DAO Office Note 97-05

Office Note Series on Global Modeling and Data Assimilation

Richard B. Rood, Head
Data Assimilation Office
Goddard Space Flight Center
Greenbelt, Maryland

Design of the Goddard Earth Observing System (GEOS) Parallel Physical-space Statistical Analysis System (PSAS)

P. M. Lyster*, J. W. Larson*, C. H. Q. Ding‡,
J. Guo†, W. Sawyer*, A. da Silva, I. Štajner**
Data Assimilation Office
NASA/Goddard Space Flight Center, Greenbelt, Maryland

Additional Affiliations:
* Joint Center for Earth System Science (JCESS), University of Maryland
† General Sciences Corporation (a subsidiary of
Science Applications International Corporation).
**Universities Space Research Association.
‡ National Energy Research Scientific Computing Center.

Goddard Space Flight Center
Greenbelt, Maryland 20771
February 1997

Abstract

The Physical-space Statistical Analysis System (PSAS) comprises the core analysis scheme for the Goddard Earth Observing System (GEOS) atmospheric data assimilation system. The Data Assimilation Office (DAO) of the National Aeronautics and Space Administration (NASA) developed both GEOS and PSAS, and are currently in the process of constructing a parallel implementation of GEOS. Here we discuss the progress in the parallel implementation of PSAS, which has culminated in the construction of *the prototype parallel PSAS*. This document derives from two week-long workshops that were conducted at the NASA/Goddard Space Flight Center Data Assimilation Office (DAO) on September 30 to October 4, and October 28 to November 1, 1996. The purposed of these workshops was to review the requirements for the developmental parallel Physical-space Statistical Analysis System (PSAS) and to lay the groundwork for the prototyping and design of the operational PSAS that will be a part of the Goddard Earth Observing System data assimilation system GEOS 3.0. This will use the *Message Passing Interface* (MPI) library, with some heritage C and Fortran 77 code, and with an overarching Fortran 90 (f90) modular design.

Contents

1 Introduction

The Physical-space Statistical Analysis (PSAS) algorithm is a significant part of the GEOS atmospheric data assimilation system that is used by the Data Assimilation Office (DAO). Apart from the considerable technology surrounding the data I/O, storage, transmission to and from data facilities, and data visualization, the core components of GEOS are an atmospheric general circulation model (GCM), a coupler that interpolates the GCM output onto the observation grid and calculates the difference between the forecast and observation (known as *innovations*), a data quality control (QC) system that checks these innovations for suspect values, and the analysis scheme PSAS (Figure 1). These are compute-intensive algorithms that, because of the nature of the underlying physical models, are highly coupled.

The DAO is preparing to move its data assimilation system to advanced computing platforms. This will be part of its regular operation, although an important role is expected for the Mission to Planet Earth (MTPE) system in the coming years. Key components of the gridpoint-based GCM (Takacs *et al.* 1996) have been parallelized, and are expected to be incorporated into the system in the coming year. In 1994, the core components of the developmental serial version of PSAS were taken over by computer scientists at the Jet Propulsion Laboratory (JPL) as part of the High Performance Computing and Communications (HPCC) project. The algorithm (PSAS JPL) was parallelized using distributed-memory Single Program Multiple Data (SPMD) message-passing approach (Ding and Ferraro 1995).

The workshops that are summarized in this document were intended to first summarize the requirements (Stobie 1996) and to initiate the prototyping and design for the parallel system GEOS 3.0. It should be noted that the current scientific (serial) versions PSAS 2.0 and parallel PSAS JPL have diverged considerably since 1994. This was the result of a conscious decision that was made at the time because it was known that the serial code was undergoing considerable scientific development and change. It is our intention to merge the parallel technology with the newest version PSAS 2.1 and generate the first unified parallel algorithm that we designate PSAS 3.0. Thereafter the development of parallel code will not diverge from the scientific production code. In order to plan the message-passing methodology, the approach we took was to review the JPL parallel code (the first week September 30 to October 4) and then study the data life cycle in the serial PSAS 2.1 (October 28 to November 1). PSAS JPL is written in C (approximately 15,000 lines of code) that uses the *Message Passing Interface* (MPI) library, and calls low-level Fortran 77 subroutines

(7,500 lines). We are most interested in how this technology will transfer to the environment that is planned for PSAS 2.1, which uses f90 extensively and MPI. Key issues are the efficiency of on-processor code and message-passing functions, and the load balancing of the new algorithm. To do this, both the software constructs (f90 types) and the physical layout of memory need to be considered; the latter is especially important for RISC-based processors such as the ones the DAO is likely to be using. The difficult task of Configuration Management which will specify the process of merging PSAS JPL with PSAS 2.1 to generate PSAS 3.0 will be left to the GEOS 3.0 Design Team.

2 The Scientific Algorithm

Four Dimensional Data Assimilation (4DDA) is the process whereby a state forecast and observations are combined to form a best estimate, or analysis, of the state (Daley 1992). A forecast is derived from a model (e.g., GCM) of the system. The data for an analysis are up to 150,000 observations over a six hour period. 4DDA may be used to provide the initial conditions for a weather forecast. 4DDA is also used at the DAO and other institutions to perform reanalyses of past datasets in order to obtain a continuous, gridded, best estimate of the atmosphere for key state variables (e.g., height, wind, surface pressure, and moisture). The DAO also provides support for measurement instrument operation. DAO will provide software for an operational reanalysis 4DDA system by the year 1998 under the Mission to Planet Earth (MTPE).

As mentioned in the Introduction, the core compute-intensive components are the model GCM, the quality control QC, and PSAS. The complexity of the GCM is $\mathcal{O}(n)$, where n is the number of gridpoints on the *analysis grid* multiplied by the number of state variables. The analysis takes p observations that are inhomogeneously placed in space and time and through a statistical interpolation modifies a forecast $w^f \in I\!R^n$ to form an analysis $w^a \in I\!R^n$. The approach of PSAS (da Silva *et al.* 1995) solves a large matrix problem using the following formulation: The innovation equation

$$(HP^fH^T + R)x = w^o - Hw^f; \tag{1}$$

and the analyzed state is given by the analysis equation

$$w^a = w^f + P^fH^Tx. \tag{2}$$

where $w^o (\in I\!R^p)$ is the vector of observations; $P^f (I\!R^n \rightarrow I\!R^n)$ is the specified forecast error covariance matrix; $R (I\!R^p \rightarrow I\!R^p)$ is the specified observation error covariance

matrix; $H(\mathbb{R}^n \rightarrow \mathbb{R}^p)$ represents a generalized interpolation from the analysis grid to the observations; and $x(\in \mathbb{R}^p)$ is a vector of weights.

Multiple observations from an instrument at the same horizontal position are called *profiles*, i.e., it is assumed for the current development that members of the same profile (e.g., radiosonde or satellite measured radiances) are at the same position of latitude and longitude. It is convenient to write

$$H = F\mathcal{I}, \tag{3}$$

where $\mathcal{I}(\mathbb{R}^n \rightarrow \mathbb{R}^s)$ interpolates from the analysis grid to the *state grid* which is a grid whose horizontal locations are the locations of profiles and whose vertical locations are standard levels for a discretized forward operator. (i.e., a quasi-unstructured grid). The *observation operator* $F(\mathbb{R}^s \rightarrow \mathbb{R}^p)$ models w^o on the *observation grid* (an unstructured grid) from the interpolated variables on the state grid. (i.e., F acts on individual profiles on the state grid to produce the corresponding profiles in the observation grid). For the current formulation of PSAS, the first term HP^fH^T in the innovation equation is evaluated as $FP_s^fF^T$ where P_s^f is the approximation of $\mathcal{I}P^f\mathcal{I}^T$, evaluated directly on the state grid using specified forecast error variances and a correlation model. The correlation model is implemented using lookup tables whose coordinates are the appropriate horizontal and vertical coordinates. For PSAS JPL there is no F operator since only state variables are assimilated. One of the key changes in PSAS 2.1 is the assimilation of non-state variables. For example (Lamich and da Silva 1996) layer thickness, total precipitable water, and cloud-cleared radiances will be directly assimilated. In these cases F is the tangent linear forward model and is often obtained from instrument teams.

The innovation equation is solved using a preconditioned conjugate gradient algorithm (Golub and van Loan 1989, da Silva and Guo 1996). This is an $\mathcal{O}(N_i p^2)$ operation, where N_i is the number of iterations of the CG solver. Typically, between eight and twelve iterations are needed to produce a CG solution whose residual has been reduced by at least two orders of magnitude. Experiments in which the residuals are reduced by more than two orders of magnitude resulted in errors in x that much smaller than expected analysis errors.

while solution of the analysis equation is an $\mathcal{O}(np)$ operation.

As mentioned earlier, the innovation matrix $HP^fH^T + R$ is dense, although entries associated with locations that are separated by several correlation lengths are negligible (or zero for compactly supported correlation functions, Gaspari and Cohn 1996). In order to introduce some sparseness in $HP^fH^T + R$ and save computational

effort, the correlations beyond a preset cutoff distance are not included. For PSAS JPL, with a $6,000$ kilometer cutoff between centroids of regions the innovation matrix is approximately 26% full, and this uses in excess of 5 gigabytes of storage.

3 Requirements

The scientific and software requirements are set out in the document: Data and Architectural Design for the GEOS 2.1 Data Assimilation System Document Version 1 (Lamich and da Silva 1996), and GEOS 3.0 System Requirements (Stobie 1996). For GEOS 3.0 we outline the following requirements that pertain to the parallelization effort:

- The parallel PSAS 3.0 will be prototyped and designed along with the serial version PSAS 2.1. In particular, the code is being developed using f90, and will handle non-state variable observation operators. The formal merging of the serial and parallel coded will be done in 1997 on a schedule to be determined by the DAO GEOS 3.0 Design Team.

- The parallel design must generate scientific software that will have a long life cycle. The recent development of PSAS as a new algorithm for data assimilation, and the incorporation of f90 into GEOS 2.1 affords the opportunity to use a modular approach that allows for expandability, decreases the likelihood of bugs, and makes it easier for a larger group of scientists to use and modify the same code. It is commonly acknowledged that parallel computing, and message-passing in particular, are sufficiently complex that some effort has to be made to hide the communication modules from a substantial population of the regular programmers.

- After an extensive review of DAO computing activities, and following the recommendation of the external DAO Review Panel (Farrell *et al.* 1996), the *Message Passing Interface* (MPI) parallel library will be used. If necessary, mixed language third-party software may be used provided portable Fortran bindings are available.

- The parallel code must scale to meet the performance needs of future production efforts of the DAO. This includes a commitment to MTPE by 1998 (Zero *et al.* 1996), and ongoing commitments to HPCC (Lyster *et al.* 1995).

Because PSAS 2.1 and PSAS 3.0 are both in a design phase, the exact definition of datatypes, modularity, and interfaces has not been decided. **It is recommended that this be done as soon as possible**. In the meantime this document will address parallel questions that must be answered as part of the design phase; most importantly, can the technology of PSAS JPL be transfered to PSAS 3.0 so as to provide efficient, load balanced code that also satisfies software configuration management requirements?

Note that a number of vendors support (or will soon) f90 and MPI on their hardware. Issues of hardware will not be further discussed, except to note that message-passing is a safe approach for the design of large-scale tightly-coupled algorithms. This is because a strong coupling between the user-generated data domain decomposition and the physical layout of memory affords the ability to optimize and scale against communications (latency and bandwidth) overhead.

4 The JPL Prototype Parallel PSAS

4.1 Summary of the algorithm

The developmental serial version of PSAS was given to computer scientists at the Jet Propulsion Laboratory in 1994. They had considerable experience in parallel conjugate gradient algorithms. The key element of the algorithm is a large matrix multiply $(HP^fH^T + R)x^{'}$ where $x^{'}$ is an intermediate weight vector. The key for message-passing parallelism is to break the matrix into components that can be parceled out to different processors (domain decomposition). Each matrix block acts on an appropriate vector fragment to form a partial vector. The result of each matrix multiply is an set of partial vectors which must be summed using message passing to form the result. In this way both the total memory and the work in generating the matrix and performing the multiply are divided among the processors; it is a tenet of parallel computing that any data or process that cannot be decomposed is a risk to performance. The large memory that is available on parallel computers, such as the Intel Paragon or the Cray T3D, also affords the possibility of calculating and storing the entire forecast error covariance matrix once per analysis cycle. This has the potential for considerable saving over approach of the serial code where the matrix was reevaluated for each iteration of the conjugate gradient algorithm. Note that the present formulation uses P_s^f, which is based on the location of observations in the state grid. Therefore the structure of P_s^f is different for each analysis cycle since the observing system, in particular TOVS satellite orbits, is not fixed. It was realized that

the regional decomposition of data that was employed by Pfaendtner *et al.* (1995) in the serial PSAS could be the basis for a message-passing parallel algorithm. In the serial version an icosahedral mesh with triangular subdomains was used to form compact regions on the surface of a sphere, each of which have approximately equal numbers of observations. Correlations between members of these regions are used to form blocks of the forecast error covariance matrix P_s^f. This is a convenient way of implementing a distance-cutoff (typically $6,000$ km) between the centroids of regions so that some sparseness (26%) can be enforced on the large matrix. More importantly, is also provides a basis for the domain decomposition of the parallel PSAS JPL. The following sections summarize the parallel algorithm that was developed (Ding and Ferraro 1995). The original serial code assimilated only state variables (p_s,u_s, v_s, u, v, h, and q); this included satellite retrieved mean layer temperature converted to height which is routinely generated from TOVS measured radiances. Hence there is no accounting for the observation operator F in this algorithm.

In short, the algorithm of the parallel version of PSAS comprises the major computational steps outlined below:

- Partition observations using inertial recursive bisection scheme. This is accomplished by the call to the function
  `partitioner( Obs_handle* , MPI_Comm* )`.

- Decomposition of the innovation matrix M via a call to the block matrix distribution function
  `matrix_distr( Obs_handle* , Vec_handle* , Reg_replica_handle* , Mblk_list* , MPI_Comm* )`

- Solution of the innovation equation via the conjugate gradient (CG) method, which is performed by the function
  `equation_solve(Vec_handle* , Reg_replica_handle*, Mblk_list* , MPI_Comm* )`

- Partition of the matrix $P^f H^T$ and solution of the analysis equation. This is accomplished by the call to the function
  `foldback( Vec_handle* , MPI_Comm* )`.

Each of these steps in the algorithm will be examined in greater detail in subsequent sections. A top-level flowchart of the parallel PSAS that summarizes the above calls from `main()` is presented in Figure 2.

4.1.1 Parallel Partition

This algorithm divides p observations among N_r regions using bisection on the surface of a sphere. The data are read in as innovations ($w^o - Hw^f$) and distributed in random order (but in equal numbers) on N_p processors. In the first iteration, the observations are divided along the orthogonal cut of their combined principal axis moment of inertia. Successive iterations are performed in a tree structure that repeats the decomposition on half the remaining processors with approximately half the data. At each stage, the calculation of the principal axis is performed in parallel. As regions replicate, data are moved between different processors using split MPI communicators. The inertial division guarantees some degree of compactness of the resulting decomposition. This algorithm requires there to be power-of-two number of processors and number of regions, with $N_r \geq N_p$. Typically, $N_p = 256$, or 512, and $N_r = 512$. It is possible to modify the power-of-two restriction on numbers of regions and processors (the orthogonal cut may be made anywhere along the principal axis to give a division of observations other than 50-50), and maintain approximately equal numbers of observations per region. However if the number of regions in each processor is not fixed for all processors there may be some load imbalance in the analysis equation (section 5.1.4). A schematic for the parallel partition of observations is shown in Figure 1. Note that the decomposition is in terms of equal numbers of observations, and that the regions are not necessarily equal in area. Also, the decomposition is two dimensional and profiles are not permitted to be divided between regions (for reasons that will be discussed in section 5.1.2).

The key structures (C) for the parallel partition are shown in Appendix B.1 (`Obs_handle`) and B.2 (`Gpt_handle`). Initially, the unsorted observations are approximately equally distributed among processors. The structure `Obs_handle` references elemental structures of type `Observ` that store the observations (actually innovations, $w^o - Hw^f$) `del`, and a number of attributes (`id`, `kt`, `kx`, `rlats`, `rlons`, `rlevs`, `xyz[3]`, `SigO`, and `sigF`). The allocated memory for all the observations on each processor is contiguous. The structure `Gpt_handle` (not to be confused with gridpoints of a regular grid) is very similar to `Obs_handle` except that the elemental type (`Gpoint`) holds only the `coord` positions of the observations on a unit sphere, a sequencing `id`, and the original processor `orgn_proc` (numbered from 0 to $N_p - 1$) where each observation is initially located. Using `Gpt_handle` the parallel partitioner proceeds by only passing data of type `Gpoint` between processors. In this way minimal data are passed (i.e., the other attributes and the value aren't passed during successive bisections of the the parallel partitioner). At each bisection, each processor identifies

the observations (`Gpoint`) that need to be passed to another processor and concatenates them in a buffer. This buffer is sent using `MPI_send`, with type `MPI_byte` and the receiving processor issues an `MPI_recv` call. The buffer is then unpacked into the `Gpt_handle`. This amortizes the message-passing latency across a long message at a (relatively small) cost of an extra local memory copy per observation. Note that memory is conserved by collapsing `Gpt_handle`, i.e., filling the holes that are left by `Gpoint`s that have been passed to other processors, after each bisection. After the last bisection, each member that is referenced by `Gpt_handle` is polled for its value of `org_proc` to find the originating processor where the actual data and its attributes is located. These data are buffered with others that have the same destination processor. An `MPI_send/recv` sequence, followed by unpacking the buffers, finally gives a compact partition referenced by `Obs_handle`. In this way the efficient parallel partitioner proceeds by sending minimal data between the processors during the bisection, and then only send one large message at the end. The resulting decomposition gives approximately equal numbers of observations per region (with some variation because profiles may not be split).

4.1.2 Matrix Block Partition

In this part of PSAS the parallel decomposition of observations is used to generate a list of matrix blocks. This is used to determine on which processor a matrix block will reside. For example, typically there are 512 regions containing approximately $p \sim 10^5$ observations. The 6,000 kilometer cutoff condition gives rise to a forecast error covariance matrix that is approximately 26% full. Therefore there are about $512^2 \times 0.26 \times 0.5 \sim 35,000$ blocks (the factor of 0.5 accounts for the symmetry of the matrix). This large number of blocks (Figure 2) is distributed among the (typically 256 or 512) processors in a load balanced manner. This achieves a balanced distribution of the memory and the work in generating the matrix elements (from a lookup table) and the work in performing the matrix vector multiply. The blocks are approximately of equal size, since there are approximately the same number of observations per region. Therefore, setting an equal number of blocks per processor provides an initial guess for a load balanced distribution. Each block is stored with the associated vector ($x^{'}$) fragment that must be used to perform the matrix-vector multiply. Diagonal blocks are stored on the processor that stores the observation for that region. For the off-diagonal blocks, it is clear the the number of vector fragments on each processor may be minimized by storing the block on one of the two processors that owns the observations corresponding to one dimension of the block. This limits

the ability to simply parcel out the blocks in a deterministic and load-balanced way. From the initial state, the load-balancing scheme proceeds towards this goal by first refining the initial distribution to produce a new initial state using the following scheme:

1. The load on each processor L_i to perform work for the conjugate gradient solver is estimated, along with the *average load* $\bar{L}$. This leads to a set of imbalances $\Delta_i \equiv \bar{L} - L_i$

2. For each pair (i, j) of processors, an exchange probability $P_{ij} = \frac{L_j}{L_i + L_j}$ is calculated.

3. Use a random number $r \in [0, 1]$ to decide if processor i or processor j is assigned the block. Processor i is assigned the block if $r \leq P_i$, while processor j is assigned the block if $r > P_i$. This amounts to a *weighted coin toss.*

From this refined initial state, the block distribution is improved iteratively through the following adjustment process:

1. For iteration $m \leq N_{\text{iter}}$, for each pair (i, j) of processors deviations from the average load balance Δ_i and Δ_j are calculated, along with the exchange probability P_{ij} described above. The number of blocks on each processor, $\mathcal{N}_{\text{i}}$ and $\mathcal{N}_{\text{j}}$, respectively, are counted along with the average number of blocks on each processor $\bar{\mathcal{N}}$.

2. If $\Delta_i > 0$ and $\Delta_j < 0$, the block is transferred from processor i to processor j. If $\Delta_i < 0$ and $\Delta_j > 0$, the block is transferred from processor j to processor i.

3. If both Δ_i and Δ_j have the same sign, then if $\mathcal{N}_{\text{i}} - \mathcal{N}_{\text{j}} > -\delta\bar{\mathcal{N}}$, then the block is moved from processor i to processor j.

4. Finally, for all other cases not covered above, the block is assigned to a processor based on the weighted coin toss described in the refinement of the initial state.

Following this scheme for $N_{\text{iter}} = 10$ and $\delta = 1/20$ results in a load imbalance of approximately 10%. The messages that must be exchanged during the matrix block partition are matrix indices and geometrical data. No complex structures, such as in the parallel partition, are exchanged until the block partition is complete.

4.1.3 Solve Parallel Conjugate Gradient

The conjugate gradient algorithm uses block preconditioning, and is described in Golub and van Loan (1989) and da Silva and Guo (1996). Computationally, the core is the large matrix-vector multiply. In the parallel partitioner the innovations are separated into regions that are distributed among the processors. These provide the initial condition for the vector iterate (x'). The vector is actually composed of vector fragments. The nature of distributed-memory parallel processing is that matrices and vectors are rarely represented as a whole on any one processor (an exception is sometimes made for the purpose of performing I/O). Depending on how the off-diagonal blocks are distributed, some of these vector fragments must be replicated on multiple processors. Furthermore the algorithm that generates the blocks uses lookup tables with inner loops over same-data types. Hence the vectors are reordered and referenced by a third structure `Vec_handle` (Appendix B.3). This is similar to the previously described structures except that the elemental type `Vec_region` references long vectors of attributes and data. In this case message-passing is easy since the same datatypes are effectively already buffered. It may be possible to sum the partial vectors using MPI split communicators and the `MPI_reduce(...,MPI_sum,...)` function. This uses a butterfly-tree algorithm. It turns out that the hand written code optimized using `MPI_send/recv` functions is more efficient since the number of processors holding any particular vector segment is small compared to the total number of processing units (C. Ding personal communication).

For PSAS JPL an older, considerably less efficient algorithm was used to perform the table lookup (it has been made faster in the present development PSAS by an order of magnitude). Hence, the generation of the matrix elements was a dominant part of the cost. The savings in evaluating the matrix only once per analysis cycle was considerable; effectively N_i (the number of CG iterations) multiplied by the cost of generating the matrix.

4.1.4 Calculate Analysis Increment

The analysis increment is (Eq. 2) $P^f H^T x$. The operator $P^f H^T \equiv P^f \mathcal{I}^\mathcal{T}$ represents the forecast error covariance between the observation grid and the analysis grid. The domain decomposition for this must account for the unstructured distribution of the observations and the structured analysis grid. The solution vector x for the conjugate gradient algorithm is decomposed in the same manner as the observations (Section 4.1.1). The decomposition for the analysis grid is based on

the fact that gridpoints within the $6,000$ mile cutoff of the location of an observation are affected by the corresponding weight in the vector x. A deterministic load-balanced algorithm proceeds as follows: an equal area rectangular distribution of gridpoints is generated (Figure 3). The boundaries of these regions are along latitude and longitude coordinate axes. At higher latitude the boundaries of the regions in the latitude-longitude plane is altered to keep the area in each region fixed, with a single cap over the poles. The gridpoints in these equal-area regions are thinned longitudinally; this is also performed increasingly with higher latitude in such a way that the number of gridpoints in each equal area region is the same. In this manner the matrix $P^f H^T$ may be generated as a number of equal-sized blocks whose centroids are within the $6,000$ km cutoff distance of the centroids of each region of x. Since there is a fixed number of regions per processor (i.e., usually 1 or 2) the number and size of the corresponding matrix blocks and the work that is performed in the partial matrix-vector multiply is the same on all processors. The structures that are used in this are `Grd_handle` and `Gvec_handle` these have the property that they are *pointers to pointers*, and are dereferenced at the lowest level of the calling tree in the foldback process, `create_gvec_regions(grd_handle, gvec_handle)` and `analysis_inc(vec_handle, gvec_handle, grd_handle)`. The partial vectors are then combined by using `MPI_All_reduceCP`.

4.2 Summary of Timings for PSAS JPL

Table 1. shows the timing for PSAS JPL for $80,000$ observations (model resolution $2.5^o \times 2^o \times 14$ levels) on 512 processors of the Intel Paragon at the California Institute of Technology. The solver achieves 18.3 gigaflop/s (77 megaflop/s per processor), which is 36% of peak performance for 512 processors. The parallel partitioner takes 3.1 seconds. This is a relatively small cost; the communications are complex but because of the strategy of sending minimal data and buffering there is little overhead. The calculation of block distribution lists uses a small amount of communications to pass simple lists between processors. During the replication of observation regions, vector fragments referenced by `Vec_handle` are sent between processors. These are relatively long messages and have little overhead. The calculation of matrix entries is very time consuming (23.8 seconds), but is performed only once per analysis. The solver uses BLAS level 2 library calls (`sgemv`) and messages (`MPI_SEND` and `MPI_RECV`) to send vector fragments of $x^{'}$. This is iterated $N_i \sim 100$ times [1], taking 36.4 seconds.

[1] This number of iterations differs from the value of N_i cited earlier because the version of PSAS that was parallelized had a more stringent convergence criterion than subsequent versions of PSAS.

The dominant cost of the analysis equation (referred to in Ding and Ferraro (1995) as "fold back") is from the generation of the matrix elements. Since one dimension of $P^f H^T$ is n which is larger than p this takes longer than the generation of the innovation matrix. The communication (1.5 seconds) involved in reassembling the analysis grid vector is similar to the assembly of vector fragments in the innovation equation.

For the innovation equation:

Task	*Time(sec)*
Read input data	14.6
Partition observations	3.1
Calculate block distribution lists	3.8
Replicate observation regions	3.3
Calculate matrix entries	23.8
Solve parallel CG	36.4
Miscellaneous	1.7
Total	87.0

Secondly, for the analysis equation:

Task	*Time(sec)*
Create grid partition	0.4
Assemble and evaluate $P^f H^T x$	67.6
Reassemble model vector	1.5
Total	69.5

Table 1. Timings for PSAS JPL on the 512 processor Intel Paragon at the California Institute of Technology.

The flop rates for runs that were recently performed on the Cray T3D at the NASA/Goddard Space Flight Center are shown in Figure 4. The closed circles are for a problem with $51,990$ observations and the diamonds are for $79,938$ observations.

Version	*solve conj grad.*	*analysis equation*
1994 PSAS	9120. (sec)	9000.
1996 PSAS	750.	750.
PSAS JPL	87.	71.

As was mentioned earlier, the condition cited here is more stringent than was later learned to be necessary.

Table 2. Comparison of timings for two versions of the serial PSAS and parallel PSAS JPL.

Table 2 shows a comparison between wall-clock times for three versions of PSAS: the serial code that was given to JPL as run on a single processor Cray C90; a recent optimized serial PSAS also run on a Cray C90; and a PSAS JPL run on 512 processor of the Intel Paragon. The older serial code is slower than JPL PSAS by two orders of magnitude, mainly due to the higher net flop rate of the parallel processor (approximately 10 gigaflop/s to 1 gigaflop/s) and the calculation of the matrix elements only once per conjugate gradient solve. The more recent, optimized, PSAS runs faster on the C90 because the algorithm for table lookup has been improved (making more use of redundancy and using nearest-neighbor lookup). In the future the need to form the error covariance matrices only once may be relaxed. This is discussed further in section 7.

5 Developmental PSAS 2.1 and 3.0

5.1 Summary of PSAS 2.1

The developmental serial PSAS 2.1 differs from older versions in two main ways: the use of f90 modules and types to modernize the software; and the incorporation of observation operators. Fortran 90 may impact message-passing through changes in the way observations are buffered in the parallel partitioner and vectors are passed in the replication and solve subroutines. Fortran 90 may also affect single-processor performance if pointers are not used carefully. PSAS 3.0 is the parallel version, and this paper deals mostly with the special problems associated with its development.

The observation operator significantly changes the algorithm (Lamich and da Silva 1996). The left hand side of the innovation equation becomes $(FP_s^fF^T + R)x$, where F is the tangent linear observation operator. The forecast error covariance matrix is formulated in state space (in particular P_s^f is dimensioned on the unstructured state grid of observation profiles). This gives rise a new representation of data on a state grid (section 2). Appendix C shows prototypes of f90 types `Obs_Vect` and `State_Vect`. Both of these types are built around (unbreakable) profiles. Attributes that do not vary along a profile are aggregated into types `Obs_Att` and `State_Att`. The `Obs_Vect` has all the attributes that facilitate the quality control functions and the generation of the F and R operators. The `State_Vect` has only those attributes that are needed to generate the forecast error covariance matrices (in particular, the

quality control `qc`, metadata index `km`, sounding index `ks` are left out, and the positions of the profiles on the unit sphere `xyz` are included). The observation operators act on profiles (or aggregates of profiles – soundings – at the same location). Hence the manner in which the `Obs_Vector` profiles are sorted in memory is not that important to performance. The observation error covariance matrix R may couple soundings, so it may be necessary to co-locate soundings in memory. At the lowest subroutine level, the P_a^f operator is generated using loops over of pairs of locations of profiles, each dimension of which corresponds to the same variable. Hence it is important to sort `State_Vect` *in memory* by data type (`kt`). The f90 types that are used, such as `State_Vect` or `Obs_Vect` actually reference allocated memory space (a memory handler). In general, it is the location of variables in memory that affects performance. If members of profiles or vectors are not appropriately sorted in memory then the inner loops may have to dereference f90 pointers and there may be a performance degradation. These are issues that affect both a serial and parallel application – especially on RISC-based processors (i.e., most scientific computers). The following sections summarize other aspects of the new data types, and how they may affect message passing. A summary of performance aspects is given in section 8.

5.1.1 Parallel Partition

This section compares the parallel approach of PSAS 3.0 that uses f90 datatypes in comparison with the approach of PSAS JPL that uses the C language and structures. As discussed in the previous section PSAS JPL uses `Gpt_handle` and `Obs_handle` structures in the parallel partition. `Gpt_handle` has reduced data (xyz position on the unit sphere and the identifying number of the originating processor) that allows the successive bifurcations of the partition to proceed efficiently. Is is sufficient here to show the analogous approach that uses f90 datatypes, and to indicate how this will give comparable performance. Considerable prototyping will have to be done in the future.

The type `Gpt_holder_f90` that is defined in Appendix D performs the equivalent function as `Gpt_handle`. The usage is:

```
type(Gpt_holder_f90) gpt

!There are numRegs regions in the problem
gpt%numRegs = numRegs
allocate(gpt%gpt_region(1:numRegs))
do index = 1,numRegs
```

```
 gpt%gpt_region(index)%numGpts=numgpts(index)
 allocate(gpt%gpt_region(index)%gpoints(1:numgpts(index))
 do iobs=1,numgpts(index
  gpt%gpt_region(index)%gpoints(iobs)%coord(1)=...x position of obs..
!  ...these data are derived from Obs_Vect...
 enddo
enddo
```

The parallel partition proceeds in the same way as PSAS JPL. At each stage, a subset of the data of type `Gpoint_f90` are buffered and passed between processors. There are (at least) two ways to do this:

(i) The data that are selected to be communicated are extracted from the holder `gpt` and inserted into two buffers of type `real` (for `xyz` positions) and `integer` (for `orgn_proc` and `id`). The buffers are communicated and unpacked at the recieveing processor. This is guaranteed to work since it respects the language types, but the loops that perform the buffering are clumsy and perhaps inefficient.

(ii) Allocate a buffer that stores a periodic array of type `Gpoint_f90`:

```
type Gpt_buffer
   type(Gpoint_f90),pointer::p(:)
end type Gpt_buffer

type(Gpt_buffer) gpt_buf

allocate(gpt_buf%p(1:max_obs_buffer))
```

This buffer can be directly filled with the data that needs to be passed between processors. The message will use the form
`MPI_send(gpt_buf,npoints*size(Gpoint_f90),MPI_BYTE,...)`.
This has been prototyped, and shown to work for the f90 compiler with MPI library on a DEC multiprocessor. However it may not be portable since f90 may not always interpret a subroutine argument as a simple pointer to a block of memory (Hennecke 1996).

In Section 4 it was shown how the structure `Obs_handle` references data with all the attributes sequentially located in memory with each observation value. In the last step of the parallel partitioner the identifying numbers of the originating

processors are used to buffer the observations and their attributes and send them to their destination processors. On the other hand, the type `Obs_Vect` that is shown in Appendix C.1 stores profiles sequentially. Redundant attributes for each profile are stored once per profile in the type `Obs_Att`. The bisection process generates a type `Gpt_holder_f90` that specifies the originating processor for the actual data. If the data are at least sorted into profiles before the parallel partition then entire profiles and attributes from `Obs_Vect` may be buffered and sent to the destination processor. This eliminates the need to send unnecessary attributes, and eliminates the need for equivalent structure to `Obs_handle` where all observations are treated as separate atomic units (along with attributes). Hence, `Obs_Vector` may fulfill a role in both message passing and in generating the R and F operators. This indicates that the f90 types `Gpt_holder_f90` and `Obs_Vect` may be used to perform the same function in the message-passing f90 parallel partitioner as the C structures `Gpt_handle` and `Obs_handle` do in PSAS JPL.

5.1.2 Matrix Block Partition

PSAS 2.1 includes, for the first time, the tangent linear observation operator F. At the end of the parallel partition, the observations are decomposed in profiles that are referenced by `Obs_Vect`. After that, the adjoint of the observation operator is used to form $F^T x$, which is referenced by `State_Vect`. This is an embarrassingly parallel calculation since operators F/F^T transform profiles of the observation grid ($\in \mathbb{R}^p$) into profiles of the state grid ($\in \mathbb{R}^s$). Therefore the forward model and its adjoint do not need to be parallelized – a blessing since they may be dusty-deck serial code – it is mainly for this reason that profiles are not permitted to be broken in the parallel partitioner (section 4.1.1 and 5.1.1). For the subsequent evaluation of $P_s^f F^T x$ there is a choice of maintaining a decomposition that has approximately equal numbers of observations in each region or transforming to another representation where equal numbers of the state grid values are in each region. The latter more closely resembles the approach of PSAS JPL because the resulting blocks of P_s^f are approximately of equal size. The parallel partitioner of PSAS JPL initially assumes that all blocks of the innovation matrix are the same size (i.e., they cost the same to generate the block and perform the submatrix-vector multiply). The subsequent iterative process, whereby load-balance is ensured, uses the size of the block as a cost function. Therefore, if we use an approximately even decomposition of the state grid the subsequent load balance algorithm will be the same as PSAS JPL. For example, the convergence of the load-balance calcualation would be as rapid, and the extent of load balance would

be expected to be the same (PSAS JPL achieves about 10% load balance after 10 iterations of the balancing process). The load balance of the operation Fx is not necessarily assured in this algorithm. However, since the number of profiles $\sim 10^4$ is much larger then the number of processors there would be a natural convergence toward load balance due to the large numbers. Also, six hours of TOVS soundings (20,000 profiles, each of which has 20 channels) takes 5 minutes of processing for the radiative transfer calculation on a 50 megaflop/s DEC uniprocessor (J. Joiner, private communication). Therefore, at 10 gigaflop/s the calculation should take about 1.5 seconds which is considerably less than the cost of generation of P_s^f and the matrix-vector multiply (Table 1). If necessary, the operators corresponding to different types of profiles could be costed and the result used to modify the cost function for the evaluation and use of blocks P_s^f (i.e., use a more sophisticated cost function other than the size of the blocks).

Note that the operation Rx is performed in the decomposition of the observation grid. It may be necessary to sort `Obs_vect` by sounding in order to preserve in-memory locality. This may help performance in a RISC-based processor. Other than that, Rx is load balanced and should not require fundamental changes between PSAS JPL and PSAS 3.0.

5.1.3 Solve Parallel Conjugate Gradient

For PSAS 3.0, the conjugate gradient algorithm is not significantly different than PSAS JPL. The algorithms for replication of fragments of `State_Vect` and the subsequent parallel sum of partial vectors uses values that are sorted into long vectors of the same type. This is the same as PSAS JPL which used the structure `Vect_handle`.

5.1.4 Calculate Analysis Increment

The message-passing that is required to calculate $P_s^f F^T x$ is similar to PSAS JPL. Matrix block lists and long vectors need to be replicated and passed as necessary. Once again load balance may be an issue. As discussed in section 5.1.2, because of the large number of profiles and the relatively small cost, the load balance for $F^T x$ may not depend on whether x is decomposed in terms of equal number of the `Obs_Vect` or `State_Vect`. However to ensure load balance of the subsequent matrix generation and matrix-vector multiply, the vector $F^T x$ (of type `State_Vect`) should be decomposed into equal numbers in regions. In this case PSAS 3.0 can use the same decomposition of the analysis grid `Anal_Vect` as PSAS JPL (Figure 3). This will guarantee that blocks of P_s^f are of equal size and there will be the same number

on all processors. The decomposition is load balanced and static (there is no iterative load-balancing process here). This may be a problem if we relax the restriction of equal sized regions and equal number of regions on each processor, because the size and number of matrix blocks assigned to processors may become disparate.

6 The Parallel Evaluation of Forecast Error Covariance Matrices

6.1 PSAS JPL, PSAS 2.1, and PSAS 3.0

One of the advantages of parallel computing is that the memory scales proportionally with the number of processors (subject to cost) and may allow for the storage of the entire forecast error covariance matrix. As described in section 4.2, if the matrix is stored once there is a time saving proportional to the number of iterations of the conjugate gradient solver N_i. This explains in part the impressive improvement in wall-clock time between the serial PSAS in 1994 and the parallel PSAS JPL (Table 2). Table 2 also shows how the serial algorithm has been made an order of magnitude more efficient through improved algorithms for calculating the matrix elements. However there is still some saving in calculating the elements once. For a machine which is memory deficient it may be necessary to develop an algorithm that dynamically decides whether to store or recalculate matrix elements. The error covariance operator P_s^f is not evaluated as a single matrix for current serial versions of PSAS, but is instead a series of sparse operators that act successively on a vector (A. M. da Silva, Personal Communication). Thus, it is not a matrix that is stored, but rather a structure of coefficients. This may make it difficult to evaluate a cost function for the load balancing algorithm, although the large numbers of profiles per processor should ameliorate the problem.

Currently, for an isotropic and separable formulation of the correlation function, the lookup tables may be stored in memory on each processor. However, as we relax these assumptions the lookup tables will grow (possibly up to the size of P^f) and may have to be decomposed in a similar way as the matrix blocks (section 4.1.2). The next section discusses a related implementation.

6.2 Issues for Evaluating P^f on the Analysis Grid

Evaluation of a forecast error covariance matrix is a computationally intensive part of solving the innovation and analysis equations. The amount of computation required

for brute-force evaluation of a general forecast error covariance matrix followed by the matrix-vector multiplication is proportional to the dimensions of the matrix. Therefore, in the current PSAS, the forecast error covariance matrix is evaluated directly on the state grid instead on the larger analysis grid. We propose an efficient algorithm which applies to a wide class of forecast error correlation functions. The algorithm uses symmetries of the analysis grid to greatly reduce the number of correlation function evaluations over brute-force methods.

In the current PSAS, the product of matrices $\mathcal{I}P^f\mathcal{I}^T$, where P^f is the $n \times n$ and $\mathcal{I}$ is $s \times n$ matrix, is approximated by the $s \times s$ matrix P_s^f, where s is typically smaller than n by an order of magnitude. Generally, brute-force evaluation of a full $k \times k$ forecast error covariance matrix requires $(k^2 + k)/2$ covariance function evaluations. Brute-force matrix-vector multiplication requires k^2 multiplications and $k^2 - k$ additions. In the future, the number of observations p will increase, thus increasing s, as well. If s becomes larger than n, and the interpolation matrix $\mathcal{I}$ is implemented as a very sparse matrix, it will be more efficient to use $\mathcal{I}P^f\mathcal{I}^T$ instead of P_s^f.

For a wide class of forecast error covariance functions, the use of $\mathcal{I}P^f\mathcal{I}^T$, with the algorithm for evaluation of P^f presented below, is computationally efficient even if the analysis grid is refined so that n remains an order of magnitude larger than s. Denote the multidimensional (*i.e.*, multi-level) random field of geopotential height forecast errors by

$$\mathbf{h}(\mathbf{p}) = \{h_1(\mathbf{p}), h_2(\mathbf{p}), \ldots, h_m(\mathbf{p})\}, \tag{4}$$

where $h_1(\mathbf{p}), h_2(\mathbf{p}), \ldots, h_m(\mathbf{p})$ are random fields on the sphere (Earth's surface) of geopotential height forecast errors corresponding to pressure levels $1, 2, \ldots, m$. The covariance of $\mathbf{h}(\mathbf{p})$ is given by the matrix of covariance functions

$$\Sigma(\mathbf{p}_1, \mathbf{p}_2) = \{B_{jk}(\mathbf{p}_1, \mathbf{p}_2)\}, \tag{5}$$

and the correlation of $\mathbf{h}(\mathbf{p})$ is given by the matrix of correlation functions

$$? \, (\mathbf{p}_1, \mathbf{p}_2) = \left\{ \frac{B_{jk}(\mathbf{p}_1, \mathbf{p}_2)}{B_{jj}(\mathbf{p}_1, \mathbf{p}_1)^{\frac{1}{2}} B_{kk}(\mathbf{p}_2, \mathbf{p}_2)^{\frac{1}{2}}} \right\}, \tag{6}$$

where B_{jk} is a covariance function of the random fields h_j and h_k, $j, k = 1, 2, \ldots, m$. The algorithm we propose applies to models where ? depends only on parameters which are preserved under the symmetries of the uniform longitude-latitude grid on the sphere. For instance the distance

$$d(\mathbf{p}_1, \mathbf{p}_2) \tag{7}$$

between $\mathbf{p}_1$ and $\mathbf{p}_2$ and the absolute value of the latitudes

$$|\varphi_1| \quad \text{and} \quad |\varphi_2| \tag{8}$$

of $\mathbf{p}_1$ and $\mathbf{p}_2$ are preserved under rotations and reflections of the uniform longitude-latitude grid on the sphere (*cf.* Definitions 2.4 and 2.5, Gaspari and Cohn 1996). The correlation ? currently used in the PSAS depends only on $d(\mathbf{p}_1, \mathbf{p}_2)$.

The covariance matrix P^f, which is the analysis grid evaluation of multi-level covariances between all the state variables (h, u, v, q, p_s u_s and v_s) can be written as a product of a diagonal matrix D of standard deviations, and a correlation matrix C

$$P^f = DCD. \tag{9}$$

Denote by C_{hh} the block of C of geopotential height forecast error covariances, *i.e.*, the evaluation of ? on the analysis grid. Denote by $C_{hh}(j,k,\varphi_1,\varphi_2)$ the block of C_{hh} which is the evaluation of B_{jk} between the points on circle of latitude φ_1 and the points on circle of latitude φ_2. It is assumed that the points on each circle are sorted with increasing longitude. Denote the number of points on each circle of constant latitude by I. In the current PSAS, $I = 144$. If ? depends only on the parameters (7) and (8), due to the symmetries of the analysis grid, the matrix C_{hh} has the following structure. Each of the blocks $C_{hh}(j,k,\varphi_1,\varphi_2)$ is a symmetric circulant matrix (see Figure 7), so there are $2I$ elements with identical values. Moreover, there are three more blocks of C_{hh} identical with $C_{hh}(j,k,\varphi_1,\varphi_2)$,

$$\begin{aligned} & C_{hh}(j,k,\varphi_1,\varphi_2) \\ = \; & C_{hh}(j,k,-\varphi_1,-\varphi_2) \\ = \; & C_{hh}(k,j,\varphi_2,\varphi_1) \\ = \; & C_{hh}(k,j,-\varphi_2,-\varphi_1). \end{aligned} \tag{10}$$

Therefore, there are $8I$ identical elements of the matrix C_{hh}.

The following is a brief description of the algorithm for evaluation of C_{hh} and computation of the matrix-vector product $\mathbf{z}_h = C_{hh}\mathbf{y}_h$, which exploits the fact that $8I$ elements of C_{hh} are identical. For every set of $8I$ identical matrix elements of C_{hh}, their value is computed by one evaluation of the correlation function. These $8I$ matrix elements are needed in computation of some, typically $8I$, coordinates of the product vector $\mathbf{z}_h$. All these coordinates of $\mathbf{z}_h$ are updated, that is the product of the element of of C_{hh} with a coordinate of $\mathbf{y}_h$ is added to the previous value of a coordinate of $\mathbf{z}_h$. This process is repeated until $\mathbf{z}_h$ is computed. A sequential version

of this algorithm was implemented on Cray C-98. It was up to 42 times faster than the algorithm which was evaluating every element of the correlation matrix.

A parallel version of this algorithm has been implemented in FORTRAN 77 using MPI on the Cray T3D. The processors are organized in a three-dimensional virtual topology. The distribution of the work was done in a "card shuffling" manner, that is each processor works with every n^{th} circle of constant latitude and pressure level, where n depends on the number of available processors. This distribution of the work has the property that increasing the support of the correlation function or refining the grid results in a more balanced load. On a grid with 8 levels and meshes of 2 degrees in latitude and 2.5 degrees in longitude using a correlation function with support of 3000 km, the speedup of this algorithm was 75.5 on 128 processors.

In this algorithm only one value of $8I$ identical matrix elements is stored at one time. The algorithm can be modified so that an $I \times I$ matrix block $C_{hh}(p_1, p_2, \varphi_1, \varphi_2)$ (equal to three other blocks of C_{hh} in equation (10)) is evaluated and stored. The matrix-vector multiplication of this block with four subvectors of $\mathbf{y}_h$ can be performed using a BLAS routine.

This algorithm was described for the block C_{hh} of C for simplicity. If correlation functions between state variables are modeled by the functions which are currently used in the PSAS, the same algorithm applies to evaluation of the the entire matrix C and matrix-vector multiplication $\mathbf{z} = C\mathbf{y}$.

7 Optimization and Load Balance

For message-passing optimization the messages should be few and buffered. For the algorithm that involves communicating lists and information about geometry there are sufficiently few bytes that this is not a significant burden. For the communication of vector fragments the data are already effectively buffered in vectors. Ding and Ferraro (1995) have developed a sophisticated algorithm that optimizes the accumulation of partial sums without using the `MPI_reduce` function. For the communication of observation and profile data Ding and Ferraro also buffer data and thus amortize the startup cost of messages.

For RISC-based technology the on-processor optimization performance depends on the ability to move data efficiently through cache to the arithmetic units and then back to memory. The process of calculating a covariance matrix and using BLAS calls is, in a sense, optimizing cache because the matrix is an optimal form for the input data. For cases where C or f90 pointers are used to refer to data, they should

be dereferenced outside of inner loops. The quantities Rx and $F^T x$ are calculated one profile (or sounding) at a time. $P_s^f(F^T x)$ is calculated using BLAS functions. $F(P_s^f F^T x)$ is calculated one profile (or sounding) at a time. The evaluation of matrix elements, which is a significant cost, may require prototyping and cache optimization. In particular, indirection is a major problem for cache optimization. For example, modern RISC-based processors may have up to two orders of magnitude difference between the time it takes to access data in cache (cache is usually kilobytes in size) and the time to access remote data in physical memory. It is often more efficient to separately sort data and thus avoid indirection on inner loops. For example, for isotropic horizontal correlation functions it may be useful to sort the profile pairs in order of increasing distance before indexing the lookup table in a loop.

The load-balancing algorithm of Ding and Ferraro was very successful. We have shown in section 5.1.2 that for PSAS 2.1 and PSAS 3.0 with observation operators there is no conceptual difficulty with following the same approach. However, the analysis equation may create problems because it is not self adjusting; it relies on the fact that blocks of P_s^f on each processor in the folding-back decomposition (Figure 3) are the same size and cost. This would be difficult to maintain if we broke the requirement of having the same number of regions of observations on each processor.

8 Input/Output and the PSAS-GCM Interface

The calculation of the innovation requires transformation from the analysis grid on which the forecast is generated and an observation grid. In principle this is not an excessive cost in terms of message-passing because the model grid which has $p \sim 10^6$ variables (i.e., $\sim 10^7$ bytes) would take approximately 0.1 seconds to be arbitrarily transformed on a parallel computer with a net bandwidth of ~ 100 megabytes/s (these back of the envelope calculations always represent an underestimate because of the hidden cost of message latency). The difficulty is one of writing efficient, modular software that transforms data between the structured analysis grid (it may be the domain decomposition of the GCM) and the unstructured observation grid. Similarly, on the back end of PSAS efficient software needs to be developed for transforming between the domain decomposition for the analysis grid of PSAS and that of the GCM. Writing the analysis grid or observation data streams (ODS, da Silva and Redder 1995) to disc using, say, MPI-IO will also need these grid transformation tools. It will be so easy to get beaked to death by a thousand ducks.

9 Parallel Quality Control

The present on-line quality control is a two-stage process (Seablom *et al.*, 1991). The gross check compares innovations ($w^o - Hw^f$) against a specified upper bound. For any even distribution of profiles on processors, such as given by the parallel partition (section 4.1.1) the gross check is obviously load-balanced and embarrassingly parallel. Those observations that fail the gross check are flagged and a subsequent buddy check is performed. This involves comparing the suspect value with a statistically interpolated estimate based on a number of nearest-neighbor unflagged observations. The comparison amounts to data indirection with its concomitant cache inefficiency. Unless a significant number of observations are flagged it is probably not efficient to sort the data for buddy check. Efficiency is further complicated by a dependency in the comparison loop that allows re-accepted data to influence not-yet-buddy-checked data. The parallel quality control algorithm of von Laszewski (1996) uses a domain decomposition based on the analysis grid. Clearly, it makes more sense to use the parallel partition of observations (section 4.1.1) as a basis for the quality control in PSAS 3.0, in effect leveraging the work that is already done for the domain decomposition of PSAS. Overlap regions of redundant data are used to parallelize the loop for the buddy check. Because of the redundancy, some processors have to wait for others to complete their fragment of the buddy-check loop, thus giving rise to a potentially pathological load imbalance. von Laszewski calculates that the number of observations that fail the buddy check is usually small enough that the load imbalance does not adversely affect overall performance. However if optimal performance is required then the algorithm itself may have to be modified to eliminate the the dependency in the buddy-check loop, rendering the buddy-check embarrassingly parallel.

10 Reproducibility

There are two aspects of reproducibility for PSAS. First, the same problem (i.e., same values of all physical and numerical parameters) should give bitwise identical results, with the exception that roundoff may vary with the number of processors, N_p. This may occur for the partial vector sum algorithm (section 4.1.2 and 4.1.4) where, for different numbers of processors, additions may be performed in a different order. Often, the guarantee of bitwise identical results are helpful for debugging. For PSAS (and a lot of other parallel applications, especially those that use standard "reduction" library functions) this can only be guaranteed for runs on the same

numbers of processors.

The second aspect of reproducibility is that results (e.g., the value of the analysis increment) may vary up to the middle order bits when the number of regions, N_r, is changed (i.e., truncation, but not necessarily "error"). Of course, there are a number of other parameters that affect truncation (e.g., number of iterations of the conjugate gradient N_i) but the number of regions is a special case because it is coupled to the configuration of the parallel computer. At present, N_r must not only be a power of two (because of the recursive bisection algorithm) but it must be an integer multiple of the number of processors in order that the analysis equation be load balanced. Typically, PSAS JPL is run with $\sim 10^5$ observations, 512 regions, and 256 or 512 processors. For the Intel Paragon, with about 8 gigabytes of available memory, we cannot run with fewer than 256 processors because of the storage of P_s^f. This may turn out to be too inflexible in terms of the use of PSAS as a production/scientific tool on a range of computing platforms. The obvious modification is to allow the recursive partitioner to select a non-powers-of-two decomposition of observations into regions. In this case, to ensure load balance the algorithm for the analysis equation would have to be self adjusting in a similar way as the load-balancing process for the matrix block partition (section 4.1.2) A second problem with truncation may arise for inhomogeneous observation patterns. The parallel partitioner will generate relatively large regions (in terms of physical area) where data are sparse. In this case the use of the data centroids of regions as a criterion for applying the correlation cutoff may be inconsistent. A more rigorous approach may base the cutoff on the minimum distance between vertexes of regions. This in turn will adversely affect load balance of the analysis equation, which assumes blocks of P_s^f to be approximately of the same size. Once again, the solution may be a self adjusting load balancing algorithm for the analysis equation.

At present, compactly supported correlation functions (that are exactly zero beyond a fixed distance) are being used in PSAS. Another issue related to truncation is that applying a cutoff to matrix blocks based on distances between centroids of data applies a harsher truncation than the case where the compact correlation function is evaluated pointwise on an observation grid. This potential problem is worse for parallel PSAS because the regions are unstructured, as opposed to the present scientific PSAS which uses structured (icosahedral) regions.

11 Portability, Reusability, and Third-Party Software

There are several key functions in PSAS JPL that may be modified and installed as (C-based) library for PSAS 3.0. It should be noted that these higher level functions of PSAS JPL were written with clean modular interfaces (often a single input and single output handles), which should make their modification fairly simple. Providing precautions are taken with the interfaces, there should be no problem calling C-based libraries from f90 code – especially when the f90 compiler has a Fortran 77 heritage (J. Michalakes private communication).

- The parallel partitioner was based on a general design that has a wider range of applications than earth science (Ding and Ferraro 1995). Hence it is a sturdy algorithm and may be modified for PSAS to allow for a non-power-of-two number of regions. We may also want to implement a flexible decomposition based on the observation grid and/or the state grid (section 5.1.2). This should be prototyped.

- The matrix block partitioner may be easily modified to include a more general cost function for the load-balancing process. As pointed out in section 5.1.2 this is straightforward for an data decomposition based on the state grid.

- The custom algorithm for combining partial vectors in the parallel matrix solve may be packaged as a library function.

- The analysis grid decomposition and algorithm for the analysis equation could be packaged, although considerable modification may be required to allow for self-adjusting load balanced algorithm.

Finally, as discussed in section 5.1.1 there may be some portability problems where f90 types are used as arguments for MPI message-passing functions (Hennecke 1996). This should be prototyped, even to the extent of generating a set of diagnostic functions to run on target parallel platforms libraries and compilers.

Acknowledgments

The original design of the parallel PSAS was developed in consultation with Robert Ferraro of the Jet Propulsion Laboratory. We would also like to acknowledge useful discussions with Max Suarez and Dan Schaffer at Goddard. The work of Chris

Ding at the Jet Propulsion Laboratory and Peter Lyster and Jay Larson at the Data Assimilation Office was funded by the High Performance Computing and Communications Initiative (HPCC) Earth and Space Science (ESS) program, contract number NCCS5-150.

References

DAO Staff: Algorithm Theoretical Basis Document for Goddard Earth Observing System Data Assimilation System (GEOS DAS), Data Assimilation Office Goddard Space Flight Center, Greenbelt, MD 20771.

Ding, H. Q., and R. D. Ferraro 1996: An 18GFLOPS Parallel Data Assimilation PSAS Package, *Proceedings of Intel Supercomputer Users Group Conference 1996*. To be published in Journal of Computers and Mathematics; also Ding, H. Q., and R. Ferraro, 1995: A General Purpose Parallel Sparse Matrix Solver Package, *Proceedings of the 9th International Parallel Processing Symposium*, p. 70.

Farrell, W. E., A. J. Busalacchi, A. Davis, W. P. Dannevik, G-R. Hoffmann, M. Kafatos, R. W. Moore, J. Sloan, T. Sterling, 1996: *Report of the Data Assimilation Office Computer Advisory Panel to the Laboratory for Atmospheres.*

Gaspari, G., and S. E. Cohn, 1996: Construction of Correlation Functions in Two and Three Dimensions. *DAO Office Note 96-03.* Data Assimilation Office, Goddard Space Flight Center, Greenbelt, MD 20771.

Golub, G. H. and C. F. van Loan, 1989: *Matrix Computations*, 2nd Edition, The John Hopkins University Press, 642pp.

Hennecke, M., 1996: A Fortran 90 interface to MPI version 1.1. RZ Universität Karlsruhe, Internal Report 63/96.
`http://ww.uni-karlsruhe.de/~Michael.Hennecke/`

Lamich, D., and A. da Silva, 1996: Data and Architectural Design for the GEOS-2.1 Data Assimilation System Document Version 1. *DAO Office Note 97-??* (in preparation). Data Assimilation Office, Goddard Space Flight Center, Greenbelt, MD 20771.

von Laszewski, G. 1996: The Parallel Data Assimilation System and its Implications on a Metacomputing Environment. PhD. Thesis Computer and Information Science Department, Syracuse University.

Lyster, P. M., and Co-I's, 1995: Four Dimensional Data Assimilation of the Atmosphere. A proposal to NASA Cooperative Agreement for High Performance Computing and Communications (HPCC) initiative.

Pfaendtner, J., S. Bloom, D. Lamich, M. Seablom, M. Sienkiewicz, J Stobie, A. da Silva, 1995: Documentation of the Goddard Earth Observing System (GEOS) Data Assimilation System – Version 1. *NASA Tech. Memo. No. 104606*, Vol. 4, Goddard Space Flight Center, Greenbelt, MD 20771. Available electronically on the World Wide Web as `ftp://dao.gsfc.nasa.gov/pub/tech_memos/volume_4.ps.Z`

Seablom, M., J. W. Pfaendtner, and P. E. Piraino, 1991: Quality Control techniques for the interactive GLA retrieval/assimilation system. *Preprint Volume, Ninth Conference on Numerical Weather Prediction,* October 14-18, Denver, CO, AMS, 28-29.

da Silva, C. Redder, 1995: Documentation of the GEOS/DAS Observation Data Stream (ODS) Version 1.01, *DAO Office Note 96-01.* Data Assimilation Office, Goddard Space Flight Center, Greenbelt, MD 20771.

da Silva, A., J. Guo, 1996: Documentation of the Physical-space Statistical Analysis System (PSAS). Part I: The Conjugate Gradient Solver, Version PSAS-1.00. *DAO Office Note 96-02.* Data Assimilation Office, Goddard Space Flight Center, Greenbelt, MD 20771.

da Silva, A., J. Pfaendtner, J. Guo, M. Sienkiewicz, and S. Cohn, 1995: Assessing the Effects of Data Selection with DAO's Physical-space Statistical Analysis System. *Proceedings of the second international symposium on the assimilation of observations in meteorology and oceanography*, Tokyo Japan, World Meteorological Organization.

Stobie, J. 1996: GEOS 3.0 System Requirements.

Zero, J., R. Lucchesi, R. Rood, 1996: Data Assimilation Office (DAO) Strategy Statement: Evolution Towards the 1998 Computing Environment.

Appendix A: List of Symbols and Definitions

A.1 List of Symbols

n	number of analysis gridpoints $\times$ number of variables	$n \sim 10^6$
p	number of observations	$p \sim 10^5$
s	dimension of the state vector (size of the state grid)	
w^a	gridded analysis state vector	$\in I\!R^n$
w^f	gridded forecast state vector	$\in I\!R^n$
w^o	observation vector	$\in I\!R^p$
H	tangent linear generalized interpolation operator	$H : I\!R^p \rightarrow I\!R^n$
$\mathcal{I}$	interpolation operator	$\mathcal{I} : \mathcal{IR}^{\setminus} \rightarrow \mathcal{IR}^{\int}$
F	tangent linear observation operator	$\mathcal{I} : \mathcal{IR}^{\int} \rightarrow \mathcal{IR}^{\surd}$
P^f	forecast error covariance matrix	
	defined on the analysis grid	$P^f : I\!R^n \rightarrow I\!R^n$
P_s^f	forecast error covariance operator	
	defined on the state grid	$P_s^f : I\!R^s \rightarrow I\!R^s$
R	observation error covariance	$R : I\!R^p \rightarrow I\!R^p$
p_s	sea level pressure	hPa
u_s	zonal surface wind speed	m/s
v_s	meridional surface wind speed	m/s
u	upper air zonal wind speed	m/s
v	upper air meridional wind speed	m/s
h	pressure level height	km
q	moisture mixing ratio	g/kg
N_p	number of processors used for a run	
N_r	number of regions used to partition the observations	
N_i	number of iterations of the conjugate gradient solver	

A.2 Definitions

Analysis grid	The latitude-longitude-pressure grid of the model analyzed fields (a structured grid)
Observation grid	The grid of locations of observations (an unstructured grid)
State grid	The grid of locations of analysis grid

	interpolated to the horizontal location of the observation profiles (a quasi-structured grid)

A.3 Versions of algorithms

GEOS 2.1	Goddard Earth Observing System Data Assimilation System that will incorporate observation operators.
GEOS 3.0	Goddard Earth Observing System Data Assimilation System that will incorporate parallel software.
PSAS 2.1	The version of PSAS with observation operators.
PSAS 3.0	The parallel version of PSAS with observation operators.
PSAS JPL	The earlier developmental version of PSAS (1994) that was parallelized by Jet Propulsion Laboratory.

A.4 PSAS JPL Datatypes

`Obs_handle`	Structure that references innovations ($w^o - Hw^f$) and observation attributes (value and attributes are co-located in physical memory, and the total dataset is stored in a contiguous buffer).
`Gpt_handle`	Structure that references the (x,y,z) locations of profiles projected onto the unit sphere (each atom (x,y,z) is co-located in memory, and the total dataset is stored in a contiguous buffer).
`Vec_handle`	Structure that references vectors of observations, and attributes such that each value is represented in physical memory as a long vector, and each attribute is separately in physical memory as a long vector.
`Grd_handle`	Structure that references indices and arrays that describe the decomposition of the analysis grid for the parallel analysis solve.
`Gvec_handle`	Structure that references the grid values on the domain decomposed analysis grid, stored in physical memory as a long vector.

A.5 PSAS 2.1 Datatypes

`Obs_Vect`	The f90 type which references the observations that are sorted into profiles by `kr, (ks), kt, kx, lat, lon, (ks), lev` i.e., the vector $\in I\!R^p$
`State_Vect`	The f90 type which references the state grid that is sorted into profiles by `kr, kt, lat, lon, lev` i.e., the vector $\in I\!R^s$
`Anal_Vect`	The f90 type which references the analysis (and forecast) grid; each variable $(u_s, v_s, p_s, u, v, h, q)$ are stored in physical memory as long vectors.

A.6 Data Attributes

kr	Region
ks	Sounding
kt	Type
lat	Latitude
lon	Longitude
km	Meta-data
qc	Quality Control
val	Value, or innovation

Appendix B: PSAS JPL Data Structures

B.1: Obs_handle Structure

```
/* data structures for observations.   Written by Chris H. Q. Ding at JPL */
#include "ktmax.h"

/* structure for an individual observation point */
typedef struct {
  int   id ;
  int   kt ;
  int   kx ;
  Rvalue rlats;
  Rvalue rlons;
  Rvalue rlevs;
  Rvalue xyz[3];
  Rvalue del;
  Rvalue sigO;
  Rvalue sigF;
} Observ;

/* all observation point data in a region. */
typedef struct{
  int reg_id;
  int numObs;
  int ktlen[KTMAX];
  int kt_typlen[KTMAX];
  int totlen;      /* length in bytes of this object with variable length */
  Rvalue  cent_mass[3]; /* center of mass of all observation points */
  Rvalue  extension[3]; /* rms distance in all 3 directions         */
  Observ *observs; /* sorted when pass to matrix build */
  Observ observLoc;
} Obs_region;

/* top structure to hold all obs_regions address on this processor */
typedef  struct{
  int numRegs;
```

```
  Obs_region ** obs_regions;
  int obs_regions_limit;
} Obs_handle;
```

B.2: Gpt_handle Structure

```
/* data structures in partitioner.c   Written by Chris H. Q. Ding at JPL */

#define  NDIM    3 /* dimension of space, Partition bases on x,y,z */

/* structure for an individual geometric point on the surface */
typedef struct {
  Rvalue coord[NDIM];
  int   orgn_proc; /* original processor when partitioning started */
  int   id;        /* sequencial id number when partitioning started */

/* Both orgn_proc and id are for keeping track of grids
 * movement purose, and are not referred to in partitioner. */
} Gpoint;

/* all gpoints in a region. Used for partition purpose */
typedef struct{
  int  reg_id;
  int  numGpts;
  int  totlen;  /* length in bytes of this object with variable length */
  Rvalue  cent_mass[NDIM];
  Rvalue  extension[NDIM];
  int    gpoints_limit; /* used during parallel partitioning,
   indicating # of gpoints allocated for *gpoints */
  Gpoint *gpoints;
  Gpoint gpointLoc; /* gpoints points to this location */
} Gpt_region;

/* top structure to hold all gpt_region address in this processor */
typedef  struct{
  int numRegs;
  Gpt_region ** gpt_regions;
  int gpt_regions_limit;
} Gpt_handle;
```

B.3: Vec_handle Structure

```
/*************************************************************************
    Data structure for observations stored as 20 arrays, each of them has
a length "reglen" and is pointed by a pointer defined in vec_region.

    These arrays are stored in memory immediately following the memory for
vec_region itself, so the whole thing can be moved in one piece.

    The total length of the structure and arrays is "size" bytes.

    To save memory in the folding back part, one could store only
rlats, rlons, rlevs, xobs, yobs, zobs, qcosp, qsinp, qcosl, qsinl,
kx, kt and xvec vectors, 13 vectors, instead of 20 vectors needed for
the correlation matrix part. Not yet implemented.
*************************************************************************/
/*  Written by Chris H. Q. Ding of JPL  */

/*
#include "ktmax.h"
#include "Rvalue.h"
#include "maxsizes.h"
*/

typedef struct {
/* The rlats, rlons, rlevs, xobs, yobs, zobs, kx, kt, id vectors
   are directly from obs_region */
Rvalue *rlats; /* latitudes */
Rvalue *rlons; /* longitudes */
Rvalue *rlevs; /* pressure levels */
Rvalue *xobs; /* xobs = cos(rlats)*cos(rlons) */
Rvalue *yobs; /* yobs = cos(rlats)*sin(rlons) */
Rvalue *zobs;   /* zobs = sin(rlats) */

/* qcosp, qsinp, qcosl, qsinl are computed in form_vec_regions() */
Rvalue *qcosp; /* cos(rlats) */
```

```
Rvalue *qsinp; /* sin(rlats) */
Rvalue *qcosl; /* cos(rlons) */
Rvalue *qsinl; /* sin(rlons) */
Rvalue *del; /* used only to calculate bvec = del*Dinvii.
   Current, xvec is stored here */

/* The following 5 vectors, sigO, sigF, Onorm, Fnorm and Dinvii
   are used only for constructing the correlation matrix */
Rvalue *sigO; /* used only to calculate Onorm = sigO*Dinvii */
Rvalue *sigF;    /* used only to calculate Onorm = sigF*Dinvii */
Rvalue *Onorm;   /* could use the same memory for sigO */
Rvalue *Fnorm;   /* could use the same memory for sigF */
Rvalue *Dinvii; /* Dinvii = 1/sqrt(sigO**2 + sigF**2) */

/* bvec is used only for solving the correlation matrix */
Rvalue *bvec;   /* could use the same memory for del  */

/*
Rvalue *xvec; solution to the CG part. This array differs from all other
   arrays in that is it allocated right before CG part
*/
int  reglen; /* number of observation points in this region */
int  ityplen[KTMAX];
int  *kx;               /* kx type for each obs */
int  *kt; /* kt type for each obs */
int  *id; /* sequence id from the sequential preprocessing part*/
int  reg_id;
int  float_offset; /* no. of bytes from start of this structure to rlats*/
int  Rvalue_offset; /* no. of bytes from start of this structure to bvec */
int  int_offset; /* no. of bytes from start of this structure to kx */
long size; /* total number of bytes used by this region */
Rvalue cent_mass[3]; /* center of mass, for checking correlaion purpose */
int  is_owned;     /* =1 if owned, =0 if not owned */
} Vec_region;

/* The top structure to hold all vec_regions address on this processor */
```

```
typedef  struct{
  int numRegs;
  Vec_region ** vec_regions;
  int vec_regions_limit;
  char *mem_non_owned_regs; /* starting memory location for non-owned regs */
  int num_owned_regs;
  int owned_regs_list[MAXREGNS]; /* list of reg_ids for owned vec_regions */
} Vec_handle;
```

B.4: Grd_handle Structure

```
/* grids related data structures. Written by Chris H. Q. Ding of JPL */

/* active grids --- those grids after decimation.
   static grids --- the grids before decimation, i.e., those basic grids.

   Each grd_region essentially defines a template for the grids in the
   region. They are all active grids since gvec_region will generate vectors
   based on these grids.

   All global_vector are based on active grids.
   All universal_vectors are initially based on active grids. After
     expand_uvec(), universal_vecs are based on static grids.
*/

/* The structure to define a grd_regions */
typedef struct{
  int reg_id;
  int num_grids;  /* # of active grids in this region */
  int num_corr;   /* # of obs regions correlated to this region */
  int num_gds_latlong[2]; /* # of active latitude grids and longitude grids */
  int start_loc; /* starting location in the active universal vector */
  int n_long_grids_zone; /* # of active longitude grids at this zone */
  Rvalue start_latlong[2]; /* lat-long cordinates of the lower-left corner */
  Rvalue cent_latlong[2];
  Rvalue cent_mass[3];
} Grd_region;

/* The top structure to hold all grd_regions on this processor */
typedef  struct{
  int numRegs;
  Grd_region ** grd_regions;
  int grd_regions_limit;
  int tot_active_grids; /* sum of active num_grids of all grd_regions */
 /* this is the currently used grids after decimation */
```

```
  int n_long_grids[TOT_LAT_GRIDS] ; /* # of active longitude grids
       at each latitude. */
  int tot_static_grids; /* # of static grids on the surface */
  int tot_grids_lat;  /* # of static latitudinal num_grids */
  int tot_grids_lon;  /* # of static longitudinal num_grids */
} Grd_handle;
```

B.5: Gvec_handle Structure

```
/* grids related data structures. Written by Chris H. Q. Ding of JPL */
#define NFVECS 9 /* # of Rvalue vector in gvec_region */

/* Note: all num_grids, max_numgrds, tot_numgrds in gvec_region refers
 to "active" grids, i.e., those grids after decimation.  */

/* The structure to define a gvec_region */
typedef struct{
  int   reg_id;
  int   num_grids; /* # of active grids in the gvec_region */
  int   num_corr;  /* number obs_regions this gvec_reg correlates to */
  int   *obs_corr_list;  /* list of correlated obs_region reg_ids */
  int   global_offset; /* starting position in the global_array */
  int   size;   /* =sizeof(Gvec_region) + num_grids*9*sizeof(float) */
  int    *kx;   /* points to a vector of all 0's, no memory allocated */
  Rvalue  *glevs;  /* points to a vector of all glev's, no memory allocated */
  /* memory allocated for the following 9 Rvalue vectors, starting at strLoc */
  Rvalue  *glats;
  Rvalue  *glons;
  Rvalue  *xgrid;
  Rvalue  *ygrid;
  Rvalue  *zgrid;
  Rvalue  *qcospg; /* cos(lat) */
  Rvalue  *qsinpg; /* sin(lat) */
  Rvalue  *qcoslg; /* cos(lon) */
  Rvalue  *qsinlg; /* sin(lon) */
  Rvalue  cent_mass[3];
  /*
  INC_SUBVEC **inc_subvecs; ** increment sub-vectors for many kt and level **
  int    num_inc_subvecs;
  */
/*  Grd_region  *grd_link;   points to the corresponding grd_region */
  int    *vec_reg_corrlist;  /* list of correlated vec_regions, by reg_id */
  Rvalue  strLoc; /* starting location of the 9 float vectors */
```

```
} Gvec_region;

/* The top structure to hold all grd_regions on this processor */
typedef  struct{
  int         n_gvec_regs;      /* number of gvec_regions on this proc */
  Gvec_region **gvec_regions;   /* array of pointers */
  int         gvec_regions_limit; /* # of allocated gvec_regions pointers */
  Rvalue      *PHmatrix;    /* points to memory for largest PH matrix blk */
  int         PHmatrix_limit; /* size of the largest PH matrix blk */
  int         max_numgrds;  /* max of any single gvec_region on this proc */
  int         tot_numgrds;  /* sum of num_grids on all gvec_regions on this p*/
  Rvalue       preslevels[MAXLEVELS];
  int         nlevels; /* # of pressure levels to be folded back */
  int         *kx_buffer; /* buffer for the kx-array for grids */
  Rvalue       *plev_buf;  /* buffer for the pressure level-array for grids */
  Rvalue       *plev_ptr[MAXLEVELS]; /* pointer array for each pressure level */
} Gvec_handle;
```

Appendix C: GEOS 2.1 Data Types

C.1: Obs_Vect Type

```
type Obs_Att
 integer::kr,kt,kx,ks,km
 real::lat,lon
end type Obs_Att

type Obs_Prof
 integer::nlev
 type(Obs_Att)::att
 real, pointer :: val(:)
 integer, pointer::qc(:)
 real, pointer::lev(:)
end type Obs_Prof

type Obs_Vect
 integer::nprof
 type(Obs_Prof), pointer :: prof(:)
end type Obs_Vect
```

C.2: State_Vect Type

```
type State_Att
 integer::kr,kt
 real::lat,lon
 real::x,y
end type State_Att

type State_Prof
 integer::nlev
 type(State_Att)::att
 real, pointer :: val(:)
```

```
 real, pointer::lev(:)
 real, pointer::z(:)
end type State_Prof

type State_Vect
 integer::nprof
 type(State_Prof), pointer :: prof(:)
end type State_Vect
```

C.3: Anal_Vec Type

```
type Anal_Vect
 real, pointer::us(:,:)
 real, pointer::vs(:,:)
 real, pointer::slp(:,:)
 real, pointer::u(:,:,:)
 real, pointer::v(:,:,:)
 real, pointer::h(:,:,:)
 real, pointer::q(:,:,:)
end type Anal_Vect
```

Appendix D: GEOS 3.0 Proposed Data Types

D.1: Gpt_handle_f90

Note that Gpt_holder_f90 emulates the structure of Gpt_handle that is used in PSAS JPL.

```
type Gpoint_f90
 real::coord(3)
 integer::orgn_proc
 integer::id
end type Gpoint_f90

type Gpt_region_f90
 integer::reg_id
 integer::numGpts
 integer::totlen
 real::cent_mass(3)
 real::extension(3)
 integer::gpoints_limit
 type(Gpoint_f90),pointer::gpoints(:)
end type Gpt_region_f90

type Gpt_holder_f90
 integer::numRegs
 type(Gpt_region_f90),pointer::gpt_region(:)
end type Gpt_holder_f90
```

Appendix E: Notes on the JPL Parallel PSAS Code

E1. Where the JPL Parallel PSAS is Archived

The code is a public-domain code available through the High Performance Computing and Communications (HPCC) Software Exchange, and can be obtained via anonymous ftp at the universal resource locator (URL)

`ftp://hera.gsfc.nasa.gov/pub/hpcc/PSAS.T3D.FIXED/mpiPSAS.tar`

Note that this tar file contains both the source code and sample data, and thus is quite large. The total disk space needed for the tar file alone is over 61 Mbytes, and the directory tree structure that results when it is untarred occupies over 70 Mbytes of disk. To be safe, one should have on the order 150 Mbytes to download, compile, and run the JPL PSAS.

E.2 About the Source Code for the Parallel PSAS

The parallel PSAS is a hybrid code, comprising 105 files of C source code (15107 lines of source code), along with 47 FORTRAN 77 files (7623 lines of source code). The source code is organized in a directory hierarchy summarized below:

- `distr`: This directory contains code that accomplishes the distribution of matrix blocks of the innovation matrix and related codes.
- `fill`: Organize and fill in blocks of the forecast error covariance matrix and related codes.
- `fold_old`: Solution of the innovation equation. This operation is sometimes referred to as the *foldback*, since it involves folding the CG solution x from the observation grid to the analysis grid.
- `ftrn_new`: FORTRAN codes containing the parameterizations of forecast error covariances, *et cetera*. One could argue that the bulk of the science behind PSAS lies here in these routines.
- `include`: The header files for all the C programs.
- `io`: Input/Output-related codes, including `read_data.c`, `read_parameter.c`, *et cetera*.
- `misc`: miscilaneous service routines, such as `print_matrix.c`, `printflag.c`, *et cetera*.

- `mv`: The parallel matrix-vector multiplication code lies here, along with the rest of the parallel conjugate-gradient solver (`cg_solver`) source code.
- `part`: Source code for both the parallel and sequential partitioners and their related routines.
- `src`: The driver program, `main.c`. As mentioned in the previous section, this is the directory from which PSAS is compiled via the command `make all`
- `run`: Directory from which one runs the parallel code. Sample input data, along with file `param.in`, which contains the control parameters for the parallel PSAS. The file `Read.me` contains detailed instructions on running the code, information regarding test cases, *et cetera*.

E.3 Building the Executable JPL Parallel PSAS

Once one has downloaded the tar file `mpiPSAS.tar`, the procedure outlined below should be followed to build the executable. One should note that the current distribution of the code is configured to be compiled on either the Intel Paragon or the Cray T3D (default).

1. Create a directory as your main PSAS directory from which will serve as the root, whose descendents will include directories containing all the source code, support data, and a directory from which to execute the code. The suggested directory name is mpiPSAS (parallel PSAS). On Unix systems (and in this section a unix-like operating system is assumed from this point foward), this directory can be created by typing the command `mkdir mpiPSAS`
2. Next, go to the mpiPSAS directory (use command `cd mpiPSAS`). Move the tar file `mpiPSAS.tar` to this directory (or, for that matter, move to this directory and download the tar file `mpiPSAS.tar` into this directory). Unpack the tar file using the command `tar -xvf mpiPSAS.tar` which will create the directory structure containing the source code, data, *et cetera*.
3. At this point, it is necessary to set some environment variables associated with the compilation process using `make`, and also the execution of the code. In the directory mpiPSAS, typing the command `source ./setup` will accomplish this. For the shells `csh` or `tcsh`, one can alter the `.cshrc` file to set this variable by adding the line `setenv PSASHOME mpiPSAS` Adding this line to the `.profile`

file is necessary for users of the `ksh` and `bash` shells. In any event, this action sets the environment variable `PSASHOME`, which can be verified by the command `echo $PSASHOME`

4. Now it is possible to proceed with the installation of the code. In the directory `mpiPSAS/src` type the command `make all` This is a lengthy process that ultimately produces the executable file `main`, which is then moved to the directory `mpiPSAS/run`.

E.4 Running the JPL Parallel PSAS

Once the code has been compiled, and is configured to run on `nprocs` processing elements, it may be run on the Cray T3D via the command `main -npes nprocs`

In order to run correctly, the contents of the parameter file `param.in` have to be specified correctly. Below is a listing of the parameters found in `param.in`, along with their significance. A sample listing of `param.in` can be found in Appendix E.5. Note that the entries in the parameter file must be in order and with no omissions.

- `datafile=` the input data file containing innovations.
- `incvfile=` the output data file, containing the analysis increment vectors
- `nobs=` the number of observations
- `totregns=` the total number of observation regions
- `max_iter=` the maximum number of iterations for the Conjugate Gradient (CG) solver
- `precond_iter=` maximum number of iterations for the preconditioner to the CG solver
- `tolerance=` fractional change at which CG solver terminates successfully
- `precond_tol=` same as above for the preconditioner CG solver
- `decimate=` 1 for equal-area grids, 0 otherwise
- `write_incvc=` 0=no write; 1= to binary file incvec.out; 2=to stdout
- `want_slu=` 1 fold back sea level u-wind, 0 otherwise

- `want_slv=` 1 fold back sea level v-wind, 0 otherwise
- `want_slp=` 1 fold back sea level pressure, 0 otherwise
- `want_uwnd=` 1 fold back up air u-wind, 0 otherwise
- `want_vwnd=` 1 fold back up air v-wind, 0 otherwise
- `want_hght=` 1 fold back up air geo hight, 0 otherwise
- `want_mixr=` 1 fold back up air mix ratio, 0 otherwise
- `plevel_low=` Upper air pressure level lower limit
- `plevel_high=` Upper air pressure level upper limit
- `ncols=` Number of columns of processors
- `nrows=` Number of rows of processors
- `printflag=` Printging flags. set printflag=-1 will give all options
- `ebugflag=` Debugging flags. set printflag=-2 will give all options
- `printproc=` The processor on which data are printed.
- `stopflag=` > 0 stop execution at the ith place. ≤ 0 no stop
- `freeflag=` Free parameter for convenience. Can be used for anything
- `rho=` Parameter used in simulated annealing load-balance algorithm for maxtrix block distribution (currently not used)
- `beta=` Parameter used in simulated annealing load-balance algorithm for maxtrix block distribution (currently not used)
- `rm_seed=` Random number seed for maxtrix block distribution load balancing scheme
- `niter=` Number of iterations for maxtrix block distribution load balancing scheme.

E.5 Procedures for Modifying the Source Code

The makefile system for the source code is a multi-directory, two-pass system with all the necessary cross-checking of dependencies. All relations are specified in one and only in one place. A modification of the `makefile` (*e.g.*, adding a source code file) should be made in `makefile.org`, *not* `makefile`. The dependencies on the header files will be accounted for automatically. After a change of `makefile.org` in one or any directories, all you need to do to re-compile the codes is to type `make all` in the `mpiPSAS/src` directory.

If one wishes to modify any of the C or FORTRAN source code, or modify any of the header (`.h`) files, one can, following this procedure:

1. Go to the relevant directory, and edit the C (`.c`), FORTRAN (`.f`), or header (`.h`) file
2. In that directory, type the command `make all`
3. Go to the `mpiPSAS/src` directory, and type the command `make all` to build a new executable `main`. Note that one can skip step 2 and simply execute this step with the same result.

If one wishes to modify the source code by adding further source code files (C or FORTRAN), it is accomplished as follows:

1. Go the relevant directory and add the new source code file
2. Change the `makefile.org` file to reflect this new source code
3. Type the command `make makefile` in that directory, which will create a new `makefile`.
4. Type command `make all` in that directory.
5. Go to the directory `mpiPSAS/src` and type the command `make all` to get a new executable `main`. Note that this step alone can accomplish what was outlined in the two previous steps.

If one wishes to refresh file dependecies, change or split header files into more header files, *et cetera*, such actions are equivelent to rebuilding the entire PSAS system. Here is what needs to be done subsequent to such changes:

1. Execute the command `touch makefile.org` in the appropriate or all directories. This can also be done via `touch */makefile.org` from the directory `mpiPSAS/src`.

2. Go to the directory `mpiPSAS/src` and type the command `make all` to build the new executable `main`.

Finally, it should be noted that there are files present in the source code directory hierarchy that are needed by `make` to correctly implement the cross-directory compilation procedure. If these files are missing or incorrectly modified, `make` could fail.

- The file `mpiPSAS/compile` contains predefined flags and variables used by `make`.
- In each of the source code subdirectories, there should be a `makefile`, which contains make rules for the files in that directory
- In each of the source code subdirectories, there is a file entitled `LINKFILES`, in which filename and dependency information to the master makefile in `mpiPSAS/src`. It is generated from the local `makefile`.

E.6 Sample Input Parameter File param.in

```
datafile=dataCRAY.51990  /* input data file */
incvfile=incvec.51990    /* output data file, increment vectors */
nobs=51990         /* number of observations */
totregns=512       /* number of observation regions */
max_iter=150      /* max iteration in Preconditioned Conjugate Gradt solver */
precond_iter=150  /* max iter in the Preconditioner which is also a CG solver */
tolerance=0.001  /* residual reduction factor in PCG solver */
precond_tol=0.001 /* residual reduction in the Preconditioner itself */
decimate=1      /* 1 for equal-area grids, 0 otherwise */
write_incvc=1   /* 0=no write; 1= to binary file incvec.out; 2=to stdout */
want_slu=0      /* 1 fold back sea level u-wind, 0 otherwise */
want_slv=0      /* 1 fold back sea level v-wind, 0 otherwise */
want_slp=0      /* 1 fold back sea level pressure, 0 otherwise */
want_uwnd=1     /* 1 fold back up air u-wind, 0 otherwise */
want_vwnd=1     /* 1 fold back up air v-wind, 0 otherwise */
want_hght=1     /* 1 fold back up air geo hight, 0 otherwise */
want_mixr=1     /* 1 fold back up air mix ratio, 0 otherwise */
plevel_low=19.0   /* pressure level lower limit : min=9.0 */
plevel_high=1501.0  /* pressure level upper limit : max=1001.0 */
ncols=0         /* the # of processors on columes */
nrows=1         /* the # of processors on rows */
printflag=0     /* printging flags. set printflag=-1 will give all options */
debugflag=0     /* debugging flags. set printflag=-2 will give all options */
printproc=0     /* the processor on which data are printed. */
stopflag= 0     /* >0 stop execution at the ith place. <=0 no stop. */
freeflag=-19    /* Free parameter for convenience. Can be used for anything */
rho=1.0         /* for maxtrix block distribution */
beta=10.        /* for maxtrix block distribution */
rm_seed=12345   /* for maxtrix block distribution */
niter=10        /* for maxtrix block distribution */
/* Don't change the sequential order of lines. They are important */
```

Appendix F: Workshop Participants

Workshop I: September 30 - October 4, 1996, DAO, Goddard Space Flight Center (GSFC), Greenbelt, Maryland.

- P. M. Lyster (chair), Meteorology Department and Joint Center for Earth System Science (JCESS), University of Maryland.
- A. M. daSilva, General Sciences Corporation and NASA Goddard Space Flight Center (GSFC)
- C. H. Q. Ding, Lawrence Berkeley National Laboratory
- Jing Guo, General Sciences Corporation
- J. W. Larson, Department of Earth and Atmospheric Sciences, Purdue University[2]
- I. Štajner, Universities Space Research Association

Workshop II: October 28 - November 1, 1996, DAO GSFC.

- P. M. Lyster (chair), Meteorology Department and Joint Center for Earth System Science, University of Maryland.
- A. M. daSilva, General Sciences Corporation and NASA Goddard Space Flight Center (GSFC)
- Jing Guo, General Sciences Corporation
- J. W. Larson, JCESS, University of Maryland
- W. Sawyer, ELCA Informatique[3]
- I. Štajner, Universities Space Research Association

[2]Current Affiliation: Dept. of Meteorology and JCESS, University of Maryland
[3]Current Affiliation: Dept. of Meteorology and JCESS, University of Maryland

Appendix G: Figures

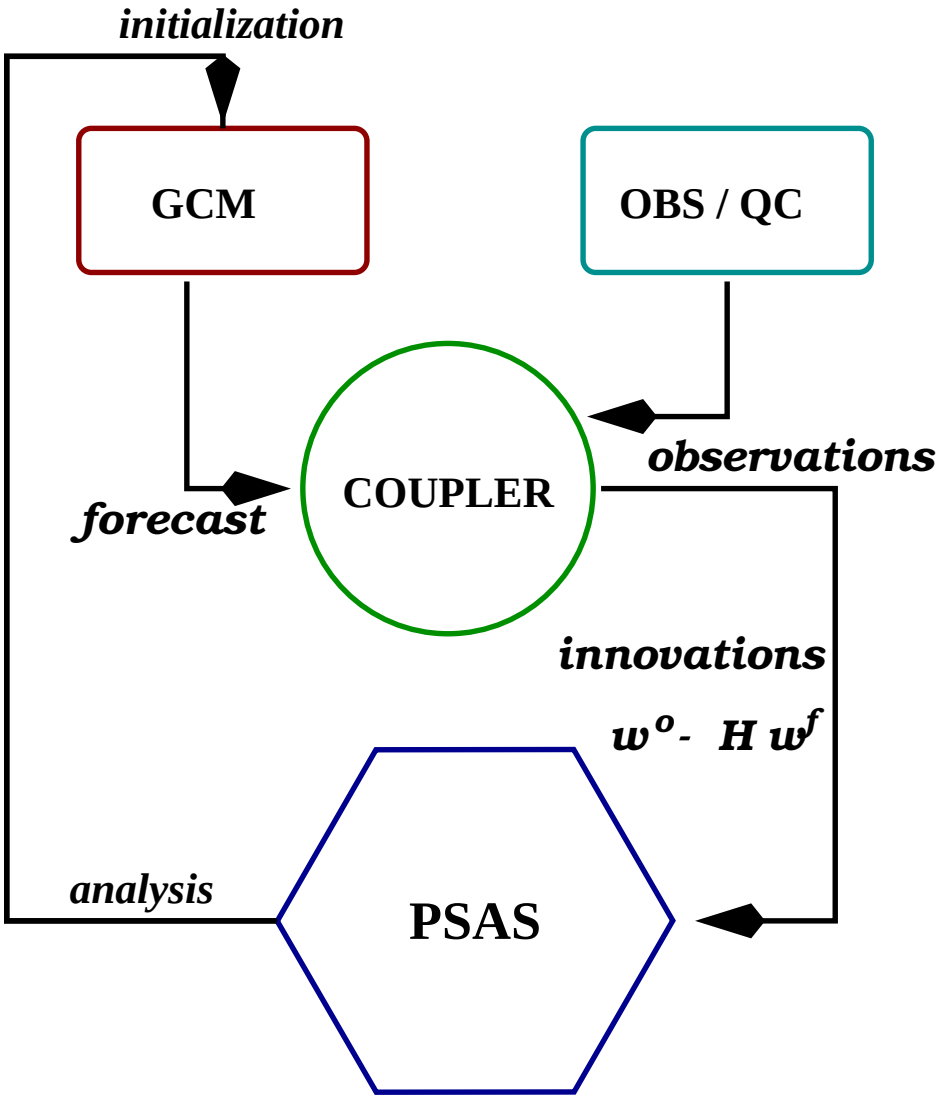

Figure 1: PSAS and its role in GEOS-DAS.

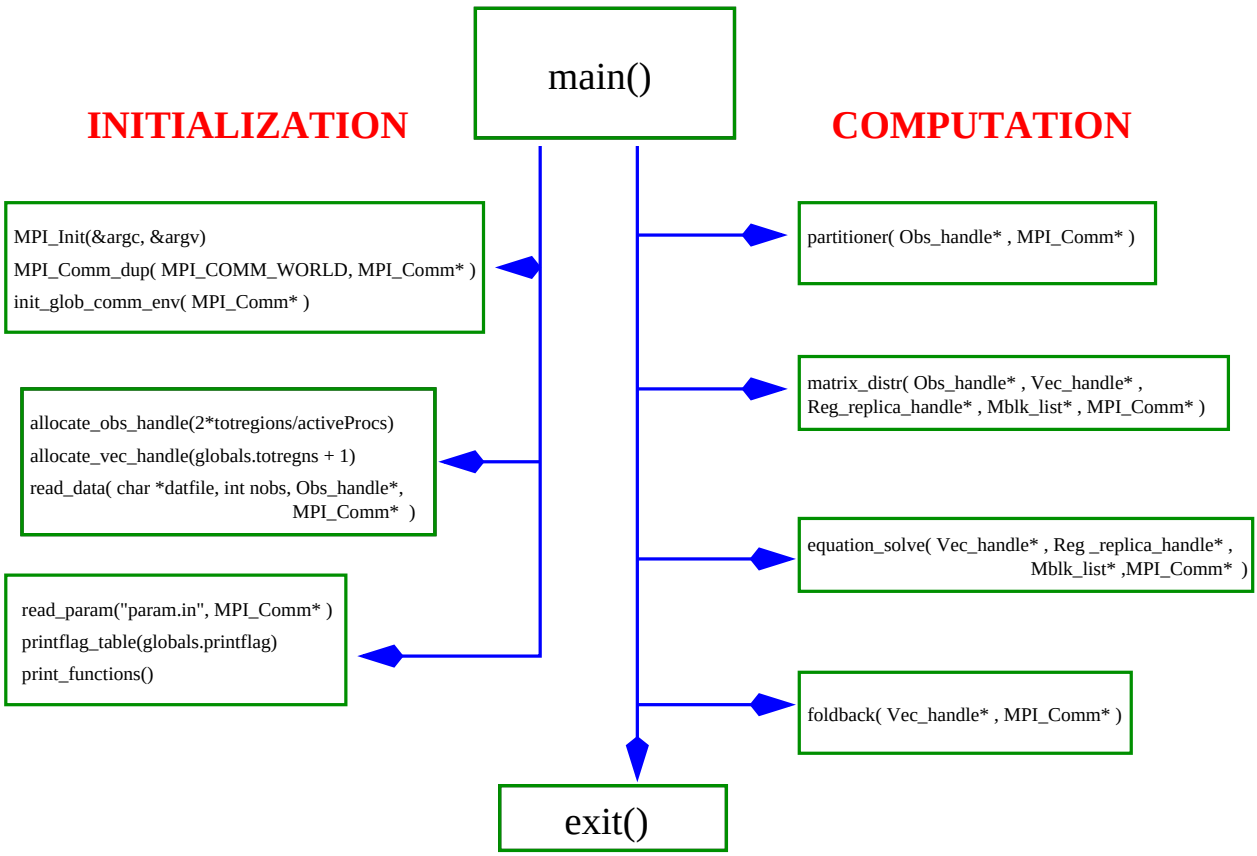

Figure 2: Top-level flowchart of parallel PSAS.

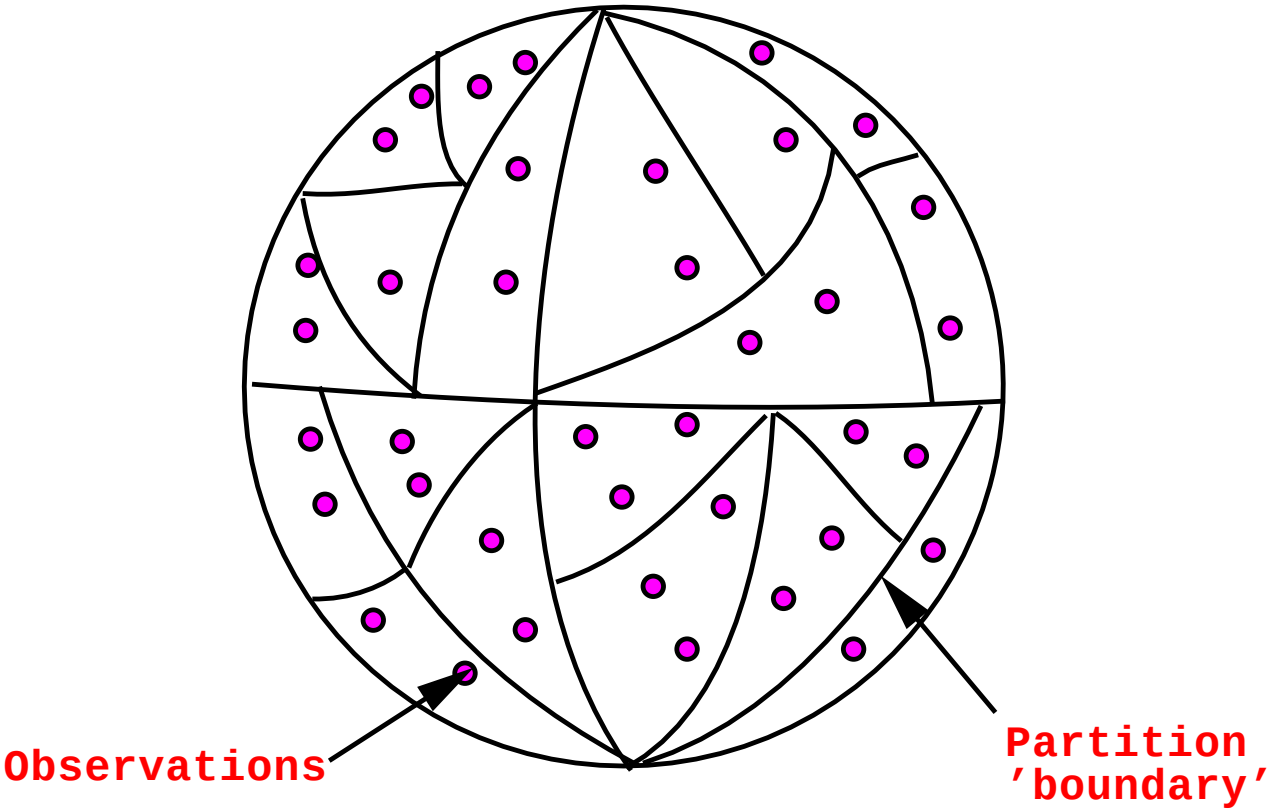

Figure 3: Parallel partition of observations into regions using recursive bisection.

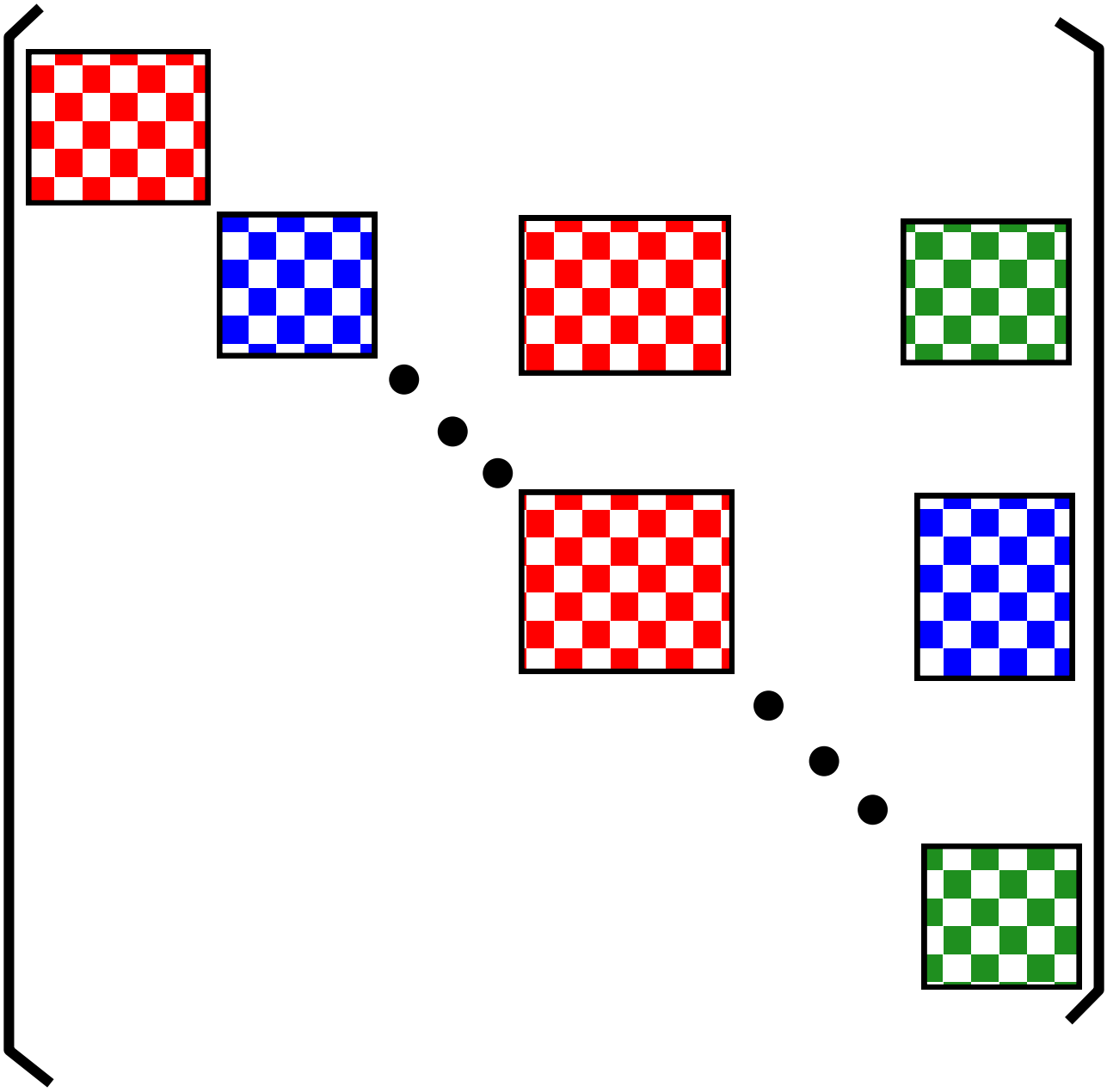

Figure 4: Matrix-block distribution of the innovation matrix $HP^fH^T + R$ The upper-half blocks of the complete symmetric matrix are assembled on each processor.

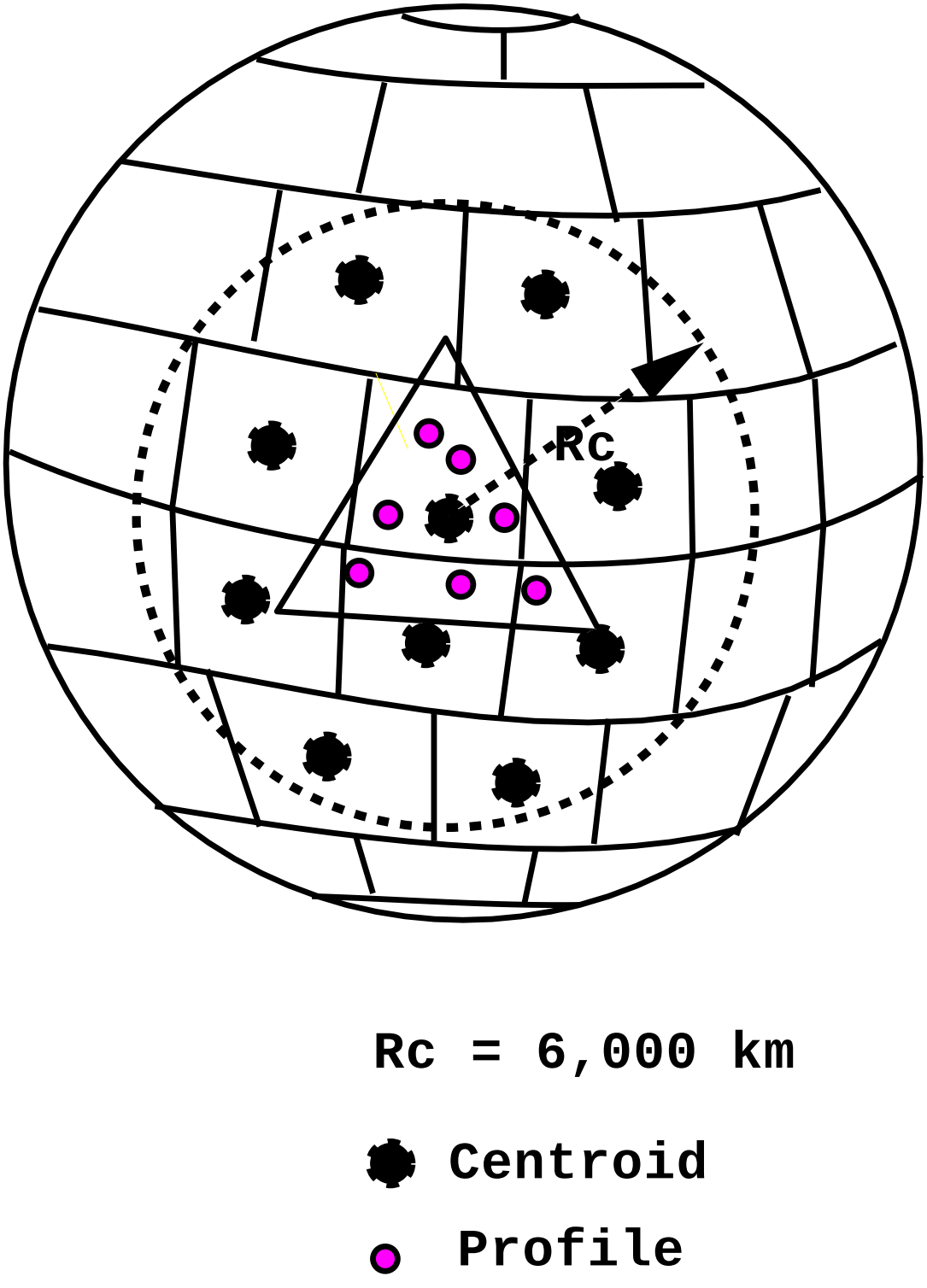

Figure 5: The load-balanced analysis grid decomposition. The triangle shows one of the regions of the parallel partitioner of observations. Blocks of $P^f H^T$ within 6,000 km of the centroid of this region are stored on the processor that has the corresponding vector fragment of x

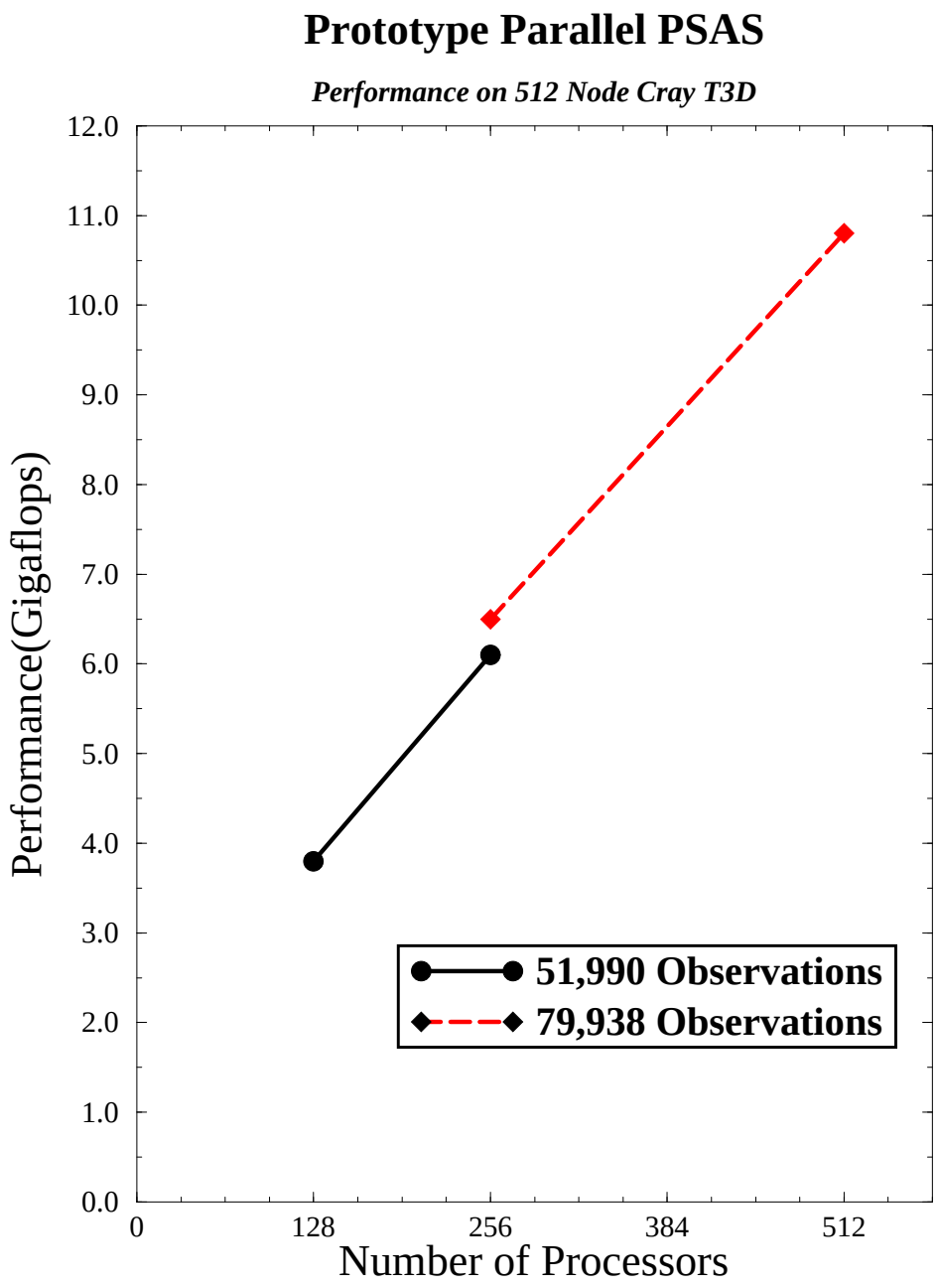

Figure 6: Performance (gigaflop/s) of the PSAS JPL on the Goddard 512 processor Cray T3D using the MPI message-passing library. The closed circles are for a problem with $51,990$ observations and the open circles are for $79,938$ observations.

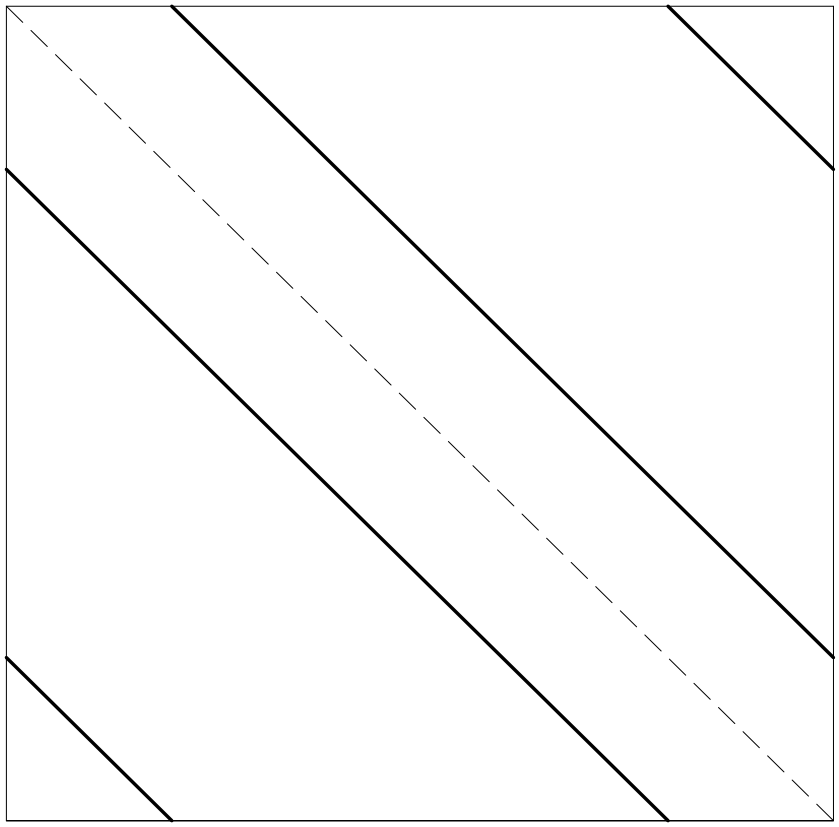

Figure 7: Every $I \times I$ block $C_{hh}(j,k,\varphi_1,\varphi_2)$ is a circulant symmetric matrix. The elements along its l^{th} and $(I-l)^{th}$ superdiagonals and subdiagonals are identical for every l, $0 \leq l \leq I-1$.