High-Performance Computing and Four-Dimensional Data Assimilation: The Impact on Future and Current Problems
Final Report

Miloje S. Makivić*
Project Leader
(July 11, 1996)

Abstract

This is the final technical report for the project entitled: "High-Performance Computing and Four-Dimensional Data Assimulation: The Impact on Future and Current Problems", funded at NPAC by the DAO at NASA/GSFC. First, the motivation for the project is given in the introductory section, followed by the executive summary of major accomplishments and the list of project-related publications. Detailed analysis and description of research results is given in subsequent chapters and in the Appendix.

---

*Principal Investigator: Prof. Geoffrey C. Fox
Northeast Parallel Architectures Center (NPAC)
Syracuse University
111 College Place, Syracuse, NY 13244-4100
e-mail: gcf@npac.syr.edu, miloje@npac.syr.edu
Mosaic http://www.npac.syr.edu/

I. INTRODUCTION

The Goddard Data Assimilation Office (DAO) is engaged in on-going operations and development using four dimensional data assimilation (4DDA). It is being increasingly recognized that 4DDA methods are likely to play an essential role in the understanding of processes involved in the earth system ([1]). Data Assimilation involves using the latest models of the atmosphere and ocean along with estimation theoretic methods for melding actual observational data (atmospheric, land surface, and oceanic) into the models so as to obtain a best estimate of the evolving state of the system. This best estimate is evaluated as large datasets that are produced every three or six hours for mulit-year periods of time, corresponding to past as well as up-to-date observations. The datasets are physically consistent, gridded values of important physical quantities (wind speed, temperature, water vapor and trace chemical mixing ratios etc.). In some cases accurate values of unobserved quantities can be provided, or values at the positions of the observations that are more accurate than individual observations themselves. In others, the model equations help to provide physical consistency constraints between distinctly different variables [2]. One of the main functions of the DAO is to provide the datasets that result from these analyses to the large scientific community as part of NASA's support for the national research effort into the earth's climate as well as global change. In the future most of the data for our assimilations will be provided by satellites as part of NASA's Mission To Planet Earth (MTPE) project [1].

The computational tasks are similar to those of operational weather prediction centers, but the computer requirements are probably larger because of the comprehensive data sets that must be provided by the DAO to the scientific community. Apart from the computing needs of scientific and ongoing development research at the DAO, one of the main users of resources is the 'Reanalysis' production projects. This is the process of running a multi-year assimilation using historical data and a 'frozen' assimilation system. Typically the DAO will run a complete reanalysis for the period 1979-present about once every two years. This will be done to take advantage of scientific and software improvements in the system. At present the frozen version of the system is GEOS-1 DAS. An important metric of the DAO's work is the number of days assimilated per wall clock day. At present, the reanalysis effort uses about 20 percent of the resources of the Cray C98 at Goddard. In this mode GEOS-1 DAS produces about 30 days of assimilated datasets per day of wall-clock time. This metric should be maintained even in the context of greater requirements such as: higher resolution – the DAO is planning on running at 1 degree (longitude) by 1.25 degree (latitude) by 40 levels by 1998; apart from archived data one can expect that new data will increasingly become a strain on system resources – the GEOS DAS system ingests about 30 megabytes per assimilated day at present, and this is expected to increase by an order of magnitude by 1998; finally, the more advanced Physical space Stastical Analysis System (PSAS) that is at present being developed at the DAO will put more pressure on the data and computing resources in the future. At present the production system (GEOS-1 DAS) uses a regionalized Optimal Interpolation (OI) analysis system. In this algorithm only observations in a limited range of a model gridpoint are used to peform an analysis. While this has been a successful approach for both climate and weather assimilation it has become clear that a global approach is needed to obtain greater accuracy and to further the proposed advanced assimilation schemesi (such as the Kalman filter). PSAS is the DAO's global assimilation system that is under development. Computational issues resolved during the analysis and porting of the GEOS-1 DAS are very much relevant for the successful implementation of PSAS on future scalable HPC platforms. The production system for PSAS is almost complete, and is expected to replace OI at the DAO during 1996.

The DAO is developing a strategy for computing requirements in the next decade. One of the major goals of the present 4DDA project was to ensure a smooth transition of DAO operations to future scalable HPC platforms that will support orders of magnitude bigger comutational demands on the production assimilation system such as PSAS . This involves:

1. An examination of the potential for parallelization and scalability of operational algorithms at DAO,
2. An examination of the potential for parallelization and scalability of candidate future algorithms at DAO, and
3. Porting and benchmarking of existing codes on testbed parallel architectures.

At the Northeast Parallel Architectures Center (NPAC), these tasks were associated with the following components of the complete assimilation system:

1. Operational Mini-Volume Optimal Interpolation Analysis
2. Advection Algorithms for transport and GCM (Eulerian and Semi-Lagrangian)

The following two sections supply an overview of our major accomplishments and conclusions regarding the software modules of the DAO assimilation system which were assigned to NPAC for evaluation, analysis and parallel development. The overview is followed by a list of project publications (including the material available on-line through the WWW). Subsequent chapters supply our research results and conclusions in the form of detailed analysis of computational issues for each software module.

II. OVERVIEW OF MAJOR ACCOMPLISHMENTS

A. Optimal Interpolation (OI)

A mini-volume based optimal interpolation algorithm is currently used at the NASA Goddard Data Assimilation Office (DAO) for four dimensional data assimilation. Instead of using all observations to update a model forecast at all of the model grid points, the analysis is performed on groups of grid points called mini-volumes. Only selected observations in a preset vicinity of each mini-volume are used in each regional analysis.

We have explored two complementary approaches to parallelization of the OI code:

1. Task parallel approach
2. Data parallel approach

Task parallel approach is implemented as a Fortran 90 code with MPI message-passing library routines. It required substantial rewriting of the original Cray F77 code. Restructuring of the sequential code was an essential requirement for porting to parallel machines and it produced important improvements in the sequential algorithm as well. For example, by sorting observational data along latitudes and longitudes, we achieved an improvement by a factor of $log(N)/N$ in CPU time for sequential quality control rotines, where N is the number of observational data items.

The resultant code is modular, easily extendable and highly portable, since it is built on standards like Fortran 90 and MPI. On top of the original code, we have developed general-purpose utilities for efficient distribution of data structures and computational load in a heterogeneous metacomputer environment. These utilities are expected to be not only useful for the data decompostion required for the newer PSAS algorithm, but have also provided a viable strategy to the Suarez-HPCC (1992) grand challenge team on the parallelization of their development oceanic data assimilation system that uses the Optimal Interpolation method. The parallel implementation is well suited for a clustered environment or any distributed environment with large grain size. This work forms a basis of the Ph. D. thesis of G. von Laszewski, who is a graduate student at NPAC funded by the DAO.

Task parallel approach is based on the fact that construction and solution of the linear system of covariance matrices can be done independently for each mini-volume. Our parallel programming strategy partitions all observations and model (grid) data among processors at the beginning of the analysis. Thereafter, individual processors perform the analysis for the mini-volumes assigned to them independently from one another. While this brings maximum speed-up for the solver, it also leads to substantial data replication and redundant computations during the quality control phase.

We performed benchmarks on IBM SP-2 for up to 40 processors and obtained very encouraging results. Extensive analysis of the speed-up curves for core quality control and analysis routines are presented in the Appendix. Analysis efficiency is in the 50 for moderate machine sizes (up to 40 processors). Incorporaton of dynamic load balancing mechanisms we have developed, will bring the efficiency much closer to the theoretical limit of 100%. We expect that the load-balanced code will have excellent scaling behavior in the massively parallel limit as well (hundreds of processors).

The quality control routines also show excellent initial speed-up, but then saturates in the massively parallel limit. The primary reason can be traced to redundant computations which have to be performed due to the overlap regions between mini-volumes. As the processor number increases, computations within the fixed size overlap region overwhelm the computations within each processor's domain. We showed how to improve the scalability of quality control by exploring alternative grid partitions, as well as distributing the vertical dimension.

The problem of redundant computations in the quality control phase was also approached using a different strategy based on data parallel programming model. We showed that quality control can be efficiently implemented using a language such as High Performance Fortran. We parallelized sea-level quality control using CM Fortran, which is a dialect of HPF available on Connection Machine series. We achieved 0.5 GFLOP sustained and 3.1 GFLOP peak on a core quality control routine without any code tuning and on a small data set which did not make efficient use of the vector units on a 256 node Connection Machine CM-5. This approximated, and even exceeded, the metric that we set in negotiations at the beginning of DAO-HPCC (1992), where we estimated performance of 0.5 gigaflop/s for key modules.

The implications of the above described work on the implementation of the quality control modules is relevant to PSAS as much as to OI. This work is described in NPAC Technical Reports SCCS-713, SCCS-668 and in a paper submitted to ECMWF conference. Substantial amount of information is available on-line, via http://www.npac.syr.edu/projects/nasa. Certain generic routines which had to be developed for the HPF implementation (e.g. segmented bubble sort) are also available on-line via NPAC's High-Performance Fortran Applications repository, http://www.npac.syr.edu/hpfa.

B. Transport

We have explored parallelization potential of both Eulerian and Semi-Lagrangian schemes for treatment of advection. Eulerian schemes were examined in the context of van Leer/ Prather chemistry transport DAO codes. Semi-Lagrangian algorithm was abstracted from the experimental DAO shallow water code.

Eulerian schemes are based on stencil computations so they can be very efficiently implemented in HPF. On some hardware these algorithms can take advantage of physical proximity of processors in the communication topology (MasPar, CM-5, Intel Paragon).

All aspects of the computation could be adequately expressed using array syntax of Fortran 90 and HPF standard library. We have both HPF and CM Fortran versions of the parallel code. We developed a strategy based on gather/scatter routines to deal with load imbalances due to polar sub-cycling. We achieved 2.5 GFLOPS sustained and 6.8 GFLOPS peak on a 256 node CM-5. Due to excellent scalability, performance can be significantly higher on bigger machines and bigger problems. Resultant parallel code was integrated using AVS framework into a Simulation-on-Demand demo at NPAC and is available via http://www.npac.syr.edu/.

On the other hand, Semi-Lagrangian algorithm can be efficiently implemented only using a message-passing approach. It presents serious challenges for data parallel languages and compilers, due to communication imbalances, unstructured communications and simulation driven (i.e. dynamic) communication patterns. The core interpolation procedures in the Semi-Lagrangian approach are dominated by high-communication costs and parallel overheads due to memory management and indexing. We have implemented a one-dimensional decomposition along lattitudes, which communicates whole longitudinal strips between nodes which are configured as a logical ring. This approach takes advantage of the fact that advection winds are mostly east-west, but it lacks scalability. The FLOP rate is low because of the low FLOP count in the interpolation in comparison to book-keeping portions of the code. Our experience with 1D decomposition shows that at high resolutions memory management and local integer computation for indexing even becomes more important than communication overhead. This is one example where FLOP rate may be completely misleading as a measure of succesful parallelization. The analysis of the parallelization of the shallow water semi-Lagrangian code is available as NPAC Technical Report SCCS-694.

III. PROJECT-RELATED PUBLICATIONS

[1] von Laszewski, G., M. Seablom, M. Makivic, P.M. Lyster, S. Ranks, 1995: Design Issues for the Parallelization of an Optimal Interpolation Algorithm, *Coming of Age: Proceedings of the Sixth ECMWF Workshop on the Use of Parallel Processors in Meteorology*, eds. G-R. Hoffman and N. Kreitz, World Scientific, 290-302, NPAC Technical Report # 668, to be submitted to *Concurrency: Practice and Experience*.

[2] Natarajan, M. and M. Makivic, 1995: Case Studies in Parallelization of NASA Shallow Water Code: Results on Interpolation Techniques, *NPAC Technical Report # 694*.

[3] von Laszewski, G., M. Makivic and G. C. Fox, 1996: Interactive Parallel Program Generation, submitted to IEEE Proceedings of Frontiers'96. IEEE, October 1996. http://www.npac.syr.edu/users/gregor/frontiers.

[4] von Laszewski, G., 1995: Parallel Optimal Interpolation for Atmospheric Science, *NPAC Technical Report # 713*.

[5] Makivic, M. and Lyster, P., 1996: Experiments in Parallelization of Eulerian and Semi-Lagrangian Transport Algorithms, in preparation for *Computers in Physics*.

[6] Project Home Page *http://www.npac.syr.edu/projects/nasa/home.html*

[7] Source Codes *http://www.npac.syr.edu/hpfa/*

IV. DATA PARALLEL IMPLEMENTATION OF EULERIAN TRANSPORT ALGORITHMS

Changes in the atmosphere produced by trace gasses may have profound environmental effects. Realization of this fact renewed interest in development of constituent transport and chemistry models. Realistic modeling of atmospheric transport is a large scale computational problem which can efficiently use massively parallel computers. This Section summarizes our experiences and conclusions regarding the implementation of transport models on parallel computers. Its focus is on one aspect of the models which is most significant from parallel computing point of view: treatment of advection. Focus on advection is also motivated by the fact that it is important component in Global Circulation Models.

There is a number of important components of the complete problem which will be ignored for various reasons. For example, we will not be concerned with chemistry, because it is local and therefore trivially parallel. Diffusion processes on the other hand can be classified as stencil computations, just like advection, which implies that they do not bring new problems beyond those encountered in advection. Finally, there is a completely different set of issues related to I/O and system integration, which is given in the Section on Metacomputing and Data Assimilation. Particularly interesting problem is the interface between transport models and assimilated data sets in the complete data assimilation system, shown on the data flow diagram of the complete assimilation system in the Appendix.

The parallel system was evolved from the sequential transport code [3] developed at NASA/Goddard, which employed Van Leer scheme for horizontal advection and Prather scheme for advection in the vertical. We will first briefly describe the basics of these two algorithms. A thorough exposition of van Leer method and its application to the atmospheric transport problems is given in [3]. Then we will describe High Performance Fortran (HPF) programming model [13], followed by the exposition of its features which were used to implement advection algorithms. A complete set of tutorials and reference materials on HPF can be obtained on-line [12]. We will compare HPF implementation with a message passing (MP) implementation of Van Leer algorithm. Finally, we will generalize our conclusions to similar scientific applications.

A. Continuity Equation

The fundamental equation for modeling dynamics of ozone and other chemical constituents in the atmosphere is the mixing ratio continuty equation. The constituent mixing ratio $D = C/M$ is defined as the ratio of constituent density C and the background air density M. At NASA/Goddard Space Flight Center, continuity equation was adopted to spherical coordinates of their assimilated data model, which provides horizontal wind components and temperature field as inputs for the transport code.

The coordinates of the model are: latitide ϕ and longitude λ in horizontal and σ pressure levels in the vertical. We will also refer to the latitude, longitude and vertical direction as X, Y and Z respectively. Horizontal coordinates are partitioned onto a constant spacing $(\Delta\phi, \Delta\lambda)$ latitude- longitude grid. Earth radius is denoted by R. Vertical coordinate has eight sigma levels in the troposphere and eleven sigma levels in the stratosphere. The interface pressure is $p_{int} = 120$mb and the topmost level has pressure $p_{top} = 0.305$mb. If we denote by p_k pressure at level k of the vertical grid and by p_{sfc} the surface pressure, then sigma coordinate for k-th level on the vertical grid is given by $\sigma_k = (p_k - p_{int})/\Pi$. Π is the pressure difference which is given by $\Pi = p_{int} - p_{top}$ in the stratosphere and $\Pi = p_{sfc} - p_{int}$ in the troposphere. Wind components will be denoted by U and V in latitude and longitude direction respectively, while the vertical wind components is W.

Since we are interested in the implementation of advection algorithms only, we will not include local chemical sources and sinks. With this restriction, the three-dimensional continuity equation for the constituent mixing ratio is:

$$-\Pi\frac{\partial D}{\partial t} = D\frac{\partial \Pi}{\partial t} + \frac{1}{R}\frac{\partial}{\partial \phi}(\Pi U D) + \frac{1}{R\cos\phi}\frac{\partial}{\partial \lambda}(\Pi\cos\phi V D) + \Pi\frac{\partial}{\partial \sigma}(W D) \tag{1}$$

The first term on the right-hand side describes changes in mixing ratio due to pressure variation. The second and third terms arise from horizontal advection, and the last term is due to vertical advection. We will discuss later each of the terms in this equation in detail, since each one of them imposes a different computation and communication structure when the variables are represented by arrays and mapped onto a parallel architecture. First we turn to the description of advection algorithms.

B. Van Leer Algorithm

Horizontal advection terms are implemented using the van Leer algorithm, which is an upstream-biased monotonic gridpoint transport scheme. It exhibits self-limiting diffusion, but the upstream bias gives it a pseudo-lagrangian character which drastically reduces phase errors. Among its major advantages are its simplicity and known ability to time-split. Its local nature provides good error propagation characteristics. This is very important when transport winds are obtained from assimilated data sets, which have a lot of noise in the polar regions. Because it is monotonic, it does not require filling to preserve positive constituent densities, which is important if correlations among constituents have to be conserved.

We will describe the algorithm using the one dimensional advection equation:

$$\frac{\partial C}{\partial t} + \frac{\partial (uC)}{\partial x} = 0 \tag{2}$$

which is discretized as:

$$C^{n+1}_{j+1/2} = C^{n}_{j+1/2} - \frac{\Delta t}{\Delta x_{j+1/2}}(\Psi_{j+1} - \Psi_j) \tag{3}$$

The notation is as follows: integer index j denotes boundary between adjacent cells, where time-averaged fluxes Ψ_j and velocities u_j are defined, index $j+1/2$ denotes cell centers, where density $C_{j+1/2}$ is defined, n is the time-step index, while Δt and $\Delta x_{j+1/2}$ denote, as usual, temporal and spatial grid intervals.

Time-averaged fluxes are given by:

$$\Psi_j = u_j \left[C^n_r - \frac{1}{2}\left(\frac{u_j \Delta t}{\Delta x_r} - \mathrm{sgn}(u_j)\right) \Delta_r C \right] \tag{4}$$

where $r = j - (1/2)\mathrm{sgn}(u_j)$. One possible choice for the finite difference operator $\Delta_r C$ which guarantees monotonicity is given by (θ denotes step function):

$$\Delta_r C = 2(C_r - C_{r-1})(C_{r+1} - C_r)\theta((C_r - C_{r-1})(C_{r+1} - C_r))/(C_{r+1} - C_{r-1}) \tag{5}$$

Equations 3, 4 and 5 constitute basic equations of the van Leer method. Horizontal advection is performed for latitude and longitude directions independently and the corresponding changes in the mixing ratio are added before the vertical advection step is performed.

C. Prather Algorithm

Prather algorithm is non-diffusive, it satisfies conservation laws by construction, and it is stable up to the CFL limit. It is based on conservation of second-order moments of constituent mixing ratio D during advection. The latitude-longitude-sigma pressure level grid divides the whole physical volume into a number of small boxes. The extent of k-th box along σ-coordinate direction is $\delta = \sigma_{k+1} - \sigma_k$. Within each box, mixing ratio is represented by a linear combination of orthogonal second order polynomials:

$$D(\sigma) = m_0 D_0(\sigma) + m_1 D_1(\sigma) + m_2 D_2(\sigma) \tag{6}$$

where

$$\begin{aligned} D_0(\sigma) &= 1 \\ D_1(\sigma) &= \sigma - \delta/2 \\ D_2(\sigma) &= \sigma^2 - \delta\sigma + \delta^2/6 \end{aligned} \tag{7}$$

Since we are using this method to treat vertical advection only, there is no horizontal spatial dependance within a box. Moments S_i of the mixing ratio distribution are obtained as integrals over a box volume:

$$S_i = \int d\sigma D(\sigma) D_i(\sigma) = \frac{m_i \delta}{(i+1)!} \tag{8}$$

The advection step is treated as a process of deconstructing and reconstructing boxes in such a way that second-order moments are conserved. For the sake of simplicity, we will assume that vertical wind component W is in the direction of increasing σ. Each box (k) is split into two sub-boxes: (1) the first sub-box contains mass which will remain in the box and its bounds are given by $\sigma_k \leq \sigma < \sigma_{k+1} - W_k \Delta t$, (2) the second sub-box contains mass which will be transported to the downstream box ($k+1$), and it is bound by $\sigma_{k+1} - W_k \Delta t \leq \sigma_{k+1}$. For each of the sub-boxes one represents mixing ratio distribution using an appropriate set of orthogonal polynomials (c.f. equation 7), with respect to the sub-box's own center of mass. The corresponding moments for each of the sub-boxes will be denoted by superscripts L (for the first sub-box) and U (for the second sub-box). If we denote by α the ratio of volumes of the U (i.e. transported) sub-box to the original box, the moments of the sub-boxes can be reconstructed from the moments of the original box:

$$\begin{aligned} S_0^U &= \alpha\left[S_0 + (1-\alpha)S_1 + (1-\alpha)(1-2\alpha)S_2\right] \\ S_1^U &= \alpha^2\left[S_1 + 3(1-\alpha)S_2\right] \\ S_2^U &= \alpha^3 S_2 \\ S_0^L &= (1-\alpha)\left[S_0 - \alpha S_1 - \alpha(1-2\alpha)S_2\right] \\ S_1^L &= (1-\alpha)^2(S_1 - 3\alpha S_2) \\ S_2^L &= (1-\alpha)^3 S_2 \end{aligned} \tag{9}$$

Prather advection step consists of computing sub-moments U and L for each of the grid boxes and then transporting the U moments to the respective downstream boxes. The distribution after the advection step is obtained by reconstructing new moments in each grid box from the mass that was left in a box during the advection step (L moments) and the newly transported mass (U moments from the upstream box). This implies that sub-moments of the reconstructed box $(k+1)$ after the advection step are:

$$S_i^L(k+1) = S_i^U(k) \qquad S_i^U(k+1) = S_i^L(k+1) \tag{10}$$

Given these sub-moments one can compute the moments of the complete grid box:

$$\begin{aligned} S_0 &= S_0^U + S_0^L \\ S_1 &= \alpha S_1^U + (1-\alpha)S_1^L + 3(1-\alpha)S_0^U - 3\alpha S_0^L \\ S_2 &= \alpha^2 S_2^U + (1-\alpha)^2 S_2^L + 5\alpha(1-\alpha)(S_1^U - S_1^L) + 5(1-2\alpha)(1-\alpha)S_0^U - 5\alpha(1-2\alpha)S_0^L \end{aligned} \tag{11}$$

Given the new moments of the full grid box, one can easily compute moment coefficients m_i and the quadratic polynomial which describes new mixing ratio distribution (c.f. equations 6 and 8). All moments must be initialized prior to the calculation. In case of full three-dimensional Prather advection there are alltogether 12 moments, including the off-diagonal ones (xy,yz,zx).

D. Parallel Implementation Strategies

Extrapolation of current trends in high-performance computing indicates that Multiple-Instruction Multiple-Data distributed-memory architectures will probably be the most cost-effective hardware platforms. This includes tightly coupled systems as well as networked workstations, which are conceptually similar. The only important difference is that tightly coupled systems have much higher communication bandwidth, which means that they can better exploit finer grain parallelism. We will discuss implementation of transport algorithms on MIMD platforms.

Advection algorithms may be characterized as data parallel, since both van Leer and Prather methods require that the same sequence of computations be applied to all elements of data arrays. In this case domain decomposition is an effective parallelization strategy. The arrays are partitioned among the processors (nodes) of a parallel machine and each node performs the same set of computations on array elements it owns. Whenever a computation requires knowledge of array elements owned by another node, messages are exchanged between the two nodes. The structure of the parallel code then consists of segments where independent computations are done within each node, followed by segments where information between nodes is exchanged via calls to message-passing library routines. The code can be structured so that the same program is executed in every node, but nodes may branch according to their processor ID, which is essential for scheduling of SENDs and RECEIVEs in the communication phase. This is known as the Single-Program-Multiple-Data (SPMD) programming approach. It should be noted that communication calls in SPMD programs frequently provide implicit synchronization barriers, since the computation cannot proceed until the required array elements are received by all of the requesting nodes. Advection algorithms can be classified as loosely synchronous.

The SPMD parallelization strategy with explicit message-passing requires that programmer manages computation and communication phases. Until recently, that was the only possible approach on the majority of parallel architectures. While this approach allows maximum flexibility it requires substantial effort to be expanded for communication scheduling and local memory management. The programmer also has to take into account hardware parameters and topology in order to produce efficient code by minimizing communication costs. Message-passing code is littered with calls to communication libraries which have nothing to do with the basic algorithm to be implemented. This reduces code legibility and makes the whole process of parallel application development more error prone.

Another approach is to design new high-level langauges which will preserve, at the application level, two basic tenets of sequential programming: single thread of control and global name space. We implemented atmospheric transport code in High Performance Fortran, which is among the most prominent of such languages. HPF language specification is a de facto standard adopted in 1993 as a result of collaboration of academic institutions and hardware and software vendors. Dialects of HPF already existed on Connection Machine and MasPar computers and at least 4 compilers are expected to be commercially available in 1995.

The basic principle of HPF is that parallelism must be explicitly stated by the programmer. For example, the compiler does not attempt to discover parallelism by a sophisticated dependency analysis of loops, which is an approach taken by other parallel languages. There are basically three ways a programmer expresses parallelism: array syntax, parallel constructs and compiler directives. HPF may be considered a superset of Fortran 90 with a small number of extensions to support effective use of MIMD machines. In both Fortran 90 and HPF arrays are first class objects. Array syntax allows the programer to manipulate complete arrays and to express parallelism transparently: since the array elements are distributed among the nodes of a parallel processor, an array instruction is executed by all nodes simultaneously. Translation between addreses in the global name space and addresses of array elements in local node memories is done by the system. Also, if the array operation requires that array elements be fetched from remote node memories, appropriate communication calls are generated by the compiler transparently to the programmer. Array syntax alone is not sufficient to guarantee good performance. Non-uniform memory access on MIMD computers implies that data locality is crucial for good performance. Because of that, some of the most important HPF extensions to Fortran 90 consist of compiler directives for data distribution among nodes, which provides the programmer with a set of high-level tools for management of data layout. At present, only simple and regular layouts can be achieved using compiler directives, in contrast to the flexibility of the message-passing approach. New mechanisms are being developed to provide users with a standard interface to flexible data partitioners via compiler directives.

We will now describe HPF features which were used to implement the 3D van Leer/Prather sequential transport code on parallel platforms. Whenever appropriate, we will make a comparison with the 2D van Leer message-passing code, and discuss new language features which would be beneficial for this one and similar applications.

E. Parallel Implementations of Atmospheric Transport

1. Data Layout

Domain decomposition strategy requires that arrays be distributed among the nodes of a parallel machine. HPF provides a set of compiler directives which determine mapping of individual array dimensions onto the processor array. This mapping is generally achieved in following stages:

1. Definition of a template using the **TEMPLATE** directive. Template is not a true array, just a description of an abstract index space, so there is no memory allocation for the template. It cannot be passed as a formal argument to a subroutine, but it can be **INHERIT**ed.

2. Definition of an abstract processor array using **PROCESSORS** directive.

3. Alignment of data arrays with the template using **ALIGN** directive

4. Distribution of the template onto the processor array using **DISTRIBUTE** directive. All array elements which are aligned with the elements of the template reside on the same processor. Possible partitioning patterns are: **BLOCK**, **CYCLIC** and *. **BLOCK** distributes an array dimension in contigous blocks, **CYCLIC** distributes in a round-robin fashion, while * specifies that the array dimension should not be distributed. **BLOCK** is analogous to **:NEWS** and * corresponds to **:SERIAL** in CM Fortran.

5. Mapping of the abstract processor array onto the hardware. This final stage is implementation dependent and is done by the run-time system transparently to the programmer.

The fundamental advantage of this process is that code design is centered around the topology of the physical problem, while the mapping from the problem topology to the communication hardware topology is done by the system in a (presumably) optimized fashion for that hardware.

The examination of van Leer and Prather algorithms shows that computation on a grid point requires knowledge of nearest neighbors only. This suggests that **BLOCK** type distribution is the best choice. Density and velocity fields are conformal although they are defined on physically staggered grids. They are aligned to the same template **Density**, which guarantees locality of memory references. Important arrays which define pressure levels and pressure increments between successive σ levels are one-dimensional. Arrays which determine surface pressure on grid centers and edges are two-dimensional. The most effective strategy to maintain memory locality with these non-comforming arrays is data replication. Using the **SPREAD** intrinsic, lower dimensional arrays are replicated into 3D arrays. The simplest replication strategy is to dimension all replicated arrays as the density and velocity arrays and to align them to the same **Density** template. If memory is at premium, the replicated arrays should be dimensioned as the procesor array and then aligned to the **Density** template in a strided fashion.

If an array is declared as**DYNAMIC**, it can be redistributed and realigned during the course of the computation, using executable directives **REDISTRIBUTE** and **REALIGN**. Arrays are implicitly realigned and redistributed across subroutine boundaries, if the distribution of the actual and dummy arguments is different. It is frequently necessary to redistribute data in order to maximize either memory access locality or parallelism during different phases of computation. In the advection code, bottom pressure layer and the interface layer between troposphere and stratosphere require special treatment. During vertical advection step, it is necessary to separate computations on the bottom layer from the rest, while horizontal advection treats all pressure levels uniformly. Because of this, bottom σ level is passed to the Prather subroutine as a 2D array. The dummy 2D argument is spread across the whole machine and the actual argument is a slice of the 3D array. The implicit mapping across a subroutine boundary allows one to choose the best data layout to implement a computational module, while the interface between modules is handled transparently by the run-time system.

2. Load Balancing

Parallel efficiency is diminished if the computational load is not balanced, since the idle processors are underutilized resource. The major source of load imbalance in the advection code is sub-cycling in the polar regions, which happens only during the advection step in the longitudinal direction. A trivial solution to this problem is to decrease the value of time step interval Δt uniformly over the globe, so that the CFL condition is satisfied everywhere with the given Δt. The computation is load-balanced at the expense of doing physically unnecessary computations in the equatorial regions. Instead, the GSFC code implements a two-time-scale strategy: a small time step Δt_1 is used for polar regions, while a bigger $\Delta t_2 = 5 \cdot \Delta t_1$ is used in the equatorial region ($|\lambda| < 70^o$). In principle, one could use a multitude of time scales for different latitude slabs. Since the computational complexity of advection step depends on the location of a grid point, any load-balanced decomposition strategy must take this into account.

The simplest decomposition strategy would distribute only X and Z dimensions of the three-dimensional arrays. This is accomplished with following directives:

```
INTEGER, PARAMETER:: lx=88, ly=144, lz=19 !grid points along lat, lon, z
REAL, DIMENSION(lx,ly,lz):: mixr          !mixing ratio array
INTEGER, PARAMETER:: nx=4, nz=4           !number of processors along
                                          !lat and z directions
!HPF$ PROCESSORS proc(nx,nz)               !processor array
!HPF$ TEMPLATE, DIMENSION(lx,lz):: temp2D !2D template
!HPF$ ALIGN mixr(:,*,:) WITH temp2D(:,:)  !collapse 3D mixing ratio array
                                          !onto 2D template
!HPF$ DISTRIBUTE temp2D ONTO  proc        !distribute template and aligned
                                          !arrays onto processor array
```

Computation is trivially load balanced, since the computational complexity varies along the locally cached Y direction. One drawback of this approach is its lack of scalability due to excessive communication costs. Scaling properties can be described by a simple model:

$$T_{2D} = T_{comm} + T_{comp} = [2L_y\tau_{comm} + \tau_s] + \tau_{VL}\frac{N_g}{P} \tag{12}$$

In this equation, T_{2D} denotes time required to perform a single van Leer step on a single pressure level using the two-dimensional decomposition. Grid size in x and y direction is L_x and L_y respectively. Total number of grid points on a single level is $N_g = L_x \cdot L_y$. Architecture specific parameters are defined as follows: P is the number of processors assigned to a single pressure level, τ_{comm} is the average time to transmit a floating point number across the network,

τ_s is the start-up time for the communication channel, and τ_{VL} is the CPU time required for van Leer update of a single grid point. If both grid size and number of processors are increased in such a way that computational load per processor is kept fixed, i.e., $N_g/P = const$, communication time diverges as $T_{comm} \propto L_y \propto N_g^{1/2}$.

Three-dimensional decompositions have better scaling properties. The following set of directives will distribute all three dimensions across a processor array:

```
INTEGER, PARAMETER:: lx=88, ly=144, lz=19
REAL, DIMENSION(lx,ly,lz):: mixr
INTEGER, PARAMETER:: nx=4, ny=4, nz=2     !number of processors along
                                          !lat, lon and z directions
!HPF$ PROCESSORS proc(nx,ny,nz)           !processor array
!HPF$ TEMPLATE, DIMENSION(lx,ly,lz)::  temp3D   !3D template
!HPF$ ALIGN mixr WITH temp3D
!HPF$ DISTRIBUTE temp3D ONTO  proc        !distribute template and aligned
                                          !arrays onto processor array
```

A simple model of computational cost per van Leer step is given by:

$$T_{3D} = T_{comm} + T_{comp} = \left[2\frac{L_y}{P^{1/2}}\tau_{comm} + \tau_s\right] + \tau_{VL}\frac{N_g}{P} \tag{13}$$

Since $L_x \propto L_y \propto N_g^{1/2}$, total computation time is a homogeneous function of the scaling argument N_g/P. Therefore, by increasing the machine size we can solve bigger problems in the same amount of time.

We augmented the three-dimensional decomposition with an explicit load-balancing strategy to handle sub-cycling, as shown in the following code segment:

```
! del_mixr  = temporary array to store changes due to X advection
! u         = X component of transport winds
! mixr_hlat = temporary array for high-latitude sections of mixr
! mixr_mlat = temporary array for mid-latitude section of mixr
! u_hlat    = temporary array for high-latitude sections of u
! u_mlat    = temporary array for mid-latitude sections of u
! del_mixr_hlat = temporary array for high-latitude sections of del_mixr
! del_mixr_mlat = temporary array for mid-latitude section of del_mixr
! trigs_hlat, trigs_mlat  = tables with trigonometric factors (precomputed)
! pres_mlat, press_hlat   = pressure arrays for mixing ratio to mass conversion
! ndt=5       subcycling ratio
! lslt1, lslt2 = boundary grid indices for South polar region
! lhlt1, lhlt2 = boundary grid indices for North polar region
! lmlt1, lmlt2 = boundary grid indices for mid-latitude region
! hlat_width = (lslt2-lslt1+1) + (lhlt2-lhlt1+1) integer parameter
! mlat_width = (lmlt2-lmlt1+1)  integer parameter
!
real, dimension(nlon,hlat_width,nlev):: mixr_hlat,u_hlat,trigs_hlat,del_mixr_hlat
real, dimension(nlon,hlat_width,nlev):: pres_u_hlat,inv_pres_hlat
real, dimension(nlon,mlat_width,nlev):: mixr_mlat,u_mlat,trigs_mlat,del_mixr_mlat
real, dimension(nlon,mlat_width,nlev):: pres_u_mlat,inv_pres_mlat
!
!HPF$ DISTRIBUTE mixr_hlat(block,block,block) ONTO proc
!HPF$ DISTRIBUTE mixr_mlat(block,block,block) ONTO proc
!HPF$ ALIGN  u_hlat     WITH mixr_hlat
!HPF$ ALIGN  trigs_hlat WITH mixr_hlat
!HPF$ ALIGN  pres_hlat  WITH mixr_hlat
!HPF$ ALIGN  u_mlat     WITH mixr_mlat
!HPF$ ALIGN  trigs_mlat WITH mixr_mlat
!HPF$ ALIGN  pres_mlat  WITH mixr_mlat
!
! gather high and mid latitudes into temporary arrays
```

```
!
mixr_hlat(:,lslt1:lslt2,:)=mixr(:,lslt1:lslst2:)
mixr_hlat(:,lslt2+1:hlat_width,:)=mixr(:,lhlt1:lhlt2,:)
mixr_mlat=f3(:,lmlt1:lmlt2,:)
u_hlat(:,lslt1:lslst2,:)=u(:,lslt1:lslst2:)
u_hlat(:,bound1+1:hlat_width,:)=u(:,lhlt1:lhlt2,:)
u_mlat=u(:,lmlt1:lmlt2,:)
!
! van Leer step on high and mid latitude temporaries
!
CALL XadvHlat(del_mixr_hlat,u_hlat,mixr_hlat, trigs_hlat, inv_pres_hlat, pres_u_hlat, ndt)
CALL XadvMlat(del_mixr_mlat,u_mlat,mixr_mlat, trigs_mlat, inv_pres_mlat, pres_u_mlat)
!
! scatter temporaries into original arrays
!
del_mixr(:,lslt1:lslst2:)= del_mixr_hlat(:,1:bound1:)
del_mixr(:,lhlt1:lhlt2,:)= del_mixr_hlat(:,bound1+1:hlat_width,:)
del_mixr(:,lmlt1:lmlt2,:)= del_mixr_mlat
```

The X-advection step is split into a mid-latitude advection and high-latitude advection. High-latitude advection performs ndt $= \Delta t_2/\Delta t_1 = 5$ van Leer steps on the near-polar-cap portions of the mixing ratio array, for each mid-latitude advection step on the near-equator portion of the array. To achieve maximum parallelism and load balance, polar regions of an array, which reside on disjoint subsets of processor node array, are gathered into a temporary array which is distributed across all nodes, before the high-latitude step. Mixing ratio changes at high latitudes are written into a conformal temporary array. After the high latitude step, this temporary array is scattered back into the original array which stores mixing ratio changes for the whole globe. The same gather/scatter is done with the mid-latitude array portions, before the Y and Z advection steps are done on the original arrays. Load balancing is achieved at a price of extra communication overhead required for the two gather/scatter phases.

The performance model of Equation 13 must be augmented with the communication overhead of the gather/scatter operations. This is in general quite complicated, since the gather/scatter overhead $\tau_{GS}(N_g, P, \Omega_P)$ depends on the problem size N_g, machine size P and the inteconnect network topology Ω_P. For simplicity, we will treat the gather/scatter operation as a broadcast along Y direction, which may be considered as an upper bound for the real overhead. For hypercube topology with cut-through routing and sufficient bandwidth to eliminate link contention, time required for a gather/scatter operation is:

$$\tau_{GS}(N_g, P, \Omega_P) = \alpha\left(\frac{l_y}{L_y}\right)\tau_{comm}\frac{N_g}{P} + \tau_s + \beta\cdot\log(P) \tag{14}$$

where β is a platform dependant parameter. Factor $\alpha(l_y/L_y)$ depends on the ratio of numbers of mid-latitudes and all latitudes, l_y/L_y. Total time for a van Leer X advection step of a load-balanced three-dimensional decomposition is:

$$T_{3D}^{LB} = T_{3D} + 2\tau_{GS} = 3\tau_s + +2\frac{L_y}{P^{1/2}}\tau_{comm}\left(\tau_{VL} + 2\alpha\left(\frac{l_y}{L_y}\right)\tau_{comm}\right)\frac{N_g}{P} + \beta\cdot\log P \tag{15}$$

Because of the logarithmic term, load-balanced three-dimensional decomposition is weakly non-scalable as well. However, if the load per processor is kept fixed, i.e. $L_y \propto P^{1/2}$, logarithmic divergence in T_{3D}^{LB} is weaker than the linear one in T_{2D}, which means that for sufficiently large machine, three-dimensional decomposition is superior. Situation is more complicated for architectures with lower connectivity. For example, it is impossible to eliminate link contention on a mesh based interconnect topology. In this case, leading terms in the gather/scatter overhead are given by:

$$\tau_{GS}(N_g, P, \Omega_P) = \alpha(\frac{l_y}{L_y})\tau_{comm}\frac{N_g}{P} + \tau_s + \beta\cdot\frac{l_y}{L_y}P - \gamma P^{1/2} \tag{16}$$

assuming a square processor array.

Scaling analysis is valuable for predicting performance in asymptotic regimes, but may be misleading in practical applications, where problem size and machine size are not freely adjustable parameters. Maximum and minimum problem size are determined by the limitations of the physical model represented by partial differential equations.

Frequently, this implies that the variations in problem size are not big enough for a scaling analysis to be relevant. Instead, given a limited range of problem and machine sizes, one has to pick a data distribution strategy which will minimize total computation time. With explicit message-passing, changing data layouts requires that the communication portion of the code be redesigned completely. Because of that, one has to decide on the optimal layout carefully *a priori*, which requires building detailed computational cost models and knowledge of hardware and software parameters. Even if one adheres to standards like MPI, this restricts practical code portability (including nominally portable libraries, unless they support switching between different layouts), since a particular layout may have excellent performance on some platforms and poor performance on others. One of the advantages of programming in HPF is that one can easily implement and experiment with different data layouts by simply changing template distributions and alignment directives. This makes *a posteriori* code optimization for a target architecture and problem size simple and inexpensive. Portability is enhanced by programmer's reliance on the run-time library optimizations to achieve good performance across a wide range of platforms.

F. Evaluation of Language Features

The most important operations in van Leer and Prather algorithms can be loosely classified as stencil computations. Due to time -splitting, each advection step requires a simple stencil which includes nearest neighbors along a given direction. Stencil computations, combined with periodic boundary conditions along X direction, are easily expressed using the Fortran 90 **CSHIFT** intinsic:

```
 SUBROUTINE XadvMlat(D,DTOVDXL,U1,F1,PS1INV,PSU,UPSTREAM)

 REAL DTOVDXL(ILMM,L1:L2,IKMM), BOT(ILMM, L1:L2, IKMM),
! DNEW(ILMM, L1:L2, IKMM),D(ILMM, L1:L2, IKMM),
! DELTA(ILMM, L1:L2, IKMM), TOP(ILMM, L1:L2, IKMM),
! FLUX(ILMM, L1:L2, IKMM)

 REAL LPSU(ILMM, L1:L2,IKMM), LFLUX(ILMM,L1:L2,IKMM),
! LU1(ILMM, L1:L2,IKMM), RD(ILMM, L1:L2, IKMM),
! LD(ILMM, L1:L2, IKMM), LDELTA(ILMM, L1:L2, IKMM)

 rd =   CSHIFT(d, DIM=1, SHIFT=-1)
 ld =   CSHIFT(d, DIM=1, SHIFT=1)
 lu  =  CSHIFT(u, DIM=1, SHIFT=1)
 lpsu = CSHIFT(psu, DIM=1, SHIFT=1)

 IF (.NOT.upstream) THEN
       top = (d-rd) * (ld-d)
       bot = ld - rd
       WHERE((top.GT.0.).AND.(bot.NE.0.))
          delta = 2 * top / bot
       ELSEWHERE
          delta = 0.
       END WHERE
 ELSE
       delta = 0.
 END IF

 ldelta = CSHIFT(delta, DIM=1, SHIFT=1)

 WHERE(lu.GE.0.)
    flux  =  d + (.5 * (1 - lu * alpha) * delta)
 ELSEWHERE
    flux  =  ld - (.5 * (1 + lu * alpha) * ldelta)
 END WHERE
```

```
      lflux = flux
      flux = CSHIFT(flux, DIM=1, SHIFT=-1)
      delta = -alpha*inv_press*(lu*lflux*lpsu - u*flux*psu)

      RETURN
      END
```

With free boundary conditions along Y direction we may use either **EOSHIFT** (end-off) intrinsic or even the array sub-section notation, as shown below:

```
      WHERE(VFLD(:,1:IJMM,:).GE.0.0)
         FLUX(:,1:IJMM,:)  =  D(:,0:IJMM-1,:) + (.5 *
     $               (1.- VFLD(:,1:IJMM,:) * DTOVDXJ) *
     $               DELTA(:,0:IJMM-1,:))
      ELSEWHERE
         FLUX(:,1:IJMM,:)  =  D(:,1:IJMM,:)
     $               - (.5 * (1.+ VFLD(:,1:IJMM,:)*DTOVDXJ) *
     $                 DELTA(:,1:IJMM,:))
      END WHERE
```

Calculation of fluxes differs for points with positive and negative velocities, due to the upstream character of the algorithm. This is expressed using the Fortran 90 masked array assignments (**WHERE** statements).

Polar caps are treated using the well-mixed approximation to avoid singularities associated with spherical metrics. This approximation requires computing net flux into the polar caps, which is expressed using the **SUM** intrinsic. The resultant 1D array is then **SPREAD** back into the 3D density array.

```
! South pole
        SUMSP = SUM(FLUX(:,1,:),DIM=1)
! North pole
        SUMNP = SUM(FLUX(:,IJMM,:),DIM=1)
! Polar caps are top and bottom strips of the 3D grid
        forall(ik=1:ikmm) DNEW(:,0,ik) =   -POLCON*SUMSP(ik)
        forall(ik=1:ikmm) DNEW(:,IJMM,ik) = POLCON*SUMNP(ik)
```

SPREAD is one of the Fortran 90 array construction functions, which also include functionality to **MERGE** and **RESHAPE** arrays. **SUM** is representative of the complementary class of reduction operations. Reductions map an array argument into an array of lower dimensionality by performing an operation (such as sum, product and others) on subsets of array elements which are defined by the arguments **DIM** and **MASK**. **DIM** specifies the dimension of the array along which the reduction should be performed and **MASK** makes the operation conditional upon a logical array argument.

Constructors and reductions involve global redistributions of data. Efficient implementation of these global operations is essential for good performance since they involve collective communication routines. This fact is recognized in HPF design which provides a number of most common and valuable global operations as intrinsic routines. Most of them are inherited from Fortran 90, but the HPF intrinsic library provides additional routines. Among the most important HPF intrinsics are:

1. array reduction functions
2. Array combining scatter functions
3. Array prefix and suffix functions (scans)
4. Array sorting functions
5. Bit manipulation functions
6. Vector and Matrix multiplication functions
7. Elemental intrinsic functions
8. Array location fubctions
9. Mapping inquiry subroutines

From the long list of Fortran 90 and HPF intrinsic routines, advection code relies mostly on constructors, reductions, scans, elemental intrinsic functions and array location functions.

Prather step consists of two phases: deconstruction of volume boxes and their subsequent reconstruction. Deconstruction phase can be made purely local by alignment of density and velocity arrays. In the reconstruction phase, moments transferred from the upstream box must be communicated to the downstream box. In the code segment below, we show how this can be accomplished in HPF using the array section notation combined with a **WHERE** construct conditional upon the sign of the velocity.

```
C     RECONSTRUCT TROPOSPHERIC, INTERFACE AND STRATOSPHERE BOXES
      WHERE(SGARRAY(:,:,1:KTOPM1).LT.0.)
         EPS(:,:,1:KTOPM1) = F(:,:,1:KTOPM1)/(F(:,:,1:KTOPM1)+ CON(:,:,2:KTOP))
         EPS1(:,:,1:KTOPM1) = 1.-EPS(:,:,1:KTOPM1)
         TEMP(:,:,1:KTOPM1)=EPS(:,:,1:KTOPM1)*C(:,:,2:KTOP)-
     1    EPS1(:,:,1:KTOPM1)*F0(:,:,1:KTOPM1)
         C(:,:,2:KTOP) = C(:,:,2:KTOP) + F0(:,:,1:KTOPM1)
         CZZ(:,:,2:KTOP)=EPS(:,:,1:KTOPM1)*EPS(:,:,1:KTOPM1)*
     1     FZZ(:,:,1:KTOPM1)+EPS1(:,:,1:KTOPM1)* EPS1(:,:,1:KTOPM1)*
     1      CZZ(:,:,2:KTOP)+5.0*(EPS(:,:,1:KTOPM1)*EPS1(:,:,1:KTOPM1)
     1        *(CZ(:,:,2:KTOP)-FZ(:,:,1:KTOPM1))-(EPS1(:,:,1:KTOPM1)-
     1          EPS(:,:,1:KTOPM1))*TEMP(:,:,1:KTOPM1))
         CZ(:,:,2:KTOP) = EPS(:,:,1:KTOPM1)*FZ(:,:,1:KTOPM1)+
     1     EPS1(:,:,1:KTOPM1)*CZ(:,:,2:KTOP)+ 3.0*TEMP(:,:,1:KTOPM1)
      ELSEWHERE
         EPS(:,:,1:KTOPM1) = F(:,:,1:KTOPM1) / (F(:,:,1:KTOPM1)
     1    +CON(:,:,1:KTOPM1))
         EPS1(:,:,1:KTOPM1) = 1. - EPS(:,:,1:KTOPM1)
         TEMP(:,:,1:KTOPM1) = -EPS(:,:,1:KTOPM1)*C(:,:,1:KTOPM1) +
     1     EPS1(:,:,1:KTOPM1)*F0(:,:,1:KTOPM1)
         C(:,:,1:KTOPM1) = C(:,:,1:KTOPM1) + F0(:,:,1:KTOPM1)
         C(:,:,1:KTOPM1) = C(:,:,1:KTOPM1) + F0(:,:,1:KTOPM1)
         CZZ(:,:,1:KTOPM1) = EPS(:,:,1:KTOPM1)*EPS(:,:,1:KTOPM1)*
     1     FZZ(:,:,1:KTOPM1)+EPS1(:,:,1:KTOPM1)*EPS1(:,:,1:KTOPM1)*
     1      CZZ(:,:,1:KTOPM1) + 5.* (EPS(:,:,1:KTOPM1)*
     1       EPS1(:,:,1:KTOPM1)*(-CZ(:,:,1:KTOPM1)+FZ(:,:,1:KTOPM1))+
     1        (EPS1(:,:,1:KTOPM1)-EPS(:,:,1:KTOPM1))*TEMP(:,:,1:KTOPM1))
         CZ(:,:,1:KTOPM1) = EPS(:,:,1:KTOPM1)*FZ(:,:,1:KTOPM1)+
     1    EPS1(:,:,1:KTOPM1)*CZ(:,:,1:KTOPM1)+3.*TEMP(:,:,1:KTOPM1)
      END WHERE
```

The bottom pressure layer requires special treatment since there cannot be flow across the Earth surface. In principle, this could be handled by the **WHERE** construct, but it leads to load imbalances. Load imbalance is eliminated by separating 2D arrays which correspond to the bottom layer from the 3D arrays which describe the rest of troposphere and stratosphere. For maximum parallelism, both 2D and 3D arrays are spread across the whole machine before the Prather step is performed. After the completion of the Prather step, arrays are merged into a single 3D array which is used for van Leer advection. Load balance is achieved by incurring the merge communication overhead.

Stencil computations are naturally expressed using array syntax because they execute mappings between conformal array sections. There are many occasions where parallelism must be expressed via more complex mappings of array indices. In this case, parallelism is expressed explicitly using the **FORALL** statement. A very simple example is the code segment below which initializes Rausch-Williamson winds to test the advection algorithm:

```
      D2 = (360. / N)
      F0 =  -D2/2.
      DTR = PI / 180.
      FORALL(IL=1:N) DL(IL) = (IL-1)*D2 * DTR
      FORALL(IL=1:N) DL2(IL) = (F0 + (IL-1)*D2) * DTR
      U1 = 2. * PI * A  / (256. * 900.)
      FORALL(IJ=0:JGMAX, IL=1:N)
     $  U(IL,IJ,:) = U1*(COS(B)*COSLAT(IJ)+SIN(B)*SIN(DG(IJ))*COS(DL2(IL)))
```

```
 FORALL(IL=1:N)
$  V(IL,:,:) = -U1 * SIN(B) * SIN(DL(IL))
```

Multiple **FORALL** statements can be combined into a **FORALL** construct. The execution of the statement can be made conditional upon a scalar logical expression as an extension to the **WHERE** statement and one may call user-defined **PURE** functions to emulate Fortran 90 elemental functions. Programmer can also use the **INDEPENDENT** directive to assert that a code segment does not produce dependencies which would prevent parallelization. It may be used by the compiler to perform additional optimizations.

FORALL is a versatile, convenient and powerful construct which can be used to accomplish many tasks, including array construction, indirection, scatter and scans. However, we encountered serious inefficiences and problems with its implementation on present compilers. Particularly problematic were our attempts to implement scans by combining **FORALL** statements with reduction intrinsics. Because of this, we consistently tried to avoid **FORALL** in favor of combinations of array expressions or array intrinsics whenever possible.

Atmospheric science applications are generally heterogeneous in their treatment of horizontal and vertical directions. For example, general circulation models (GCM) are split into two components: dynamics, which integrates fundamental partial differential equations quite similar to the ones considered here, and physical parametrization, which characterizes sub-grid scale processes as functionals of grid variables. The physical parametrization algorithms usually couple grid points along the vertical direction in a global fashion. They can be frequently expressed in terms of simple reduction operations and scans. Given any commutative and associative operation $\times$ and an array A, scan is defined as a collection of results: $(A(1), A(1) \times A(2), A(1) \times A(2) \times A(3), \ldots, A(1) \times A(2) \cdots \times A(N))$, i.e. reduction with saved intermediate results. Scans have a serial flavor but they can be very efficiently implemented in parallel, particularly when there is hardware support like on CM-5.

A simple example of a scan is the computation of pressure. Where available, we compute the pressure by calling directly the intrinsic function. If the library function is not available, simple scan may be expressed via a **FORALL** statement with a simple reduction function,

```
FORALL(K=2:KTOP) F1(:,:,K-1) = SUM(F1(:,:,K-1:KTOP),dim=3)
```

Segmented scans may be expressed using a segment **MASK** argument in the reduction function:

```
FORALL(K=2:KTOP) F1(:,:,K-1) = SUM(F1(:,:,K-1:KTOP),dim=3)
```

The amount of communication along the vertical direction in most atmospheric science codes may warrant an approach where the vertical direction is not distributed if the machine is small enough relative to the grid size. On platforms like CM-5 reductions and scans may take advantage of the specialized communication hardware, but the number of pressure layers is usually too small to amortize the latency costs, and it seems to be always better to serialize the vertical direction. Again, the decision to distribute vertical dimension can be postponed until after the trial runs are performed, since it can be implemented by simply changing the processor array directive from

```
!HPF$ PROCESSORS proc(nx,ny,nz)
```

to

```
!HPF$ PROCESSORS proc(nx,ny,1)
```

Finally, we would like to emphasize how much is code clarity and compactness enhanced by the expressive capabilities of Fortran 90 array notation. For example, triplet notation semantics allowed us to represent gather/scatter operations by a few lines of HPF code. Each line of the HPF code which just copies one array section into another, must be implemented as a sizable *block* of message-passing code, consisting of a control block, index arithmetic for send and receive buffers, manipulation of temporary buffers and at least one SEND/RECEIVE pair. Message passing codes are usually much longer than their sequential counterparts, unlike HPF codes which are frequently shorter.

In fairness to message-passing approach, HPF codes are not nearly as memory efficient. Reliance on array syntax sometimes requires that new temporary arrays be used or that scalar temporary variables be turned into arrays. This happens whenever the **WHERE** construct replaces sequential **DO** loops. Another advantage of the message passing approach is that it allows us to implement arbitrary data decompositions, so we could in principle dispose of the gather/scatter step altogether. One possibility we considered is to distribute arrays into rectangular boxes of different sizes, so that the product of the number of grid points per box times the computational complexity per grid point is fixed. This decomposition is load balanced and there is no global communication overhead due to gather/scatter. However, the bookkeeping required to manage communication between boxes of varying shape and size for varying number of nodes and grid sizes is overwhelming. This is the reason why in our message-passing implementation we opted to use a small enough Δt to satisfy the CFL condition for the whole globe.

This discussion touches upon one of the defficiencies of the current version of HPF. It has semantic support for regular data layouts, which is sufficient for a large class of applications, but it lacks support for irregular data structures or non-uniform data layouts. For transport algorithm, support for static non-uniform partitioner would trivially solve the load-balancing problem. In other applications, like those which involve adaptive grids, dynamic load balancing and re-partitioning is essential. In either case, language functionality would be greatly enhanced by a high-level interface to a user-specified partitioner or repartitioner. A number of research efforts is directed towards providing such an interface.

G. Performance

The purpose of our research was to examine problems which arise in the parallel implementation of transport algorithms and to evaluate how successfully can they be solved using the two major parallel computing paradigms. We were particularly interested in examining the features of the new data parallel language, High Performance Fortran, and how effectively can they be mapped onto the transport algorithms. Because of that, we were not interested in obtaining detailed performance measures for different problem sizes and machines. Furthermore, HPF compilers are far from mature and any attempt to provide performance measures is bound to be short-lived.

The table below shows typical performance data for the CM Fortran implementation of the van Leer algorithm on a 144 x 88 x 14 model grid with load balancing. One thing to note is the very low FLOP rate for Y advection because of the bottleneck introduced by the well-mixed polar cap approximation.

Number of Processors	32	256
X Advection Peak GFLOPS	1.12	6.78
Y Advection Peak GFLOPS	0.63	4.38
X Advection Average GFLOPS	0.51	2.37
Y Advection Average GFLOPS	0.053	0.30

Production systems may require multiple data layouts for optimum performance, which depend on the machine size and problem size. In the advection codes we examined, there are regimes where either of the three decomposition strategies dominates, so it is essential to have the capability to redistribute data with minimum effort. By encapsulating data layout into a set of high-level primitives, HPF enables effective *a posteriori* optimization.

H. Summary and Conclusions

We implemented prototype parallel atmospheric transport codes using a high-level data parallel language (High Performance Fortran) as well as a low-level message-passing approach (Intel native communication library). The parallel systems evolved from sequential Fortran 77 codes developed at NASA Goddard Space Flight Center. Eulerian transport algorithms such as van Leer and Prather are characterized mostly by local computations which can be efficiently expressed in data parallel paradigm. We showed how the language features of HPF can be used to compactly and effectively express computations in the advection algorithms. We showed that implementation in High Performance Fortran offers important advantages in comparison with the low-level message-passing approach. One of the most important advantages is the higher level of abstraction of the parallel system which is presented to the application programmer. As a result, data layout management tasks are separated from the rest of the application. Such an implementation requires less coding, offers enhanced portability and a practical strategy of inexpensive *a posteriori* experimentation with different data layouts for optimization. The limitation of this approach is that one can currently implement only regular layouts. Because of this restriction one is frequently forced to incur overheads for data redistributions required for load-balancing. This problem should be solved in the near future by integration of general-purpose, flexible partitioners into the language. Gridpoint methods in atmospheric and earth sciences are generally characterized by regular, static data structures and static computational flow-graphs, which makes them suitable candidates for efficient implementation in High Performance Fortran and other data parallel languages. This implies that grid based general circulation models are also appropriate for parallelization using HPF. Adaptive mesh refining techniques will be easy to implement once user-specified partitioners become incorporated into the language.

Even semi-Lagrangian methods, which result in irregular and dynamic communication patterns can be efficiently implemented in HPF on equal area grids like icosahedral. The uniform nature of icosahedral grid is an important advantage over the standard latitude-longitude grid. By tuning time step Δt on icosahedral grid, one can guarantee that communication will be nearest neighbor only, which is impossible with latitude-longitude grid. In this case, a global reduction is necessary every time step to ensure that Δt is properly sized, which makes HPF global intrinsics particularly valuable. Semi-Lagrangian algorithm is an important case where considerations of parallel efficiency influence the design of the primary data structures. Usually, the data structures of the original sequential implementation determine the parallel implementation. In this sense, a large number of parallel codes evolve "adiabatically" from sequential codes. However, sometimes there are important benefits in designing an algorithm from scratch with its parallel implementation in mind.

In a production environment, advection or transport code constitutes a module which must be integrated into the complete system, which may be spread across a heterogeneous meta-computer. Neither the data-parallel languages nor the low-level approach offer a seamless and convenient system integration environment. We are particularly interested in the system integration requirement of the complete data asimilation system. There seem to be two complementary approaches to integration across architectures and even paradigms. Fortran M, for example, provides tools for integration of data and task parallelism at the single language level. An important feature of Fortran M is capability to allocate resources among tasks dynamically. On the other end of the spectrum is exogenous integration which separates module integration paradigm (like AVS) from the module implementation paradigm (like HPF). Complete assimilation system can be conceptualized as a static data-flow diagram which can be conveniently mapped onto an AVS network. We are currently investigating the possibility of using AVS framework for system integration. This approach turned out to be very successful in other meta-computer applications, particularly in those cases where links between modules are static. One of the limitations of the present AVS design is that exchange of data between modules is done through the kernel process and cannot use the full communication bandwidth of a parallel machine. This is not very restrictive in our case, because of the inherently sequential structure of the assimilation loop data-flow graph. We have implemented the van Leer/ Prather advection routines as AVS modules and linked them with AVS supported visualization modules.

V. PARALLELIZATION OF NASA/GODDARD SHALLOW WATER SEMI-LAGRANGIAN CODE

The purpose of this component of the 4DDA project is to investigate suitable domain decompositions that lead to efficient parallelizations of the interpolation techniques used in the Semi-Lagrangian advection scheme. In this report we present the results of our investigation using the lasagna decomposition of the input domain of the bilinear and bicubic interpolations used in NASA's 2-D Shallow Water Model.

Our approach to the usage of the lasagna decomposition reduces the amount of communication needed for either the bilinear or bicubic interpolations. The criteria for goodness of the algorithm is based on the ratio of the useful work done to the amount of communication required.

A. Introduction

Weather forecasts are essential for the planning of many activities. They need to be accurate and available in time. The basis of weather forecasting are Numerical Weather Prediction (NWP) models which assimilate data and predict future conditions of atmosphere. Given the requirement that forecasts be available in real-time, the numerical models must be efficient. There is also considerable interest in building quality data-sets or model predictions of weather conditions for research purposes and as case studies. These involve processing of voluminous data using the numerical models in an efficient manner.

One chronic problem with NWP models is that the longest time step that can be chosen is decided by stability rather than accuracy. This is alleviated to a considerable degree in the Semi-Lagrangian Transport Model using a semi-implicit time differencing scheme [8,9]. This was demonstrated for a limited area shallow water model (one layer, homogeneous fluid). An excellent review of Semi-Lagrangian integration schemes for atmospheric models is treated by [10].

Later work addressed the deficiencies associated with polar singularity in the latitude-longitude co-ordinate system [7]. NASA implemented a modified model based on this paper

Another notable issue is that for accuracy to be improved, among other things higher resolutions are required. This calls for efficient implementations on powerful machines. Effort has been underway to implement these models on parallel computers [1].

In order to establish the most suitable paradigm, certain modules of NASA's serial implementation were identified as test-cases. Initial studies at Syracuse University and NASA have indicated that the interpolation techniques deployed occupied a total running time of 25% of running time in the serial code. It was also observed that the usage of the construct "forall" did not have a good speed-up in the data parallel prototype on the CM-5. This can be attributed to the irregular, non-local pattern of data accesses. In addition the decomposition was carried out in the North-South axis as shown in Fig. 1; as one expects winds to be mostly in the East-West direction. This lead to a very poor utilization of the nodes when there was a small grid size and one of the grid axes declared as serial or local to a node. In such a case the data is distributed among only a few nodes along the news or parallel axis leaving remaining processors with no work [5] The default news:news (square or ravioli) decomposition showed better results. This is because the run time library is able to ensure a better utilzation of nodes with the default decomposition. But this decompostion does not take advantage the fact that the winds are of a East-West nature.

Hence, the message passing model was proposed as an alternate parallelization technique as it offers a greater flexibility in choice of domain decomposition. One does not need to depend on the vagaries of run-time libraries or systems to decide the utilization of nodes. Scalable decompositions may be developed. Additionally, the decomposition can be adapted to ensure that the pole regions are treated differently than regions near the equator. This is because, at the poles the winds move around the pole (entire latitude). This is another disadvantage using data parallel techniques which permit only simple decompositions. Hence, nearest neighbor communication is not possible at the pole for the default square decomposition on CM-5 when using a latitude-longitude grid.

A lasagna decomposition along the N-S axis was adopted as the first test case as seen in Fig. 2. This was postulated to have mostly nearest neighbor communication in the N-S direction. The polar region problem is avoided as the lasagna strips are in E-W direction. The algorithm uses a message-passing version of the data-parallel construct "cshift". The maximum number of "cshifts" is determined at run-time at every time -step for each component. The necessary data from the circulated wind "strips" is collected and a local interpolation is performed at every time step. Two strategies and their merits are discussed in the next section.

B. Metrics

We propose some measurements for assessing the goodness of the algorithm as well as giving direction to future work :

- Overall running time
- Ratio of calculation to communication.
- The maximum displacement in terms of nodal distance in the N-S direction.
- Speed-up of interpolation
- Accuracy of calculations

The first criterion shows how well the algorithm performs with respect to the serial code as the problem size grows. The numerical value is of not much interest. But, a very low figure would indicate that the algorithm was ill-conceived. The second may give a clue as to what could be a suitable ratio of the partition dimensions in more complex decompositions. It would also verify if the assumption that the winds are mostly E-W is true. The third criterion indicates how well the algorithm will scale when a global 3-D Semi-lagrangian model is used. As far as the fourth criterion is concerned, speedups will be obtained by comparing the performance of the code on a 32 node CM-5 with the performance on a single node on the CM-5. The last criterion is important as it indicates the presence of errors other than roundoff. This done by finding out if the difference between the interpolated values obtained from both the serial and parallel implementations are homogenous for a given resolution.

---

[1] Weather Modelling project undertaken at NASA Goddard Laboratory, Maryland

C. Design

The problem of parallelizing the bilinear and bicubic interpolation is first considered by using a simple approach. In this lasagna decomposition each node receives an input partition of the latitude, longitude grid and each of the two wind components [Fig 2]. Each node then computes a set of points (ordered pairs) (ii,jj) from the latitude, longitude grid, based on which an interpolation is carried out. In case of the bilinear interpolation, associated with the set of points (ii,jj) there exist four sets of points (ii,jj+2), (ii,jj+3), (ii+1),(jj+2), (ii+1,jj+3) which are required for the interpolation [Fig. 3]. In case of the bicubic interpolation there are sixteen such sets of points [Fig. 4] [2].

In the first approach to interpolation, the algorithm is outlined below.

- Do for all (ii,jj)
  - Find the 4 points (ii,jj+2), (ii,jj+3), (ii+1,jj+2), (ii+1,jj+3)
  - Obtain the values of the wind component at above points.
    - If the value is not available locally, *fetch* from the appropriate neighboring node
  - Find the interpolation point.
  - Compute the value at interpolation point.
- End of do loop

It is obvious that though the above approach is simple, it is impractical to implement because of the huge communication overhead involved. For, every point that is not available locally, a communication call will have to be generated to fetch the value at that point. There are at least two approaches which lower the communication overhead. In both the approaches strips of the wind components are shifted from node to node. The shifting occurs in both the southern and northern directions. The amount of shifting is determined by the approach discussed in the next section.

D. Pre-Determination of Degree of Circulation

If the assumption that the winds in the North-South direction mostly result in nearest neighbor communicationis valid , then the following strategy may be used. For any given global indice (ii, jj), we determine the node it belongs to. The inter-nodal distance is then simply the difference between the owner node and user node. But, this distance has a direction, i.e. the "owner" node may be reached in the North or South Direction [Fig. 5]. Hence, this is converted to the inter-nodal distance between the two nodes when going in the North direction. Moreover, if the inter-nodal distance exceeds half the number of nodes, the shorter path is in the Southern direction. For, either interpolation there are two extreme points in the North-South direction [Fig. 3,4]. The inter-nodal distance is computed for both the points separately, for all the indices on a node. The maximum inter-nodal distance in a direction (both North and South are evaluated) is then found by a global reduction. The two maxima in the North-South direction govern the circulation.

The algorithm for *the pre-determination of degree of circulation* is below

- ny = Number of nodes (processors) used
- Let 'myaddr' denote the address of the node executing the code
- Let 'ownp0' denote nodal address where value 'jj(i,j)+0' resides
- Let 'ownp3' denote nodal address where value 'jj(i,j)+3' resides
- Number of elements in E-W direction for a partition is imp3
- Number of elements in N-S direction for a partition is range
- for j = 1, range

[2] Hereafter, the discussion will consider only the bilinear interpolation, but the same reasoning holds good for the bicubic interpolation. Appendix A covers the bicubic interpolation algorithm.

- for i = 1, imp3
  - ownp0 = jj(i,j) / partition width
  - ownp3 = (jj(i,j) + 3) / partition width
  - Find the inter-nodal distance in *northern direction* from 'myaddr' to both ownp0 and ownp3
    - distp0 = ownp0 - myaddr
    - distp3 = ownp3 - myaddr
    - if (distp0 .lt. 0),distp0 = ny + distp0
    - if (distp3 .lt. 0),distp3 = ny + distp3
  - Find the shortest inter-nodal distance for both ownp0, ownp3, and update the local maximum inter-nodal distances if required

- Find the global max. northern dist. (reduction operation)
- Find the global max. southern dist. (reduction operation)

Expansion of Updation of local maximum inter-nodal distances

- Let distp0 be the inter-nodal distance in *northern direction* from 'myaddr' to ownp0
- Let distp3 be the inter-nodal distance in *northern direction* from 'myaddr' to ownp3
- If distp0 .gt. (ny / 2)
  - distp0 = ny - distp0
  - if (max. southern dist .lt. distp0) max. southern dist = distp0
- Else
  - if (max. northern dist .lt. distp0) max. northern dist = distp0
- If distp3 .gt. (ny / 2)
  - distp3 = ny - distp3
  - if (max. southern dist .lt. distp3) max. southern dist = distp3
- Else
  - if (max. northern dist .lt. distp3) max. northern dist = distp3

E. Reconstruction of the Sets of Points Approach

In the second approach we wish to fully reconstruct the four associated sets of points (ii,jj+2), (ii,jj+3), (ii+1),(jj+2), (ii+1,jj+3) in four arrays for every node. Then the four arrays are indexed to obtain the four points required for interpolation on every node. As a wind component has been partitioned as shown in Fig. 2, there is an upper and a lower bound on the index jj in the North-South direction that decides which values of a wind component may be found locally. Thus, at every shift of the wind strips, the four arrays maybe partially reconstructed using the new bounds obtained with every shift. At the end of the shifting the arrays should be fully reconstructed. However, there will be points such that the points (ii,jj) will be local but (x,jj+2) or (x,jj+3) may not be local as seen in figure 3 [3]. Thus it will not be possible to reconstruct *entirely locally* the four associated sets of points for all jj that that fall within the bounds of the node's partition. *If the reconstruction is not done entirely locally, there would be additional communication overhead to fetch the missing values one at a time, as in the first approach.* This hitch may be rectified by initially creating a 3-element wide strip in the Northern direction by fetching those elements from the neighboring nodes [Fig. 6]. The creation of the Northern buffer is created before performing any shifts. The

[3] x maybe ii or ii + 1. This does not matter as in the East-West direction the elements of the wind component are local to a node.

original partition and this northern buffer is now termed a segment. These segments are shifted instead of the original partition. The algorithm for the second approach is outlined as follows, which is executed once for each of the two wind components.

First level refinement for the algorithm.

- Number of elements in E-W direction for a partition is imp3
- Number of elements in N-S direction for a partition is range
- Pre-determine the degree of circulation
- Construct the northern buffer.
- Select elements from segment
- Fetch segments and select elements from southern direction
- Fetch segments and select elements from northern direction
- Perform local interpolation

In this approach, one is guaranteed that if the value at jj(i,j) is resident on a node, then the value at jj(i,j)+3 is resident [4]. Therefore, only the node where jj(i,j) is resident, need be considered for determination of the local maximum inter-nodal distance. This the only deviation from the strategy outlined in the previous subsection.

Algorithm for *construction of northern buffer*

- Copy original wind partition to temporary buffer (depart)
- If partition width less than 3 then
  - While northern buffer incomplete
    * Send depart to southern node
    * Receive partition into northern buffer from northern node
    * Copy received partition into depart
  - End of while
- Else (partition width is .ge. 3)
  - Send original partition to southern node
  - Receive new partition in northern buffer from northern node
- Append northern buffer to original wind partition. This will be called segment.

[4] As there is the 3-element northern buffer appended to the original partition.

Algorithm for *fetching of segments from southern direction*

- Current partition bounds = Original partition bounds
- Copy original segment into temporary buffer (depart)
- For number of hops in southern direction
  - Send upper & lower bounds of current partition to northern node.
  - Receive upper & lower bounds of new partition from southern node.
  - Send 'depart' to northern node.
  - Receive new segment from southern node in buffer 'landing'.
  - Current partition bounds = new partition bounds
  - Copy 'landing' into 'depart'
  - Select elements from landing (new segment)

Algorithm for *fetching of segments from northern direction*

- Current partition bounds = Original partition bounds
- Copy original segment into temporary buffer (depart)
- For number of hops in northern direction
  - Send upper & lower bounds of current partition to southern node.
  - Receive upper & lower bounds of new partition from northern node.
  - Send 'depart' to southern node.
  - Receive new segment from northern node in buffer 'landing'.
  - Current partition bounds = new partition bounds
  - Copy 'landing' into 'depart'
  - Select elements from landing (new segment)

Algorithm for *selection of elements from segment*

- Number of elements in E-W direction for a partition is imp3
- Number of elements in N-S direction for a partition is range
- Input :
  - lower & upper partition bounds [Fig. 6]
  - segment (corresponding to partition as in Fig 6)
  - set of points {(ii,jj)}
  - arrays to hold the four sets of points - w1, w2, w3, w4
- For i = 1, imp3
  - For j = 1, range
    * If (jj(i,j) .le. upper) .and. (jj(i,j) .ge. lower)
      · w1 = segment(ii(i,j), jj(i,j) +2)
      · w2 = segment(ii(i,j), jj(i,j) +3)
      · w3 = segment(ii(i,j)+1, jj(i,j) +2)
      · w4 = segment(ii(i,j)+1, jj(i,j) +3)

Algorithm for *bilinear interpolation.* Fig. 3 explains the terms delx, dely.

- Number of elements in E-W direction for a partition is imp3
- Number of elements in N-S direction for a partition is range
- for j = 1, range
  - for i = 1, imp3
    * value at interpolated point = (1.0 - delx(i,j)) * ((1.0 - dely(i,j)) * w1(i,j) + dely(i,j) * w2(i,j)) + delx(i,j) * ((1.0 - dely(i,j)) * w3(i,j) + dely(i,j) * w4(i,j))

The drawback with this approach is that the communication overhead for creating the northern buffer increases when the width of an individual strip is smaller than the overlap region [Fig. 7].

F. Reconstruction of Segment Approach

The third approach recognizes the fact that given the set of computed points (ii,jj) on each node, the node only needs to see a segment of the entire wind component to perform a local interpolation; there is no necessity for reconstructing the associated four sets of points. This segment is not the same as in the second approach, in that its size is not a static quantity. The size of the segment depends on how far back one needs to go along the fluid trajectory to perform an interpolation, and hence is a dynamic quantity. Thus, no extra communication is needed to establish a northern buffer. Additionally, only the *original* partitions of the wind component need be shifted to create the segment. The circulated wind strips are appended as shown in Fig. 8. Some, book-keeping needs to be done in order to the relative position of the partition in the segment and for generation of local indices. The interpolation is then carried out locally. The four points are obtained by the use of the local indices. The algorithm is outlined below.

The *first-level refinement* for the third approach. The individual steps are elaborated subsequently.

- Number of elements in E-W direction for a partition is imp3
- Number of elements in N-S direction for a partition is range
- Pre-determine the degree of circulation
- Copy the partition into a large buffer called segment

- Fetch partitions from the southern direction and append to segment.
- Fetch partitions from the northern direction and append to segment.
- Convert global indices to local indices.
- Perform the interpolation locally.

The pre-determination of the degree of circulation is exactly as outlined in section 3.1.

The algorithm for *fetching of partitions from the southern direction and appending to the segment* is as below.

- Current partition bounds = Original partition bounds
- Copy partition into temporary buffer (depart)
- For number of hops in southern direction
  - Send upper & lower bounds of current partition to northern node.
  - Receive upper & lower bounds of new partition from southern node.
  - Send 'depart' to northern node.
  - Receive new partition from southern node in buffer 'landing'.
  - Update parameters needed for generation of local indices.
  - Move up the previously filled columns in the segment by width of the new partition (southern partitions occupy lower indices in the segments [Fig. 8]).
  - Append the new partition to the 'southern end' of the segment.
  - Update width of segment.
  - Copy 'landing' into 'depart'

The algorithm for *fetching of partitions from the northern direction and appending to the segment* is as below.

- Current partition bounds = Original partition bounds
- Copy partition into temporary buffer (depart)
- For number of hops in northern direction
  - Send upper & lower bounds of current partition to southern node.
  - Receive upper & lower bounds of new partition from northern node.
  - Send 'depart' to southern node.
  - Receive new partition from northern node in buffer 'landing'.
  - Update parameters needed for generation of local indices.
  - Append 'landing' to the 'northern end' of the segment; northern partitions occupy higher indices in the segment [Fig 8].
  - Update width of segment.
  - Copy 'landing' into 'depart'

The local indices are generated with the following considerations :

- The first column in any segment does not necessarily correspond to column '1' in the wind component.
- There maybe a wrap around in the southern direction; i.e. the southern most node communicated may cross the south pole.
- Similarly there may be a wrap around in the northern direction.

The algorithm *generation of local indices* is as below. Fig. 9 illustrates the algorithm.

- co_lower = lower bound of non-wrapped part of segment
- wr_lower = lower bound of wrapped part of segment
- wrapd = width of wrapped part of segment.
- Let 'njj' denote the set of new (local) indices
- If there is no wrap around in either direction
  - for i = 1, imp3
    * for j = 1, range
      · njj(i,j) = jj(i,j) - co_lower + 1
- If there is wrap around in southern direction
  - for i = 1, imp3
    * for j = 1, range
      · if jj(i,j) .ge. wr_lower then njj(i,j) = jj(i,j) - wr_lower + 1
      · Else njj(i,j) = jj(i,j) + wrapd
- If there is wrap around in northern direction
  - for i = 1, imp3
    * for j = 1, range
      · if jj(i,j) refers to the wrapped part then njj(i,j) = jj(i,j) + jmp3 - co_lower + 1
      · Else njj(i,j) = jj(i,j) - co_lower + 1

The algorithm for *the bilinear interpolation* is as below. This is similar to the algorithm in the serial code, excepting that local indices are used to index a segment of the wind component on every processor.

- Let 'seg' denote segment
- Let 'njj' denote new (local) jj
- Number of elements in E-W direction for a partition is imp3
- Number of elements in N-S direction for a partition is range
- for j = 1, range
  - for i = 1, imp3
    value at interpolated point = (1.0 - delx(i,j)) * ((1.0 - dely(i,j)) * seg(ii(i,j), njj(i,j)+2)) + dely(i,j) * seg(ii(i,j), njj(i,j)+3) + delx(i,j) * ((1.0 - dely(i,j)) * seg(ii(i,j)+1, njj(i,j)+2) + dely(i,j) * seg(ii(i,j)+1, njj(i,j)+3))

G. Implementation

The project was implemented on CM-5 (CMOST 7.3) using the native message passing library [6]. This was developed as a proof of concept that the message passing paradigm was a suitable way of implementing a parallel version of the Semi-Lagrangian Transport code. It is hence highly modular and has about five levels of subroutine calls. Though this allows for the easy testing of the code, it conflicts with the requirements of high performance computing in that the overall running time drastically increases. This prompted us in measuring computations without the function calling overhead. Therefore the running time of a production code would be between the timings with the function call overhead and without the overhead. Ideally we would like it to be closer to the timings without the overhead.

The third approach was implemented and instrumented for measuring performance as discussed in the next section.

H. Performance

The timings were obtained on CM-5 on CMOST 7.3 on a 32 node partition. The same code was run on a single node of the CM-5 in order to measure the speedup [5]. The performance is discussed for both bicubic and bilinear interpolations with respect to the five criteria discussed in the Introduction. They are as follows :

- The overall running time.
- The maximum displacement in terms of nodal distance in the N-S direction.
- Ratio of calculation to communication.
- Speed-up of interpolation
- Accuracy of calculations

Before discussing the performance issues, we will first define the following terms :

- Averaged Maximum Number of hops to northern nodes NI.
- Averaged Maximum Number of hops to southern nodes SI.
- Total Number of hops in both directions $TI = (NI + SI)$.
- The total communication time per iteration of fetching or gathering partitions from southern (or northern nodes) T_{exp}. This is the sum of the time for sending and receiving a bound of the partition plus sending and receiving a partition. This time is derived by measuring the communication time for various message sizes.
- Time required for bicubic (or bilinear) interpolation on the 32 node partition (without function overhead) T_{int}.
- Time required for bicubic (or bilinear) interpolation on the single node (without function overhead) T_{int_s}.
- Speedup of bicubic (or bilinear) interpolation $S_{int} = T_{int_s}/T_{int}$.
- Time required for reduction of address calculations on the 32 node partition (without function overhead) T_{ra}. This overhead is a result of the implementation of the code for the Bicubic Interpolation alone and does not show up at the design stage.
- Time required for reduction of address calculations on the single node (without function overhead) T_{ra_s}.
- Speedup of reduction of address calculations $S_{ra} = T_{ra_s}/T_{ra}$.
- Time required for conversion of the global indices to local indices on the 32 node partition (without function overhead) T_{li}.
- Time required for conversion of the global indices to local indices on the single node (without function overhead) T_{li_s}.
- Speedup of conversion to local indices $S_{li} = T_{li_s}/T_{li}$.
- Time taken for node-to-node communication (circulation of partitions of winds and the bounds of the partition) on the 32 node partition T_{co_n}.
- The product of TI and T_{exp} is $T_{co_{ne}}$. This is approximately equal to T_{co_n} .
- Time required for book-keeping and buffer copying in creation of segment (no function calling overhead or node-to-node comunnication is accounted for) on the 32 node partition T_{bc}.

---

[5] The flow of program execution within a subroutine is however, not necessarily the same. This results in less than ideal speedup for some parts of the code.

- Time required for book-keeping and buffer copying in creation of segment (no function calling overhead is accounted for) on the single node T_{bc_s}. There is no communication required on the single node for creation of segment.
- Time required for creation of segment on the 32 node partition $T_{cs} = T_{bc} + T_{co_n}$.
- Speedup of buffer copying in creation of segment $S_{bc} = T_{bc_s}/T_{bc}$. This indicates how well the buffer copying alone scales when comparing the timings on the 32 node and the single node in CM-5. The next speedup includes the cost of the node-to-node communication which is a part of the "creation of segment" function It is a "gathering" process. .
- Speedup of creating a segment $S_{cs} = T_{bc_s}/T_{bc}$. This includes the node-to-node communication overhead in the computation of the speedup.
- Time taken for global maximum operation on the 32 node partition T_{co_g}.
- Time taken for finding local maxima in determining degree of circulation (no global reduction involved) on 32 node partition T_{dcl}.
- Time taken for determining degree of circulation $T_{dc} = T_{dcl} + T_{co_g}$.
- Time taken for finding local maxima in determining degree of circulation on single node partition. Note the single node does not need a global maximum operation T_{dcl_s}.
- Speedup of finding local maxima (calculations) $S_{dcl} = T_{dcl_s}/T_{dcl}$. This indicates how well the computation alone scales when comparing the timings on the 32 node and the single node in CM-5. The next speedup includes the cost of the global maximum operation.
- Speedup of finding degree of circulation $S_{dc} = T_{dcl_s}/T_{dc}$. This speedup includes the cost of the global maximum operation.
- Time taken for latitude, longitude index calculations (no function calling overhead) on the 32 node partition T_{ll}.
- Time taken for latitude, longitude index calculations (no function calling overhead) on the single node T_{ll_s}.
- Speedup of latitude, longitude index calculations $S_{ll} = T_{ll_s}/T_{ll}$.
- Total calculation (overall) time (with function calling overhead and timing overhead) on the 32 node partition T_{ca_o}. This is the sum of T_{ca_a} and function calling overhead.
- Total communication time (includes both global communication and node to node communication) on the 32 node partition $T_{co} = T_{co_g} + T_{co_n}$
- Ratio of calculation time (overall) to communication time on the 32-node CM-5 $R_a = T_{ca_o}/T_{co}$
- Total calculation time (adjusted, i.e. no function calling overhead) on the 32 node partition $T_{ca_a} = T_{ll} + T_{dcl} + T_{bc} + T_{li} + [T_{ra}] + T_{int}$ [6]
- Ratio of calculation time (adjusted) to communication time on the 32-node CM-5 $R_a = T_{ca_a}/T_{co}$.
- Total calculation time (adjusted, i.e. no function calling overhead) on the single node $T_{ca_{as}} = T_{ll_s} + T_{dcl_s} + T_{bc_s} + T_{li_s} + [T_{ra_s}] + T_{int_s}$.
- Speedup (adjusted) $S_a = T_{ca_{as}}/(T_{ca_a} + T_{co})$.
- Total (Overall) running time T_{ot} (includes function calling overhead, communication overhead and timing overhead) on a 32 node partition on CM5. $T_{gt} = T_{ca_o} + T_{co}$. This is also the sum of T_{ca_a}, T_{co} and function calling overhead.

[6] T_{ra} is shown in square brackets as there is no reduction of address calculations for Bilinear Interpolation. Similarly T_{ra_s} is shown in square brackets for the definition of $T_{ca_{as}}$.

- Total (Overall) running time T_{ot_s} (includes function calling overhead and timing overhead) on a single node on CM5. This is the sum of $T_{ca_{a_s}}$ and function calling overhead.

- Speedup (overall) $S_o = T_{gt_s}/T_{gt}$.

The first table shows the non-linear nature of the communication overhead versus message size. The overhead decreases as message size increases from 0 to 8 bytes. Somewhere between a size of 8 to 16 bytes, the overhead starts increasing again and again becomes lower at 32 bytes. The machine also exhibits a high latency; for example messages of size 64 bytes takes nearly the same amount of time as a message of 256 bytes for a synchronous send and receive.

a. **Table 1 : Communciation Time vs Message Size**

Message Size (bytes)	Communication Time (ms)
0	0.226
1	0.200
2	0.176
4	0.172
8	0.155
16	0.221
32	0.218
64	0.232
256	0.250
512	0.300
1280	0.428
2048	0.799
5120	1.584
7168	1.751
10000	2.591
22528	5.218

Table 2 shows the message size used at a given resolution. From Tables 1 and 2, we derive the values for T_{exp}. T_{exp} is the sum of the times for communicating the bound of a partition (4 bytes long) and the partition itself (message size at given resolution).

b. **Table 2 : Message Size Used At Given Resolution**

Resolution (bytes)	Message Size (bytes)
(8x64)	256
(64x64)	2048
(128x128)	5120
(128x256)	7168
(256x512)	22528

1. Bicubic Interpolation

The reconstruction of segment approach was adopted for the bicubic interpolation. Table 3 shows the overall results with the function calling overheads. Table 4 shows the overall results without function calling overhead [7] . Tables 5 to 9 show the functional breakup of the timings in Table 4.

[7] The function calling overhead does not apply to the timing of the communication routines.

Table 3 : Overall Results

Resolution (I,J)	T_{ot} ms	T_{ot_s} ms	S_o	T_{ca_o} ms	T_{co} ms	R_o
(8,64)	14.62	190.00	13.0	11.86	2.76	4.29
(64,64)	63.60	1146.20	18.0	58.41	5.18	11.26
(128,128)	211.19	4500.30	21.2	213.76	8.17	24.94
(128,256)	396.70	9122.00	23.0	388.00	8.66	44.78
(256,512)	1740.40	——	–	1718.5	21.91	78.43

Table 4 : Results Without Function Calling Overhead

Resolution (I,J)	T_{ca_a} ms	R_a	$T_{ca_{as}}$ ms	S_a
(8,64)	4.85	1.75	85.1	11.18
(64,64)	26.42	5.09	522.8	17.5
(128,128)	92.04	11.27	1985.00	19.8
(128,256)	173.45	20.01	4034.00	22.15
(256,512)	714.75	32.62	——	——

Table 5 : Functional Breakup of Table 2 Timings

Resolution (I,J)	T_{ll} ms	T_{ll_s} ms	S_{ll}
(8,64)	0.58	15.16	26.14
(64,64)	3.20	92.34	28.86
(128,128)	12.76	364.7	28.58
(128,256)	27.57	732.5	26.56
(256,512)	99.24	——	——

Table 6 : Functional Breakup of Table 2 Timings

Resolution (I,J)	T_{co_g} ms	T_{dcl} ms	T_{dc} ms	T_{dcl_s} ms	S_{dcl}	S_{dc}
(8,64)	0.0774	0.59	0.667	13.82	23.42	20.70
(64,64)	0.0829	3.30	3.38	77.26	23.41	22.88
(128,128)	0.0930	12.99	13.08	305.68	23.53	23.37
(128,256)	0.0910	26.29	26.38	613.21	23.32	23.24
(256,512)	0.1070	108.8	108.9	——	——	——

Table 7 : Functional Breakup of Table 2 Timings

Resolution (I,J)	T_{co_n} ms	T_{bc} ms	T_{cs} ms	T_{bc_s} ms	S_{bc}	S_{cs}
(8,64)	2.69	1.51	4.2	5.52	3.66	1.31
(64,64)	5.11	6.81	11.92	33.65	4.94	2.82
(128,128)	8.08	17.31	25.39	128.77	7.44	5.07
(128,256)	8.57	24.41	32.98	293.8	12.04	8.91
(256,512)	21.8	87.35	109.15	——	——	—-

Table 8 : Functional Breakup of Table 2 Timings

Resolution (I,J)	T_{li} ms	T_{li_s} ms	S_{li}	T_{ra} ms	T_{ra_s} ms	S_{ra}
(8,64)	0.274	2.16	7.88	0.85	15.03	17.68
(64,64)	1.55	13.77	8.88	5.42	89.55	16.52
(128,128)	4.04	53.17	13.16	20.93	360.56	17.22
(128,256)	5.55	109.65	19.75	42.80	734.56	17.16
(256,512)	17.2	——	——	90.92	——	——

Table 9 : Functional Breakup of Table 2 Timings

Resolution (I,J)	T_{int} ms	T_{int_s} ms	S_{int}
(8,64)	1.04	34.02	32.7
(64,64)	6.13	216.25	35.28
(128,128)	24.02	777.20	32.15
(128,256)	46.81	1550.9	33.12
(256,512)	311.29	——	——

Table 10 : Experimental Confirmation of Node-to-Node Communication Time T_{co_n}

Resolution (I,J)	NH	SH	TH	T_{exp} ms	$T_{co_n e}$ ms
(8,64)	4.0	1.5	5.5	0.422	2.32
(64,64)	4.2	1.5	5.7	0.971	5.53
(128,128)	3.1	2.0	5.1	1.756	8.96
(128,256)	2.0	2.0	4.0	1.923	7.69
(256,512)	2.0	2.0	4.0	5.390	21.56

Since the code has 5 levels of subroutine calling, there is approximately a factor of 2 difference between the overall timings with and without the function calling overhead. A good production code would have a *overall timing* somewhere between these two figures [8].

The *inter-nodal distance* seems to be fairly static for both components (north, south directions inclusive). On an average this is 4 "cshifts" in the Northern direction or 1.5 "cshifts" in the Southern for a resolution of 8 x 64. This appears to stabilize at 2 "cshifts" in either direction at higher resolutions as seen from Table 10. This is close to the assumption that the winds generate mostly nearest neighbor communication.

The *ratio of calculation to communication* (R_o or R_a) (Tables 3,4) falls with increasing resolution because of two factors, namely :

- High latency in the communciation overhead. Due to high latency, it takes almost the same time to send and receive a message of 256 bytes as a message of size 64 bytes.
- The decreasing total number of cshifts TI. The total number of "cshifts" falls from 5.5 to 4.

It is obvious that this algorithm is dominated by overheads other than communication at high resolutions, which will be established shortly [9].

It is also noticed since the function calling overhead is high, the ratio R_a is less than R_o. Consequently, the *speedup* S_a is lower than S_o. However, S_o itself is much lower than ideal. At low resolutions, this is primarily due to the high communication overhead [10], but does not hold true at higher resolutions. Hence, a complete picture of speedup will not emerge without a detailed discussion at the functional level of the code. We will first tabulate the functions in order of increasing speedup for the resolution of 8x64 (Table 11) from the data in Tables 3 to 7 and then analyze the reasons for the performance. We will do this similarly for a higher resolution of 128x256 (Table 12).

[8] It is not desirable to eliminate totally the function calling overhead except where it is in excess. For example, one should inline the calls to the actual interpolation routine as it is called within an inner loop of a doubly nested loop.

[9] Communication overhead is only 1.28 % at a resolution of 128x256.

[10] At a resolution of 8 x 64, the ratio R_o is as low as 4.29.

a. **Table 11 : Functional Breakup of Timings at a Resolution of 8 x 64:**

Function	Speedup	%age of $T_{ca_a} + T_{co}$	%age of T_{ot}
Creation of Segment (cs)	1.31	53.98 %	28.1 %
Buffer Copying in above (bc)	3.66	19.79 %	10.3 %
Conversion to Local Indices (li)	7.88	3.65 %	1.9 %
Reduction of Address Calculation (ra)	17.68	11.14 %	5.8 %
Degree of Circulation (dc)	20.70	8.83 %	4.6 %
Computation alone in above (dcl)	23.42	7.68 %	4.0 %
Latitude, Longitude Index Calculations (ll)	26.14	7.68 %	4.0 %
Interpolation (int)	32.7	13.45 %	7.0 %

The "creation of segment" function shows a very low speedup primarily because it involves node-to-node communication with high latency. But, this does not account for all the loss in speedup (speedup of buffer copying is also at a low of 3.66). This is because there are more places in the 32-node run where there is *buffer copying* [11]. The term buffer copying includes memory to memory copying and the book-keeping associated with it. There is some load imbalance in the memory to memory copying, especially at the low resolutions, due to nodes at the poles having to deal with a larger partition (8x5 instead of 8x2) as shown in Figure 2. These nodes also have to handle with the pole-wrap problem [12] which are absent in other nodes, leading to an additional load imbalance. The load imbalance is apparent from the fact that at the next higher resolution (64x64), though the grid points have increased by 8, the amount of time spent in buffer copying increases by 4. Otherwise the buffer copying scales with number of grid points. In the single node we find that the time spent in buffer copying scales well with the number of grid points.

Taking into account the implementation and the sizes of buffers involved in both the runs, the single node run performs only about 6 times the amount [13] of buffer copying as the 32 node run. This is slightly more than the speedup of 3.66. However, due to the load imbalance the drop in speedup can be expected . As the buffer copying alone takes 19.79 % of running time (without function calling overhead), this has a significant effect on the overall speedup [14] .

The speedup of "conversion to local indices" is also low. This could be due to an error in measurement of a very small interval of time (0.27 ms) in the 32-node run. There is also the possibility that due to a load imbalance that affects the nodes corresponding to the polar regions which have to translate indices that cross the poles, that the speedup is low. This code involves testing of a condition in the innermost loop of a doubly nested loop. This does not occur in the single node run where only a straight forward translation to local indices takes place. But, the impact of this function is slight as it constitutes only 3.65 % of the running time (which is the least of all the functions).

The "reduction of address calculations" constitutes about 11.14 % of the running time. The number of times it is called for the 32-node run is equal to the number of grid points in a partition. There are 16 grid points at a resolution of 8x64. Therefore the approximate time per call is only 53 μs which spans 8 integer operations. The measurement of such a small time is prone to errors, which may have led to a somewhat moderate speedup of 17 to 18. [15]

The "degree of circulation" has a moderate speedup of 20.7. When the global maximum operation is left out, the speedup is about 23.4, which is fairly high. As the global maximum operation is a constant overhead, its effect at higher resolutions becomes negligible. As the code involves testing of conditions and as the quantities involved are dynamically varying with both time and space, it is difficult to pinpoint how the speedup is affected. Additionally this function has no significance for the single node run.

The "calculation of latitude, longitude indices" shows a good speedup of 26.1. This involves a mix of integer and floating point arithmetic.

The actual "bicubic interpolation" shows more than ideal speedup. The timing of this routine has the same drawback as the "reduction of address calculations". As the time intervals involved are very small, about 65 μs, the measurements are prone to errors. However, it may be concluded that this function has a good speedup even after

---

[11] Though both the single node and the 32 node run the same code, because the single node has no communication to perform, it does not need to perform the buffer copying associated with the sending and receiving of partitions. In other words the single node program traces a different path through the code. We will hereafter refer to this as a *run*.

[12] Fetching of partition by wrapping around the pole, needs more book-keeping.

[13] Not to be confused with number of places

[14] Inclusive of the communication cost, it takes the lion's share of the running time at 53.98 %.

[15] Even the timings for the single node run suffer from the same defect.

discounting for any errors.

The overall speedup S_a is therefore poor because functions which have poor speedups (< 16) constitute 57.6 % of running time (with no function calling overhead), those with medium speedup ($16 \leq speedup \leq 23$) constitute 20.0 % and those with good speedups 22.2 % (> 23) [16].

b. **Table 12 : Functional Breakup of Timings at a Resolution of 128x256:**

Function	Speedup	%age of $T_{ca_a} + T_{co}$	%age of T_{ot}
Creation of Segment (cs)	8.91	18.1 %	8.31 %
Buffer Copying in above (bc)	12.04	13.40 %	6.15 %
Reduction of Address Calculation (ra)	17.16	23.53 %	10.8 %
Conversion to Local Indices (li)	19.75	3.05 %	1.4 %
Degree of Circulation (dc)	23.24	14.49 %	6.65 %
Computation alone in above (dcl)	23.32	14.42 %	6.62 %
Latitude, Longitude Index Calculations (ll)	26.56	15.14 %	6.95 %
Interpolation (int)	33.1	25.70 %	11.8 %

At this resolution cache miss penalties play an important role as seen from Tables 5 to 9 where there is a jump in speedups from the 128x128 to 128x256 resolution [17].

Using the same reasoning as in the case of 8x64 resolution for the "creation of segment" function, the single-node run has about 6 times the amount of *buffer copying* as the 32 node-run. However, the speedup of buffer copying is 12.04 which is two times more than expected. This is most likely due to caching effects. It can also be seen that there is an increase of 1-1/2 times in T_{bc} compared with a two-fold increase in the number of grid points from a resolution of 128x128. This indicates that the load imbalance due to unven partition size and pole wrapping is becoming less severe compared to a resolution of 8x64.

The node-to-node communication is still expensive as the speedup inclusive of this cost falls to 8.91 from 12.04. However, the adverse effect of the "creation of segment" function is less on the overall speedup as it only takes 18.1 % of the running time compared to 53.98 % at the 8x64 resolution.

The speedup of "conversion to local indices" increases to 19.75. This is certainly due to cache misses, because at a resolution of 128x128 the speedup is only 13.2. At the 128x128 resolution the accuracy of the timing measurements should be good as there are 512 iterations taking place. The speedup is not better than 19.75 probably because of the load imbalance problem at the nodes corresponding to the poles.

The measurement of the total time for the "reduction of calculations" suffers from the drawback discussed earlier. The speedup for this function shows little variance (Table 8). But since it now constitutes 10.8 %, it does little to contribute to a better speedup.

The "degree of circulation" has a fairly high speedup of 23.24. The global maximum operation now constitutes a negligible amount of the time spent in the function, as seen in the difference in speedups S_{dcl} and S_{dc}.

The "calculation of latitude, longitude indices" shows a good speedup of 26.56.

The actual "bicubic interpolation" as before shows more than ideal speedup due to the drawback discussed earlier.

Thus the overall speedup S_a is better because functions which take major part of running time (55.35 %) have good speedups (> 23). About 12 % of the code has medium speedup ($16 \leq speedup \leq 23$) and the rest has a poor speedup (< 16).

An *accuracy* of upto (but not including) the fifth decimal place was obtained between the parallel and serial implementations.

Table 10 confirms the *accuracy of the node-to-node communication overhead* . The results from the secondary experiment, $T_{co_{ne}}$ and T_{co_n} match closely.

---

[16] The classification of poor, medium and good speedups may seem arbitrary. But this distinction serves the purpose considering that timings were obtained on a 32 node machine.

[17] It was not possible to run at a resolution of 256x256 on the single node due to lack of memory.

2. Bilinear Interpolation

The reconstruction of segment approach was also adopted for the bilinear interpolation. Table 13 shows the overall results with the function calling overheads. Table 14 shows the overall results without function calling overhead [18] . Tables 15 to 19 show the functional breakup of the timings in Table 14. We obtain very similar results for certain functions in the code, because the code is essentially the same excepting for the actual interpolation routine. There is also no "reduction of address calculation" function in this code.

Table 13 : Overall Results

Resolution (I,J)	T_{ot} ms	T_{ot_s} ms	S_o	T_{ca_o} ms	T_{co} ms	R_o
(8,64)	10.50	96.88	9.23	8.22	2.27	3.62
(64,64)	41.12	589.6	14.34	36.63	4.49	8.176
(128,128)	128.46	2280.0	17.74	112.03	6.43	18.97
(128,256)	240.47	4717.0	19.60	232.05	8.42	27.56
(256,512)	955.20	——	–	930.68	24.53	37.95

Table 14 : Results Without Function Calling Overhead

Resolution (I,J)	T_{ca_a} ms	R_a	$T_{ca_{as}}$ ms	S_a
(8,64)	3.37	1.48	47.31	8.39
(64,64)	16.17	3.60	290.3	14.05
(128,128)	52.63	8.18	1142.0	19.34
(128,256)	98.37	11.68	2349.1	22.03
(256,512)	381.45	15.55	——	——

Table 15 : Functional Breakup of Table 2 Timings

Resolution (I,J)	T_{ll} ms	T_{ll_s} ms	S_{ll}
(8,64)	0.61	14.72	24.09
(64,64)	2.95	91.87	31.14
(128,128)	11.19	364.8	32.60
(128,256)	22.53	733.0	32.60
(256,512)	91.23	——	——

[18] The function calling overhead does not apply to the timing of the communication routines.

Table 16 : Functional Breakup of Table 2 Timings

Resolution (I,J)	T_{co_g} ms	T_{dcl} ms	T_{dc} ms	T_{dcl_s} ms	S_{dcl}	S_{dc}
(8,64)	0.0756	0.60	0.67	11.55	19.40	17.21
(64,64)	0.0797	3.27	3.35	69.09	21.12	20.64
(128,128)	0.0907	12.88	12.97	276.50	21.47	21.31
(128,256)	0.0890	25.72	25.81	556.30	21.63	21.55
(256,512)	0.0946	102.7	102.79	——	——	——

Table 17 : Functional Breakup of Table 2 Timings

Resolution (I,J)	T_{co_n} ms	T_{bc} ms	T_{cs} ms	T_{bc_s} ms	S_{bc}	S_{cs}
(8,64)	2.20	1.47	3.67	5.50	3.74	1.50
(64,64)	4.41	5.98	10.39	34.59	5.78	3.33
(128,128)	6.34	14.50	20.84	128.69	8.88	6.18
(128,256)	8.33	24.66	32.99	296.50	12.02	8.99
(256,512)	24.43	87.54	111.97	——	——	—-

Table 18 : Functional Breakup of Table 2 Timings

Resolution (I,J)	T_{li} ms	T_{li_s} ms	S_{li}
(8,64)	0.28	2.19	7.96
(64,64)	1.56	13.63	8.74
(128,128)	3.86	53.13	13.76
(128,256)	5.44	108.9	20.02
(256,512)	17.2	——	——

Table 19 : Functional Breakup of Table 2 Timings

Resolution (I,J)	T_{int} ms	T_{int_s} ms	S_{int}
(8,64)	0.410	13.36	32.59
(64,64)	2.40	81.11	33.8
(128,128)	10.21	318.78	31.22
(128,256)	19.25	654.56	32.81
(256,512)	82.81	——	——

Table 20 : Experimental Confirmation of Node-to-Node Communication Time T_{co_n}

Resolution (I,J)	NH	SH	TH	T_{exp} ms	$T_{co_n e}$ ms
(8,64)	3.33	1.67	5.0	0.422	2.11
(64,64)	3.33	1.67	5.0	0.971	4.86
(128,128)	2.00	1.93	3.93	1.756	6.90
(128,256)	2.0	2.0	4.0	1.923	7.692
(256,512)	2.0	2.0	4.0	5.390	21.56

Since the code has 5 levels of subroutine calling, there is approximately a factor of 2 difference between the overall timings with and without the function calling overhead. A good production code would have a *overall timing* somewhere between these two figures. We would like the timings to be closer to the value of $T_{ca_a} + T_{co}$ than T_{ot}.

The *inter-nodal distance* seems to be fairly static for both components (north, south directions inclusive). On an average this is 3.33 "cshifts" in the Northern direction or 1.67 "cshifts" in the Southern for a resolution of 8 x 64. This appears to stabilize at 2 "cshifts" in either direction at higher resolutions as seen from Table 10. This is close to the assumption that the winds generate mostly nearest neighbor communication.

The *ratio of calculation to communication* (R_o or R_a) (Tables 13,14) falls with increasing resolution because of two factors, namely :

- High latency in the communciation overhead. As observed earlier, it takes almost the same time to send and receive a message of 256 bytes as a message of size 64 bytes.
- The decreasing total number of cshifts TI. The total number of "cshifts" falls from 5.0 to 4.

It is obvious that this algorithm is dominated by overheads other than communication at high resolutions, which will be established shortly [19].

It is also noticed since the function calling overhead is high, the ratio R_a is less than R_o. Consequently, the *speedup* S_a is lower than S_o. However, S_o itself is much lower than ideal. At low resolutions, this is primarily due to the high communication overhead [20], but does not hold true at higher resolutions. Hence, a complete picture of speedup will not emerge without a detailed discussion at the functional level of the code. [21]. We will first tabulate the functions in order of increasing speedup for the resolution of 8x64 from the data in Tables 13 to 19 (Table 21) and then analyze the reasons for the performance. We will do this similarly for a higher resolution of 128x256 (Table 22).

a. **Table 21 : Functional Breakup of Timings at a Resolution of 8 x 64:**

Function	Speedup	%age of $T_{ca_a} + T_{co}$	%age of T_{ot}
Creation of Segment (cs)	1.50	65.07 %	34.95 %
Buffer Copying in above (bc)	3.74	26.06 %	14.00 %
Conversion to Local Indices (li)	7.96	4.96 %	2.67 %
Degree of Circulation (dc)	17.21	11.88 %	6.38 %
Computation alone in above (dcl)	19.40	10.64 %	5.7 %
Latitude, Longitude Index Calculations (ll)	24.09	10.82 %	5.8 %
Interpolation (int)	32.59	7.27 %	3.90 %

The "creation of segment" function shows a very low speedup primarily because it involves node-to-node communication with high latency. As discussed in the case of the Bicubic Interpolation, this does not account for all the loss in speedup. There is some load imbalance in the memory to memory copying, especially at the low resolutions, due to nodes at the poles having to deal with a larger partition (8x5 instead of 8x2) as shown in Figure 2. These nodes also have to handle with the pole-wrap problem which are absent in other nodes, leading to an additional load imbalance. The load imbalance is apparent from the fact that at the next higher resolution (64x64), though the grid points have

[19] Communication overhead is only 2.64 % at a resolution of 128x256.

[20] At a resolution of 8 x 64, the ratio R_o is as low as 3.62, whereas at 128x256 R_o reaches a high of 37.95.

[21] Another notable aspect is that the overall speedups for the Bilinear Interpolation is lower than that for Bicubic Interpolation. This is because the actual interpolation routine for the Bilinear Interpolation involves only 6 floating-point multiplications compared to 36 for the Bicubic Interpolation.

increased by 8, the amount of time spent in buffer copying increases by 4. Otherwise the buffer copying scales with number of grid points. In the single node we find that the time spent in buffer copying scales well with the number of grid points.

The speedup of buffer copying is also low, as there are more places in the 32-node run where there is buffer copying. Taking into account the implementation and the sizes of buffers involved in both the runs, the single node run performs only about 6 times the amount [22] of buffer copying as the 32 node run. This is slightly more than the speedup of 3.66, but this is expected due to load imbalance. As the buffer copying alone takes 26.06 % of running time (without function calling overhead), this has a significant effect on the overall speedup [23] .

The speedup of "conversion to local indices" is also low, like the Bicubic Interpolation. This could be due to an error in measurement of a very small interval of time (0.28 ms) in the 32-node run. There is also the possibility that due to a load imbalance that affects the nodes corresponding to the polar regions which have to translate indices that cross the poles, that the speedup is low. But, the impact of this function is slight as it constitutes only 4.96 % of the running time (which is the least of all the functions).

The "degree of circulation" has a moderate speedup of 17.21. When the global maximum operation is left out, the speedup is about 19.40, which is medium. As the global maximum operation is a constant overhead, its effect at higher resolutions becomes negligible. As the code involves testing of conditions and as the quantities involved are dynamically varying with both time and space, it is difficult to pinpoint how the speedup is affected. Additionally this function has no significance for the single node run.

The "calculation of latitude, longitude indices" shows a good speedup of 24.09 which involves a mix of integer and floating point arithmetic.

The actual "bilinear interpolation" shows more than ideal speedup. The timing of this routine has the same drawback as the "reduction of address calculations". As the time intervals involved are very small, the measurements are prone to errors. However, it may be concluded that this function has a good speedup even after discounting for any errors.

The overall speedup S_a is therefore poor because functions which have poor speedups (< 16) constitute 70.02 % of running time (with no function calling overhead), those with medium speedup ($16 \leq speedup \leq 23$) constitute 11.88 % and those with good speedups 18.1 % (> 23) [24].

b. **Table 22 : Functional Breakup of Timings at a Resolution of 128x256:**

Function	Speedup	%age of $T_{ca_a} + T_{co}$	%age of T_{ot}
Creation of Segment (cs)	8.99	30.89 %	13.72 %
Buffer Copying in above (bc)	12.02	23.09 %	10.25 %
Conversion to Local Indices (li)	20.02	5.09 %	2.20 %
Degree of Circulation (dc)	21.55	24.17 %	10.73 %
Computation alone in above (dcl)	21.63	24.08 %	10.70 %
Latitude, Longitude Index Calculations (ll)	32.60	21.09 %	9.37 %
Interpolation (int)	32.81	18.02 %	8.01 %

As discussed in the case of the Bicubic Interpolation, at this resolution, the jumps in speedups compared to the 128x128 resolution is due to cache misses. [25]

Using the same reasoning as in the case of 8x64 resolution, the single-node run has about 6 times the amount of buffer copying as the 32 node-run for the "creation of segment" function. However, the speedup of buffer copying is 12.04 which is two times more than expected. This is most likely due to caching effects. It can also be seen that there is an increase of 1.7 times in T_{bc} compared with a two-fold increase in the number of grid points from a resolution of 128x128. This indicates that the load imbalance due to unven partition size and pole wrapping is becoming less severe compared to a resolution of 8x64.

The node-to-node communication is still expensive as the speedup inclusive of this cost falls to 8.99 from 12.02. As the "creation of segment" function still takes 30.89 % (13.72 % with function calling overhead) of the running time, it affects the overall speedup adversely.

[22] Not to be confused with number of places

[23] Inclusive of the communication cost, it takes the lion's share of the running time at 65.07 %.

[24] The classification of poor, medium and good speedups may seem arbitrary. But this distinction serves the purpose considering that timings were obtained on a 32 node machine.

[25] It was not possible to run at a resolution of 256x256 on the single node due to lack of memory.

The speedup of "conversion to local indices" increases to 20.02. This is certainly due to cache misses, because at a resolution of 128x128 the speedup is only 13.76. At the 128x128 resolution the accuracy of the timing measurements should be good as there are 512 iterations taking place. The speedup is not better than 20.0 because of the load imbalance problem at the nodes corresponding to the poles.

The "degree of circulation" has a fairly high speedup of 21.55 The global maximum operation now constitutes a negligible amount of the time spent in the function, as seen in the difference in speedups S_{dcl} and S_{dc}.

The "calculation of latitude, longitude indices" shows more than ideal speedup of 32.60. This is due to errors in measurement of timings.

The actual "bicubic interpolation" as before shows more than ideal speedup due to the drawback discussed earlier.

Thus the overall speedup S_o or S_a is better than at a resolution of 8x64. The speedup is only medium because functions which take major part of running time (60.15 %) have poor or medium speedups (≤ 23). About 30.89 % of the code has poor speedup (< 16) , another 29.26 % has medium speedup ($16 \leq speedup \leq 23$) and the balance 39.85 % the rest has a good speedup (> 23).

An *accuracy* of upto (but not including) the fifth decimal place was obtained between the parallel and serial implementations.

Table 20 confirms the *accuracy of the node-to-node communication overhead* . The results from the secondary experiment, $T_{co_{ne}}$ and T_{co_n} match closely.

I. Conclusions

A reasonable speedup maybe achieved by the message passing version. This problem is hard to parallelize because of the following reasons. At low resolutions, the speedup is poor because of very high communication overhead [26], some load imbalance, and a fair amount of book-keeping and memory management involved. At higher resolutions the amount of book-keeping and memory management dominate the communication [27] and load imbalance overheads. But, unfortunately, these overheads are required for the working of the parallel algorithm and are difficult to reduce. Hence, there is only a medium speedup. If the algorithm were to be ported to a machine with a low latency communication network, the book keeping and memory managment overheads will still result in only a medium speedup. However, there is scope for improving the overall running time by reducing the function call overhead.

[26] For the bicubic interpolation, T_{co} takes 57 % as much time as T_{ca_a}. For the bilinear interpolation, T_{co} takes 68 % as much time as T_{ca_a}, which is even worse.

[27] T_{co} takes only about 30 % as much time as T_{bc} for either the bicubic or the bilinear interpolations. T_{bc} itself takes about 14 % and 24.9 % of T_{ca_a} for the bicubic and bilinear interpolations respectively.

J. Appendix A: Bicubic Interpolation

The bicubic interpolation is used where more accuracy is desired than the bilinear interpolation. [28] The bicubic interpolation used here is a two-step one-dimensional lagrange polynomial (cubic) interpolation.

The inputs to the bicubic interpolation consists of sixteen values arranged on a 3 cell x 3 cell grid as shown in figure 10. Using the sixteen values four cubic interpolations can be carried out using the four points available at each value of ii (in the North-South direction). Then another cubic interpolation can be performed on the four resultant values in the East-West direction (all of which reside on a straight line which is dely units away from the line y = jj + 3) to obtain the desired value.

Let $P(y) = a_0 + a_1 y + a_2 y^2 + a_3 y^3$ be a polynomial of third degree. The standard equation for the lagrange polynomial interpolation is given by,

$$P(y) = \sum_{i=0}^{3} f_i L_i(y) \tag{17}$$

where,

- y is the point at which interpolation takes place.
- P(y) represents the desired value.
- f_i are the known values on the grid at points y_i.
- $L_i(y)$ are the Lagrange polynomials such that $L_i(y) = \Pi^3_{k \neq i, k=0}(y - y_k)/(y_i - y_k)$

As shown in figure 10, considering the North-South direction. Let the position of point of interpoation be considered relative to the lower right corner of the *grid square* in which it is located. It is Δ_y units away from the lower right corner in North-South direction.

In our usage [figure 10], let

- y_i represent jj+2, jj+3, jj+1, jj+4, $0 \leq i \leq 3$
- f_i corresponds to the value of w_k at y_i, $i = (k-1) mod 4$, $1 \leq k \leq 16$

As the grid points are evenly spaced, the computation of the Lagrange polynomials can be simplified as below,

$$L_0 = [(y - y_1)(y - y_2)(y - y_3)]/[(y_0 - y_1)(y_0 - y_2)(y_0 - y_3)]$$

$$L_0 = [(1 - \Delta_y)(1 + \Delta_y)(2 - \Delta_y)]/(-1)(1)(-2) = 0.5 * (2.0 - \Delta_y) * (1.0 - \Delta_y^2) = \Delta_{y2} * (1.0 - \Delta_y^2)$$

where, $\Delta_{y2} = 0.5 * (2.0 - \Delta_y)$

Similarly,

$$L_1 = (\Delta_y^2 + \Delta_y) * \Delta_{y2}$$

$$L_2 = (\Delta_y^2 - \Delta_y) * \Delta_{y2}/3$$

$$L_3 = (\Delta_y^2 - 1.0) * \Delta_y/6$$

A similar reasoning holds for East-West direction. Let the subscript 'y' be used for quantities in the North-South direction and 'x' for East-West. The algorithm for the bicubic interpolation is as follows.

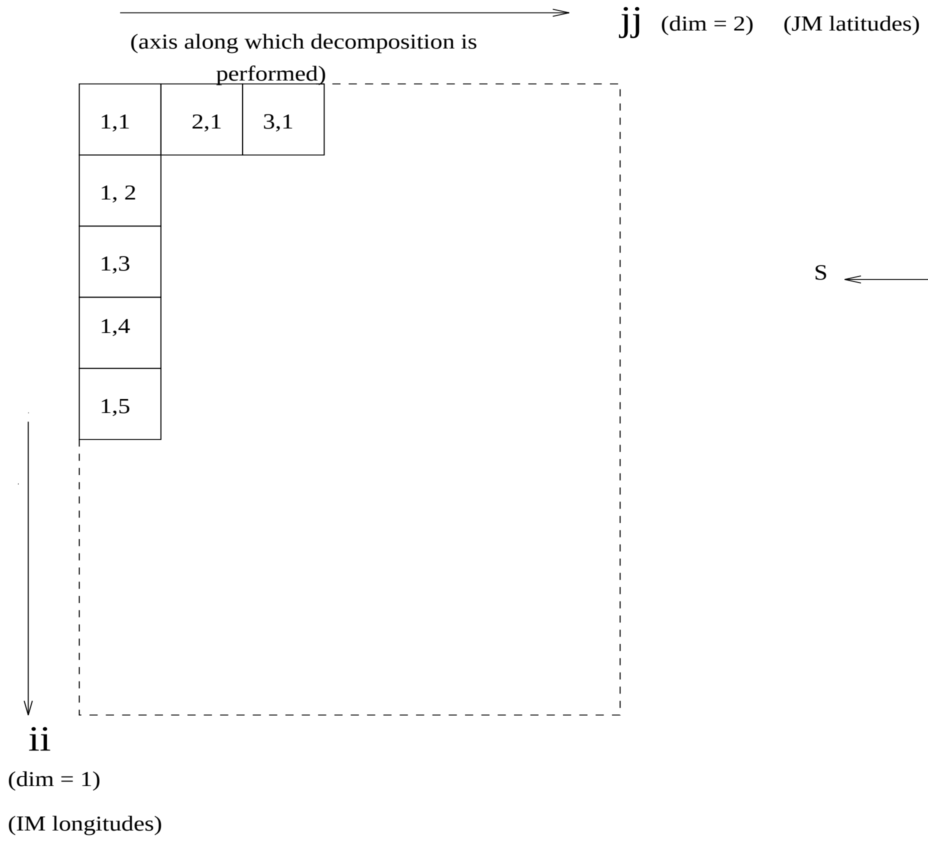
- Inputs

---

[28] The bilinear interpolation is more economical than the bicubic interpolation and is hence used where more accuracy is not required.

  - Sixteen values at the grid points as in figure 10
  - Relative positions Δ_x, Δ_y

- Compute terms Δ_{y2} and Δ_y^2 - 2 multiplications
- Compute terms Δ_{x2} and Δ_x^2 - 2 multiplications
- Compute Lagrange polynomials L_{iy} - 6 multiplications
- Compute Lagrange polynomials L_{ix} - 6 multiplications
- Perform four lagrange interpolations in North-South direction - 4 x 4 multiplications
- Perform the final lagrange interpolation in East-West direction - 4 multiplications. This obtains the desired value in a total of 36 multiplications.
- Return the desired value



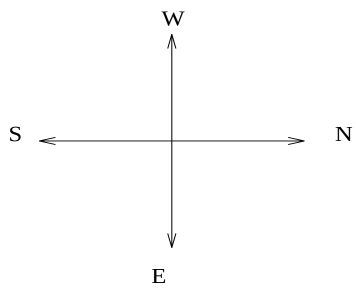


Fig. 1 Original Grid : Dimensions And Orientation. Also shown is the axis along which lasagnia decomposition is performed.

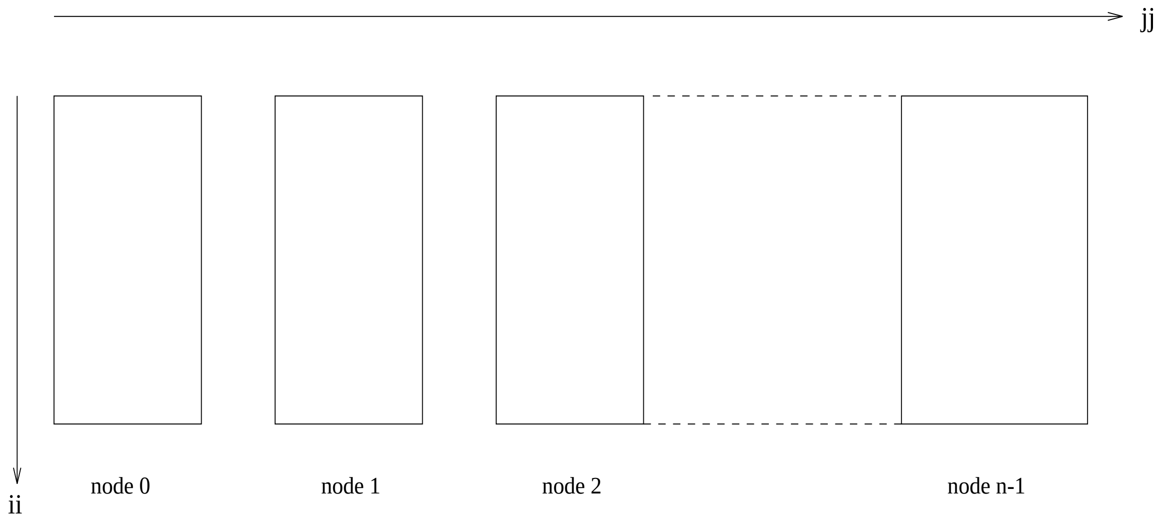
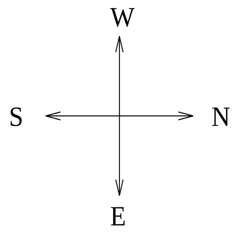


Fig. 2 Decomposed grid. Each node has a partition of the original data. As the original grid had dimensions which were not a power of 2, the last node picks up a larger partition than the rest.



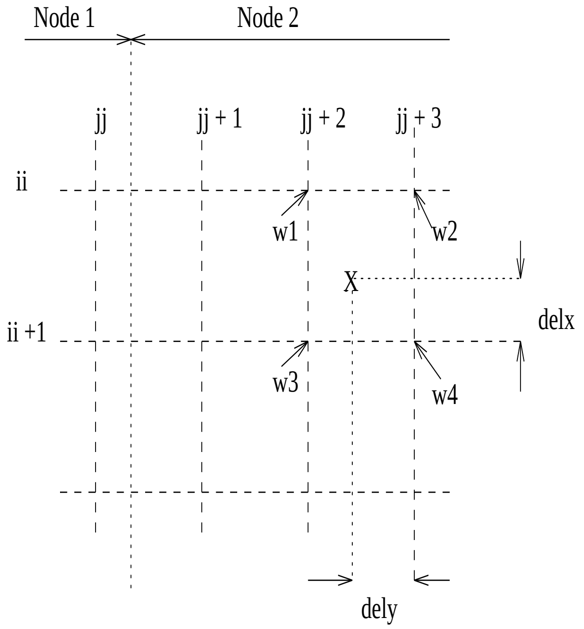


Fig. 3 Bilinear Interpolation. The 'X' marks the point which is found on interpolation. The figure also shows the case where the value at jj is resident on a node (node 1) but values at jj + 1 onwards are off-node.

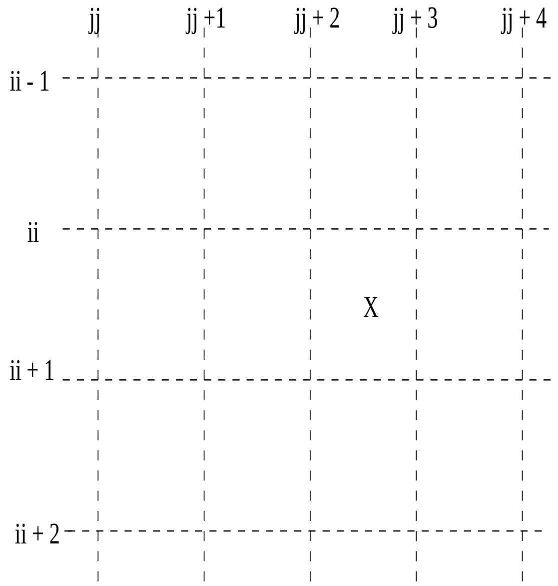


Fig. 4 Bicubic Interpolation. The 'X' marks the point which is found on interpolation.

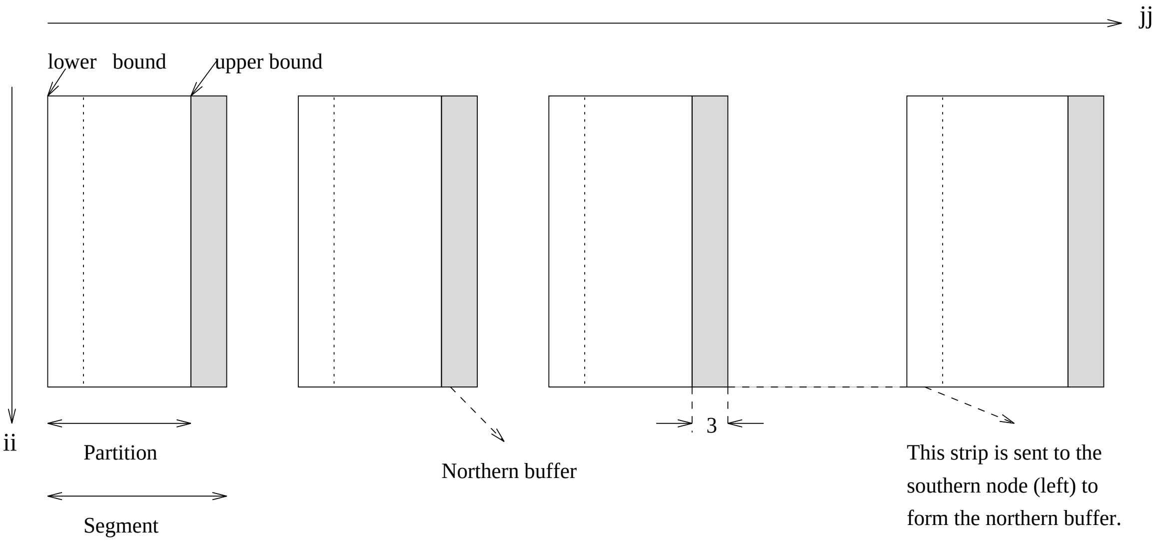
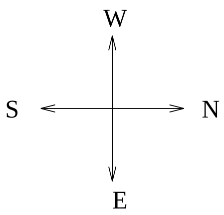
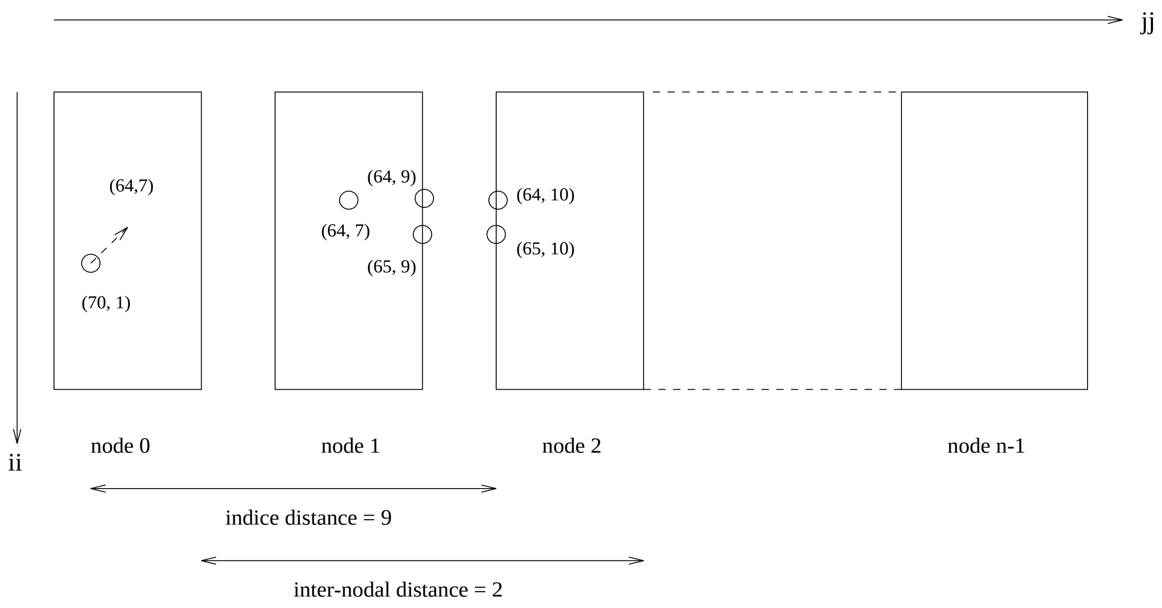


Fig. 6. A fixed width (3 elements wide for bilinear interpolation) is constructed on every node that overlaps with the northern neighbor.





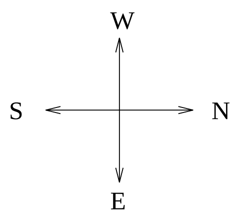


Fig. 5 The figure shows how the inter-nodal distance is computed. In node 0, the global indice jj has the value 64 at location (70,1) and global indice ii has the value 7. To be able to perform the interpolation the farthest indice needed is jj = 10 (jj + 3) in the Northern direction for the bilinear interpolation. However, if jj were such that the points were more than half-way around the globe, the southern most point jj + 2 is furthest. For simplicity we use jj instead of jj + 2 in our calculations. These two are the most extreme points for the bilinear interpolation and are used in the calculation of the maximum inter-nodal distance for each jj.

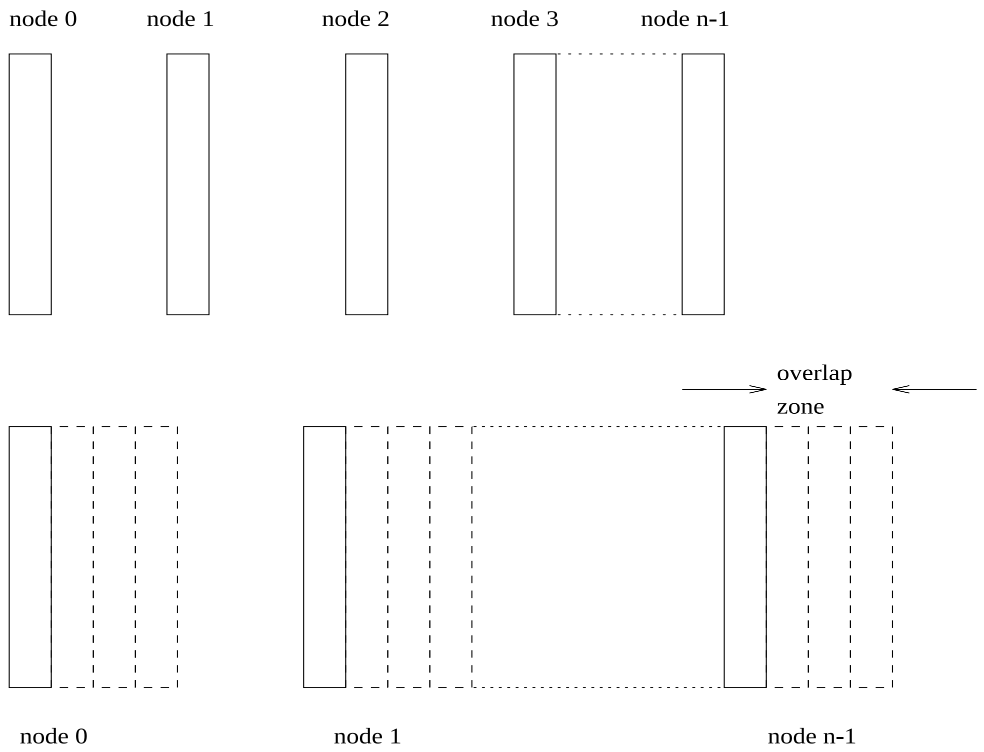


Fig. 7 The top half of figure shows a case where an individual partition is smaller than the overlap zone. In this case two or more communication steps have to be carried out only for establishing the overlap zone as shown in the bottom half of the figure. This maybe avoided as subsequent communication steps fetch data which include the data in the overlap zone. An improved algorithm is explained in the next figure.

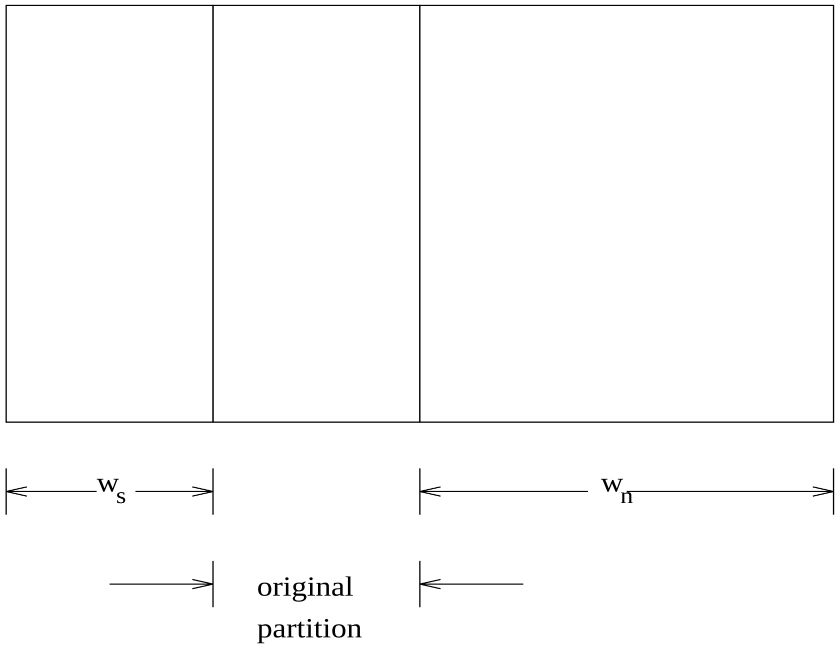


Fig. 8 The figure shows the resulting block in a node after all communication steps. There is no northern buffer created, however. Each node adds on a strip of width w (with the suffix s for southern and n for northern) in the North-South directions, depending on how far along the fluid trajectory one has to traverse to be able to perform an interpolation. An individual strip width is the same as that of a partition, i.e. w (s or n) is a multiple of the partition width.

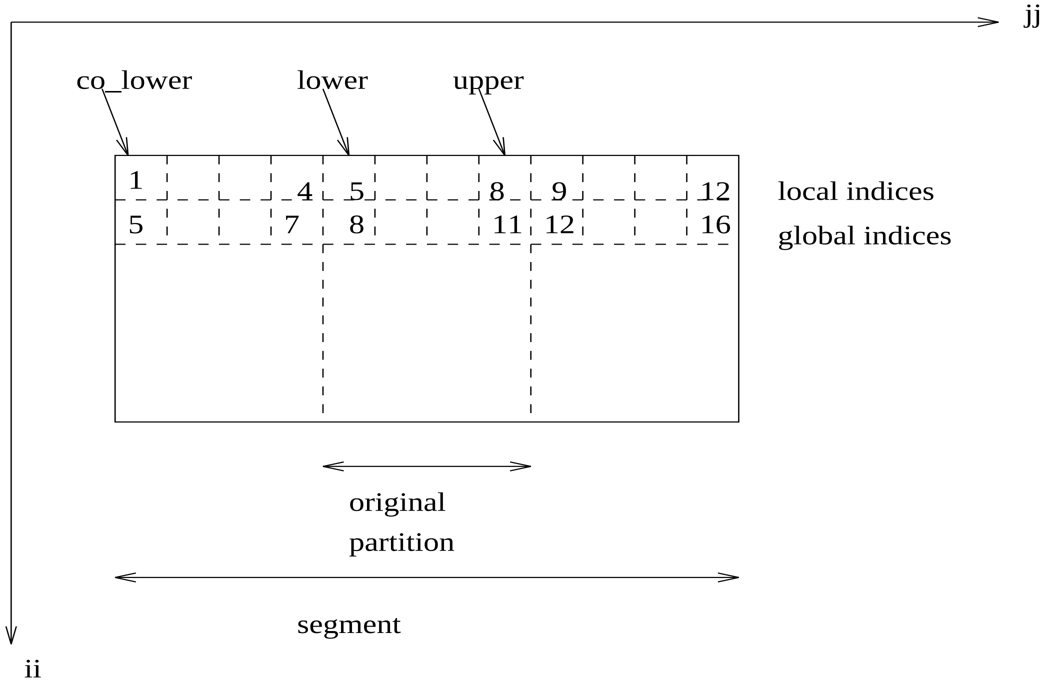


Fig 9 a. This figure shows a node which has no wrap around after the segment has been fully reconstructed in either north or south directions. Global indices translate to local indices in the North - South direction according to the formula

Local index = Global Index - co_lower + 1

$(10, 7) \equiv (10, 7 - 5 + 1) = (10, 3)$

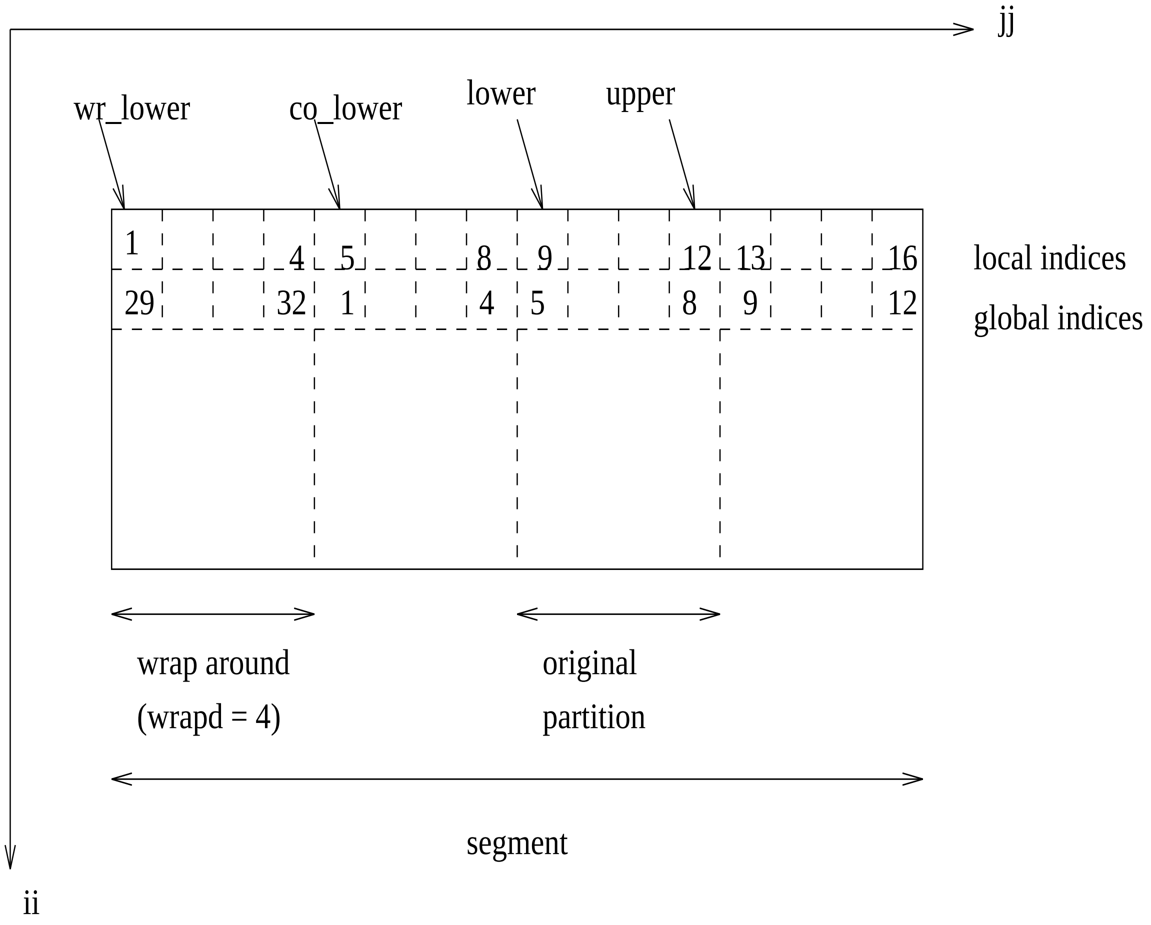


Fig 9 b. This figure shows a node which has a wrap around in the southern direction after the segment was fully reconstructed. Global indices translate to local indices in one of two ways.

Case a) The global index refers to a value in the wrapped part .

Local Index = Global Index - wr_lower + 1

$$(10, 30) \equiv (10, 30 - 29 + 1) = (10, 2)$$

Case b) The global index refers to a value in the non-wrapped part. .

Local Index = Global Index + wrapd

$$(10, 9) \equiv (10, 9 + 4) = (10, 13)$$

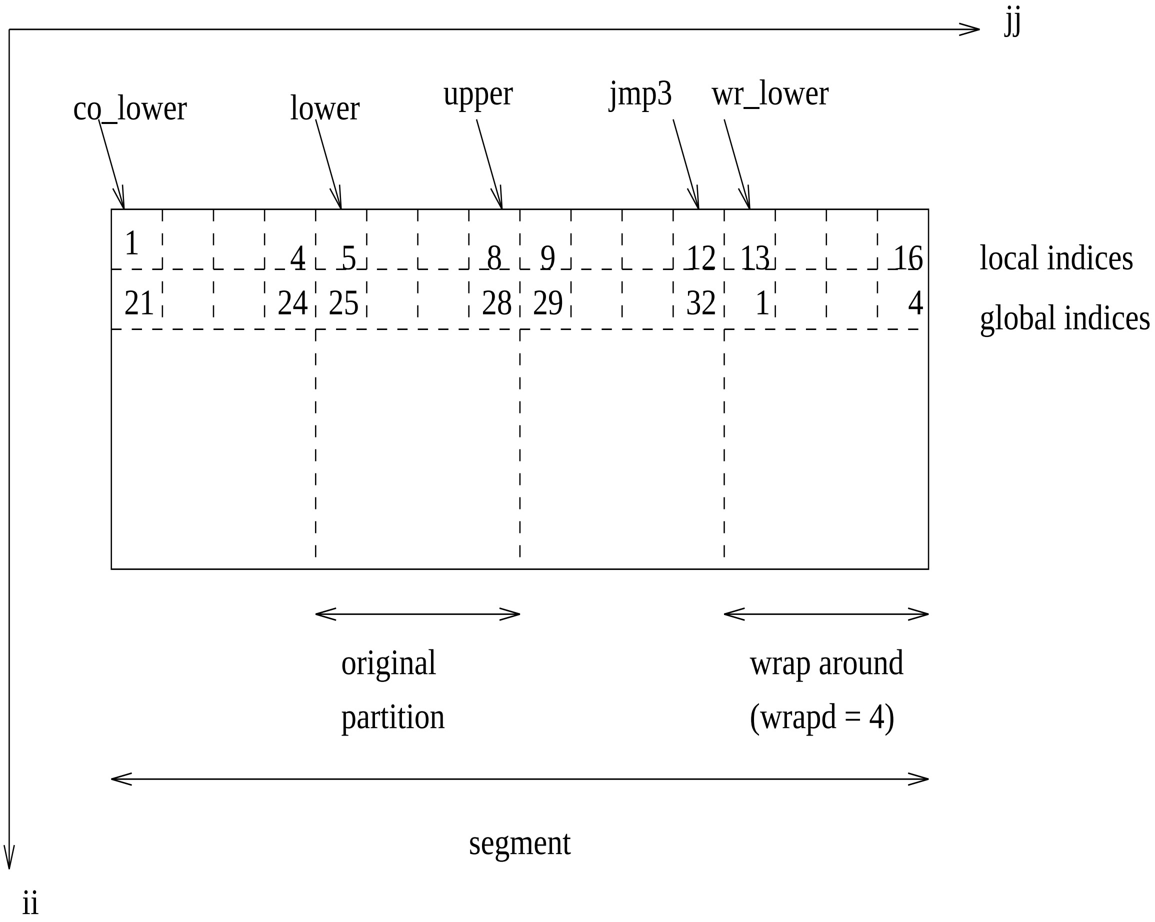


Fig 9 c. This figure shows a node which has a wrap around in the northern direction after the segment was fully reconstructed. Global indices translate to local indices in one of two ways.

Case a) The global index refers to a value in the wrapped part .

Local Index = Global Index + jmp3 - co_lower + 1

$$(10, 3) \equiv (10, 3 + 32 - 21 + 1) = (10, 15)$$

Case b) The global index refers to a value in the non-wrapped part. .

Local Index = Global Index + co_lower + 1

$$(10, 21) \equiv (10, 21 - 21 + 1) = (10, 1)$$

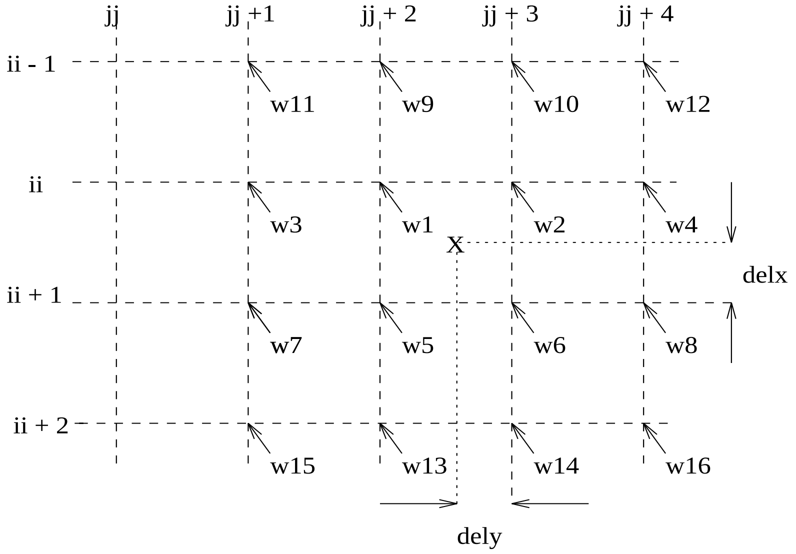
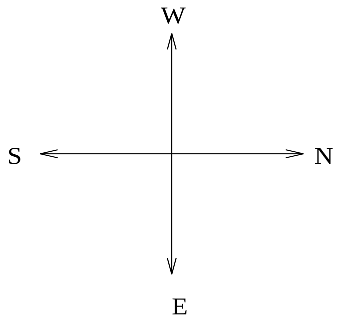


Fig. 10 Bicubic Interpolation. The 'X' marks the point which is found on interpolation.



VI. DATA PARALLEL IMPLEMENTATION OF QUALITY CONTROL FOR DATA ASSIMILATION SYSTEM GEOS DAS 1

Four-Dimensional Data Assimilation is a Grand Challenge problem. It is a process of combining short term predictions from numerical models with experimental observations to produce minimum expected error estimates of the evolving state of Earth's atmosphere. One of the missions of the NASA/Goddard Data Assimilation Office (DAO) is to generate research-quality assimilated data sets for Earth Science, and in particular for the Earth Observing System (EOS).

Data assimilation plays a central role in the Mission to Planet Earth enterprise. The assimilation process transforms a heterogeneous and inhomogeneous observational data set into a time-continuous data set on a regular spatial grid. On one hand, data assimilation propagates model information into observational-data-sparse regions; on the other hand it is a valuable reality check of model predictions.

Development of advanced assimilation systems requires continual and timely reanalysis of multi-year data sets. Future assimilation systems must incorporate chemical, oceanic and land-surface dynamics, in addition to the traditional meteorological domain. Therefore, inherent computational requirements, combined with continuing reprocessing of historical data sets, establishes data assimilation as one of the most compute-intensive tasks among NASA's Earth Science efforts. It is expecetd that by the year 2000 DAO will require 150 GF sustained and 110 TB of storage. By the year 2004, these numbers will grow to 1200 GF and 500 TB respectively.

Extrapolation of current trends in high-performance computing clearly indicates that massively parallel and distributed processing is the only way to satisfy DAO performance demands at an acceptable cost. Therefore, to ensure viability of present and forthcoming DAO software in future high-performance computing environments and a smooth transition to future scalable HPC platforms, it is essential to perform:

- Examination of the potential for parallelization and scalability of operational algorithms at DAO,

- Examination of the potential for parallelization and scalability of candidate future algorithms at DAO, and
- Porting and benchmarking of existing codes on testbed parallel architectures.

These tasks would be daunting even without the added aggravation of a rapidly changing software and hardware environment which characterizes the current state of parallel computing. However, emergence of two standards of parallel computation, which are among the major recent developments in this field, will have profound effects on DAO software development.

High Performance Fortran (HPF), which is the industry standard set of parallalization-enabling extensions of Fortran 90, promises top performance and portability on a wide variety of physically distributed memory platforms. Message Passing Interface (MPI) standard will enable portability of message-passing codes without sacrificing communication hardware performance. In addition to investigating intrinsic parallelism of DAO applications, our goal was to explore possibilities and limitations offered by these two software environments, in light of the DAO operational data assimilation system. One fundamental advantage of HPF is the abstraction of the underlying hardware from the application developer. In this article, we will explore the benefits and pifalls of the HPF implementation of the operational data assimilation system. We will first provide a brief introduction to the underlying algorithms and related HPF language features. We will then describe parallelization strategies in detail and explain how are they related to the HPF semantics. We will consider issues such as performance but will not dwell too much on them, since the HPF compiler technology is not mature enough to provide desired levels of performance. Instead, we are more interested in the features of the language itself, hoping to leverage upon futures improvements in compiler technology. Finally, we will summarize our findings and outline likely future directions for parallelization of the next generation of DAO data assimilation software, Physical-space Statistical Analysis System (PSAS).

A. Description of GEOS-DAS-1 Optimal Interpolation (OI) Analysis System

The current version of the assimilation system performs multivariate statistical interpolation analysis of sea-level pressure and surface wind data over the oceans and geopotential height and wind fields on constant pressure surfaces. In multivariate analysis pressure and temperature data are coupled with wind fields. The system also performs univariate analysis of water vapor mixing ratio on constant pressure surfaces. The spatial grid of GEOS-DAS-1 analysis system is a $2.0^o \times 2.4^o$ latitude/longitude horizontal grid replicated over 14 analysis levels in the vertical direction. The levels correspond to constant pressure surfaces at $(20, 30, 50, 70, 100, 150, 200, 250, 300, 400, 500, 700, 850, 1000)mb$. The analysis is coupled to a 20 pressure level, $2.0^o \times 2.4^o$ horizontal resolution General Circulation Model (GCM) for the troposphere and lower stratosphere.

We will describe the features of the GEOS-DAS-1 system which are relevant from the viewpoint of parallel algorithm implementation only. For a complete description of the production assimilation system at DAO reader is referred to [4]. Also, we cannot dwell on mathematical or statistical content of the analysis equations. For a thorough introduction to the subject of statistical interpolation, reader is referred to [2].

Statistical interpolation attempts to find optimal combination of erroneous experimental observations and erroneous first guess (model prediction) which, by definition, minimizes expected error in the analyzed quantity. Model prediction at the analysis time τ will be denoted by vector $\vec{y}_f$ (f is for first guess). The array of observations in a 6-hour window centered on τ will be denoted by $\vec{x}$. It is important to keep in mind that model arrays live on the regular spatial (latitide-longitude) grid, while observational data have a continuum of locations. The analyzed data arrays are conformal with model arrays in the horizontal directions. However, model variables are defined on σ-coordinates, while the analysis data are defined on "mandatory" pressure level surfaces, as described earlier. Analysis arrays will be denoted by $\vec{y}_a$. Analysis, model and observations are related through the analysis equation:

$$\vec{y}_a = \mathbf{M}(\vec{y}_f) + \mathbf{W}\left[\vec{x} - \mathbf{HM}(\vec{y}_f)\right] \tag{18}$$

$\mathbf{M}$ denotes mapping function between model variables (temperature(T), wind(U, V), specific humidity (μ), surface pressure (P_0) and σ-coordinates to the analysis variables (geopotential height (Z), wind(U, V), water vapor mixing ratio(ρ), sea-level pressure(P_{sl})) and pressure coordinates. $\mathbf{H}$ is an operator which interpolates model variables at the observation locations using their values on surrounding model grid points. Matrix $\mathbf{W}$ contains the analysis weight: each weight determines size of correction to a first guess value at a given grid point from a given observation. Number of rows in $\mathbf{W}$ is equal to the number of model grid points, while the number of columns equals the number of observations. It will be referred to as the gain matrix.

The gain matrix $\mathbf{W}$ is computed by minimizing the expected error variance of the analyzed data. It can be shown [4] that the formal solution is:

$$\mathbf{W} = \mathbf{P}_f \mathbf{H}^T \left[\mathbf{H}\mathbf{P}_f \mathbf{H}^T + \mathbf{R}\right]^{-1} \tag{19}$$

where $\mathbf{P}_f = < \vec{\epsilon}_f \vec{\epsilon}_f^T >$ is the first guess (background) error covariance matrix and $\mathbf{R}_f = < \vec{\epsilon}_o \vec{\epsilon}_o^T >$ is the observation error covariance matrix. It is assumed that errors are unbiased and that background and observation errors are uncorrelated. For a detailed description of the specification of background and observation error covariances, the reader is referred to [4]. The analysis error covariance matrix is obtained as:

$$\mathbf{P}_a = [\mathbf{I} - \mathbf{W}\mathbf{H}]\, \mathbf{P}_f \tag{20}$$

Analysis error variances are computed as diagonal metrix elements of $\mathbf{P}_a$.

Data ingest and preprocessing are required to convert raw data sets into an input data set acceptable for the analysis algorithm described above. While this is a complex multi-stage procedure in the production system, we will be mostly concerned with data quality control, which is computationally most demanding. Since the focus of this article is on the application of High Performance Fortran to the Optimal Interpolation problem, issues related to I/O and system integration are left for forthcoming publications.

GEOS-DAS-1 employs a two-stage quality control technique: a gross error check and a "buddy" check [4]. The gross check compares the difference, δ, between an observation and the first guess value interpolated onto the observation location, and a phenomenologically established threshold:

$$\delta^2 \geq \left(\sigma_o^2 + \sigma_f^2\right) \times T \tag{21}$$

In this equation, σ_o is the observation error variance, σ_f is the first guess error variance and t is a $O(1)$, height-dependant, subjectively established tolerance factor. If this test evaluates to true, the observation is flagged as suspect and passed to the second quality control stage, the "buddy" check. Buddy check performs a successive-corrections analysis of the suspect datum. If the equation 21 still evaluates to true after the analysis, the observation is rejected. Rejection rates are usually between 5 and 10%.

A typical production assimilation run will process of the order of 10^5 observations. It is expected that this number will grow to 10^6 within a decade or so. Therefore, the observation error covariance matrix will grow from $\propto 10^10$ matrix elements at present to 10^12 in the near future. Background error covariance matrix scales as the square of the spatial grid size and has around 2×10^8 elements at a $2^o \times 2.5^o$ resolution. It is clear that the assimilation problem size is determined by the observation data set. It is also clear that the formal solution given by equation 19 is impossible to implement in practice even on massively parallel machines, because of the observation error covariance matrix size. Effective treatment of the observation data drives both the approximation strategies and the parallel implementation.

The approximation strategy employed by GEOS-DAS-1 consists of partitioning of the global analysis problem into a large number of independent localized analysis problems of manageable size. First, the model and analysis grid is divided into clusters of nearby grid points in latitude/longitude direction. The number and layout of the grid points in a cluster depends on latitude and is hardwired for a particular resolution. Each cluster is extended into the third dimension, so that the three-dimensional volume is divided into a number of non-overlapping cylinders. Each cylinder constitutes a **mini-volume**. Second, the innovation vector covariance matrix is replaced by a localized approximation which applies for *all* grid points in a mini-volume. The mini-volume technique scans for observations within a predefined scan radius from the *center* of a mini-volume. Only these observations are used to construct the covariance matrix. Essentially, a physically motivated cut-off is introduced into the error correlation functions. This enables independent solutions for each mini-volume.

The mini-volume technique differs from the conventional analysis method because it does not scan for observations within the scan radius from *each* grid point. Therefore, data selection and solution of covariance matrix are performed once for all grid points in a mini-volume. Scan radius is presently set at $1600km$. Both the scan radius and layout of mini-volumes are obtained experimentally to avoid discontinuities across mini-volume boundaries. Mini-volume definitions are give in the Appendix.

Equation 19 is now applied for each mini-volume using the local approximation to the innovation vector covariance matrix. The computationally expensive task is the inversion of the innovation vector covariance matrix, which is accomplished using the standard Cholesky algorithm.

The above analysis algorithm is implemented in sequential modules for Surface Analysis, Moisture Analysis and Upper Air Height and Wind Analysis. The modules cannot be spawned as independent tasks since the latter ones depend on the outputs of the former ones. At present, we have parallelized the Surface Analysis module. Since it operates on smaller data sets than the other two, it is easier to debug and verify results as well as less demanding on I/O resources. On the other hand, it is a better candidate to explore parallelization issues, since it is likely to generate higher communication overheads than the other two, as shown later on. Since the algorithms are almost identical, our analysis and conclusions are valid for all three modules, unless stated otherwise explicitly.

To summarize, the computational flow of the OI analysis is as follows:

- Quality Control
    1. Gross check
    2. Buddy check
- Construction of mini-volume covariance matrices
    1. Data search within scan radius
    2. Data sort and selection
    3. Connovation vector covariance matrix preparation
- Solution of the analysis equation for each mini-volume
    1. Computation of the gain matrix by applying the Cholesky method to Equation 19
    2. Computation and storage of analysis data

We will now turn to the explanation of parallelization strategies for each computational stage.

B. Overview of Parallelization Strategies

Parallelization strategies for the mini-volume based OI algorithm are developed around three major types of data objects:

1. model variables
2. observational data
3. mini-volume data

Model variables for Moisture and Upper Air Analysis are represented by three-dimensional arrays representing values of physical quantities on a uniformly discretized grid of latitude/longitude/pressure(height) coordinates. Model data for Surface Analysis are represented by two-dimensional arrays, corresponding to pressure, latitude and longitude wind components on the latitude/longitude grid on the lowest pressure level.

Observational data are implemented as a linear list (array) of structures. Structure fields correspond to different data attributes (lattitude, longitude, source, type, quality, value etc.) Mini-volume data constitute vectors and matrices which define the linear system, given by Equation 19, associated with each mini-volume. Mini-volume index can be viewed as an extra dimension in the label space of these arrays. Alternatively, mini-volume data can be viewed as stacks of vectors and matrices, obtained from the model and observational data through a sequence of operations detailed below. Parallelization strategy requires partitioning of data objects at each stage of the computation in the most effective fashion, which depends on the transformations undergone by the data objects.

Quality Control phase is entirely independent of the mini-volume concept. Both gross check and buddy check are characterized by sequences of operations applied on each item in the observation data array. Therefore, this phase can be described as data parallel. Majority of operations can be described using the array syntax of Fortran 90. Some transformations involve maps between the observational data index space and model data index space. The major operation which couples observational data and model data is the interpolation from model grid onto the locations of the observational data. It is this interaction between data and model index spaces which makes Quality Control highly non-trivial for parallelization. In the parlance of data parallel languages, model and data arrays are non-conformal. When the arrays are partitioned among the nodes of a parallel processor, mappings between the two index spaces will be accompanied by non-local memory accesses. Minimizing the number of non-local accesses is essential for an efficient parallel implementation.

Mini-volumes appear in the second stage of the computation, where for each mini-volume one identifies the observational data which belong to the mini-volume (i.e. observations located within the scan radius from mini-volume center) and then selects the best data items among them which will be used for further computation. Using the locations and types of the selected observations one constructs observational error covariance matrices. The two covariance matrices are combined via Equation 19 into the innovation error covariance matrix, which, in turn, defines the linear system for each mini-volume via Equation 19. By the end of this stage the original model and observation arrays disappear as first class objects by being transformed into mini-volume stacks.

Mini-volume linear systems are independent from one another. In the third stage, the independent mini-volume linear systems are solved using the Cholesky algorithm. Model and observational data objects are completely irrelevant during this stage. Once the solution is completed, though, the results from individual mini-volume problems must be concatinated and written into the analysed data array, which is conformal with the model array along latitude and longitude coordinates.

It is clear from this flow analysis that, at different stages, different paradigms of parallel computation may apply. Quality control can be most conveniently viewed as data parallel, while the other two stages are task-parallel-like. One must carefully consider penalties and trade-offs in performance and development costs which arise from single-paradigm or multi-paradigm implementations.

Given the time and manpower constraints, we have initially adopted task parallelism as our main parallelization strategy for the production version of the OI code. This approach is based on the partitioning of mini-volume stacks among the nodes of a parallel processor. Each processor is responsible for the mini-volumes embedded in its assigned geographical region. At the beginning of the calculation *all* observational data relevant for the mini-volumes owned by a processor are shipped to the processor. After that, processors are almost independent of one another, apart from exchanges which may arise from the dynamic load-balancing strategy. Besides conceptual simplicity, the main advantage of this approach is that it minimizes the number of changes which must be made in the original sequential code, particularly in the routines for construction of covariance matrices and the Cholesky solver. It can also be easily load balanced. Search for observation data is done within a scan radius which is much larger than the geographical region assigned to each mini-volume (see Figure 1 in the Appendix). Hence, the major shortcoming of this approach is due to the existence of overlaps between these search regions. The relative amount of overlap increases as the number of mini-volumes per processor decreases, thus limiting scalability.

Most of the redundant computations appear in the quality control stage, which is not surprising, given that quality control really manipulates model and observation arrays and can be naturally expressed as a data parallel algorithm. It is not easy to predict the overall percentage of time spent in Quality Control for future assimulation systems, since many more observation attributes may be added and more sophisticated quality control procedures may be introduced. It is likely that Quality Control will take of the order of a quarter up to a third of the overall CPU time. This was the initial motivation to develop an experimental data parallel implementation along with the task parallel production code. While developing the data parallel version using High Performance Fortran as a platform, it became clear that OI code may serve as an excellent real-life benchmark for both language features and compiler technology. In the next Section we describe the HPF implementation of the Surface Analysis Module in the OI code.

Given the large granularity of distinct computational phases, the optimal approach to parallelization of a major production code like OI would be to implement each phase as a module, using the most appropriate paradigm within each module, and then deal with paradigm integration using either language level support (e.g. EXTRINSIC mechanism in HPF, Fortran M) or, more preferably, higher level integration environments (like AVS or Khoros, for example). This is the approach we adopted for parallelization of the Physical-Space Statistical Analysis System (PSAS), the next generation of the DAO assimilation software.

C. High-Performance Fortan Implementation of the Surface Analysis Module

1. High Performance Fortran Overview

High Performance Fortran (HPF) is derived from the experimental Fortran D langauge [11] with the goal to preserve, at the application level, two basic tenets of sequential programming: single thread of control and global name space. HPF language specification is an evolving standard initially adopted in 1993 as a result of collaboration of academic institutions and hardware and software vendors. Dialects of HPF have already existed on Connection Machine and MasPar computers. A number of fully HPF compliant compilers will be commercially available in 1996. For a detailed exposition of HPF design, philosophy and semantics, the reader is referred to [12].

The basic principle of HPF is that parallelism must be explicitly stated by the programmer. For example, the compiler does not attempt to discover parallelism by a sophisticated dependency analysis of loops, which is an approach taken by other parallel languages. There are basically three ways a programmer expresses parallelism: array syntax, parallel constructs and compiler directives. HPF may be considered a superset of Fortran 90 with a small number of extensions to support effective use of MIMD machines. In both Fortran 90 and HPF arrays are first class objects. Array syntax allows the programer to manipulate complete arrays and to express parallelism transparently: since the array elements are distributed among the nodes of a parallel processor, an array instruction is executed by all nodes simultaneously. Translation between addreses in the global name space and addresses of array elements in local node memories is done by the system. Also, if the array operation requires that array elements be fetched from remote node memories, appropriate run-time library calls are generated by the compiler transparently to the programmer.

Data parallel applications are characterized by sequences of operations which are performed on all members of a data structure. This type of parallelism can be achieved by partitioning the data structure among the nodes of a multiprocessor. Nodes may share a common memory or have physically disjoint memories or both. HPF design goal is to achieve maximum performance for any kind of memory organization. Non-uniform memory access on physically distributed memory machines implies that data locality is crucial for good performance. HPF has a number of features designed to exploit data locality to achieve optimum performance across different hardware parameters. Among the most important HPF extensions to Fortran 90 are compiler directives for data distribution and alignment among the nodes, which constitute a programmer's high-level toolbox for management of data layout. This should be contrasted with intricacies of micro-management of data layout in message-passing environments.

Mapping of data arrays onto the processor array is typically done in two stages: first, a group of data objects are aligned among themselves, using *ALIGN* or *REALIGN* directives; then, the aligned objects are mapped onto an *abstract* processor array (as defined by the *PROCESSORS* directive) using the *DISTRIBUTE* or *REDISTRIBUTE* directives. These steps are under programmer's control. In the final stage, the abstract processor array is mapped in a system dependant fashion onto the physical hardware, transparently to the application programmer. The fundamental advantage of this process is that code design is centered around the topology of the physical problem, while the mapping from the problem topology to the communication hardware topology is done by the system in a (presumably) optimized fashion.

Possible distribution patterns are: **BLOCK**, **CYCLIC** and *****. **BLOCK** distributes an array dimension in contigous blocks, **CYCLIC** distributes in a round-robin fashion, while ***** specifies that the array dimension should not be distributed. **BLOCK** is analogous to **:NEWS** and ***** corresponds to **:SERIAL** in CM Fortran.

2. Data Layout

Parallelization of the **Quality Control** submodule within the Surface Analysis Module is based on one-dimensional *BLOCK* decomposition of both model arrays and data arrays. The important model arrays are pressure (P), latitudinal (U) and longitudinal (V) wind components. These are 2D arrays dimensioned as $IM \times (JM + 1)$, where IM is the number of latitude strips and JM is the number of longitude strips in the discretized latitude/longitude grid.

In the OI algorithm, references to model arrays usually involve clusters of nearby points. For example, model data have to be interpolated onto the observation locations, which presently requires four points on a grid rectangle surrounding the observation location. By partitioning model arrays into fixed width strips using template

```
!HPF$ PROCESSORS pgrid(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE (BLOCK,*) ONTO pgrid :: psl
!HPF$ ALIGN WITH psl :: uwind,vwind
```

we ensure high degree of locality if the number of grid points per processor is not too small. For a $2.5\deg X2\deg$ resolution, $IMX(JM + 1) = 144X91 = 10000$, which means that decomposition granularity is sufficiently large even for massively parallel machines. Granularity improves for finer resolutions and for other Analysis modules, such as Mixing Ratio and Height-Wind Analysis, which use three-dimensional modell arrays. The array extent in the third dimension is equal to the number of pressure layers, which is of the order of few tens. The two-dimensional decomposition is therefore efficient for most of the present and future parallel machines. Higher-order interpolation schemes would also increase efficiency by increasing complexity of local computation.

In the task-parallel implementation, observational data are modeled as one-dimensional array of structures. Structure fields correspond to the attributes of an observation at a given location. This implementation is elegant and flexible. Flexibility is quite important, given that new data attributes will be added to the future generation DAS, which can be easily accomplished by adding new records in the data structure. Since HPF precursors such as CM Fortran and MasPar Fortran do not support user-defined data types, in our present HPF implementation observational data are modeled as a standard 2D array. This is a short term solution which allows us maximal portability on present platforms. The first array dimension lists the observational data and it is distributed in a BLOCK fashion across available processors. The second array dimension is serialized and it is used to store observation attributes. In the present version of the assimilation system, there are 10 attributes:latitude, longitude, type...

The storage pattern of the parallel code is different from the storage pattern of the original Cray CFT77 code: in latter one, observational data are stored as a one dimensional array, where all observation latitudes are stored first, followed by all longitudes and similarly for other attributes. At various stages of computation, the Cray program constructs indices into this array, which are used by **gather** procedures to fetch necessary data. This strategy allows efficient use of memory banks and vector hardware, but it is unacceptable for a distributed memory where data locality is of paramount importance. Frequently, several attributes of an observation are needed in a single code segment.

For example, both latitude and logitude are needed to compute the distance of an observation from the center of a mini-volume. The original storage pattern virtually guarantees that the required attributes would be scattered across multiple processors. On the other hand, the present observational data decomposition, which is achieved using the following template:

```
!HPF TEMPLATE(BLOCK,SERIAL)
```

guarantees locality of memory references to different attributes of the same observation. The only possibility to reuse old program arrays would be to enforce a CYCLIC distribution pattern on the original 1D array, but it is easy to see that it does not work for arbitrary numbers of processors and/or variable size data arrays. Data arrays must be dynamically allocated for effective partitioning of variable sized data sets.

Quality Control can be roughly divided into Gross Check and Buddy Check phase. During the Gross Check, each observation is tested against model prediction at the location of the observation. Observation is flagged as suspect if the difference is larger that the expected observation error multiplied by an empirically adjusted factor of $O(1)$. The important operation during the Gross Check is interpolation of model pressure and winds onto the observation locations. It involves non-conforming data and model arrays. For each observation, one has to obtain indices of model data for the 4 grid points on the grid rectangle which surrounds the observation. With the $(BLOCK, SERIAL)$ decomposition, computation of indices into the model arrays does not incur any communication costs. Once the index arrays are computed, FORALL construct must be used to gather model data into 4 arrays which are conformal with the observational data array. The gather phase involves indirection with substantial communication costs. For example, on machines like CM-5, gather involves router communications which are much costlier than structured grid communications (NEWS). Communication costs are amortized somewhat by extensive checking of the model grid data, which increases number of operations on conformal arrays 16 times in comparison with straightforward interpolation. As a result, the interpolation routine is quite efficient even on a high latency machine like CM-5.

After the interpolations, Gross Check can proceed without any communication overheads, thanks to conformity of arrays. After completion of Gross Check, suspect observations are flagged and one proceeds with the Buddy Check. For each suspect observation, one searches a region of 1000km in radius in the vicinity of the suspect point and identifies observations which passed the Gross Check. Once identified, these observations are used in a successive correction scheme to improve upon the suspect observation. If this does not help, the observation is discarded for further analysis. The search is implemented as a sequential loop over suspect observations. Within this loop, the inner loop over all observational data is parallelized by distributing observational data array. The major communications overhead comes from broadcasting the information about the suspect point. Another major communication overhead comes from a global reduction (SUM) on the array which contains weights assigned to each observation data in the successive corrections scheme. These two overheads reduce the peak FLOP rate from 3.7 GFLOPS on a 256-node CM-5 to 0.7 GFLOPS sustained. Peak rate is achieved in those parts of the loop body which manipulate observation data array and arrays conformal with it, i.e. those without communication costs.

This version of the parallel algorithm follows closely the original sequential algorithm. Additional speed-up can be obtained by redesigning the sequential code. The observational data can be sorted along latitudes and longitudes before the quality control routines are called. The costs of the parallel sort scale as N/P logN, where N is the number of items to be sorted and P is the number of processors. After the sort, complexity of buddy check scales as const X N, in comparison with the original algorithm which has a complexity of N^2. It is clear that the strategy with sorting wins on large enough observational data sets. The sort was implemented in the task parallel version, but not yet in the data parallel version.

After the completion of Quality Control, data parallel algorithm and the original algorithm begin to diverge significantly. In the sequential code, an analysis driver routine first establishes a loop over mini-volumes. Within the body of the loop, the following tasks are executed:

1. **Search** for observations within 1600km of the center of the mini-volume

2. **Selection** of observational data which belong to the mini-volume: data are first sorted according to their proximity to the mini-volume center, their type and their quality. Afterwords, 150 or less data with highest score determined by a set of semi-empiric criteria are selected. The outcome of the selection process is an index list into the data array.

3. **Construction** of correlation matrices: Index lists computed in the previous step are used to gather the chosen observation data into workspace arrays. These workspace arrays are then used to compute correlation matrices for the mini-volume.

4. **Solution** of the linear system: the major step is Cholesky inversion of the correlation matrix. Thge result is a set of weights for all grid points in a mini-volume.

5. **Store** of the computed weights for the mini-volume grid points.

Due to the sequential nature of processing of the mini-volumes, memory requirements are not excessive, since only the final results, i.e. the computed weights have to be stored. Workspace arrays are reused and overwritten with the information from the next mini-volume.

Parallelization strategy cannot be centered on parallelizing the inner body of this loop, i.e. parallelization of the Analysis algorithm for a **single** mini-volume. There are at least three major reasons why this strategy is unacceptable:

- Indirection into the observational data array would incur high communication costs since data needed for one mini-volume analysis are typically spread across many processors.
- A single mini-volume problem is typically too small. Asa result, parallel implementation of the Cholesky solver would be inefficient and load-imbalanced since mini-volume matrix sizes may range from just a few to a few hundred.
- Usage of distributed static temporary arrays in the Construction phase would lead to load-imbalance problems since number of mini-volume data fluctuates . On the other hand, dynamic allocation of workspace arrays for each mini-volume would incur additional system overhead.

Therefore, our parallelization strategy is to distribute the outermost loop over mini-volumes. As a first step, we separated Search, Selection and Construction phases from the Solution stage and redesigned them to handle arrays of mini-volume matrices, instead of one matrix at a time.

The main complication of this approach is that intermediate results now have to be stored, which is typical of data parallel algorithms. All relevant variabl es in the Analysis stages, including the workspace arrays now acquire another dimension, corresponding to the mini-volume index. Life is further complicated by the fact that it is not known what are the sizes of the arrays, since the size of each mini-volume data set is variable and known only at run-time. Given that large amounts of memory are required in the data pa rallel startegy, this has important consequences for the design of the parallel algorithm, particularly on machines where virtual memory is not available. This is the cas e on the CM-5, which was the original platform for the development of the parallel code.

In the Solution stage the original sequential code runs in an embarrasingly parallel fashion, due to mini-volume independence. The transition from data parallel to task parallel is effected by the EXTRINSIC mechanism of HPF or the somewhat similar global/local programming model on the CM-5.

In addition to being able to accomodate the program arrays in memory, in a data parallel implementation one has to accomodate the temporaries created by the runtime system. Because of that, we divide the whole mini-volume set into an adjustable number of groups. The parallel program driver processes one group at a time in a sequential loop. Each group corresponds to a contiguous subset of the mini-volume indices. Using the triplet notation of Fortran 90, the array sections with relevant mini-volume data are spread across the whole parallel machine.

Before the Search phase, a number of temporary arrays of adjustable size are allocated. These arrays are going to be used to store data items identified in the Search and Selection phases. They are essential for successful memory management of the Analysis phase.

In the Search phase, we have a sequential loop over mini-volume indices in a group. Parallelization comes from distributing the observation data array. Each processor loops independently through the observations it owns to find out which observations are within 1600km of the mini-volume's center. These observations are flagged and their attributes (latitudes, longitudes, type, quality, etc.) are gathered using FORALL construct into previously allocated temporary arrays. The temporary arrays essentially act like stacks. The data from the next mini-volume are concatinated to the data from previous mini-volumes, i.e. they are pushed onto the stack. If the stack size is exceeded, the search records the mini-volume index that caused stack overflow and the rest of the Analysis is done on the mini-volumes which fit into the stack size. Once the Analysis is finished, all stacks are cleared and the Search begins anew with the mini-volume that caused stack overflow. This procedure guarantees that there will always be sufficient memory regardless of the number of observations in a mini-volume. In case of an upward fluctuation in the number of observations in a mini-volume group, the worst that will happen is that fewer mini-volumes will be processed at a time.

The stacks are passed to the Selection routines. Along with stack arrays, the Selection routines receive a mini-volume map array which identifies the mini-volume indices associated with data items listed in the stack arrays. The map array is conformal with stack arrays and is also dynamically allocated. For example, the stack elements which belong to mini-volume number M will have number M written in the conformal segment of the map array. There are multiple stack arrays which list latitudes, longitudes, data types etc. First we use segmented bitonic sort to rearrange data lists for each mini-volume in increasing distance order from the mini-volume center. The sequential code uses heap sort, but it was easier to parallelize the bitonic sort. The performance of any segmented sort depends on the processor number and distribution of the mini-volume data set sizes in a complicated fashion. In the worst hypothetical case, it can happen that a single large data set from one mini-volume is placed on a single processor, while all other processors have evenly distributed much smaller mini-volume data sets. Then, although the number of data is equal on all processors, due to the peculiar segment size distribution, the sort is severely imbalanced. Since we experimented with a single data set only, it was difficult to see how much of the problem this would be in a typical analysis run. Since there is an imbalance in geographic observation data distribution, one would expect to have significant load imbalances in typical sorting procedure. We plan to experiment with a statistical cure, where mini-volume indices are rehashed in a random fashion.

The sorted data are searched to pick highest quality data of given types. These searches are done on the stack arrays with use of the mini-volume map array. The central operation here is counting and gathering of selected data into their permanent. Counting is achieved by first flagging the chosen data by 1 against the discarded data which are flagged by 0, and then using the FORALL construct with the SUM intrinsic in the body of FORALL. The procedure can be thought of as the segmented scan operation.

Again, FORALL is used to gather the selected data into their permanent storage. Since up to 150 data items are kept for each mini-volume, once the selection of data is completed, memory requirements are more predictable. Furthermore, many more data can be identified within a mini-volume during the Search routine then the maximum number of selected data. Therefore, the permanent storage is implemented as dynamically allocated 2D arrays with mini-volume index in one dimension and the list of data for a given mini-volume in the other dimension (up to 150 of them). Dynamic allocation is essential again because it is known only at run-time which mini-volumes in a group will fit into the stack size.

Parallelization of the Construction stage is achieved by distributing the mini-volume index dimension across available processors. Again, the simple decomposition over mini-volumes may lead to significant load imbalances. We plan to apply first the rehashing procedure to the mini-volumes as a statistical cure for load imbalance. Another approach would be to go to the EXTRINSIC procedure immediately and treat both Construction and Solution phase as task parallel. Then, simple load balancing procedures using explicit message passing can be designed, as described in the Section on task parallel implementation.

The goal of the data parallel implementation was not do design an optimized and fully integrated Analysis system. We wanted to experiment with an emerging data parallel language, High-Performance Fortran and explore benefits and limitations of data parallel syntax for a complex code with non-conformant data structures, non-uniform memory requirements and even computational paradigm. It was also very important to asses the benfits of code development in HPF, given that Fortran 90 is becoming the standard development platform at DAO for future assimilation systems. The results are quite encouraging, although the present generation of compilers are still not up to the demands of this code. The code could be compiled within a reasonable amount of time and run with acceptable performance only under CM Fortran. This is the most mature HPF-like compiler, but it is also much more restrictive than ful HPF. Extensive benchmarking is therefore left for future work. What is quite important is that majority of computations can be naturally mapped onto the array syntax anad array constructs of Fortran 90. FORALL construct, global reductions and scans are also extremely useful. Of course, parallel programming would not be possible without dynamic memory allocation. We believe that a set of utilities for segmented sorting, searching and counting would be beneficial not only for this code, but for a large number of applications. Finally, our load balancing problems would be realitevly easy to solve if we had a more flexible partitioning scheme as a part of the language. This is hopefully something that HPF II will implement. DO INDEPENDENT would simplify a number of tasks that are now implemented using FORALL or WHERE if possible, Unfortanately, it is not properly supported by present compilers. With flexible partitioning and DO INDEPENDENT capable of handling subroutine calls, we could implement the complete load-balanced Solution phase without resorting to the EXTRINSIC mechanism.

VII. REFERENCES

[1] Asrar, G. nad J. Dozier, 1994: *EOS: Science Strategy for the Earth Observing System*, AIP Press, New York, 119pp.

[2] Daley, R., 1991: *Atmospheric Data Analysis*, Cambridge University Press, New York, 457pp.

[3] Allen, D.J., Douglass, A.R., Rood, R.B., and Guthrie, D.P., 1991: Application of a monotonic upwind-biased transport scheme to three-dimensional constituent transport calculations, *Mon. Wea. Rev.*, **119**, 2456-2464.

[4] Pfaendtner, J., S. Bloom, D. Lamich, M. Seablom, M. Sienkiewicz, J. Stobie and A. da Silva, 1995: Documentation of the Goddard Earth Observing System (GEOS) Data Assimilation System Version 1, NASA Tech Memo. 104606, **4**, 44pp.
[5] Thinking Machines Co., 1993: CM Fortran Reference Guide **Ch. 12**
[6] Thinking Machines Co., 1993: CMMD User's Guide and Reference Manual
[7] McDonald, A.; Bates, J.R., 1989: Semi-lagrangian Integration of a Gridpoint Shallow Water Model on the Sphere. *J. American Meteor. Soc.*, **117**, 130-137
[8] Robert, A., 1981: A stable numerical integration scheme for the primitive meteorological equations. *Atmos. Ocean*, **19**, 35-46
[9] Robert, A., 1982: A semi-Lagrangian and semi-implicit numerical integration scheme for the primitive meteorological equations. *J. Meteor. Soc. Japan*, **60**, 319-324
[10] Staniforth, A.; Cot'e, J., 1991: Semi-Lagrangian Integration Schemes for Atmospheric Models - A Review. *Monthly Weather Review*, **119**, 2206-2203
[11] Bozkus, Z., Choudhary, A., Fox, G., Haupt, T. and Ranka, S., 1993: Fortran 90D/HPF compiler for distributed memory MIMD computers: design, implementation and performance results, *Proceedings of Supercomputing 93*, Portland, OR, p.351.
[12] NPAC Repository of HPF Applications, URL *http://www.npac.syr.edu/hpfa/*.
[13] High Performance Fortran Forum (HPFF), 1993: High Performance Fortran Language Specification, *Scientific Programming* **2**, No.1.