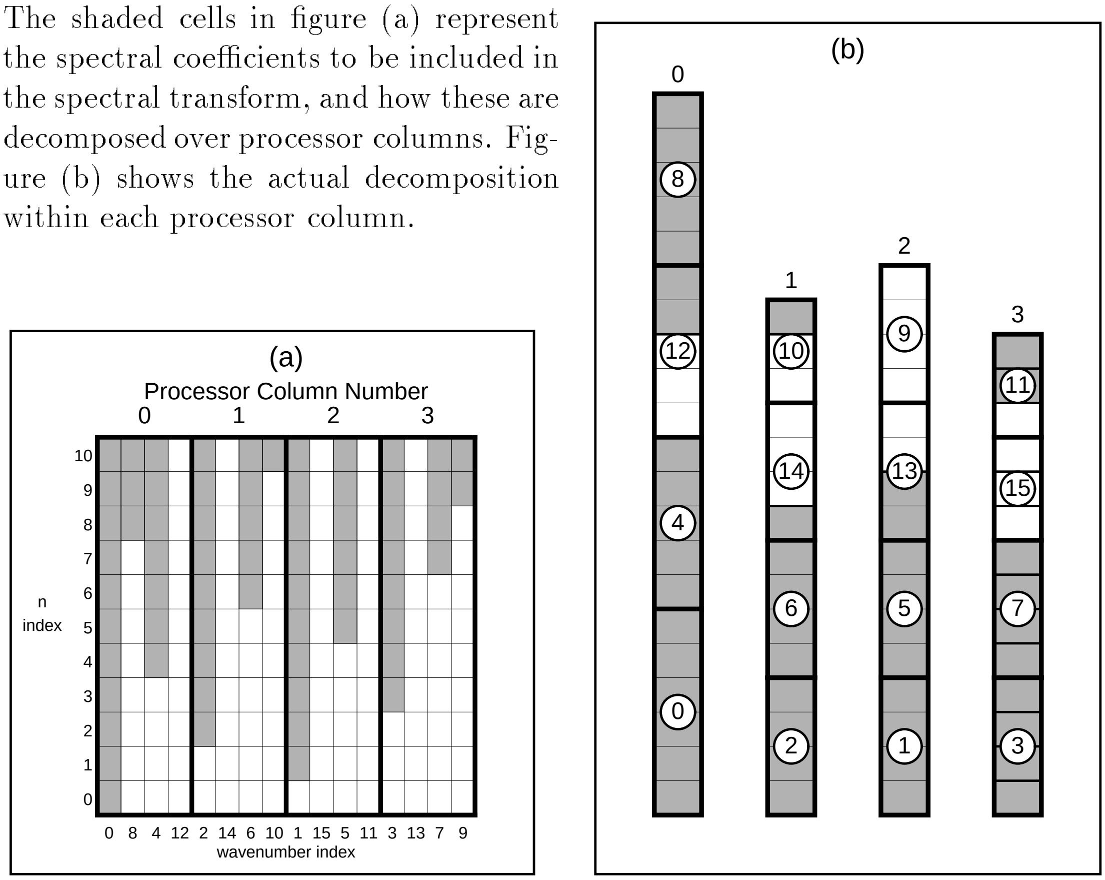
We first consider the decomposition of the physical domain. A linear partitioning function is applied in the longitude direction leaving the ordering of the longitude indices unchanged. In the latitude direction the partitioning function given in Eq. (8) is used in order to pair up corresponding latitude lines in the north and south hemispheres. Since there are 4 processors in each direction, each processor is responsible for a subblock of 8×4 data points. This is shown in Figure 1(a), in which the thicker lines show the divisions between processors, and each small cell represents one data point. The assignment given in Eq. (9) is used to uniquely associate each data subblock with a processor. A linear assignment function is applied in the longitude direction, while in the latitude direction a Gray code assignment function is used. This assignment is shown in Figure 1(a) by the numbered circles, which indicate the number of the processor assigned to each data subblock.

Since we evaluate the Fourier coefficients of *real* functions in the longitude direction, only the coefficients, ξ^m, for $m = 0, 1, \ldots, I/2$, need be explicitly stored. Also, the imaginary parts of ξ^0 and $\xi^{I/2}$ are identically zero, so the real part of $\xi^{I/2}$ can be stored as the imaginary part of ξ^0. The Fourier domain, therefore, is only half the size of the physical domain, as shown in Figure 1(b), with each processor containing a 4×4 subblock of Fourier coefficients. However, since the Fourier coefficients are complex numbers the total storage required for a real function and its Fourier coefficients is the same, and the transform can be done in-place. In performing the FFTs the ordering of the latitude indices is unchanged. However, the order of the wavenumber index is scrambled according to the partitioning function given in Eq. (10). This function permutes the indices into bit-reversed order, and then reorders them so that indices m and $I/2 - m$ are adjacent. Thus, in Figure 1(b), the wavenumber indices are arranged in successive pairs, each of which sums to 16. The first two wavenumber indices, 0 and $I/4$, are exceptions since no communication of data is necessary to extract the corresponding Fourier coefficients. The assignment of data subblocks to processors is as prescribed in Eq. (11), that is, the ith subblock in the wavenumber direction is assigned to column $R_{d_x}(i)$ of the processor grid, and the jth subblock in the latitude direction is assigned to row $G(j)$ of the processor grid.

The decomposition of the spectral domain is illustrated in Figure 2. It should be recalled that the spectral transform is truncated at $m = M$, and that for a particular value of m the index n runs from m to M. In this example $M = 10$, so wavenumber indices 11, 12, 13, 14, 15, and 16 are discarded. This is shown in Figure 2(a) by empty columns. The shaded columns in Figure 2(a) represent the spectral coefficients included in the spectral transforms. Thus, for example, the first column represents the coefficients $\xi_0^0, \xi_1^0, \ldots, \xi_{10}^0$. The decomposition over wavenumber index is the same as in Fourier space, and determines which column of the processor grid ξ_n^m lies in for fixed m.

Figure 2(a) only shows which spectral coefficients are to be included in the decomposition of the spectral domain. The actual decomposition is shown in Figure 2(b). In each column of processors the spectral coefficients to be used are arranged as a single array by running first over the n index, and then over the m index. This is shown, for example, in the first column of Figure 2(b). Here the first 11 shaded boxes represent ξ_n^0 for $n = 0, 1, \ldots, 10$; the next 3 unshaded boxes represent ξ_n^8 for $n = 8, 9, 10$; and the next 7 shaded boxes represent ξ_n^4 for $n = 4, 5, \ldots, 10$. The shading of boxes is chosen to clearly distinguish different m values, and has no other significance. Having produced, for each processor column, a one-dimensional array of spectral coefficients, this array is decomposed among the processors in each column using a linear partitioning function

Figure 2. The decomposition of the spectral domain over a 4×4 grid of processors. The shaded cells in figure (a) represent the spectral coefficients to be included in the spectral transform, and how these are decomposed over processor columns. Figure (b) shows the actual decomposition within each processor column.



and a Gray code assignment function. Thus, in Figure 2(b), processor 0 is assigned the first 6 spectral coefficients in the first column, and processors 4, 12, and 8 each are assigned 5 spectral coefficients. The decomposition of the spectral domain cannot be expressed as the Cartesian product of two one-dimensional decompositions, rather it should be regarded as a set of N_x one-dimensional decompositions, each over N_y processors. Figure 2(b) shows the load imbalance that arises from discarding spectral coefficients with $m > M$. Processor 0 has 6 coefficients, while other processors such as 10 and 11 have only 3 coefficients. The impact of this imbalance will be discussed in Section 5.3.

4. Algorithmic Details.

4.1. The Sequential FFT Algorithm. The sequential, I-point, forward FFT algorithm consists of k steps, where $I = 2^k$. In each step, $I/2$ "butterfly" computations are performed to update the data values in-place. Each butterfly takes two data items whose array indices differ by exactly one bit, and uses them to compute two new values that replace the values of the original two data items. At the end of the k steps the Fourier coefficients are obtained in bit-reversed order, and the computational complexity is $O(I \log I)$. The FFT algorithm, therefore, consists of an outer loop over k steps, and an inner loop over $I/2$ butterfly computations.

The computational complexity can be reduced by a factor of two if we seek the FFT of a real function, rather than that of a complex function. The method followed was that given in *Numerical Recipes* [7], which for completeness we shall now outline. Given the real array, f_j for $j = 0, 1, \ldots, I-1$, to be transformed, we generate a complex array, h, of length $I/2$, the real and imaginary values of which are the even and odd points in f, respectively. Thus,

$$\Re(h_j) = f_{2j}, \qquad \Im(h_j) = f_{2j+1}, \qquad (j = 0, 1, \ldots, I/2 - 1) \tag{13}$$

After performing a complex $I/2$-point FFT on the array h, to give the Fourier transform, H, the transform, F, of the original real function f can be extracted as follows;

$$F_m = \frac{1}{2}\left(H_m + H^*_{I/2-m}\right) - \frac{i}{2}\left(H_m - H^*_{I/2-m}\right) e^{2\pi i m/I} \qquad (m = 0, 1, \ldots, I/2) \tag{14}$$

where x^* denotes the complex conjugate of x, and $i = \sqrt{-1}$. Since $F^*_{I-m} = F_m$, in general we need store only the spectrum for $m = 0$ to $m = I/2$. For the spectral method used in this work the spectrum is truncated at some value M satisfying $I \geq 3M + 1$, so the upper half of the spectrum is not needed in any case. To perform the inverse transform, the complex transform H can be recovered from Eq. (14), and an $I/2$-point inverse FFT is performed on this array, leading directly to the real-valued array, f.

4.2. The Parallel FFT Algorithm. The parallel FFT of 2^k points on a d-dimensional hypercube has a d-step parallel phase, followed by a $(k-d)$-step sequential phase [3,8]. The parallel phase is similar to the first d steps of the sequential algorithm, except that interprocessor communication is required to bring the pairs of data points updated in each butterfly into the same processor. In addition, in both the parallel and sequential phases, the number of "local" butterflies done per step in each processor is $i/2$, where i is the number of data items per processor.

In the early parallel FFT algorithms (for example [4]), communication was performed within the inner loop of the algorithm. Thus, a message was sent in each pass through the inner loop, each time incurring a communication latency cost. For early hypercubes, such as the Caltech/JPL Mark II hypercube, this latency cost was small compared with the total cost of sending a message, and so inner loop communication did not degrade performance too much. However, for modern machines the overhead due to latency for short messages can dominate the communication cost, and this precludes the use of inner loop communication for most algorithms on machines like the Intel iPSC/860 hypercube. We, therefore, move the communication in the parallel phase of the FFT to the outer loop. This reduces the communication latency by a factor of about $i/2$, and also reduces the communication volume by a factor of 2.

In step r of the parallel phase of the algorithm ($r = 0, 1, \ldots, d-1$), processor p swaps half of its data items with half of those in processor q, where q is the number obtained by flipping bit $(d - r - 1)$ of p. Bits are numbered in order of increasing significance, starting from 0. If bit $(d-r-1)$ of processor p is 0 then p swaps its upper $i/2$ data items with the lower $i/2$ data items of processor q; otherwise, p swaps its lower $i/2$ data items with the upper $i/2$ data items of q. This communication of data in the parallel phase can be expressed as a series of d bitwise exchanges in the binary representation of the global index. Swarztrauber [8] refers to these types of bitwise transformation of the global index as i-cycles (see also [6]). The global index consists of k bits. For a linear partitioning function the $(k-d)$ least significant bits give the local index within a particular processor, and if the assignment function is the identity function the d most significant bits give the processor number. In step r of the parallel phase, bit $(k-d-1)$ of the global index, which is the most significant bit of the local index, and bit $(k-r-1)$ are swapped. After the d steps of the parallel phase the original bits of the global index have been permuted as follows;

$$\begin{array}{ccc} \text{Original bit order} & & \text{Bit order after } d \text{ steps} \\ (k-1, k-2, \ldots, 1, 0) & \Rightarrow & (k-d-1, k-1, k-2, \ldots, k-d, k-d-2, \ldots, 1, 0) \end{array}$$

Now one more communication step exchanging bits $(k-1)$ and $(k-d-1)$ results in the following ordering,

$$(k-d, k-1, k-2, \ldots, k-d+1, k-d-1, \ldots, 1, 0)$$

The $(k-d)$ least significant bits are now in their original order, indicating that the ordering of data items within each processor has been left unchanged by the $(d+1)$ communications. The d most significant bits, however, have been cyclically shifted one bit to the right. This means that the net effect of the communication is to place the data that would otherwise be in processor p in processor $R_d(p)$. Thus, the assignment function has changed from the original identity function, $A(i) = i$, to $A(i) = R_d(i)$. These changes occur in addition to the bit reversal produced by the FFT algorithm, which changes the partitioning function from $\Pi(j) = j$ to $\Pi(j) = B_k(j)$.

Having performed the FFT on the complex array, H, it is still necessary to combine the data items at array indices m and $I/2 - m$ for $m = 0, 1, \ldots, I/4$ in order to get the FFT of the original real array (see Eq. (14)). To do this the data to be combined need to be in the same processor, and this requires that the partitioning function be further modified to the form given in Eq. (10).

4.3. Integration over Latitude. Having evaluated the Fourier coefficients, $\xi_n^m(\mu_j)$, along each latitude line, μ_j, the spectral coefficients are found by summing over latitude,

$$(15) \qquad \xi_n^m = \sum_{j=0}^{J-1} \xi^m(\mu_j) P_n^m(\mu_j) w_j = \sum_{i=0}^{N_y-1} \left(\sum_{j \in \mathcal{L}_i} \xi^m(\mu_j) P_n^m(\mu_j) w_j \right) = \sum_{i=0}^{N_y-1} T_n^m(i)$$

where $\mathcal{L}_i$ is the ith index subset in the partitioning over latitude. The partial sums, $T_n^m(i)$, can be evaluated within each processor with no communication. Thus, the evaluation of the spectral coefficients, ξ_n^m, requires the summation of N_y partial sums. This summation is performed independently in each column of the processor grid using a ring algorithm, and is described in detail in [11]. The ring algorithm proceeds in

$N_y - 1$ steps. Initially each processor evaluates the partial sums, T_n^m, for the spectral coefficients assigned to its neighbor in the decomposition of the spectral domain. In the first step of the ring algorithm each processor passes these partial sums one step around the ring. Each processor receives a set of partial sums, and evaluates its contribution to each. The contributions are then added to the partial sums. After $N_y - 1$ such steps all contributions have been summed, and the decomposition of the spectral coefficients is as described in Sections 3.3 and 3.4.

5. Results and Discussion. We shall first describe the incremental steps taken in developing the concurrent code for solving the shallow water equations, and then results for the optimum implementation will be presented and discussed. In all cases the code was compiled with release 1.1 of The Portland Group i860 Fortran compiler, with the default optimization level of 1.

The process of implementing the shallow water equation code on the 128-node Intel iPSC/860 hypercube began with a sequential version suitable for executing on a Unix workstation. In the first phase of the concurrent implementation the data were decomposed only over the y direction, which corresponds to latitude in the physical and Fourier domains and to spectral coefficient index in the spectral domain. The data were not decomposed over the x direction (the latitude/wavenumber direction). This allowed the original FFT routines in the sequential code to be used, and effort was focused on optimizing the parallel summation over latitudes in Eq. (2). In the second phase of the implementation, decomposition over both the y and x directions was investigated.

5.1. Parallel Summation. Results for the parallel summation in Section 4.3 have been presented and discussed at length in [11], so we shall just summarize these results here. In [11] the problem domains were partitioned in only the y direction, so $N_x = 1$. In the physical and Fourier domains each processor is responsible for a set of latitude lines, as in Eq. (8). In the first attempt at parallelizing the code the spectral coefficients were duplicated in all processors. Each processor evaluated its contribution to each spectral coefficient, and these contributions were then summed over all processors. In this case the inverse transform could be found locally since each processor holds all the spectral coefficients. This approach is simple to implement, requiring few changes to the original sequential code. However, it was found to be unacceptable due to high communication costs, and the duplication of computation in the spectral domain. The duplication of spectral coefficients also wastes memory.

The second approach also partitioned the spectral coefficients, as described in Section 3. This eliminated most of the redundant computation, and also improved the concurrent efficiency of the summation phase of the spectral transform. This summation was performed using a ring pipeline, as described in Section 5.4, and the application ran efficiently for problem sizes of interest on the 128-node Intel iPSC/860 hypercube. For the T340 problem running on 128 nodes a speed of 340 Mflops was achieved.

5.2. Parallel FFTs. Decomposing the data over just the y direction limits the number of processors that can be brought to bear on the solution of the shallow water equations, and hence the extent to which the problem's inherent parallelism can be exploited. In order to make effective use of the computational power of massively parallel MIMD computers, containing hundreds or thousands of processors, it is necessary to decompose data domains in the x direction as well as the y direction. In the shallow water equation code Fourier transforms are performed in the longitude direction, and so decomposition in this direction requires the development of a parallel FFT algorithm.

The basic structure of the parallel FFT algorithm has already been described in Section 4.2, and will be referred to as version 1. However, some tuning was required to get acceptable performance. The tuning concentrated on the following three areas:

1. replacing the evaluation of the complex exponentials in the butterfly calculations by a lookup table,
2. reducing communication latency by reducing the number of messages sent, and
3. masking communication costs by overlapping communication and calculation.

We refer to the code produced by replacing the evaluation of the complex exponentials by a lookup table as version 2. As shown in Table 1, version 2 runs significantly faster than version 1. For the T85 case with $N_x \times N_y = 1 \times 32$ (i.e., for sequential FFTs) the performance improved by a factor of 4. It is important to note that although the concurrent efficiency of version 1 is larger than that of version 2, its performance is poorer – judging an algorithm on the basis of efficiencies can be misleading. The efficiency is useful in gauging the scalability of an algorithm, but by itself cannot be used to determine the best algorithm on a given concurrent machine. The memory requirements for the lookup tables for the forward FFT are $\mathrm{O}(i)$, while for the inverse FFT they are $\mathrm{O}(i \log N_x)$, where i is the length of the FFT, I, divided by the number of processors, N_x. If only a single FFT needs to be computed the lookup tables may consume a relatively large amount of memory. However, in the shallow water equation code each processor, in general, evaluates several FFTs, thereby amortizing the memory cost.

The next step in tuning the parallel algorithm involved attempting to modify version 2 to reduce communication latency, and to overlap communication and calculation. As noted by Walker [9], if the communication is performed in the outer loop the butterfly calculations in a single FFT cannot be overlapped with communication. The evaluation of the complex exponentials *can* be overlapped with communication, but since we are using a lookup table this is not an option. We could put the communication back inside the inner loop, which does permit the butterfly calculation to be overlapped with communication. This may be a good approach on some machines, however, for the Intel iPSC/860 any gains from overlapping communication and calculation would be swamped by the higher latency overhead.

Fortunately the shallow water equation code contains another outer loop that we have so far ignored. Namely, the loop over the latitudes in each processor. By making the loop over latitude the inner loop (rather than the outer loop) we can perform the communication necessary at a given step of the FFT for all latitudes at the same time. This approach requires data to be packed into and unpacked out of communication buffers, and the communication buffers require memory of order the size of the original data, however the communication latency is reduced by a further factor of n_{lat}, the number of latitude lines per processor. Essentially, changing the order of the loops allows us to push the communication to an outer level of the loop hierarchy, and that is why latency is reduced.

Since the FFTs over different latitude lines are independent we can now also overlap communication and calculation. This is done by dividing the latitude lines in each processor into two equally-sized sets. While the calculations for the first set are being performed the communication for the second set is done, and *vice versa*. We refer to the code that incorporates these modifications as version 3.

In Table 1, we present results for the T85 case for versions 1, 2 and 3 on up to 128 nodes of the Intel iPSC/860 hypercube. Version 3, in which latency is lowest and communication is overlapped with computation, is clearly the best algorithm. In version 2, decomposing over the x-direction for a fixed number of processors always results in poorer performance. However, in version 3 the performance at first improves as N_x increases, and then falls off when it is increased further, as shown in Figure 3. This behavior arises because as N_x increases the time to do the FFTs also increases due to the higher concurrent overhead. However, for a fixed number of processors an increase in N_x reduces N_y, so the concurrent summation described in Section 4.3 will be performed more efficiently. An increase in N_x also increases the load imbalance in the summation phase. The net effect of these conflicting trends is to produce a shallow minimum in the plot of processing time versus N_x. However, if the grain size in both directions is large enough, then both the FFT and summation phases will be computed efficiently, and load imbalance will result in a monotonic rise in the processing time as N_x increases, as may be seen in some of the entries in Table 2 (for example, T85 on up to 16 processors).

Having determined that version 3 gives reasonably good performance, we then went on to use version 3 for problem sizes T21, T42, T85, T169, and T340, for processor grids with different sizes and shapes. The results are presented in Table 2.

Figure 4 illustrates how the shape of the processor mesh affects performance. In Figure 4 the time for 10 time steps of the T85 case is plotted as a function of the total number of processors for a purely latitudinal decomposition ($N_x = 1$), a purely longitudinal decomposition ($N_y = 1$), and for the mixed decomposition that gives the best performance. Both Figures 3 and 4 show how a mixed decomposition results in better performance, particularly for smaller grain sizes. Figure 4 also shows that for a mixed decomposition 128 processors can be used, whereas for a purely longitudinal (latitudinal) decomposition only up to 64 processors can be used due to an insufficient number of longitude (latitude) points. Thus, the mixed decomposition allows more parallelism to be exploited.

In Figure 5 the parallel efficiency of the shallow water equation code is shown for a number of problem sizes as a function of the number of processors. The results shown are for the best mixed decomposition case, except in the T340 case where memory constraints precluded a more complete investigation. Figure 5 shows that for the larger problem sizes of interest (T85, T169, T340) good parallel efficiency is achieved. The results indicate that for the T169 and T340 cases high efficiency would be achieved on

T85 timings for 10 time steps (seconds)			
$N_x \times N_y$	Version 1	Version 2	Version 3
1×32	7.98	2.05	2.02
2×16	8.14	2.39	1.82
4×8	9.70	4.21	1.78
8×4	8.88	3.55	1.85
16×2	12.18	6.39	2.05
32×1	20.12	12.69	2.32
1×64	4.43	1.43	1.59
2×32	4.29	1.43	1.30
4×16	4.96	2.22	1.18
8×8	4.52	1.84	1.17
16×4	6.15	3.23	1.25
32×2	10.09	6.33	1.38
64×1	19.01	13.32	1.70
1×128	Too few latitudes		
2×64	2.50	1.05	1.03
4×32	2.66	1.29	0.88
8×16	2.35	1.02	0.80
16×8	3.12	1.67	0.81
32×4	5.08	3.20	0.87
64×2	9.53	6.68	1.04
128×1	Too few longitudes		

Table 1. Timings for the T85 case on 32, 64, and 128 nodes of the Intel iPSC/860 hypercube. The three versions are discussed in the text.

Version 3 timings for 10 time steps (seconds)					
$N_x \times N_y$	T21	T42	T85	T169	T340
1×1	1.13	6.21	40.83	NEM*	NEM
1×2	0.61	3.06	20.43	NEM	NEM
2×1	0.81	3.72	22.55	NEM	NEM
1×4	0.36	1.62	10.41	NEM	NEM
2×2	0.48	1.91	11.42	NEM	NEM
4×1	0.61	2.12	12.10	NEM	NEM
1×8	0.24	0.94	5.43	30.39	NEM
2×4	0.33	1.05	5.86	31.25	NEM
4×2	0.44	1.17	6.19	NEM	NEM
8×1	0.51	1.34	6.66	NEM	NEM
1×16	0.21	0.61	3.10	16.15	NEM
2×8	0.28	0.64	3.14	15.94	NEM
4×4	0.37	0.71	3.25	16.42	NEM
8×2	0.42	0.80	3.48	NEM	NEM
16×1	0.49	0.92	3.90	NEM	NEM
1×32	LAT†	0.55	2.02	9.14	NEM
2×16	0.22	0.52	1.82	8.75	NEM
4×8	0.24	0.57	1.78	8.66	NEM
8×4	0.26	0.61	1.85	8.78	NEM
16×2	0.30	0.69	2.05	NEM	NEM
32×1	LON‡	0.82	2.32	NEM	NEM
1×64	LAT	LAT	1.59	5.95	31.22
2×32	LAT	0.45	1.30	5.02	NEM
4×16	0.25	0.45	1.18	4.70	NEM
8×8	0.25	0.46	1.17	4.68	NEM
16×4	0.27	0.50	1.25	4.92	NEM
32×2	LON	0.58	1.38	NEM	NEM
64×1	LON	LON	1.70	NEM	NEM
1×128	LAT	LAT	LAT	4.52	20.20
2×64	LAT	LAT	1.03	3.33	16.20
4×32	LAT	0.38	0.88	2.79	NEM
8×16	0.27	0.33	0.80	2.62	NEM
16×8	0.27	0.33	0.81	2.68	NEM
32×4	LON	0.36	0.87	2.84	NEM
64×2	LON	LON	1.04	NEM	NEM
128×1	LON	LON	LON	NEM	NEM

†Not enough latitudes. ‡Not enough longitudes. *Not Enough Memory

Table 2. Timings for the version 3 code on up to 128 nodes of the Intel iPSC/860 hypercube for a range of problem sizes.

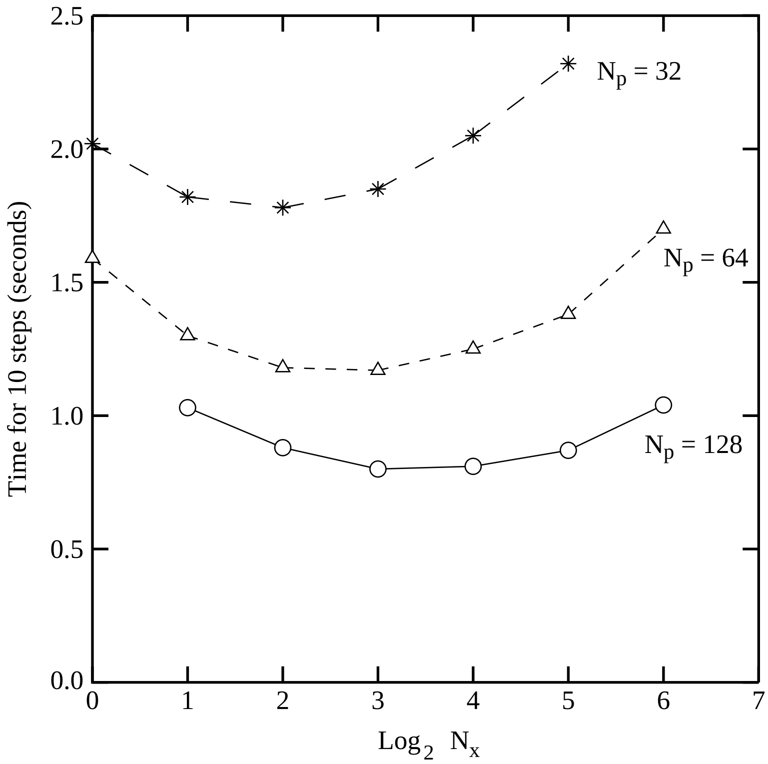


Figure 3. Performance of the T85 case as a function of the shape of the processor mesh.

larger machines, provided the characteristics of the communication system were at least as good as those of the machine used here. In particular, we expect the T169 and T340 cases to run efficiently on the 528-processor Intel Delta system.

5.3. Load Imbalance. The maximum number of spectral coefficients assigned to any column of processors is given by

$$L_{max} = (R+1)\left(M+1-\frac{N_x R}{2}\right) \tag{16}$$

where $R = \lfloor (M+1)/N_x \rfloor$. Since the same amount of work is done on each spectral coefficient, the load imbalance, ℓ_s, in the spectral domain (defined as the maximum excess computational load in any processor divided by the average load) is

$$\ell_s = \frac{2N_p \lceil L_{max}/N_y \rceil}{(M+1)(M+2)} - 1 \tag{17}$$

where $N_p = N_x N_y$ is the total number of processors. In Figure 6 we plot the load imbalance, ℓ_s, as a function of the number of processors for a range of problem sizes for the case $N_y = 1$. For the $N_y = 1$ case, the load imbalance is maximized for a given number of processors. As may be deduced from Figure 6, the load imbalance is less than 20% provided there are at least 16 longitude points per processor. Of course, the overall impact of this load imbalance on the performance of the application depends on

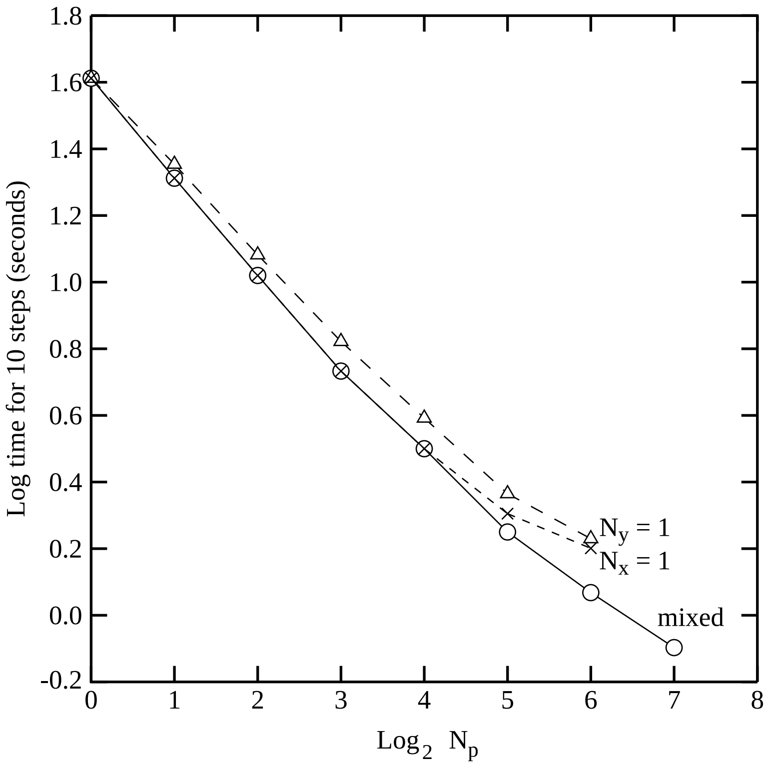


Figure 4. Performance of the T85 case as a function of the number of processors for pure latitudinal and longitudinal decompositions, and for the best mixed decomposition.

the relative amounts of computation performed in the physical, Fourier, and spectral domains.

5.4. Masking Communication Overhead. Our results have shown the importance of reducing communication overhead on concurrent multiprocessors, such as the Intel iPSC/860 hypercube, by overlapping communication and calculation. We, therefore, devote this subsection to a more general discussion of how communication and calculation can be overlapped in loosely synchronous parallel algorithms. The use of these techniques is illustrated with examples from our parallel implementation of the shallow water equations code, SSWMSB.

Loosely synchronous algorithms are characterized by a series of compute-communicate cycles in which the communication phase imposes a degree of synchronization on the processors [3]. In the computational phase between communications the processors can run completely asynchronously. Each cycle can be labeled by a global counter. Typically a cycle begins with each processor receiving data from one or more processors. The processors then independently perform some computation using the data received, and send the results to some set of processors for use in the next cycle. We shall refer to the work done by a single processor in a cycle as a *subtask*, and the process of transforming some initial data through a series of subtasks to produce some desired output will be referred to as a *task*. In general the compute phase of a subtask can be divided

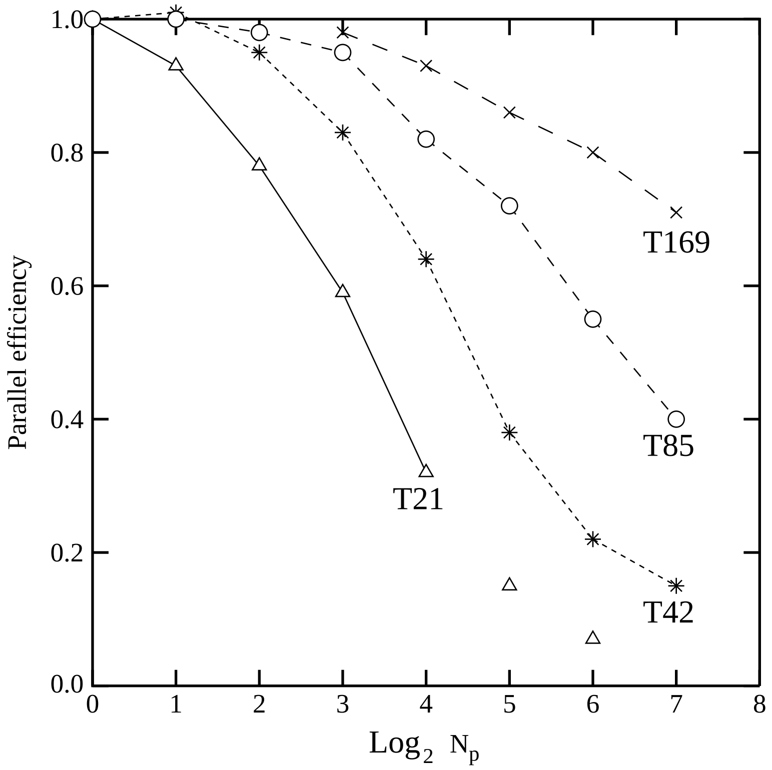


Figure 5. Parallel efficiency of the shallow water equation code for the best mixed decomposition as a function of number of processors for a number of problem resolutions. In the T21 case grain size constraints did not permit the best decomposition to be unambiguously determined for runs on 32 and 64 processors. In the T169 case memory constraints prevented a timing run on one node being done. The parallel efficiencies in these cases are, therefore, derived from an estimated one-processor runtime based on an empirical model.

into a *critical phase* in which the computation depends on data from the preceding subtask(s), and a *non-critical phase* that is independent of the preceding subtask(s). The effective use of concurrent computers characterized by high communication overhead, such as the Intel iPSC/860 hypercube used in this work, requires communication costs to be masked by overlapping communication and computation. Two approaches used in this work are to overlap the communication phases of a task with (1) the computation phases of another task, and (2) the non-critical computation phases of the same task.

A pipeline can be used to perform a set of independent tasks, the number of subtasks in each of which equals the number of processors. In a *linear pipeline* the processors are arranged in a line, and the ith subtask for each task is assigned to the processor at position i in the line. Each task is initiated in the processor at the beginning of the line, and the results are accumulated in the last processor, as shown in Figure 7(a). If all the subtasks take approximately the same length of time the linear pipeline can provide an efficient means of exploiting parallelism, particularly when the inner loop(s) of the

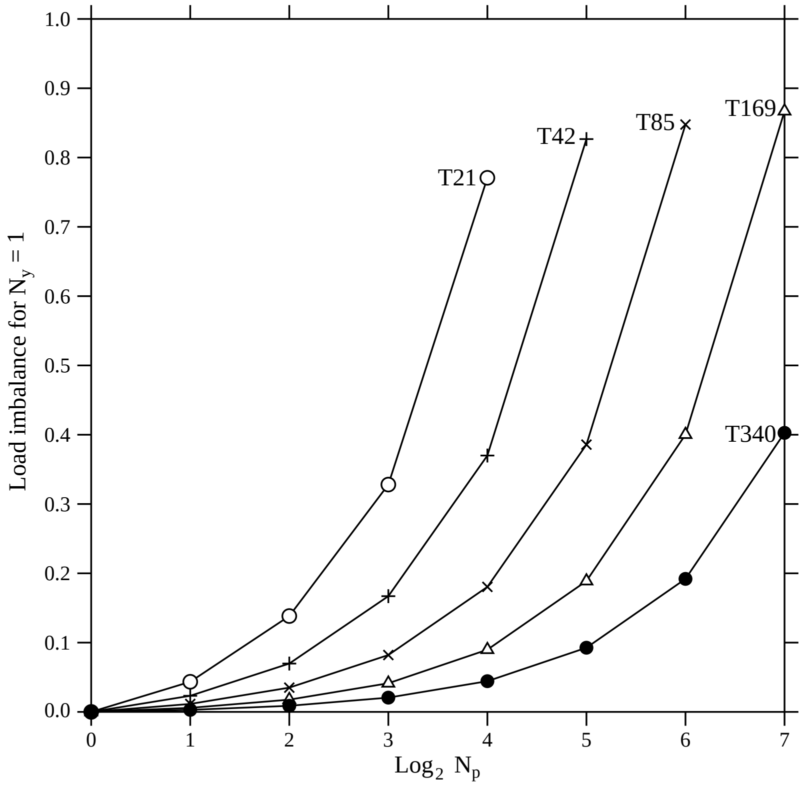


Figure 6. Load imbalance, ℓ_s, as a function of number of processors, N_p, when $N_y = 1$.

algorithm contain little inherent parallelism. This approach has been used, for example, in the solution of multiple tridiagonal systems arising in a plasma instability problem [2]. In many problems it is desirable to have the final results distributed evenly across the processors. This can be achieved by using a *ring pipeline*, in which each processor in turn initializes a task. As shown in Figure 7(b), the tasks now terminate in different processors. An advantage of the ring pipeline over the linear pipeline is that, if the number of tasks is exactly divisible by the number of processors, there is no pipeline startup cost – all processors are kept busy. The advantage of the linear pipeline is that the computation of one task can be performed while receiving data for the next task. In the ring pipeline only the non-critical computation of a task can be performed while receiving data needed for the critical computation phase of the same task. Thus, in general, a larger fraction of the computation is available for overlap in the linear pipeline.

Whereas the tasks in a pipeline algorithm are independent, in a dimensional exchange algorithm they overlap, and a subtask sends data to subtasks in different tasks. Each task can be represented as a binary tree, with each node being a subtask. The dimensional exchange algorithm overlaps these trees, as shown in Figure 7(c). On a hypercube multiprocessor a dimensional exchange algorithm involves exchanging data over each communication channel in turn. The parallel phase of a FFT is an example of a dimensional exchange algorithm.

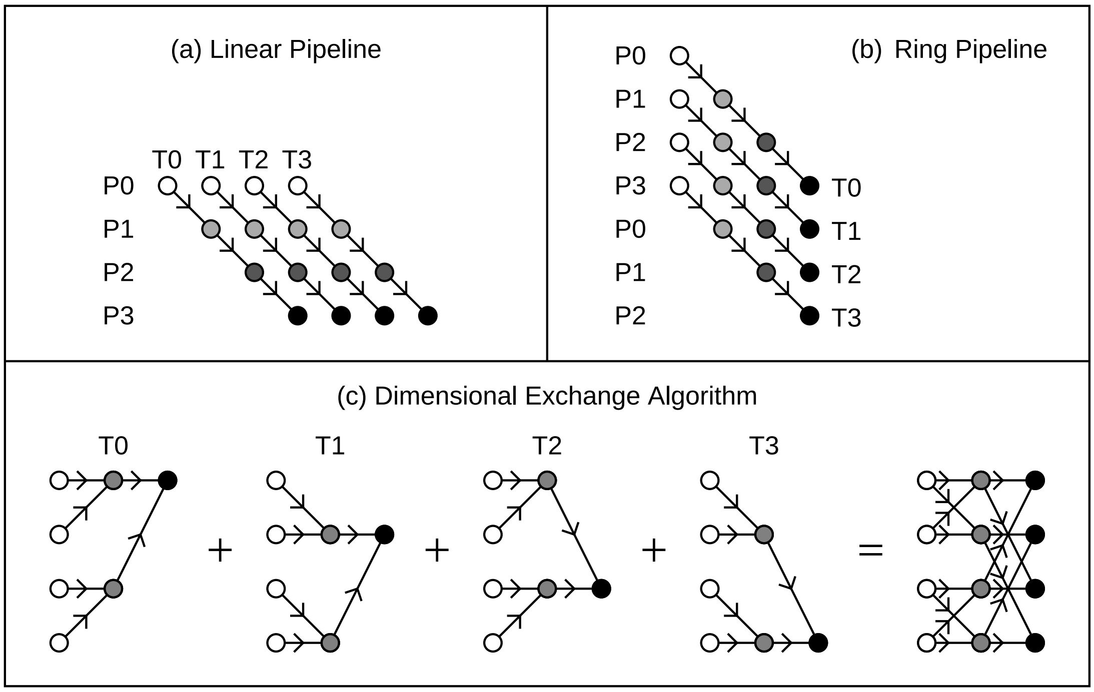


Figure 7. Schematic representations of (a) a linear pipeline, (b) a ring pipeline, and (c) a dimensional exchange algorithm. In all cases 4 tasks are shown, and the shaded circles represent subtasks. The degree of shading indicates how much of the task has been completed at a given stage – a white circle designates a subtask in the first cycle of a task, and a black circle designates the end of the task. In (a) and (b), Pn stands for the nth position in the pipeline. In (c), we show how the dimensional exchange algorithm is made up of overlapping binary trees.

In summing the contributions to the Legendre transform the non-critical phase is the evaluation of the local contributions to a spectral coefficient, referred to as T_n^m in Eq. (15). The critical phase is simply the summation of the local contribution with the running sum received from the preceding subtask. In the FFT algorithm the critical phase is the evaluation of butterfly pairs, and the non-critical phase is the determination of the complex exponential in the "twiddle factor". The ratio of the time spent communicating between two subtasks and the time for a non-critical computation determines the extent to which communication and calculation can be overlapped. In the evaluation of the spectral coefficients in Eq. (15) the time for the non-critical phase is proportional to the number of latitudes per processor, and hence the amount of overlap (and the concurrent efficiency) increases as the grain size increases in the latitude direction. In the FFT algorithm a lookup table is used to find the twiddle factors, resulting in a short non-critical phase. Thus, there is little overlap of communication and computation within a single FFT. If several FFTs need to be evaluated, as is the case in the shallow water equation code, communication and calculation can be overlapped. Taking the FFTs in pairs, the calculation in one step of one FFT can be overlapped with the communication in the other FFT, and *vice versa*. Thus, the communication and calculation phases of the two FFTs are interleaved, and we refer to this technique as *task interleaving*.

Communication latency also often significantly degrades concurrent performance, and should be minimized by sending as few messages as possible. This can be done, whenever possible, by exchanging the order of the loops over tasks and subtasks. Thus, if a code originally contains an outer loop over tasks and an inner loop over subtasks, it should be restructured so that the subtask loop is the outer loop and the task loop is the inner loop. This reduces latency by moving communication from the inner loop to the outer loop. For example, in performing the summation in the Legendre transform a single task is to find the spectral coefficient, ξ_n^m, for some m and n. The subtasks correspond to the steps in the pipeline. Thus, latency is reduced if the inner loop is over spectral coefficients, and the outer loop is over the steps in the pipeline. In this case, in each step of the pipeline, the running sum for a block of spectral coefficients is updated, rather than just for a single spectral coefficient, and blocks of coefficients are communicated. Similarly, in the evaluation of the FFTs, if a task is the evaluation of a single FFT, and a subtask involves the computation of one set of butterfly evaluations in a processor, then latency can be reduced by making the loop over FFTs (i.e., latitudes) the inner loop. Now when we perform the interleaving, instead of taking single FFTs in pairs, we interleave two blocks of FFTs each containing half the number of latitude lines per processor.

It should be noted that a dimensional exchange algorithm could also be used to perform the summation in the Legendre transform. This approach uses a version of the *fold* algorithm of Fox et al. [3]. The communication volume is the same in the *fold* algorithm and the corresponding pipeline algorithm. However, *fold* performs fewer communication steps and hence incurs a lower latency cost. On the other hand, in the *fold* algorithm less of the non-critical computation is available for overlap with communication since half of it must be done before the first communication phase. The optimum method for performing the summation is, therefore, machine-dependent, and further work is required to determine the best method on the Intel iPSC/860 hypercube, and similar machines. These issues will be pursued further in a subsequent paper.

6. Summary and Conclusions. In the climate modeling community problem sizes of interest range from T85 to T340, corresponding to grid resolutions from about 1.5 degrees to less than half a degree. For these types of problem the spectral method can be parallelized efficiently on MIMD distributed memory computers with hundreds of processors. The Intel iPSC/860 hypercube used in this work only had 8 Mbytes of memory on each processor, and this prevented a thorough investigation of the T340 case. Of the T340 runs that were performed, a 2×64 processor mesh gave the highest performance of approximately 560 Mflops. If more memory had been available we expect the performance would have been greater for less elongated processor meshes. This expectation is based on the results for T85 and T169 cases running on 128 nodes.

It was found that in all cases of interest parallel performance is significantly improved by decomposing over both coordinate directions, rather than over just one or the other. Using a mixed decomposition resulted in performance improvements of up to 42%. In addition, a mixed decomposition allowed more processors to be brought to bear on a given problem.

The Intel iPSC/860 hypercube, and similar multiprocessors, have high communication latency and throughput costs, and acceptable levels of performance are achievable only if specialized programming techniques are used. In this work, we have emphasized the importance of reducing latency by moving communication to the outermost loop possible. Another important factor is the need to overlap communication and computation. This can be done by identifying the non-critical part of each phase of computation, and overlapping this with communication. The communication must be performed using non-blocking reads and writes. Some additional buffers are required to maintain data integrity, but we have found the cost of this extra memory to be small in comparison with the benefits gained.

In the FFT algorithm the time for the non-critical computation is very short compared with the communication time, so there is no opportunity to overlap communication and computation in a single FFT. To achieve overlap we have introduced the concept of task interleaving. By alternating the computation and communication phases of a pair of independent tasks the critical computation of one task can be overlapped with the communication in the other, and *vice versa*.

If no attempt is made to reduce latency and overlap communication and computation, many of the distributed memory multiprocessors currently available are only capable of running efficiently on embarrassingly parallel problems. The techniques that have been used in this work to reduce communication costs demonstrate that it is possible to use this type of multiprocessor to effectively exploit parallelism in a much larger class of applications.

We intend to incorporate what we have learned from parallelizing the shallow water equations code into the design of a parallel version of CCM2. This will require addition issues to be addressed. In particular, in CCM2 a semi-Lagrangian method will be applied in the physical domain. This will result in load imbalance since the polar and equatorial regions must be processed in different ways, and suggests that our method of decomposing the problem domains may need to be modified. The load imbalance in the radiative calculation must also be considered in developing an efficient parallel code. See [11] for a more thorough discussion of how this work can be applied when parallelizing CCM2.

REFERENCES

[1] G. L. Browning, J. J. Hack, and P. N. Swarztrauber, *A comparison of three numerical methods for solving differential equations on the sphere*, Monthly Weather Review, 117 (1989), pp. 1058–75.

[2] B. A. Carreras, N. Dominguez, J. B. Drake, J. N. Leboeuf, L. A. Charlton, J. A. Holmes, D. K. Lee, V. E. Lynch, and L. Garcia, *Plasma turbulence calculations on supercomputers*, Int. J. Supercomputer Applications, 4 (1990), pp. 97–110.

[3] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving Problems on Concurrent Processors*, vol. 1, Prentice Hall, Englewood Cliffs, N.J., 1988.

[4] P. D. Noerdlinger and D. W. Walker, *Discrete Fourier transforms on the Mark II hypercube*, Tech. Report 337, Caltech Concurrent Computation Project, 1986.

[5] S. A. Orszag, *Transform method for calculation of vector-coupled sums: application to the spectral form of the vorticity equation*, J. Atmos. Sci., 27 (1970), pp. 890–895.

[6] R. B. Pelz, *Parallel compact FFTs for real sequences*, SIAM J. Sci. Stat. Comput., (submitted 1991).

[7] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical recipes: the art of scientific computing*, Cambridge University Press, Cambridge, England, 1986.

[8] P. N. Swartztrauber, *Multiprocessor FFTs*, Parallel Comp., 5 (1987), pp. 197–210.

[9] D. W. Walker, *Portable programming within a message-passing model: the FFT as an example*, in The third conference on hypercube concurrent computers and applications, G. C. Fox, ed., ACM Press, 1988, pp. 1438–50.

[10] D. L. Williamson, J. T. Kiehl, V. Ramanathan, R. E. Dickinson, and J. J. Hack, *Description of NCAR community climate model (CCM1)*, Tech. Report 285, National Center for Atmospheric Research, June 1987.

[11] P. H. Worley and J. B. Drake, *Parallelizing the spectral transform method*, Concurrency: Practice and Experience, 4 (1992), pp. 269–291.