ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-190

Load-Balancing Algorithms for the Parallel Community Climate Model

by

Ian T. Foster and Brian R. Toonen

Mathematics and Computer Science Division

Technical Memorandum No. 190

January 1995

This work was supported by the Atmospheric and Climate Research Division, Office of Energy Research, Office of Health and Environmental Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Contents

Load-Balancing Algorithms for the Parallel Community Climate Model

Ian T. Foster Brian R. Toonen

Abstract

Implementations of climate models on scalable parallel computer systems can suffer from load imbalances resulting from temporal and spatial variations in the amount of computation required for physical parameterizations such as solar radiation and convective adjustment. We have developed specialized techniques for correcting such imbalances. These techniques are incorporated in a general-purpose, programmable load-balancing library that allows the mapping of computation to processors to be specified as a series of maps generated by a programmer-supplied load-balancing module. The communication required to move from one map to another is performed automatically by the library, without programmer intervention. In this paper, we describe the load-balancing problem and the techniques that we have developed to solve it. We also describe specific load-balancing algorithms that we have developed for PCCM2, a scalable parallel implementation of the Community Climate Model, and present experimental results that demonstrate the effectiveness of these algorithms on parallel computers. The load-balancing library developed in this work is available for use in other climate models.

1 Introduction

Scalable parallel computer systems use a high-speed interconnection network to connect hundreds or thousands of powerful microprocessors. Each processor typically has its own memory, executes independently, and exchanges messages with other processors to synchronize execution or share data. Contemporary examples of this architecture include the Intel Paragon, Thinking Machines CM5, IBM SP, and CRAY T3D.

Science and engineering applications can often be adapted for execution on scalable parallel computers by using a technique called domain decomposition [4]. This works as follows. First, principal program data structures are decomposed into disjoint subdomains of approximately equal size. Then, each subdomain is mapped together with its associated computation to a different processor. Finally, communication is introduced to move data between subdomains when this is required for computation. Unfortunately, the performance of a program developed by using these techniques can be compromised by poor single-processor performance, by excessive interprocessor communication, or by *load imbalance*: a nonuniform mapping of computational load to processors. It is the last problem that we address in this report.

While load-balancing is an important problem of general interest in parallel computing, our particular interest is in developing efficient load-balancing algorithms for parallel

climate models. Load imbalances can arise in climate models because the amount of computation to be performed per data item is variable. This variation occurs in the model routines that perform computations concerned with physical parameterizations such as solar radiation, gravity wave drag, and convective adjustment. This component of the model is termed "physics" to distinguish it from "dynamics," which is primarily concerned with the fluid dynamics of the atmosphere. While load imbalances can also arise in dynamics, these have a different character and are not considered here.

Computational load imbalance is generally addressed by using one of two methods. Static load-balancing techniques attempt to determine a static mapping of computation to processors that minimizes total execution time. While requiring no specialized runtime mechanisms, this technique does not appear well suited to climate models, in which load distribution can change significantly during program execution. In contrast, dynamic load-balancing techniques allow the mapping of computation to processors to change during program execution. The various mappings can be defined prior to execution and applied by using a predefined schedule, or can be computed during execution. The techniques that we have developed support both mapping approaches.

The rest of this report is as follows. Section 2 describes the structure of PCCM2, the parallel climate model that we use to evaluate our load-balancing techniques. Sections 3, 4, and 5 describe the principal load imbalances that occur in PCCM2, a set of algorithms that we have developed to correct these load imbalances, and the structure of the library developed to implement these algorithms. Finally, Section 6 presents performance results for the various algorithms, and Section 7 presents our conclusions.

2 PCCM2

While much of the work reported in this paper is independent of any particular climate model, our implementation work and empirical studies have been performed in the context of PCCM2, a parallel implementation of the Community Climate Model (CCM2) developed by the National Center for Atmospheric Research (NCAR) [3]. Hence, we provide a brief introduction to the structure of this model.

Both dynamics and physics operate on a set of three-dimensional data structures with size $N_{glat} \times N_{glon} \times N_{gver}$, where N_{glat}, N_{glon}, and N_{gver} are the number of grid points in the latitudinal, longitudinal, and vertical direction, respectively. The parallel implementation uses domain decomposition techniques to decompose these data structures, and associated computation, in the two horizontal dimensions [3]. Some of these data structures are used only by dynamics or only by physics; others are shared by the two components. At each time step, a subset of these data structures is passed between the two components of the model, which are executed one after the other. Hence, it is most efficient in the absence of load imbalances to decompose physics data structures in the same way as dynamics data structures.

The dynamics data structures are decomposed as follows. Processors are divided into processors responsible for groups of latitudes (P_{lat}) and processors responsible for groups of longitudes within a latitude row (P_{lon}). For the purposes of this discussion, we restrict N_{glat}, N_{glon}, P_{lat}, and P_{lon} to powers of two. The following restrictions also apply: $P_{lat} \leq (N_{glat}/2)$ and $P_{lon} \leq (N_{glon}/4)$. The resulting structure is illustrated in Figure 1 [2]. In order to exploit symmetries in the dynamics computations, the latitudes are divided

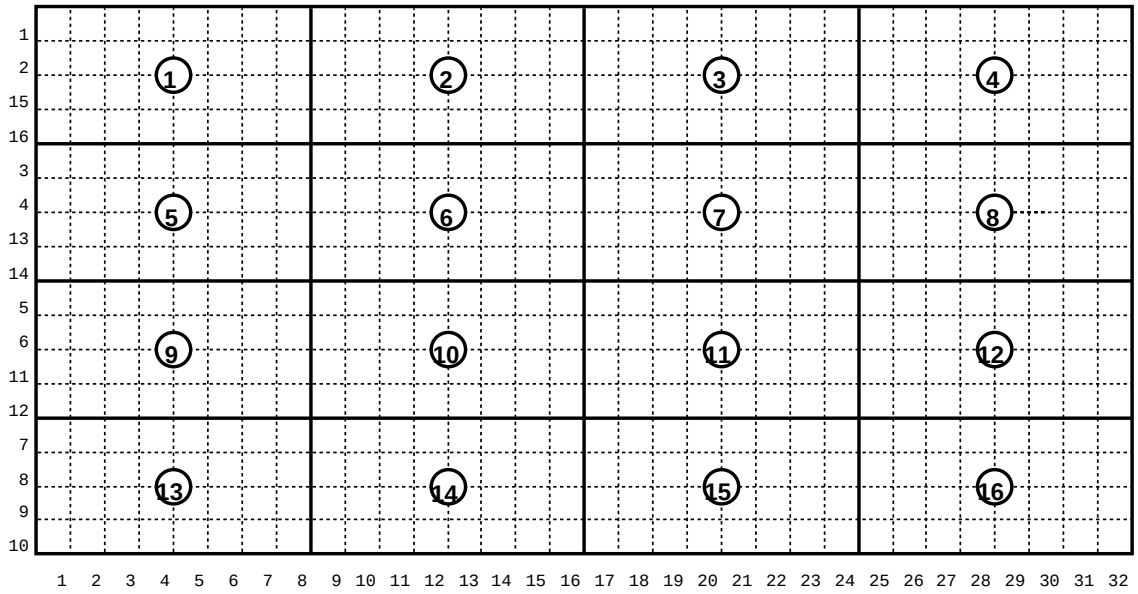

Figure 1: Initial Decomposition of a Physical Domain Consisting of 16×32 Columns Mapped onto 16 Processors

into $2 \times P_{lat}$ sections with each section containing $N_{glat}/(2 \times P_{lat})$ latitudes. For any given latitude processor row J, row J is assigned data sections J and $2 \times P_{lat} - J$. Thus, each row of latitude processors receives $N_{glon} \times N_{glat}/(2 \times plat)$ latitudes from the north and symmetrically the same number of latitudes from the south, resulting in $N_{llat} = N_{glat}/P_{lat}$ latitudes of data being assigned to each latitude processor group.

The longitudinal decomposition is a linear partitioning of the data columns. The N_{glon}columns of data found on each latitude are divided among the P_{lon}processors, resulting in $N_{llon} = N_{glon}/P_{lon}$ data columns per latitude on any given processor. Since the partitioning is linear, a longitude processor I is given data columns $(I-1) \times N_{llon} + 1$ through $I \times N_{llon}$ from each latitude assigned to it.

3 Load Imbalances in PCCM2

In the current release, PCCM2.1, physics load imbalances account for 8.1 percent of total execution time at T42 resolution on the 512-processor Intel Touchstone Delta computer. This proportion is expected to increase as other components of the model are optimized.

Three types of physics time step can be distinguished within PCCM2: partial radiation, full radiation, and no radiation [1]. Partial radiation time steps occur every hour (which, in the current implementation, is every third time step). During these time steps, the shortwave radiation calculations are performed. A full radiation time step additionally computes the absorptivity and emissivity of longwave radiation. These time steps occur once every twelve hours (every 36 time steps). The remaining time steps are referred to as "no-radiation steps," since no solar radiation computations are performed.

A study of computational load distribution within CCM2 reveals that the most significant source of load imbalance is the diurnal cycle [6]. This is due to the computationally expensive shortwave radiation calculations performed during partial radiation time steps.

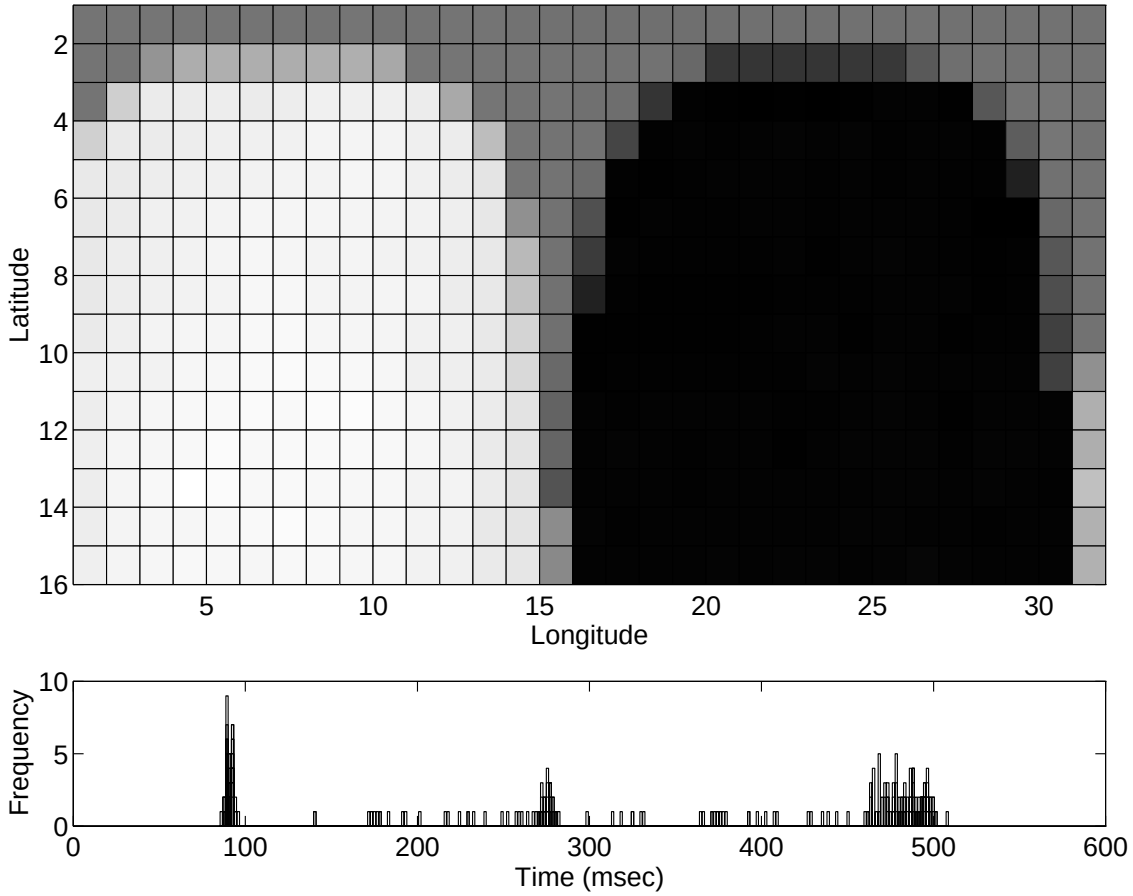

Figure 2: Physics Computation Time for a 16×32 Processor Mesh on the Intel Touchstone DELTA When Using the Same Data Distribution as Dynamics

Because these calculations are performed only for grid columns exposed to solar radiation, processors containing exposed data columns perform significantly more computation than processors with few or no exposed data columns. Hence, if PCCM2 physics data structures are decomposed in the same way as dynamics data structures, we obtain a spatial imbalance where approximately half the processors remain idle while the others perform the required radiation calculations (see Figure 2).

The load imbalance introduced by the diurnal cycle is temporal as well as spatial [6]. The earth's rotation about its axis causes the area impacted by solar radiation to continuously shift westward. In addition, the revolution of the earth around the sun results in a cyclic annual drift of the solar declination between the summer and winter solstices.

Hence, the latitudes exposed to solar radiation change over time. Furthermore, cycling between the three types of time steps introduces still another form of load variation.

Other load imbalances encountered in PCCM2 physics include land/sea imbalances, variations caused by weather patterns (e.g., convection over the Indian subcontinent during the monsoon), and the seasonal cycle [5]. While these are not currently viewed as significant performance problems, future enhancements to PCCM2 physics may introduce new forms of load imbalance with these sort of patterns.

4 Load-Balancing Algorithms

The load distribution characteristics of PCCM2 dynamics and physics, and indeed of the three different types of physics time steps, are so different that it is not feasible to utilize a single mapping of data and computation to processors. Instead, we require load-balancing algorithms that change mapping frequently — in fact, on almost every time step. In this section, we describe such a set of algorithms, each characterized by the mappings that it employs. The implementation and performance of these algorithms are considered in subsequent sections.

The library that we have developed to support our load-balancing algorithms provides a general framework that can be used to implement a wide range of algorithms. The algorithms that we develop for use in PCCM2, however, are concerned primarily with the diurnal cycle. Fortunately, the pairing of symmetric north-south latitudes in the initial distribution compensates for most of the load variation resulting from the solstitial drift. However, the continual westward movement of the heavily loaded region, caused by earth's rotation about its axis, is not compensated for in the same manner.

Within physics, computation performed within each vertical column is independent of that performed in other columns. This situation means that vertical columns and their associated computation can be migrated between processors without any significant changes to physics. We take advantage of this fact to alter the data distribution used in physics so that each processor receives an equal number of daytime and nighttime data columns. Unfortunately, this decomposition cannot easily be used in dynamics. Hence, data must be returned to the initial decomposition after completion of physics computation and prior to the invocation of dynamics.

The various load-balancing algorithms that we have developed for PCCM2 all seek to compensate for the load imbalance that results from the diurnal cycle, by migrating columns within individual latitudes (we do not migrate columns within longitudes because there is little load imbalance in the north/south direction). The algorithms differ in the frequency and patterns of migration that they employ.

The simplest algorithm that we consider swaps every other data column with the processor directly opposite itself within its latitude. This column will be located on the processor that is longitudinally $P_{lon}/2$ processors away. This algorithm causes a contiguous block of columns exposed to daylight to be dispersed to processors that contain few or no daytime columns, and hence does an excellent job of balancing load. Since each processor moves half of its data to another processor, it has the disadvantage of a large amount

of data always being transmitted. On the other hand, the communication pattern is predictable and hence amenable to optimization.

The other algorithms that we consider seek to balance load by moving a smaller number of columns. In general, this approach can reduce communication requirements and the overhead associated with the load-balancing library. With these algorithms, the solar radiation state of each data column must be determined. For our purposes, the zenith angle computations found in CCM2's `radinp` routine provide sufficient information. Using the cosine of the zenith angle, we can easily determine the columns exposed to solar radiation by checking for a value greater than zero.

Given information about which data columns are exposed to solar radiation, a second swapping algorithm can be considered. This algorithm determines the difference, d, in the number of daytime columns on a given processor and the processor $P_{lon}/2$ processors to the west. Then, $\frac{d}{2}$ daytime columns from the more heavily loaded processor are exchanged for an equal number of nighttime columns from the opposing processor. This method reduces the number of data columns being transmitted but requires that a separate mapping be generated for each radiation time step. These mappings must be either cached or computed on the fly. While caching is sufficient for the trial runs associated with this study, it is infeasible for the extended runs common to climate models.

Slightly more complex is the algorithm that moves data columns rather than simply swapping them. To accomplish this, we must estimate computation costs associated with a column. We determine the computation costs as follows. First, we determine the ratio of the computation times required for daytime and nighttime columns. We refer to this ratio as the *daypoint-nightpoint ratio*, $\frac{d}{n}$. This ratio is computed in a calibration step, prior to running the model. Using this information and the exposure information discussed previously, we can determine the cost of computation associated with the data columns on a every latitude and processor. The cost for any given latitude and processor is simply $\frac{d}{n} \times N_{day} + 1 \times N_{night}$, where N_{day} is the number of columns exposed to daylight and N_{night} is the number of columns in complete darkness.

In addition to estimating computation costs, the movement algorithm necessitates the extension of the data arrays used within physics so as to provide room for more than N_{llon} columns per latitude. Because the rest of PCCM2 assumes N_{llon} columns per latitude, and cannot easily be modified, it becomes necessary to copy data arrays from dynamics arrays to new "extended arrays" prior to calling physics. This step represents additional overhead not found in either of the swapping algorithms.

The movement algorithm attempts to move columns between a given processor and the processor offset 180 degrees in longitude, because this strategy was found to compensate well for the diurnal cycle imbalance. In this algorithm, daytime columns are moved from the more heavily loaded processor until the cost difference between the two processors is minimized. Then, the same technique is used for nighttime points, thus providing a fine–grain adjustment. Since physics requires the data columns on each processor to be contiguous, data columns may need to move locally. To minimize this local movement, we choose the data columns to be transported to opposing processors from right to left in the data arrays.

For comparison purposes, we also included a version of the well-known recursive bisection algorithm in this study. The bisection algorithm, like the movement algorithm,
linebreak

uses the daypoint-nightpoint ratio and the zenith angle to determine the cost associated with each column. The data columns within a latitude are recursively divided into two groups with approximately equal costs. The recursion continues until each processor has been assigned a contiguous set of columns.

Since all four algorithms are designed to deal with the diurnal cycle, they are applied only during radiation time steps: that is, once every hour. During non-radiation time steps, physics data structures remain in their initial decomposition. Hence, the first swapping algorithm alternates between two mappings: the initial mapping and a swapped mapping. For the purposes of this study, the other algorithms use a different mapping for each radiation time step.

5 Implementation

The load-balancing algorithms described in the preceding section are implemented by a general-purpose, configurable data movement library. This library allows the programmer developing load-balancing algorithms to specify simply the mapping of data columns to processors that is to apply within physics at each time step; the library then takes care of organizing the movement of data required to support this mapping.

Data movement is required for two purposes. At every time step, data structures shared by physics and dynamics must be reorganized from the physics mapping to the dynamics mapping prior to calling dynamics, and then back to the physics mapping prior to calling physics. In addition, data structures used only with physics must be reorganized whenever the physics mapping changes.

Obviously, we wish to minimize the amount of data communicated by the data movement library. Hence, we distinguish between the following four categories of physics data structures. Each has its own set of data movement requirements.

Input: These variables are shared by physics and dynamics. They are used to pass values from dynamics to physics, but not from physics to dynamics. Hence, they must be reorganized before calling physics, but not after.

Output: These variables are shared by physics and dynamics. They are used to pass values from physics to dynamics, but not from dynamics to physics. Hence, they must be reorganized after calling physics, but not before.

Input/Output: These variables are shared by physics and dynamics. They are used to pass values both from physics to dynamics and from dynamics to physics. Hence, they must be reorganized both before and after calling physics.

State: These variables are only used within physics. Within PCCM2, these are variables whose values are set at the beginning of the model and then remain constant or change only rarely during execution. These variables are most often found in common blocks, although they may be occasionally stored in temporary files or in-core storage. It is necessary to reorganize these variables only when the physics mapping changes.

5.1 Definitions and Data Structures

A load-balancing algorithm is represented to the data movement library as a set of *schemas* and a *schedule*. In this section, we define these terms and provide additional information on the techniques used to implement load-balancing algorithms.

As PCCM2 performs computation one latitude at a time, the load-balancing library transfers data columns to processors only within the same latitude processor group. In other words, a column may be transferred from P_1 to P_2 only if $\lfloor P_1/P_{lon} \rfloor = \lfloor P_2/P_{lon} \rfloor$. Furthermore, the data column must be assigned to the same latitude on the new and original processors. Although these restrictions limit load-balancing to a single dimension, it would not be difficult to remove these restrictions should load-balancing in the second dimension be required.

Schema: A *schema* defines a valid mapping of physics columns to processors. A mapping is represented as an $N_{glat} \times N_{glon}$ integer array in which the (i,j)-th entry identifies the processor on which the (i,j)-th physics columns is to be located. The constraints placed on a schema are that (a) the number of data columns assigned to any one processor in each latitude is no more than the constant N_{llonx}, where $N_{llonx} \geq N_{llon}$ is the maximum number of data columns per latitude that any processor has space allocated for, and (b) a column not be assigned to a processor not existing in its latitude processor group. A special schema, the *identity schema,* is defined as a mapping that assigns all columns to their initial (or home) processor. The following is an example of an identity schema for a 4×8 grid, mapped to 4 processors in a 2×2 configuration.

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

The following schema allocates ten columns to processor 1, six to processor 2, eleven to processor 3, and five to processor 4.

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

The following schema is not valid because it attempts to send a column from processor 1 to processor 3, thereby violating the restriction that the receiving processor be in the same latitude processor group as the sender.

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

Table 1: Daypoint-Nightpoint Ratios

Processors		Daypoint-Nightpoint Ratio	
Lat.	Long.	Full Radiation	Partial Radiation
4	8	1.19	4.20
8	8	1.18	4.13
8	16	1.15	3.62
16	16	1.15	3.63
16	32	1.11	3.21

Schema Set: A *schema set* is a collection of schemas from which a single schema may be selected to define the current data mapping. Every schema set must contain the identity schema as the first member of the set.

Schedule: A *schedule* specifies which schema in the schema set is to be applied at each time step.

5.2 Data Movement Library

The load-balancing system uses three data movement routines to reorganize data when switching from one schema to another. These routines, all based on a generic movement algorithm, are summarized below; they and the generic movement algorithm are described in detail in Appendix A.2.

State Reorganization: By definition, state data do not change (or at most very rarely) and hence need by moved only when the schedule calls for a schema change. The state reorganization algorithm controls the movement of state data, ensuring that transmission is minimized.

Input Exchange: The input exchange routine is responsible for transporting variables that are classified as both input and input/output. This routine also determines the number of data columns currently assigned to the processor.

Output Exchange: The output exchange algorithm is responsible for transporting variables that are classified as both input/output and output.

6 Empirical Studies

6.1 Method

The performance of the four algorithms discussed in Section 4 was measured by using an instrumented version of PCCM2. This instrumented version is based on an early release of PCCM2 in which the dynamics algorithms are not optimized. This reduces the proportion of total time taken in physics and hence the apparent impact of the load-balancing algorithms, but does not invalidate the comparison of the algorithms. In Section

Table 2: PCCM2 Performance Results for a 16×32 Mesh on the Intel Touchstone DELTA

Section	Algorithm	Time (msec)				Relative Speedup
		Full	Partial	None	Average	
Overall	bisection	3589.0	1179.5	858.6	1032.5	1.010
	movement	3592.0	1103.5	847.3	1001.8	1.041
	swapping	3489.0	1111.6	846.7	1001.0	1.042
	swapping2	3473.0	1102.8	848.2	998.9	1.044
	original	3740.0	1276.5	823.9	1043.2	
Physics	bisection	2829.0	346.5	63.7	226.9	1.185
	movement	2826.0	294.1	63.8	210.9	1.275
	swapping	2702.0	306.4	62.9	210.6	1.276
	swapping2	2700.0	296.5	63.0	207.6	1.295
	original	2998.0	504.5	47.0	268.8	

6.4, we present the performance results when the first swapping algorithm is incorporated into a more optimized PCCM2.

Several forty time step runs were performed on the Intel Touchstone DELTA at T42 resolution (64×128 grid of data columns) with the number of processors varying between 32 and 512. All of the algorithms used were specified by schema sets and schedules loaded from files. We chose to limit the runs to forty time steps since that number encompassed all types of time steps while minimizing the number of layouts that had to be buffered in memory. Although this approach allowed us to explore the diurnal cycle, we were unable to observe the effects of the seasonal cycle.

Since the movement and recursive bisection algorithms require a daypoint-nightpoint ratio, we performed several runs with different processor counts. During these runs, the time requirements for a varying number of daytime points were measured. From this information, we were able to compute the daypoint-nightpoint ratio for each of the processor counts (see Table 1).

Once the preliminary work was complete, schema sets along with schedules were generated and stored. Then, two separate sets of performance runs were performed. The first sets of runs measured the overall performance of the model. The second set gathered information about the overhead of the load-balancing system. The runs were separated to avoid probe effects from affecting overall performance measurements. A disadvantage of this approach is that the total times and measured overheads do not always add up.

Although the load-balancing system has introduced additional overhead to each time step, we find that it succeeds in reducing the load imbalance caused by the solar radiation calculations. As can be seen in Figure 3 and more vividly in Figure 4, the overall performance of the model has been improved by more than 4 percent when using the swapping algorithms on all 512 processors of the Intel Touchstone DELTA. This improvement in performance is a direct result of the near-elimination of the load imbalance within physics.

Tables 2 and 3 contain the actual execution and overhead times measured on 512 processors of the Intel Touchstone DELTA. The tables give average times over all time steps for each type of time step: full radiation, partial radiation, and non-radiation. They

Table 3: Load-Balancing Overhead for a 16×32 Mesh on the Intel Touchstone DELTA

Section	Algorithm	Time (msec)			
		Full	Partial	None	Average
Input	bisection	9.0	38.3	2.2	13.4
	movement	4.0	19.0	2.5	7.6
	swapping	20.0	19.4	2.2	8.0
	swapping2	18.0	19.4	2.2	7.9
Output	bisection	8.0	44.2	2.0	15.1
	movement	3.0	19.7	2.0	7.5
	swapping	20.0	20.0	2.0	8.0
	swapping2	19.0	19.6	2.0	7.9
State	bisection	10.0	55.0	30.4	37.4
	movement	4.0	29.3	13.2	17.8
	swapping	26.0	27.1	14.5	18.6
	swapping2	26.0	26.5	13.9	18.1
Extend	bisection	8.0	8.5	8.4	8.4
	movement	8.0	8.9	8.4	8.5
	swapping	0.0	0.0	0.0	0.0
	swapping2	0.0	0.0	0.0	0.0
Total	bisection	35.0	146.0	43.0	74.3
	movement	19.0	76.9	26.1	41.4
	swapping	66.0	66.5	18.7	34.6
	swapping2	63.0	65.5	18.1	33.9

also give the average time for all time steps. It should be noted that the latter value is not simply the average of the other three columns but is a weighted average, where the weights are based on the number of time steps executed for each type within a 24-hour period.

The overhead data in Table 3 breaks down load-balancing costs into four categories. The first three correspond to the input exchange, output exchange, and state reorganization operations described in Section 5.2, while the fourth is the copying required when moving data from the dynamics arrays to the extended arrays used in physics in the non-swapping algorithms. Note that the swapping algorithms do not incur "Extend" costs.

6.2 Results

It is clear from Figure 4 that the second swapping algorithm was the most effective on the Intel Touchstone DELTA. With this algorithm, the imbalance caused by the diurnal cycle is reduced from 6.8 percent to 0.8 percent. Clearly, this swapping algorithm succeeds in eliminating almost all of the imbalance. This result is apparent when Figures 2 and 5 are compared. Physics execution time on 512 processors, excluding the overhead of the load-balancing system, is reduced by 22.8 percent. The overall execution time for an average time step is decreased by 4.3 percent, despite the 3.3 percent of additional overhead introduced by the load-balancing system. Similar figures are also seen with the first

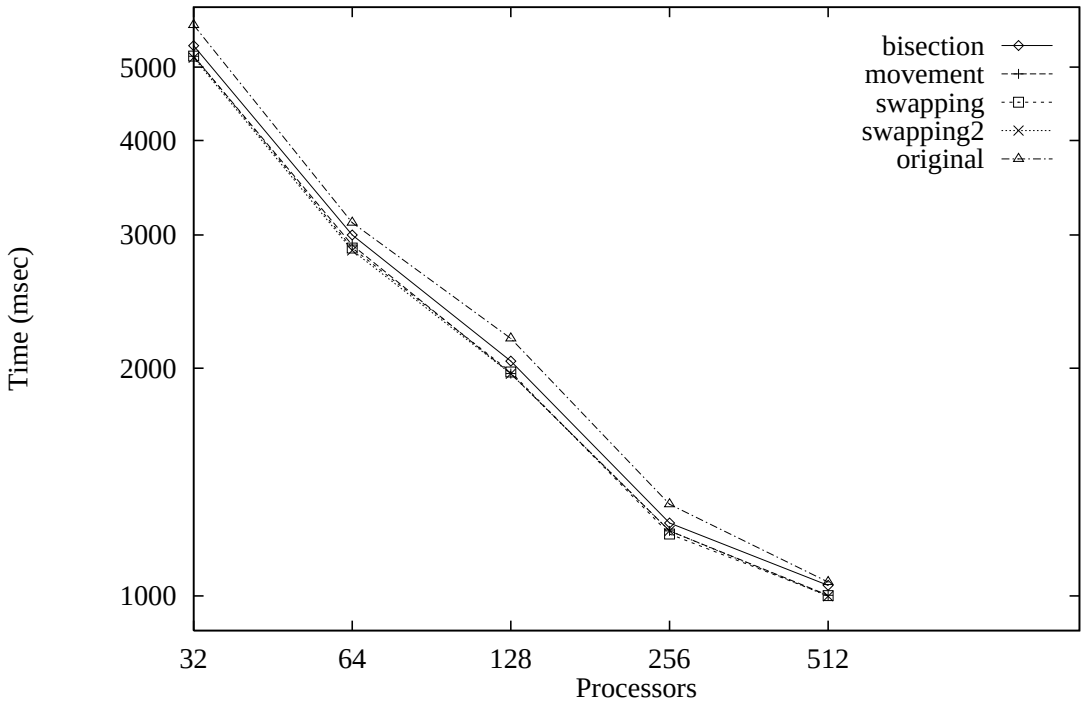

Figure 3: Overall Performance on the Intel Touchstone DELTA

swapping algorithm.

For comparative purposes, Figure 6 shows the load distribution obtained with the recursive bisection algorithm, which improved overall performance by only 1.0 percent. The reason for this algorithm's poor performance is clear: it performs much more communication than the other algorithms and, in consequence, incurs significantly higher load-balancing overheads. Additionally, the large spatial imbalance resulting from the diurnal cycle is not easily removed without reordering the columns within a latitude. Not having the ability to intersperse nighttime columns among the daytime columns, this algorithm fails to make the fine-grain adjustments necessary to balance the radiation calculations.

In theory, one would expect the movement algorithm to outperform the swapping algorithms; however, the empirical data show it to be less effective than expected. Although this algorithm communicates less data than either of the swapping algorithms, it has the additional overhead of extending the physics arrays. It also proves to be slightly less effective in balancing load. This is because it expects the computational costs associated with a daytime or nighttime column are constant. However, physics contains other imbalances besides the diurnal cycle. Although smaller in magnitude, these imbalances do have an impact, which is compensated for by swapping but not by movement.

6.3 Other Issues

The algorithms used in this study switch to an identity mapping during non-radiation time steps. Hence, physics state data are reorganized both before and after each radiation

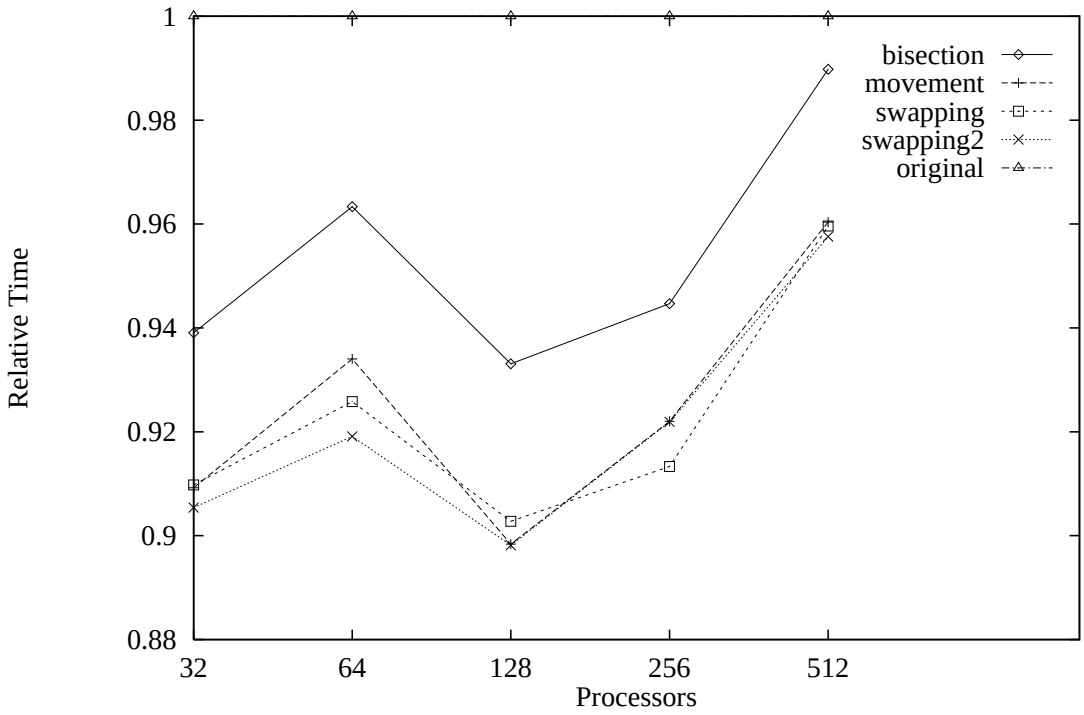

Figure 4: Relative Overall Performance on the Intel Touchstone DELTA

time step. In principle, these data could be cached on each processor in the swapping algorithm, avoiding the need for the reorganization. As can be seen by examining the partial radiation times in Table 3, however, the time required to reorganize the state data is less than the time required to exchange the input and output data on every time step. Hence, this situation is not expected to have a significant impact on performance.

Extended arrays are a source of overhead in the PCCM2 implementation of the data movement algorithms, as data must be copied to and from the extended arrays at each time step. This overhead could be avoided in a climate model that used extended arrays throughout both physics and dynamics. Our results suggest, however, that the swapping algorithm would still outperform the data movement algorithms.

While it would be possible to devise a new movement algorithm that was aware of the other imbalances that result in inefficiencies in the current movement algorithm, the additional overhead associated with this awareness would likely cancel any improvements. Additionally, for extended runs, the mappings must be generated at run time rather than precomputed and cached. Both the second swapping algorithm and the movement algorithm incur an additional overhead because a new mapping must be computed for each radiation time step. The first swapping algorithm uses only two fixed schemas and thus avoids this additional overhead.

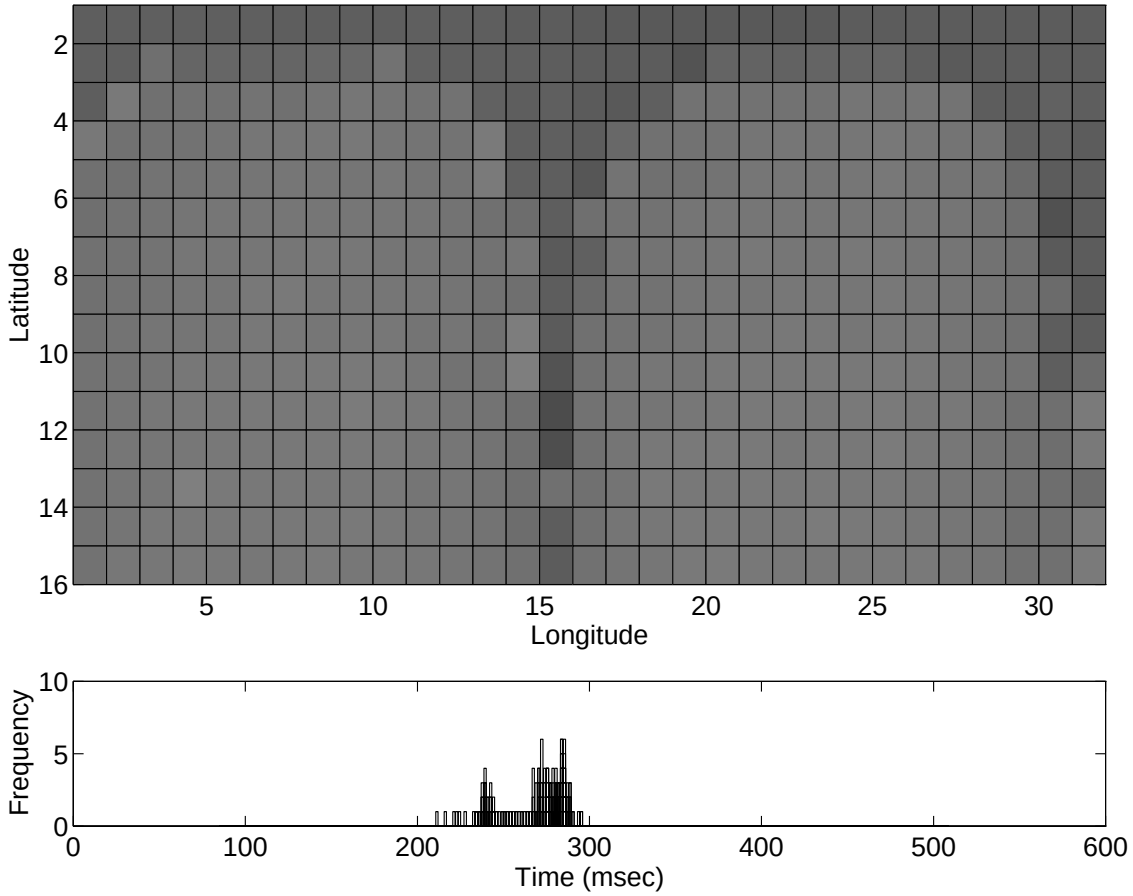

Figure 5: Physics Computation Time after Load-Balancing with the Swapping 2 Algorithm

6.4 Experiments with Optimized PCCM2

Further enhancements were made to PCCM2 independent of the version used for development of the load-balancing libraries. Many modifications were made in the dynamics portion of the code, but physics remained relatively untouched. The modifications made to the newer version of the code have resulted in a substantial performance improvement within dynamics and thus have made the physics imbalances more significant to the overall execution time. Trial runs using the first swapping algorithm indicate an overall improvement of 5.9 percent when the load-balancing code is added to the current version of the model.

7 Conclusions

The results of this work are encouraging. The swapping algorithms succeeded in significantly reducing the load imbalance, improving the total execution time by 5.8 percent. The overhead associated with the load-balancing code, however, is still rather high. In future work, we will investigate techniques for reducing the overhead. Caching state data is one possible approach. Another is to perform load balancing only on radiation physics. While less general, this requires moving far less data. In addition, the incorporation of

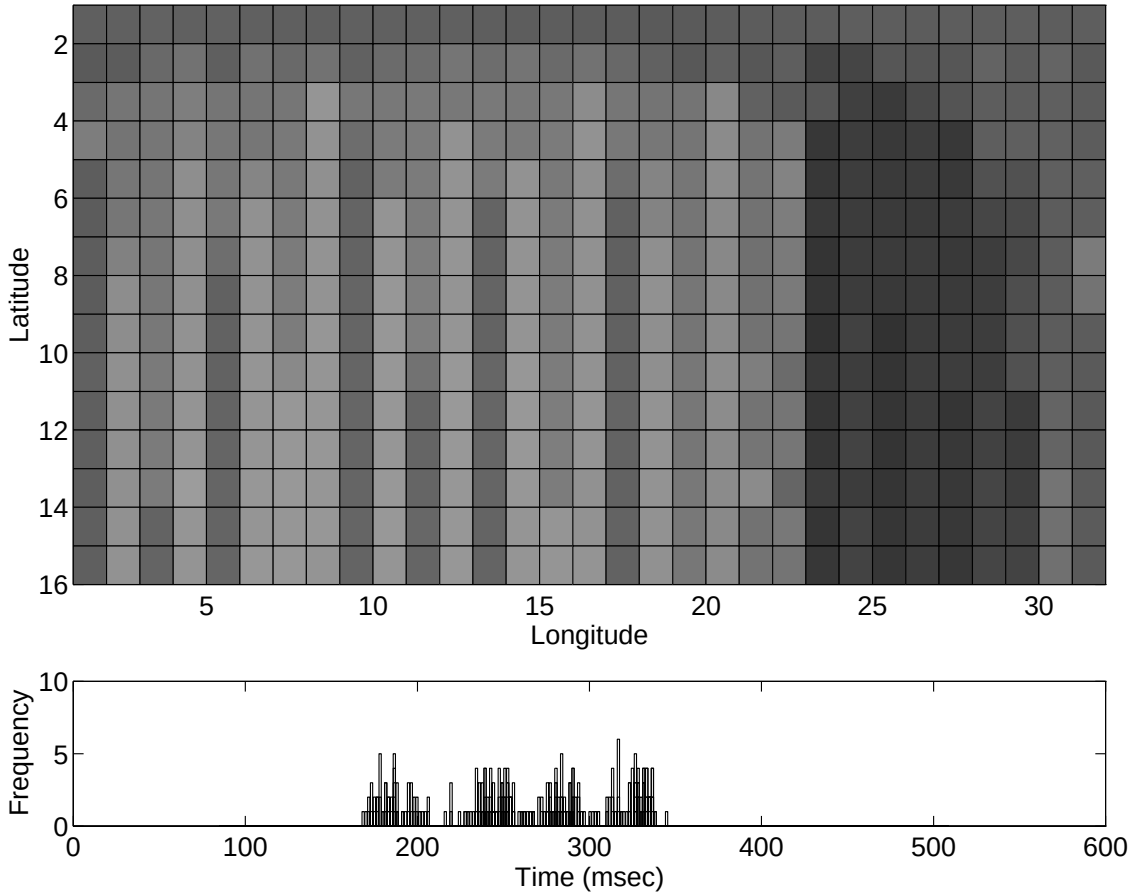

Figure 6: Physics Computation Time after Load-Balancing with the Bisection Algorithm)

a transposed-based FFT into PCCM2 appears to make it possible to integrate input and output data movement into the transpose operation used to move from latitude/vertical decomposition to latitude/longitude decomposition.

A Library Algorithms

We describe two algorithms used within the data movement library. The first generates a layout from a schema, and the second determines the communication required to move from one layout to another.

A.1 Layout Generation Algorithm

The layout generation algorithm generates a unique layout from a valid schema. Subsequent algorithms use layouts to determine data movement requirements.

A *layout* is a $N_{glat} \times N_{glon}$ array of $(processor, latitude, index)$ triples that define the exact mapping of each physics data column to a processor and data space. A column is said to be *on-processor* if the mapping places it on the same processor as specified by the identity schema. A column that is not on-processor is referred to as an *off-processor*

column. A valid layout does not include any discontinuities or "holes" in the data space mappings. In other words, the data columns must be packed to the left in each latitude on each processor. Finally, on-processor columns must remain in their initial or "home" location whenever possible.

The algorithm operates as follows:

1. Scan the entire schema, and count the number of local (on-processor), L, and foreign (off-processor), F, columns for each latitude on that processor. If on any latitude $L + F > N_{llonx}$, then signal an error.

2. Scan each latitude of the processor's component of the schema, left to right, and place on-processor columns in their "home" location. If more than F discontinuities exist in the layout for the given processor and latitude, then the rightmost on-processor columns should be moved to fill these excess "holes."

3. Scan each latitude of the schema, left to right, and place each foreign column encountered in the layout, selecting first areas of discontinuity and then free locations as the destination.

It should be noted that although the algorithm attempts to place each on-processor column at the same location as the corresponding dynamics column, success is not guaranteed. Consider the following schema:

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

Execution of algorithm `schema_to_layout` generates the following layout for this schema:

1,1,1	1,1,2	1,1,3	1,1,4	2,1,1	1,1,5	2,1,2	1,1,6
1,2,1	1,2,1	1,2,3	1,2,4	2,2,1	2,2,2	2,2,3	2,2,4
3,1,1	3,1,2	3,1,3	3,1,4	4,1,1	4,1,2	3,1,5	3,1,6
3,2,1	3,2,2	3,2,3	3,2,4	4,2,1	4,2,2	4,2,3	3,2,5

A.2 Generic Data Movement Algorithm

The generic data movement algorithm determines the communication required to move from layout L_1 to layout L_2. In order to minimize communication costs on computers with high message startup costs, it packs all data to be sent to a given processor into a single message.

1. **PostReceives:** determine which columns will be sent to the current processor from other processors, and inform the message-passing system of expected messages.

   (a) Create a receive list identifying the columns that will be sent by other processors.

      i. Initialize the receive list and a temporary transfer list to an empty state.

ii. For each $L_2(i,j) = (m,k,p)$ with $m = P$, where P is the current processor:
   A. Find $L_1(i,j) = (n,l,o)$.
   B. If $m \neq n$, add (i,j) to the transfer list.
iii. Sort the transfer list by n, the processor to be transmitted to.
iv. For each processor P:
   A. Scan the transfer list, counting the number of items transmitted, t, for processor P.
   B. If $t > 0$, add (P,t) to the receive list.
v. Save the receive list for later use by ReceiveColumns.

(b) For each entry (P,t) in the receive list:
i. Allocate a message buffer containing enough space for t data columns and addresses.
ii. Inform the message–passing system as to each buffer's location.

2. **SendColumns:** determine which currently on-processor columns are to be sent to other processors, compose a message containing those columns, and then send the messages.

(a) Create a transfer list that contains the columns present on this processor that need to be sent to other processors.
i. Initialize the transfer list to an empty state
ii. For each entry $L_1(i,j) = (n,l,o)$ with $n = P$, where P is the current processor:
   A. Find $L_2(i,j) = (m,k,p)$.
   B. If $m \neq n$, then add (i,j) to the list of columns to transmit.
iii. Sort the transfer list by m, the processor to be transmitted to.

(b) Using the transfer list, compose and send a message for each processor to which one more columns must be sent. Each message should contain the data of the local columns to be sent and their corresponding (k,p) addresses.

3. **LocalReorganization:** move columns that are remaining on-processor but need to change location in the data storage arrays. This can be accomplished invoking the following algorithm for every entry $L_1(i,j) = (n,l,o)$ with $n = P$ where P is the current processor:

(a) Find $L_2(i,j) = (m,k,p)$.
(b) If $m = n$, move the data for column (i,j) from (l,o) in the data array space to (k,p). (Note: given the current restrictions, $l = k$ will always be true)

4. **ReceiveColumns:** complete the receive process by waiting for messages to arrive and placing the data contained within the messages into the appropriate data storage arrays. The following algorithm should be executed for each (P,t) item in the receive list saved in PostReceives:

(a) Query the message–passing system for the arrival of a message from processor P waiting until one arrives.

(b) For each of the t columns of data in the preassigned message buffer:

i. Extract the data and the (k, p) address for that column.

ii. Place data into the data arrays at location (k, p).

In the context of PCCM2, the generic movement algorithm is used by the following algorithms.

State Reorganization: This algorithm is a modified version of the generic movement algorithm that maintains L_1 internally as the last layout used. This algorithm exits immediately unless $L_1 \neq L_2$.

Input Exchange: This algorithm uses the generic movement algorithm with L_1 always set to the identity schema. It also returns the number of columns assigned to a processor.

Output Exchange: This algorithm is a specific instantiation of the generic movement algorithm that sets L_2 to always be the identity schema.

B Using the Library

A library of data transport routines has been implemented and integrated into PCCM2. This library is responsible for taking data mappings from the load-balancing system and performing the necessary data transfers to obtain those mappings. In PCCM2, the library is initialized within the load-balancing startup code, `lbsetup`, which is called from the main PCCM2 routine, `ccm2`. The use of the load-balancing system by PCCM2 can be controlled using the C preprocessor macro `PP_LOAD_BALANCE`. When `PP_LOAD_BALANCE` is set to one, the load-balancing system is enabled; any other value results in its being disabled.

B.1 Schemas

In the current implementation of the load-balancing algorithms, schemas can be either loaded from a schema set file during program initialization or generated by user-supplied code during model execution. The method used is determined at compile time by the C preprocessor macro `LB_SCHEMA_GEN`. If this is defined, then the system expects that some form of code exists that will supply the schemas to the load-balancing system; otherwise, the load-balancing code will expect to find the schemas defined in a schema set file.

If the file method is used, the load-balancing initialization code calls `layout_set`. The `layout_set` routine repeatedly calls the routines `schema_read` and `layout_generate` in order to generate a set of layouts from the set of schemas found in the file schema.set. The schema.set file is read by `schema_read`, which processes the text file by reading integer values until it has obtained $N_{glat} \times N_{glon}$ values. It returns these values in a two-dimensional array. The next time the routine is called, it reads the next schema in the file and returns it providing one is found. Upon reaching end of file, `schema_read` will return a status of false causing `layout_set` to terminate. The schema.set file may be organized in any way

that proves to be visually pleasing to the user as long as the first schema defined is an identity schema and there are exactly $n \times N_{glat} \times N_{glon}$ integer elements present in the file where n is a positive integer.

If `LB_SCHEMA_GEN` is defined, then the load-balancing code expects that a routine external to the load-balancing system will be generating schemas. The schema generation routine then uses `layout_replace` to create a layout from the schema and register the layout with the load-balancing system. Like `layout_set`, `layout_replace` uses `layout_generate` to generate a corresponding layout from the supplied schema. As with the schema file method, an identity schema must be registered as the first schema.

B.2 Layouts

As stated earlier, schemas may be supplied to the load-balancing system by two different mechanisms: schema files or runtime generation. In either case, the schemas are converted to layouts by `layout_generate`. The information generated by the `layout_generate` routine is used by the data movement routines.

The layouts currently registered with the load-balancing system are stored in the variable `Layouts` defined in `layout.com`. The `Layouts` variable is a four-dimensional array with dimensions of *global latitude*, *global longitude*, *information type*, and *layout number*. Given one of the following values for the *information type* dimension, all of the necessary information can be obtained about a given layout.

LAYOUT_PROC: processor to which the data element is to be moved

LAYOUT_ROW: the local latitude on which the data element is to be placed

LAYOUT_COLUMN: the local longitude or column on which the data element is to be placed

B.3 Extended Arrays

Because some nonidentity schemas cause some processors to acquire additional physics data columns, it was necessary to extend the length of data arrays to accommodate the additional data. Although separate arrays could be allocated for the additional data, this approach would result in unnecessarily complex modifications to the physics code as well as poorer performance. The lengthening was achieved by (a) changing the dimension of the various physics data arrays from `p_lond` to a larger value `p_londx` (N_{llonx}), and (b) changing loops over these arrays to range from 1 to `m_nlonx`. The changes were required in all subroutines contained within the `phys` call tree. The existence of the extended array additions with PCCM2 are controlled by the definition of the C preprocessor macro `PHYS_EXTEND_ARRAYS`.

B.4 Data Movement Routines

The data elements in the PCCM2 computational grid are reorganized by using three routines: `state_reorg`, `input_exchange`, and `output_exchange`. The `state_reorg` routine is responsible for reorganizing the state information whenever the schedule dictates that a new schema is to be used. The `input_exchange` and `output_exchange` are responsible for reorganizing input and output information, respectively. They also share the task of

Table B-1: Reorganization subroutines

	Post	Send	Local	Receive
State	`state_post_recv`	`state_send`	`state_local`	`state_receive`
Input	`input_post_recv`	`input_send`	`input_local`	`input_receive`
Output	`output_post_recv`	`output_send`	`output_local`	`output_receive`

reorganizing input/output information. The `input_exchange` routine is called prior to the `phys` routine, while `output_exchange` is called after `phys`.

Each of the three data movement routines is broken down into four subroutines: *post for receive*, *send data elements*, *local reorganization*, and *receive data elements*. Table B-1 contains the routine names of these subroutines as they correspond to the three data movement types. The posting routines call `get_transfer_list` to obtain a list of expect data columns, compose a receive list from the transfer list, store the receive list in `Recv_List`, allocate sufficient memory from a buffer to receive the data, and tell the message-passing system what messages are expected and where in the buffer to place them when they arrive. The send routines call `get_transfer_list` to obtain a list of points to be sent and then, prior to sending, pack the data columns together so that only one message is sent to any given processor. The local reorganization routines move data columns that are local to a given processor but are no longer in the correct location in the data arrays. The receive routines, using the information stored in `Recv_List`, wait until expected messages have arrived and then unpack the messages, placing them in the specified locations within the data arrays.

Acknowledgment

Access to the Intel Touchstone DELTA was provided by the Concurrent Supercomputing Consortium.

References

[1] Bath, L., Olson, J., and Rosinski, J. User's Guide to NCAR CCM2. National Center for Atmospheric Research, Boulder, Colorado. 1992.

[2] Drake, J., Walker, D., and Worley, P. Parallelizing the Spectral Transform Method – Part II, ORNL/TM-11855. Oak Ridge National Laboratory, Oak Ridge, Tennessee. 1991.

[3] Drake, J., Foster, I., Hack, J., Michalakes, J., Semeraro, B., Toonen, B., Williamson, D., and Worley, P. PCCM2: A GCM Adapted for Scalable Parallel Computers. *Proc. AMS Annual Meeting*, AMS. 1994.

[4] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., and Walker D. *Solving Problems on Concurrent Processors*. Prentice-Hall. 1988.

[5] Michalakes, J. Analysis of Workload and Load Balancing Issues in the NCAR Community Climate Model, ANL/MCS-TM-144. Argonne National Laboratory, Argonne, Illinois, 1991.

[6] Michalakes, J., and Stevens, R. Analysis of Computational Load Distribution in the NCAR Community Climate Model. *Computer Hardware, Advanced Mathematics and Model Physics Pilot Project Final Report*, DOE/ER–0541T, pages 19–24. U.S. Department of Energy. 1992.