

# Performance Analysis and Optimization on the UCLA Parallel Atmospheric General Circulation Model Code

John Kim, Robert Ferraro  
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91009

John Farrara, Carlos Mechoso  
Department of Atmospheric Sciences, University of California, Los Angeles, CA 90024

## Abstract

An analysis is presented of several factors influencing the performance of a parallel implementation of the UCLA atmospheric general circulation model (AGCM) on massively parallel computer systems. Several modifications to the original parallel AGCM code aimed at improving its numerical efficiency, interprocessor communication cost, load-balance and issues affecting single-node code performance are discussed. The impact of some of the optimization strategies on the performance of the AGCM code as we implemented on several state-of-the-art parallel computers, including the Intel Paragon and Cray T3D, is presented and analyzed.

## 1.1 Introduction

The climate system is characterized by complex interactions and feedbacks among its components. General circulation models (GCMs) of the atmosphere and ocean are among the most powerful tools available for studies of the climate system and its variability. Numerical simulations performed using GCMs are among the most computationally expensive scientific applications because several three dimensional physical fields need to be updated at each time step by solving a large system of partial differential equations governing fluid motion on a rotating sphere, and also because a very long simulation period is required to produce statistically significant numerical results. Parallel computers are thus natural computing tools for GCM simulations.

An atmospheric GCM model was developed at UCLA by Arakawa and co-workers [1] during the seventies and the model is still being constantly upgraded by atmospheric scientists there. The first parallel implementation of the UCLA AGCM model was developed as a collaborative effort between Lawrence Livermore National Laboratory and the University of California, Los Angeles. The results presented in the paper by Wehner et al. [2] revealed that the parallel efficiency of the code on large numbers of processors (> 100) is mediocre. In other words, the code does not "scale" well to a large number of processors. Here scalability refers to the reduction of execution time as more processors are used for a fixed problem size. The main objective of our work is to analyze the AGCM algorithm components and its parallel implementation from a computational performance perspective, find bottlenecks that hinder the parallel scalability of the code, and use better algorithms and efficient parallel implementation strategies wherever possible to maximize the performance of the AGCM code on scalable parallel systems.

This paper is organized as follows: Section 2 gives a brief overview of the structure of the parallel UCLA AGCM code and an analysis of its parallel performance, Section 3 discusses our optimization strategies on the code to improve its performance on massively parallel systems, Sec-

tion 4 presents a performance comparison between the modified parallel code and the original one, Section 5 offers some of our thoughts on developing reusable template modules for GCM simulations, and finally Section 6 gives our conclusion.

## 2. Structure and performance of the parallel AGCM code

The UCLA AGCM code is a large software package which simulates many physical processes. The reader is referred to Suarez et al. [3] and references therein for a more complete description of the representations of the physical processes. There are two major components of the code: i) AGCM/hydrodynamics, which computes the evolution of the fluid flow governed by the primitive equations by means of finite-differences, and ii) AGCM/Physics, which computes the effect of processes not resolved by the model's grid (such as convection on cloud scales) on processes that are resolved by the grid. The results obtained by AGCM/Physics are supplied to AGCM/Hydrodynamics as drivers for flow simulations. The AGCM code uses a three dimensional staggered grid for velocity and thermodynamic variables (potential temperature, pressure, specific humidity, ozone, etc.). This three dimensional grid is formed by the Arakawa C-mesh [1] in the horizontal (latitude/longitude) directions with relatively small number of vertical layers (usually much fewer than the horizontal grid points). A cell in such a grid is a cube in spherical geometry with velocity components centered on each of the faces and the thermodynamics variables at the cell center. The AGCM/Hydrodynamics itself consists of two main components: a spectral filtering part and the actual finite difference calculations. The filtering operation is needed at each time step in regions close to the poles to ensure the effective grid size there satisfies the Courant-Friedrich-Lewy (CFL) condition [4], a stability requirement for explicit time-difference schemes when a fixed time step is used throughout the whole spherical finite-difference grid.

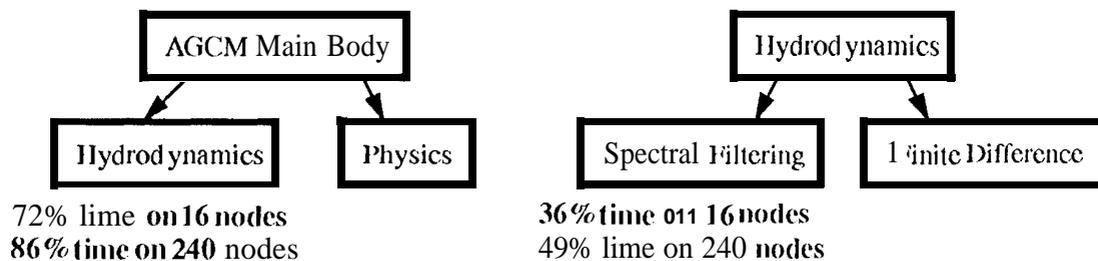


Figure 1. Execution times of major components in the UCLA AGCM code

A two-dimensional grid partition in the horizontal plane is used in the parallel implementation of the UCLA AGCM model. This choice of grid partition is based on the facts that column (vertical) processes strongly couple the grid points which makes the parallelization less efficient in the column direction, and that the number of grid points in the vertical direction is usually small. Each subdomain in such a grid is a rectangular region which contains all grid points in the vertical direction. With this grid partition, there are basically two types of interprocessor communications involved in the parallel AGCM simulation. Message exchanges are needed among (logically) neighboring processors (nodes) in finite-difference calculations; non-nearest neighbor message-passing is needed for implementing the spectral filtering operations. Timing measurements on the main components of the original parallel AGCM code, using the  $2 \times 2.5 \times 9$  (lat x long x vertical) resolu-

tion which corresponds to a 144 x 90 x 9 grid, is shown in Figure 1.

As shown in Figure 1, the AGCM main body consists of a hydrodynamics module and a physics module, with preprocessing and postprocessing parts excluded. Since preprocessing and postprocessing steps are only performed once, whereas the main body part is iterated through a time stepping loop in the AGCM simulation, the latter is absolutely dominant in terms of execution time. Comparing the two modules in the main body, we can see the hydrodynamics part is dominant in cost especially on large numbers of nodes. Furthermore, our timing analysis on the hydrodynamics part indicates that the spectral filtering is a very costly component with poor scalability to large number of nodes (see Figure 1). Although the use of spectral filtering in the UCLA AGCM model improves the computational efficiency of the finite-difference calculations by enabling the use of uniformly larger time steps, the high cost of performing the filtering seems to offset a large part of this performance gain. The inferior performance of the filtering operation is due to the use of an inefficient filtering algorithm and the existence of a severe load imbalance in the filtering stage.

It is clear from Figure 1 that in order to substantially improve the overall performance of the AGCM code, some optimization must be done first on the filtering part of the code [2].

### 3. Optimization strategies and implementations in the parallel AGCM code

There are primary two ways to improve the performance of a parallel code running on a distributed memory message-passing computer. One way is to optimize its single-node performance by using a more efficient computational algorithm, making more efficient use of cache or eliminating redundant operations in the code. Single node optimizations can usually be achieved by restructuring the data structures and other computational parts of the code. Another way is to improve its scalability to large numbers of processors so that one can either reduce the solution time for a large problem by using more processors, or can solve increasingly larger problems with more processors within a fixed amount of time. The scalability of a parallel code is affected both by the ratio of communication cost to computation cost and the degree of load imbalance in the code. As stated above, our timing results indicate the cost of spectral filtering procedure is a dominant part in the parallel AGCM code especially when running on a large number of nodes. We therefore focused our first effort on improving the overall performance of the filtering part in the AGCM code.

#### 3.1) Spectral filtering in the UCLA AGCM model

The filtering algorithm used in the UCLA AGCM model is basically a set of discrete Fourier filters specifically designed to damp fast-moving gravity-inertial waves near the poles. These wave modes become numerically unstable when the CFL condition is violated in the vicinity of the poles as a result of the increasingly smaller zonal grid distances as one approaches the poles in a uniform longitude-latitude grid. The filters contain a latitudinal dependence but are applied over the complete longitudinal domain on every vertical layer. As discussed in [1], the filtering operation takes the form of an inverse Fourier transform in wavenumber space as

$$\phi'(i) = \frac{1}{(M+1)} \sum_{s=1}^M \hat{\phi}(s) \hat{S}(s) e^{i\lambda_r s} \quad (1)$$

where  $\hat{\phi}(s)$  is the Fourier transform of a generic variable  $\phi(s)$  to be filtered,  $\hat{S}(s)$  is a prescribed function of wavenumber and latitude, but is independent of time and height. In particular, two types of filtering are performed in the UCLA AGCM code. One is the so called "strong filtering" which is applied to about one half of the latitudes around the poles in each hemisphere; the other is a "weak filtering" which is applied to about one third of the latitudes around the poles in each hemisphere. The convolution theorem for Fourier transforms states that the filtering as defined in (1) is mathematically equivalent to the convolution

$$\phi'(i) = \sum_{s=1}^M S(s)\phi(i-s). \quad (2)$$

In the original AGCM code, filtering was performed using the convolution form in (2). In its parallel implementation, the summation defined in (2) was implemented in several ways, involving either communications around "processor rings" in the latitudinal direction, or communications in binary trees [2]. Letting  $N$  denote the number of grid points and  $P$  the number processors in the latitudinal direction and since no partial summation is performed during the data transfer, the ring approach requires  $P \log P$  messages and a total transfer of  $NP$  data elements; the binary tree requires  $O(2P)$  messages and a transfer of  $O(NP + N \log P)$  data elements [2].

The high cost of the filtering compared to the rest of the hydrodynamics module as shown in Figure 1 stems from two important factors. The first is the use of convolution form (2) in physical space for the filtering. Assuming a three-dimensional grid for filtering has dimensions  $N \times M \times K$ , where  $N$ ,  $M$ ,  $K$ , are dimensions in latitudinal, longitudinal and vertical directions, respectively, the computational cost of doing convolution on the grid is of order  $O(N^2 \times M \times K)$ , whereas the cost for the rest of hydrodynamics code is of order  $O(N \times M \times K)$ . The second is the existence of a severe load imbalance caused by the fact that only subdomains at high latitudes require filtering. Solutions to these problems are somewhat obvious: (i) use the fast Fourier transform (FFT) instead of performing direct convolution for the filtering, and (ii) perform load balancing before filtering by redistributing data to be filtered from processors containing high latitude subdomains to processors containing low latitude subdomains which either have very little filtering to do or are completely idle during the filtering stage.

### 3.2) Efficient parallel spectral filtering

Since the spectral filtering is applied to lines of grid points at high latitudes and the grid decomposition for the UCLA AGCM code is a two dimensional decomposition in the horizontal plane, the FFT operation also requires interprocessor communication. There are at least two possibilities to parallelize the FFT filtering. One is to develop a parallel one dimensional FFT procedure for processors on the same rows in the processor mesh, so that this procedure can be applied to every line of data to be filtered. The second approach is to partition the data lines to be filtered and redistribute them among processor rows in the latitudinal direction so that FFT's on each data line can be done locally in each processor. The second approach essentially involves a data array transpose. These FFT filtering approaches have a computational cost of  $O(N \times \log N \times M \times K)$ . Again letting  $N$  denote the number of data elements and  $P$  denote the number of processors in the latitudinal

nal direction, the approach using the parallel one dimensional FFT requires  $O(P \log P)$  messages and a transfer of  $O(N \log N)$  data elements, while the approach using FFT with data transpose requires  $O(P^2)$  messages and a transfer of  $O(N)$  data elements. Therefore the first approach requires fewer messages but exchanges larger amounts of data than the second approach.

We chose to implement the second approach for the spectral filtering in the AGCM code. The main reason for our choice is the relative simplicity of implementing the data transpose and the possibility of using highly efficient (sometimes vendor provided) FFT library codes on whole latitudinal data lines within each processor.

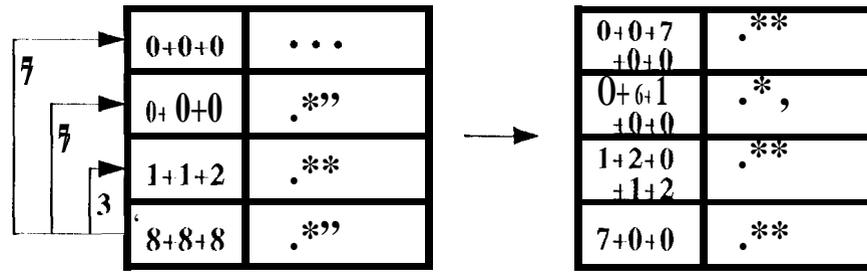


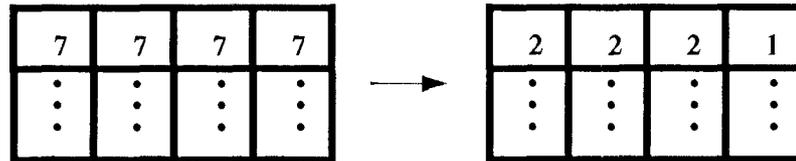
Figure 2. An illustration of data row redistribution for a load balanced filtering.

### 3.3) Load-balanced parallel FFT filtering

To solve the load-balance problem in filtering, we need to redistribute the data rows to be filtered along the longitudinal direction. In the UCLA AGCM code, the spectral filtering is performed at each time step before the finite-difference procedures are called. Weak and strong filterings are performed on different sets of physical variables, one variable at a time in the original AGCM code. To maximize the performance efficiency from the load balance procedures, we reorganized the filtering process so that all weakly filtered variables are filtered concurrently, as are all strongly filtered variables. This change is possible because there is no data dependency among weakly filtered variables, nor among strongly filtered variables in the filtering process. Based on these considerations, we decided to implement a generic load balancing module which does the following: given an  $M \times N$  processor mesh, with  $M$  processors in the latitudinal direction and  $N$  processors in the longitudinal direction, with  $L$  variables to be filtered (weakly or strongly), each with  $R_j$  ( $j = 1 \dots L$ ) rows of data to be filtered, the goal of load balancing is to redistribute the data rows in the longitudinal direction so that after redistribution, each processor will contain approximately (since total number of data rows to be filtered are usually not divisible by  $N$ )

$$\left( \sum_{j=1}^L R_j \right) / N \quad (3)$$

rows to be filtered. If it could be assumed that exactly half of the data rows in one hemisphere are to be filtered, which is the case for the strong filtering in the AGCM code, the implementation of data redistribution for load balancing would be a relatively simple task. All that would be required in this case is to redistribute data rows in a way which is symmetric about the middle latitude line in each



**Figure 3. Data row transpose in latitude direction following the row redistribution shown in Figure 2.**

hemisphere. Since we need to do load balancing for both weak and strong filterings, a more general scheme is needed. We therefore decided to implement a code module which can produce a balanced load in (3) regardless of the number of rows to be filtered in each hemisphere. Figure 2 shows an example of how data rows for three variables are redistributed in a hemisphere in an  $M \times 8$  processor mesh. The load redistribution is followed by a data row transpose and redistribution among processors in the latitudinal direction. Figure 3 shows the data row transpose performed after the row redistribution shown in Figure 2. The actual FFT filtering is performed on data rows after the data transpose, which is then followed by inverse data movements to restore the data layout which existed prior to the filtering.

Due to the generality required for the load-balancing and parallel FFT module, some non-trivial set-up code is needed to construct information which guides the data movements for the load-balancing and load-balanced parallel FFT at each time step of hydrodynamics calculation. The set-up involves substantial bookkeeping and interprocessor communications. Its cost is not an issue for a long AGCM simulation since it is done only once, and its cost is also nearly independent of AGCM problem size.

### 3.4) Loadbalancingthe physics component

The physics parameterization component in the AGCM code consists of a large amount of local computations with little interprocessor communication required under a horizontal partition of the grid. The measured parallel efficiency of the physics component with a  $2 \times 2.5 \times 9$  grid resolution is about 61% on 240 nodes on Cray T3D. Since there is no communication cost, it is only the load-imbalance in the column physics processing that drags down the parallel efficiency. The distribution of computational load in the physics component varies dynamically with space and time in the AGCM simulation. The amount of computation required at each grid point is determined by several factors, including being whether day or night, cloud distribution, and the amount of cumulus convection determined by the so-called conditional stability of the atmosphere. Adding to the difficulty of physics load-balancing is the unpredictability of the cloud distribution and the distribution for cumulus convection, which implies an estimation of computation load in each processor is required before any efficient load-balancing scheme can proceed.

Several possibilities of achieving load-balancing have been considered. One way to achieve a balanced load distribution is to perform a cyclic data shuffling among all processors. Suppose the total number of processors is  $N$ , each processor divides its local data to be processed into  $N$  pieces, sends  $(N - 1)$  pieces of the data to other processors, and receives  $(N - 1)$  pieces of data from other processors. Figure 4 shows such a data shuffling among four processors. The complete data shuffling as shown in Figure 4 guarantees a balanced load distribution as long as the load distri-

3	4	3	4
1	2	1	2
3	4	3	4
1	2	1	2

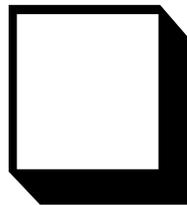


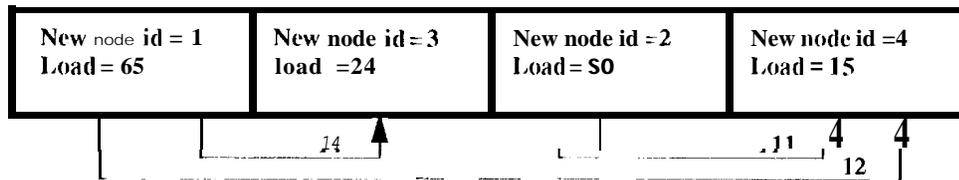
Figure 4. Scheme 1: Global data shuffling among 4 processors to achieve a balanced load distribution. Each data piece is indexed with the id of the processor where it is to be processed.

tribution within each processor is close to uniform in space, a reasonable assumption when  $N$  is large. The main drawback of this approach is the cost of performing all-to-all communications with a complexity of  $O(N^2)$ , and the division of each local data into  $N$  equal pieces for  $N$  processors does not seem to be computationally efficient when  $N$  is large.

An alternative to a complete data shuffling for load balancing, but also guaranteeing a good load distribution, is to use an approach similar to the one discussed in the previous section for filtering operations. First, the computation load for each processor needs to be computed or estimated by some means. Let us look at a specific example for the ease of our discussion. Figure 5 illustrates the steps needed to balance the load among four processors. In Figure 5A, the computing load in each processor has been figured out in some way, and an integer weight is assigned for each local load. All the nodes are then assigned a new id through a sorting of all local loads. The sorting of node ids is performed to simplify subsequent data movement which attempts to minimize the amount of interprocessor communication. With the new node ids and weights of local load available, the required data moves can be carried out in a way similar to that for balancing the filtering load, as shown in Figure 5B. Figure 5C shows the new load distribution after the data movement. It

Node id = 1 Load = 65	Node id = 2 Load = 24	Node id = 3 Load = 50	Node id = 4 Load = 15
--------------------------	--------------------------	--------------------------	--------------------------

A: initial load distribution with original node id.



B: Node ids are sorted according to local data loads. Required data moves are shown.

Node id = 1 Load = 39	Node id = 2 Load = 24+14	Node id = 3 Load = 39	Node id = 4 Load = 15+11+12
--------------------------	-----------------------------	--------------------------	--------------------------------

C: load distribution after load-balancing.

Figure 5. Scheme 2: An alternative which optimizes communication cost.

can be seen that the communication complexity of this load-balancing approach is  $O(N)$ , a significant improvement over the complete data shuffling in scheme 1. Large overhead, however, is incurred in making the optimized data moves possible which involve a number of global communications and a substantial amount of local bookkeeping. This overhead cost was not a serious performance issue in the load-balancing for filtering because it is the cost from a preprocessing step that is done only once during the entire execution of the AGCM code, but the overhead for physics load-balancing may not be overlooked because it is associated with the cost of each physics load-balancing. In addition, a decomposition of a local data load into many parts with different weights may not be a convenient thing to do.

The analysis on load-balancing scheme 2 and 3 leads us to think that it may be more practical in our case to devise a load-balancing strategy that may be less robust (if it is applied only

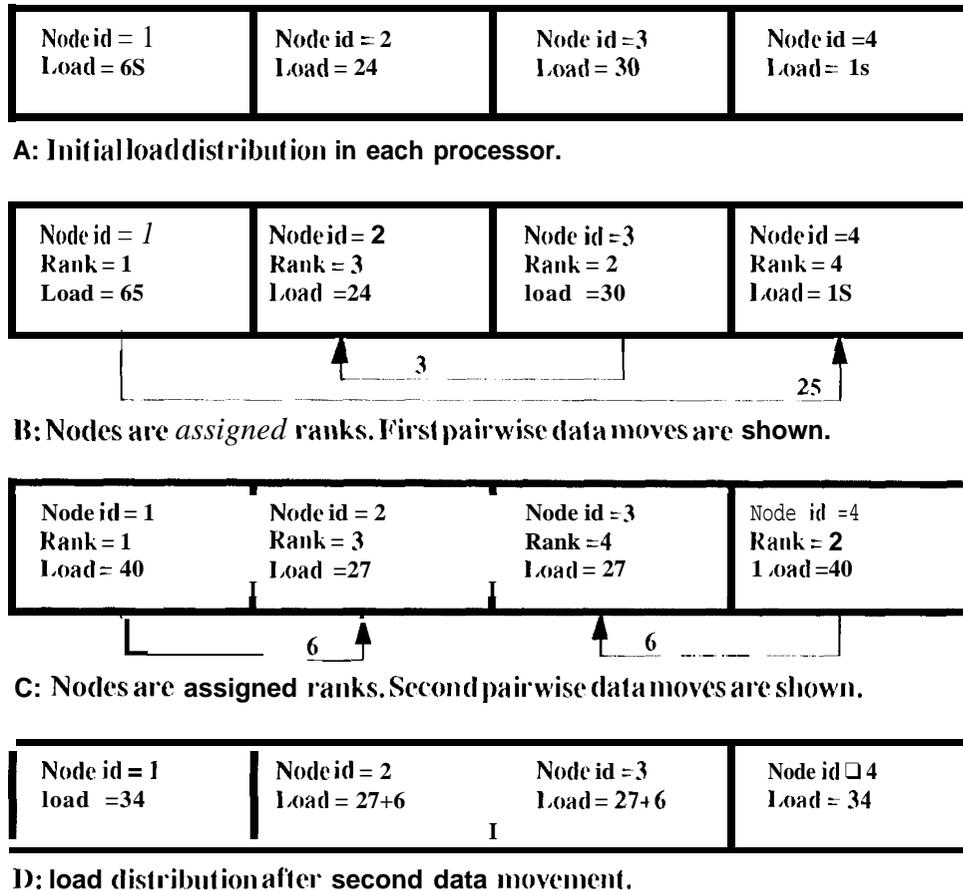


Figure 6. Scheme 3: Load-balancing with pairwise data exchanges.

once) but more cost-efficient and easier to implement. The approach that we currently decide to adopt requires only pairwise interprocessor communications for a movement and a small amount of bookkeeping. The steps for this scheme can still be illustrated by using the previous example for four processors, as shown in Figure 6. The scheme also begins with an evaluation of the local load in

each processor, as shown in Figure 6A. The data load is sorted and a rank is assigned to each processor as a result of the sorting, and a pairwise data exchange between processors with rank  $i$  and rank  $N - i + 1$  is initiated, as shown in Figure 6B. Due to the limitation of pairwise data exchange, the resulted load distribution from the first data move may not be satisfactory. If this is the case, the load sorting and pairwise data exchange can be repeated once as shown in Figure 6C. Figure 6D shows the load distribution after the second data move. Since each load-balancing cycle (sorting and pairwise data moves) is relatively low in cost, the cost of performing it a few times could still be less than that of the previous two schemes.

The number of times needed for sorting and pairwise communication in scheme 3 to achieve a satisfactory load-distribution clearly depends on the initial load distribution. To evaluate the effectiveness of scheme 3 for load-balancing the actual physics component code, we first implemented the load-sorting part in scheme 3, and use it as a tool to perform load-balancing on the physics component and to evaluate the result without actually moving the data arrays around. To estimate local computing load in each processor, a timing on the previous pass of physics component was performed at each processor and the result was used as an estimate for the current physics computing load. Tables 1-3 show the simulation results on 64, 126 and 252 nodes on Cray T3D. With  $P$  processors, the percentage of load-imbalance shown in the last column of the tables is defined as

$$\text{Average Load} = \left( \sum_{i=1}^P \text{LocalLoad}_i \right) / P$$

$$\text{PercentageOfLoad Imbalance} = \frac{(\text{MaxLoad} - \text{AverageLoad})}{\text{AverageLoad}}$$

**Table 1: Load-balancing simulation for physics with a 2 x 2.5 x 29 grid resolution on a 8 x 8 node array on Cray T3D**

Code status	Max load (second)	Min Load (second)	% of load-imbalance
Before load-balancing	11.00	4.90	37%
After first load-balancing	7.71	6.20	9%
After second load-balancing	7.10	6.60	6%

**Table 2: Load-balancing simulation for physics with a 2 x 2.5 x 29 grid resolution on a 9 x 14 node array on Cray T3E**

Code status	Max load (second)	Min Load (second)	% of load-imbalance
Before load-balancing	5.20	2.50	35%
After first load-balancing	4.00	3.14	12%
After second load-balancing	3.52	3.22	5%

**Table 3: Load-balancing simulation for physics with a  $2 \times 2.5 \times 29$  grid resolution on a  $14 \times 18$  node array on Cray '131)**

Code status	Max load (SCCO11(1))	Min load (second)	% of load-imbalance
Before load-balancing	33.34	1.112	48%
After first load-balancing	2.20	1.70	12.5%
After second load-balancing	1.92	1.80	6%

Scheme 3 can be seen as an iterative scheme that converges to a load-balanced state from a given initial load-distribution state. The “convergence rate” of the scheme clearly depends on the initial state as the results in Tables 1 - 3 indicate. On 126 and 252 nodes, it can be seen from Table 2 and 3 that application of the scheme twice to the physics component can reduce the percentage of load-imbalance to a reasonable level. One advantage of scheme 3 is its flexibility of making a compromise between the cost and accuracy of load-balance. A pairwise data exchange is only needed when the load difference in the pair of nodes exceeds some tolerance, and the iteration can stop as soon as the percent age of load -imbalance falls within the tolerance.

### 3.4) Single node performance optimization

With the use of the load-balanced FFT filtering module, we have been able to reduce the cost of filtering significantly in the parallel AGCM code (see Section 4). With the  $2 \times 2.5 \times 9$  resolution on 240 nodes, for example, the filtering cost dropped from 49% of the cost doing hydrodynamics part to about 21%. Our timing of the code indicates the cost of communication for exchanging values at ghost grid points for the finite-differencing is relatively insignificant, usually around 10% of the cost of the hydrodynamics component on 240 nodes. With a load-balanced physics component, we expect the overall execution time of the AGCM code be reduced by 10-15% on 240 nodes. We now turn our discussion to the issue of single-node performance optimization for the AGCM code. As is typical for a real-world application, the overall performance of the parallel AGCM code is well below the peak performances on both Intel Paragon and Cray T3D nodes, which is usually an indication that the cache efficiency of the code is poor. Our main goal is to improve the single-node performance of the code - minimize the execution time of the code on a single processor - with a machine-independent and problem-size robust approach (i.e. without resorting to any assembly coding). We selected a dry-convection routine from the hydrodynamics component and a cumulus convection routine from the physics component as the representative candidates for single-node performance analysis and optimization because of the heavy local computing involved in these routines and their cost weights in their respective components. Our optimization effort started from improving some of the more obvious code segments, such as eliminating or minimizing redundant calculations in nested loops, replacing some loops by Basic Linear Algebra Subroutines (BLAS) library calls for vector copying, scaling or saxpy operations. We also tried to break down some very large loops involving many data arrays in hoping to reduce the cache miss rate. When applying these strategies to the dry-convection routine, we were able to cut down its execution time on a single Cray '131) node by about 20%.

BLAS routines are usually faster than programmer's hand-coded loops in a high-level programming language for matrix-vector data processing because they were optimized for pipelining computing and cache efficiency with assembly coding. It seems, however, difficult for us to utilize the BLAS library beyond some 10W-1CVCI routines in a few places of our code. In a code based on finite-differencing schemes as the AGCM code, it is usually hard to cast major parts of computation into matrix-vector type operations. Instead, we found that a large part of the computations in our selected routines can be converted into what we call "pointwise vector-multiply", which, for example, has the form in a two-dimensional nested loop:

```

DO j = 1, N
  DO i = 1, M
    C(i, j) = A(i, j) x B(i, s)
  ENDDO
ENDDO

```

where the subscripts can be either a constant or equal to  $j$ . The computation in the above loop is not one of the operations defined in the current BLAS library (e.g. on Cray T3D). We think one possibility to achieve good performance for such a loop is to develop an optimized library routine in assembly language which can recursively perform the following operation on two vectors  $a = \{a_1, a_2, \dots, a_n\}$  and  $h = \{b_1, b_2, \dots, b_m\}$

$$a \otimes b = \{a_1 b_1, a_2 b_2, \dots, a_m b_m, a_{m+1} b_1, \dots, a_{2m} b_m, \dots\} \quad (4)$$

where it is assumed that  $n$  is divisible by  $m$ . The interface of the routine can be such that it takes as input a set of data arrays and returns the result array. If some optimization on such a pointwise vector-multiply operation is possible in terms of cache and pipelining, there is a good chance for us to improve single-node performance for the AGCM code in a portable and robust fashion.

The general idea of cache efficiency optimization is to explore data locality of an application so that the data existing in the cache can be reused as much as possible. In a finite-difference application such as the AGCM code, a major part of the local computations lie in the evaluations of finite-difference equations which involve a number of discrete fields corresponding to physical variables defined on computational grids. At each grid point indexed by  $(i, j, k)$ , the following type of code frequently occurs

$$r(i, j, k) = D_1 f_1(i, j, k) + \dots + D_m f_m(i, j, k), \quad (5)$$

where  $f_i$  ( $i = 1, \dots, m$ ) are discrete fields and  $D_i$  ( $i = 1, \dots, m$ ) are stencil operators. Although it seems natural, as done in the AGCM code, to allocate storage corresponding to discrete fields in (5) as separate data arrays, the cache efficiency in computing (5) on those separate arrays is usually rather poor when the typical array size is much larger than the cache size or when data stored in a large number of arrays are referenced in a statement of form (5), because in such cases the cache-miss rate can be very high. One alternative to allocating separate data arrays is to declare a single array for storing all the discrete fields in (5). In a Fortran code, one can thus define a "block-oriented" array of the form

$$f(m, i \dim, j \dim, k \dim). \quad (6)$$

The use of a data array of the form (6) to evaluate (5) could, in principle, reduce the cache-miss rate, because grid variables in the neighborhood of a certain cell are stored closer to each other in memory than the case when separate arrays are used. When data arrays of the size  $32 \times 32 \times 32$  in form (4) are used, our test code evaluating a seven-point Laplace stencil applied to several discrete fields showed a speed-up a factor of 5 over the use of separate arrays on the Intel Paragon, and a speed-up factor of 2.6 was achieved on Cray's 131J for the same size data arrays. Encouraged by this result, we tried the use of block array in the dry-convection routine, where about a dozen of three-dimensional arrays were combined into one single array. A performance comparison between the code with block array and the code with separate arrays did not show any advantage of using the block array. For some sizes of data array, the code with the block array underperformed the code with separate arrays. A more careful examination of the dry-convection routine revealed some conflicting factors regarding the selection of a good data structure for cache efficiency. A basic fact is that the dry-convection routine contains many different types of array-processing loops which reference a varying number of data arrays. The block array may be a better data structure for cache efficiency in a loop referencing all the grid variables in the block array, but it could be a worse data structure (than the separate arrays) for code in other loops which only reference a small subset of grid variables in the block array. It is therefore not easy to predict the overall effect on the cache performance for a non-trivial code when a block array or separate arrays are used. A side-effect of using block array is the poor readability of the code, which makes it error-prone and harder to debug.

**Table 4: AGCM timings (seconds/simulated day) with old filtering module on Intel Paragon grid resolution:  $2 \times 2.5 \times 9$**

Node mesh	Dynamics	Dynamics Speed-up	Total time (Dynamics and Physics)
1 x 1	8702	1.0	14010
4 x 4	848.5	10.3	1177
8 x 8	366	23.8	443.5
8 x 30	186	46.8	216

**Table 5: AGCM timings (seconds/simulated day) with new filtering module on Intel Paragon grid resolution:  $2 \times 2.5 \times 9$**

Node mesh	Dynamics	Dynamics Speed-up	Total time (Dynamics and Physics)
11x1	8075	1.0	11225
44 x 4	639.0	12.6	992.6
88 x 8	207.5	38.9	306.0
88x30	87.2	92.6	119.0

show that the scaling of load-balanced FFT filtering for the 9-layer model is about 4.74 running on 240 nodes versus running on 16 nodes with a parallel efficiency of 32%, and the scaling of load-balanced filtering for the 15-layer model is about 5.87 with a parallel efficiency of 39%. The improved efficiency for the 15-layer model reflects the higher ratio of local computational load over interprocessor communication cost when more vertical layers are added to the AGCM model. Although not shown here, we found the scaling of the whole AGCM code for the 15-layer model is about the same as the 9-layer model. This could be the result of the fact that in the 15-layer model, some additional load-imbalance is introduced in other parts of the AGCM code. We would even expect better scaling be achieved for the parallel filtering as well as for the overall AGCM code when a 1x1.25 by 15-layer version is used. The execution times also consistently show that the parallel AGCM code runs about 2.5 times faster on Cray 3D than on Intel Paragon.

**Table 8: Total filtering times (seconds/simulated day) on Intel Paragon for the 2 x 2.5 x 9 grid resolution**

Node mesh	Convolution	FFT without load balance	FFT with load balance
4 x 4	309.5	111.4	87.7
4 x 8	<b>240.0</b>	88.0	53.7
<b>8 x 8</b>	<b>189.5</b>	66.4	38.2
4x30	99.6	43.7	22.2
8x30	90.0	37.5	18.5

**Table 9: Total filtering times (seconds/simulated day) on Cray T3E for the 2 x 2.5 x 9 grid resolution**

Node mesh	Convolution	FFT without load balance	FFT with load balance
4 x 4	123.5	44.6	35.1
<b>4 x 8</b>	<b>96.0</b>	35.2	21.5
8 x 8	75.8	26.4	15.3
4 x 30	39.6	17.5	8.9
8 x 30	36.0	15.0	7.4

## S. Software design issues for GCM simulations

Since GCM simulation codes are typically large software packages containing tens of thousands lines of code, another goal of our work is to develop portable and reusable library modules and extensible template codes which will be useful for GCM type applications. The original parallel AGCM code was implemented in Fortran with a generic message-passing interface. The portability of the code was achieved by using macros for message-passing protocols and memory alloca-

**Table 6: AGCM timings (seconds/simulated day) with old filtering module on Cray T3D  
grid resolution:  $2 \times 2.5 \times 9$**

Node mesh	Dynamics	Dynamics Speed-up	Total time (Dynamics and Physics)
$1 \times 1$	3480	1.0	5600
$4 \times 4$	339	11.3	470
$8 \times 8$	146	26.3	177
$8 \times 30$	74	51.9	87.5

**Table 7: AGCM timings (seconds/simulated day) with old filtering module on Cray T3D  
grid resolution:  $2 \times 2.5 \times 9$**

Node mesh	Hydrodynamics	Dynamics Speed-up	Total time (Dynamics and Physics)
$1 \times 1$	3230	1.0	4990
$4 \times 4$	256	12.6	397
$8 \times 8$	83	38.9	122
$8 \times 30$	35	92.3	48

#### 4. Performance studies

Timings have been performed on the Intel Paragon and Cray T3D (Some timing on IBM SP-2 were also performed, but are not shown here) for the parallel AGCM code with the new filtering module and the results were compared to those from the original code. The message-passing portability of the filtering module was achieved by using MPI protocols in the code. Since the UCLA AGCM code uses a NITCDF input history file and we do not have a NITCDF library usable on Paragon, we had to develop a byte-order reversal routine to convert the history data to use on Intel Paragon. We here only discuss timing results obtained on Intel Paragon, which qualitatively applies to Cray T3D and IBM SP-2 as well. Tables 4-7 show comparisons of execution time for the hydrodynamics part and for the entire AGCM code (including physics part) using the 9-layer model on Intel Paragon and Cray T3D. Tables 8 and 9 show a comparison of costs for doing the filtering using different versions of filtering module with the 9-layer model, and Tables 10-11 show the costs of filtering in the 15-layer model on Paragon and Cray T3D. For all the timing runs, a  $2 \times 2.5$  horizontal grid resolution is used. In comparison to the old AGCM code, the hydrodynamics component in the new code is a little more than twice as fast on 240 nodes. The scaling (or speed-up) of the entire code also improved significantly, which is clearly a result of the load-balanced filtering. The load-balanced FFT filtering module runs about five times faster than the old convolution filtering module on 240 nodes for both the 9-layer model and the 15-layer model, Tables 8-9 and 10-11

**Table 10: Total filtering times (seconds/simulated day) on Cray T3E for the 2 x 2.5 x 15 grid resolution**

Node mesh	Convolution	FFT without load balance	FFT with load balance
4 x 4	802	304	221
4 x 8	566	205	118
8 x 8	422	150	85
4 x 30	217	96	49
8 x 30	188	81	37

**Table 11: Total filtering times (seconds/simulated day) on Cray T3E for the 2 x 2.5 x 15 grid resolution**

Node mesh	Convolution	FFT without load balance	FFT with load balance
4 x 4	320	121	88
4 x 8	226	82	47
8 x 8	168	60	34
4 x 30	86	38	19
8 x 30	75	32	15

tion protocols. This macro approach unfortunately also introduced some complications to the code maintenance and modifications. First the code needs to go through two macro preprocessors before a standard Fortran compiler can be applied, which can cause problems when porting the code to a new machine because macro preprocessors may behave differently. Embedding macros in the code also make changes to the code error-prone if one is not familiar with how macros are to be expanded. We think the portability of the AGCM code can be achieved in a simpler and more reliable way. Our approach also defines generic interfaces for possibly machine-dependent operations such as message-passing protocols and memory allocation, but the implementation of the interfaces is wrapped in a small number of subroutines. These subroutines are selectively compiled depending on the specific machine where the code is to run. We believe our approach can reduce the machine-dependent portion of the code to a minimum and thus make maintenance and modification of the code easier. We are also identifying common algorithms and operation components from GCM applications, and develop code modules which are reusable and extensible (as application templates) in different GCM applications. In our view, candidate components for GCM applications include efficient finite-difference kernels, parallel spectral filters, communication modules for exchanging ghost-point values at domain-partition boundaries, enforcing (physical) periodic boundary condition, load-balance modules, and fast (parallel) linear system solvers for implicit time-differencing schemes. We believe, within the scope of GCM applications, these code components can

be developed in a unified, highly modular and efficient manner, and we think an objected-oriented approach (at least for building the infrastructure of a generic GCM application) implemented in an advanced scientific computing language like Fortran 90 can be used in the code development. With these code components available, the prototyping and implementation of a new, portable and efficient GCM code package for distributed memory parallel machines will be a lot easier.

## 6. Conclusion and future work

We have shown our analysis and optimization strategies to improve the overall performance of the parallel UCLA AGCM code on massively parallel computers by implementing a load-balanced FFT filtering module for the hydrodynamics component, and a load-balancing module for the physics component. Performance comparisons of the AGCM codes with old and new spectral filtering modules show that a speed-up of a factor 2 is achieved as a result of our work on 240 nodes, and our analysis shows that a load-balanced physics component could improve the AGCM code performance by an additional 10~15%. We discussed our effort on the single-node performance optimization for selected subroutines from the AGCM code, including the lessons we learned from our attempt to improve the cache efficiency, and a possibility to achieve better single-node performance for the AGCM code by developing an optimized point wise vector-multiply routine. We also addressed our views on making better software design for GCM applications through developing efficient and reusable code components. A complete implementation of the load-balancing module for the physics component is being developed. Single-node performance-tuning is still one of our main on-going efforts in the performance optimization on the AGCM code.

## Acknowledgments

This work was supported in part by the NASA High Performance Computing and Communication for Earth and Space Sciences Project under Grant NAG 5-2224. The investigations reported here were conducted on a Intel Paragon operated by the Concurrent Supercomputing Consortium at Caltech and a Paragon located at Jet Propulsion Laboratory, on a Cray T3E system operated by the Jet Propulsion Laboratory, and on IBM S1<sup>2</sup> operated by NASA Ames Research Center.

## References:

1. A. Arakawa and V. Lamb, "Computational Design of the Basic Dynamical Processes of the UCLA General Circulation Model.", *Methods in Comp. Phys.* 17 (1977) 173-265.
2. M.J. Wehner, A.A. Mirin, P.G. Eltgroth, W.J. Dannevik, C.R. Mechoso, J. Farrara, J.A. Spahr, "Performance of a Distributed Memory Finite-Difference Atmospheric General Circulation Model.", *Parallel Computing* 21, 1655-1675, 1995.
3. M.J. Suarez, A. Arakawa, and D.A. Randall, "The Parameterization of the planetary boundary layer in the UCLA General Circulation Model: Formulation and Results.", *Mon. Wea. Rev.*, 111, 2224-2243, 1983.
4. "Introduction to the UCLA General Circulation Model: Its History, Present State and Future Direction", UCLA Atmospheric Science 281 Course Note, Winter 1995.