The Design and Evaluation of a High-Performance Earth Science Database

Carter T. Shock, Chialin Chang, Bongki Moon, Anurag Acharya,
Larry Davis, Joel Saltz, Alan Sussman
University of Maryland
Institute for Advanced Computer Studies and
Department of Computer Science
College Park, MD 20742
{ctso,chialin,bkmoon,acha,lsd,saltz,als}@cs.umd.edu

Abstract

Earth scientists have encountered two major obstacles in their attempts to use remotely sensed imagery to analyze the earth's land cover dynamics. First, the volume of data involved is very large and second, significant preprocessing is needed before the data can be used. This is particularly so for studies that analyze global trends using data sets that cover multiple years. In this paper, we present the design of an earth science database as well as our early experiences with it. The primary design goal of this database is to facilitate efficient access to and preprocessing of large volumes of satellite data.

Our initial design assumed that the main bottleneck in the system would be retrieving data from the disks. However, experimental results show that precise identification of all the data values corresponding to a query can take a significant amount of time. The problem is even more pronounced in designing the system to attempt to minimize time spent performing I/O. We therefore discuss a major redesign of the system that includes a reworking of the indexing scheme and a reorganization of the data on disks. Preliminary experimental results show that the redesigned system performs significantly better than the original system, providing interactive response times for local queries.

1 Introduction

Earth scientists have been thwarted by the staggering volume of remotely sensed data in their attempts to study the earth. For example, coarse-grained satellite data (about 4km per pixel) for ten days is about 7GB, while a finer-grained version of the same data (about 1km per pixel) is about 112GB. Therefore, earth scientists must use smaller post-processed products known as *Level 2* data. These Level 2 products are often projected composite images derived from many *Level 1*, or raw, data sets. A Level 2 full-globe composite image of the coarse-grained data just described is about 228MB. The result of this compositing, however, not only is that the original sensor radiometry is unrecoverable, but also that different Level 2 images must be re-projected into a common frame of reference for direct comparison. Access to the original sensor radiometry is important because a significant amount of earth science research is devoted to developing correlations between raw sensor readings from the satellite and various properties of the surface of the earth. Once composition operations are performed to produce Level 2 products, it is no longer possible to retrieve the original data. As for the projection problem, the process of generating any map-based Level 2 image requires projection of the globe onto a two-dimensional grid, resulting in spatial distortion of the data. Many different projections are used for various purposes by earth scientists; no single projected product is adequate for all potential uses. We therefore address the problem of providing a database of Level 1 data capable of producing unprojected remotely sensed images, while preserving the integrity of the original radiometry used to create these images [12]. Other current systems, such as Sequoia2000[20, 21], Paradise[5] and ARC/INFO [22] index Level 2 products and therefore do not address these issues. Such systems would be suitable for storing the Level 2 output images produced by our system.

These requirements present two major challenges for the design and implementation of a high-performance remote-sensing database. First, the database must provide low-latency retrieval of very large volumes of spatio-temporal data from secondary storage. The system described in this paper provides this capability by declustering and placement of the multi-dimensional spatio-temporal dataset onto a disk farm spread across the multiple processors of a distributed memory parallel machine, with each processor connected to multiple unshared disks. Such a configuration is suitable for providing the high I/O bandwidth necessary for fast retrieval of large volumes of data. Second, the large reduction in size between Level 1 and Level 2 data makes it imperative, to achieve good performance, that the composition operations be performed on the same machine where the data is stored. This requires careful coordination of computation, data retrieval from disk and communication between processors on the parallel machine to provide the best performance from the database.

This paper presents a database design for random spatio-temporal access into Level 1 remotely sensed data. The database is global in scope and has been implemented to run in a parallel environment. §2 discusses the design of the initial version of the database, including a discussion of the relevant characteristics of the satellite data. §3 presents the results of several experiments using the database to generate remotely sensed image products, to quantify the performance of the system. In §4 we discuss some of the weaknesses of the original design that were exposed by the experiments, and in §5 we describe a major redesign of the system and experimental results demonstrating significant performance improvements. Finally, in §6, we discuss our plans to extend and improve the database system.

2 System Design

We have designed a high performance global database for storing Level 1B remotely sensed data. Level 1B refers to data that has had sensor and satellite hardware corrections applied to the raw sensor measurements and has also been navigated (i.e. the location of each sensor measurement on the surface of the earth has been determined and stored). In addition, Level 1B data is supplied with *quality indicators* that describe the state of the sensor and satellite at the time the measurement was taken. This allows processing algorithms to reject unsatisfactory data. The current implementation stores *Advanced Very High Resolution Radiometer Global Area Coverage* (AVHRR GAC) Level 1B data. The following sections describe AVHRR GAC data, the class of queries that the system is designed to answer, and the data structures and retrieval algorithms for generating imagery from this data.

2.1 AVHRR GAC Data

AVHRR is a thermal sensor mounted on the NOAA series of satellites. As it scans the surface of the earth in a polar orbit, the AVHRR sensor measures and records data for five thermal bands. Each measurement is referred to as an *instantaneous field of view* (IFOV) and represents the thermal reflectance for a region on the surface of the earth. GAC IFOVs directly beneath the orbital path of the sensor (*on-nadir*) are approximately a circle with a diameter of four kilometers. Off-nadir GAC IFOVs are ellipsoids whose major axis increases with the distance from nadir.

The AVHRR sensor uses a rotating mirror to scan perpendicular to its orbital track. Each rotation of the mirror yields 409 IFOVs and is referred to as a *scan line*. An AVHRR GAC Level 1B data set is an accumulation of 110 minutes of scanning, or approximately 12,000 scan lines. Each scan line record in a GAC Level 1B data set includes five bands of thermal data for each of the 409 IFOVs, 51 latitude/longitude pairs for locating the IFOVs on the geoid, 51 solar zenith angles, and various other data quality indicators for the scan line. Each GAC Level 1B data file represents just over a single orbit of the sensor and also includes meta-data for the collection of scan lines, such as temporal bounds, sensor characteristics, etc.

2.2 Queries

A *query* into the database must specify information about both the input data to search and the output image format. Specification of the input data includes both the sensor(s) of interest and the temporal and spatial bounds of the data to be retrieved. The sensor(s) of interest is(are) selected from the set of sensors that have data stored in the database. The temporal bounds of the query are provided as a start and stop time specified in Universal Coordinated Time (UTC). Only data recorded between the start and stop times are used to answer the query. The spatial bounds of the query describe the user's *area of interest*, and are specified as a polygon on the geoid. Only IFOVs incident on this region are included in the result of the query.

The result of a query is one of two *types* of image. For simplicity we label the two types as *multi-attribute* images and *list* images. For both image types, the database generates pixel values in the resultant images by retrieving all of the IFOVs incident on that pixel that satisfy the spatio-temporal bounds of the query.

Multi-attribute images are those in which each pixel has several values associated with it. For example, AVHRR data consists of five separate channels. A multi-valued image for this sensor type might include a value for each of the five channels for each pixel. Attribute values are determined from the set of IFOVs contained in a pixel via a composition function supplied by the user as part of the query (e.g. `min`, `max`). The composition function should be both commutative and associative, since the system will determine the order that composition occurs on the IFOVs for each pixel (effectively based on the order that they are available to be processed). *List* images are the simplest, and potentially the largest, image type. Each pixel in a list image contains the entire set of IFOVs incident on that pixel.

Both image types specify a *grid* parameter. A grid is an equal angle subdivision of the geoid. It is specified with an origin point and an angular extent for the width and height of a cell in the grid (they need not be equal). A cell in the grid describes the spatial extent of a pixel in an output image. The origin of an image is a simple offset into this grid. The extent of an image is an offset specified by the number of grid cells.

The system provides multiple image types to give users flexibility and control over creation of the output image. First, all images conform to user specified grids. Two images with the same grid parameters are co-registered[1], greatly simplifying multi-sensor analysis. The system always preserves traceability. Queries are repeatable given identical grid parameters. A user is able to specify an output grid that contains exactly one pixel, and, by choosing a *list* image type, retrieve the set of IFOVs incident on that pixel. The system allows control of mapping errors. The grid used for a query is not necessarily linked to any projection. Therefore a user can choose to *display* the data using any projection desired, after the output image is returned.

2.3 Indexing Scheme

An AVHRR GAC Level 1B data set can be represented as an approximately 6x409x12000 array. The three dimensions of the array correspond to:

1. 5 bands and a geographic location for each IFOV,
2. 409 IFOVs for each scan line, and
3. approximately 12,000 scan lines per data set.

Unfortunately, a simple array is not sufficient to capture the spatial attributes of the data for several reasons. First, the distance between IFOVs is not uniform. Second, the distance between scan lines varies due to eccentricities in the sensor's orbit. Finally, the distance between IFOVs in a single scan line varies both with respect to the IFOVs' distance from nadir and from eccentricities in the sensor's orbit.

We considered spatial structures such as R-trees [13] and quadtrees [16] for indexing the data. Goodchild et. al. [11] suggest a hierarchical tessellation of triangles over the surface of the globe as a global spatial index. We determined that the use of a global index for all IFOVs would require replication of large amounts of meta-data for each IFOV. Furthermore, a global index whose atoms are IFOVs could be prohibitively large. Therefore some method for aggregating IFOVs was required. In order to minimize data growth from replication of meta-data and to provide the greatest flexibility, we decided to use a hierarchical indexing scheme based on separate *global* and *local* indices.

2.3.1 The Global Index

The global index is a coarse-grained spatio-temporal index of the local indices. Each entry in the global index consists of:

- spatial bounds for a local index,
- temporal bounds for a local index,
- coarse meta-data such as sensor type and number, and

[1]This co-registration is accurate only to within the spatial accuracy of the level 1B data

- a pointer to a local index.

From the perspective of the global index, a local index is an encapsulated object. A query into the global index only needs to determine if the spatio-temporal bounds of the query intersect a particular local index, so that the query can then be passed to the local index for resolution. By implementing local indices as objects, and by keeping the global index general we can avoid problems associated with global spherical approaches. The two-level design has several advantages. First, the object-oriented design allows the global index implementation to ignore individual data characteristics, which are encapsulated in the local indices. Second, the use of a global index allows for efficient indexing on attributes other than space and time (e.g. sensor type). Third, the global index can support quick pruning of large portions of the data base when presented with localized queries. Fourth, the global index can allow query previews that enable users to quickly refine their queries, without forcing large volumes of data to be retrieved from disks [6]. Finally, the global index is extensible – it is easy to include data from other sensors without re-engineering the indexing scheme or re-indexing existing data.

Because local indices are encapsulated objects, the global index implementation can be quite straightforward, mainly depending on the desired scope of the entire system. The initial implementation indexes only a handful of local indices, so a simple array that can be searched sequentially was used. The primary results presented in this paper relate to performance within local indices, so we will not discuss the global index further.

2.3.2 The Local Index

Local indices provide fine grained spatio-temporal indexing into AVHRR GAC Level 1B data. Recall that AVHRR GAC data is essentially an array. While the distance between IFOVs is not uniform, adjacency in the array does correspond to adjacency on the surface of the earth. Therefore, a good local indexing scheme should preserve the innate array qualities of the data while providing spatial indexing into that array. If we preserve the array, data selection using temporal criteria is straightforward. Meta-data provides the temporal bounds for an entire data file, and it is also possible to calculate a time for each scan line. However, spatial indexing is significantly more complex.

Before discussing the spatial indexing scheme for local indices, we discuss several preprocessing issues. The first preprocessing step is to divide the raw Level 1B file into seven files: one file for meta-data (mainly geo-location data), one file for geographic locations of IFOVs and five files for the individual band data. This approach reduces I/O overhead for queries where only one or two sensor bands must be retrieved. The file containing geographic data is an implicit key to the band data files. One latitude/longitude pair is stored for each IFOV, so that the file offset to the geographic location of an IFOV can be used to find the offset to the IFOVs sensor values in the band files.

The minimum granularity in the local index is a 16x16 block of 256 adjacent IFOVs. This approach yields 25 adjacent blocks of IFOVs for every 16 scan lines. We found that far off-nadir IFOVs are rarely, if ever, used in real analysis. For ease of implementation, the system does not index the nine IFOVs at the far ends of each scan line. We chose to index blocks of IFOVs to reduce the overall size of the local indices.

The index is a hierarchy of binary trees whose keys are quadrilaterals. In this paper we will refer to these as *block trees*. As with R-trees, the key for each interior node must be a quadrilateral that wholly contains the keys of all its children. To construct the index, a bounding quadrilateral is derived from the four corner IFOVs for each 16x16 block. Each **row** of 25 blocks (comprising 16 scan lines) serves as the leaves of a block tree. These are *horizontal trees*. The interior nodes of the tree are generated by finding the minimum bounding quadrilateral for both of the node's children. The root of each block tree is a minimum bounding quadrilateral for the entire tree. A single *vertical tree* can then be built for the file, with its leaves being the root nodes of the *horizontal trees*. The root node of the *vertical tree* is a bounding quadrilateral for the entire local index and is used to determine the spatial extent of the local index for use in the global index. The structure of a local index is shown in Figure 1.

2.3.3 Local Index Optimizations.

Spatial indexing efficiency is determined by how well the bounding quadrilaterals approximate the spatial extent of the underlying data. We refer to this quality as *fit*. The probability of a query into a local index returning no data is inversely proportional to the fit of that index's bounding quadrilateral. In other words, given a poor fit, it is more likely that a query will intersect the bounding quadrilateral, but *not* intersect any of the IFOVs within that bounding quadrilateral. Several optimizations strive to achieve good fit at low computational cost.

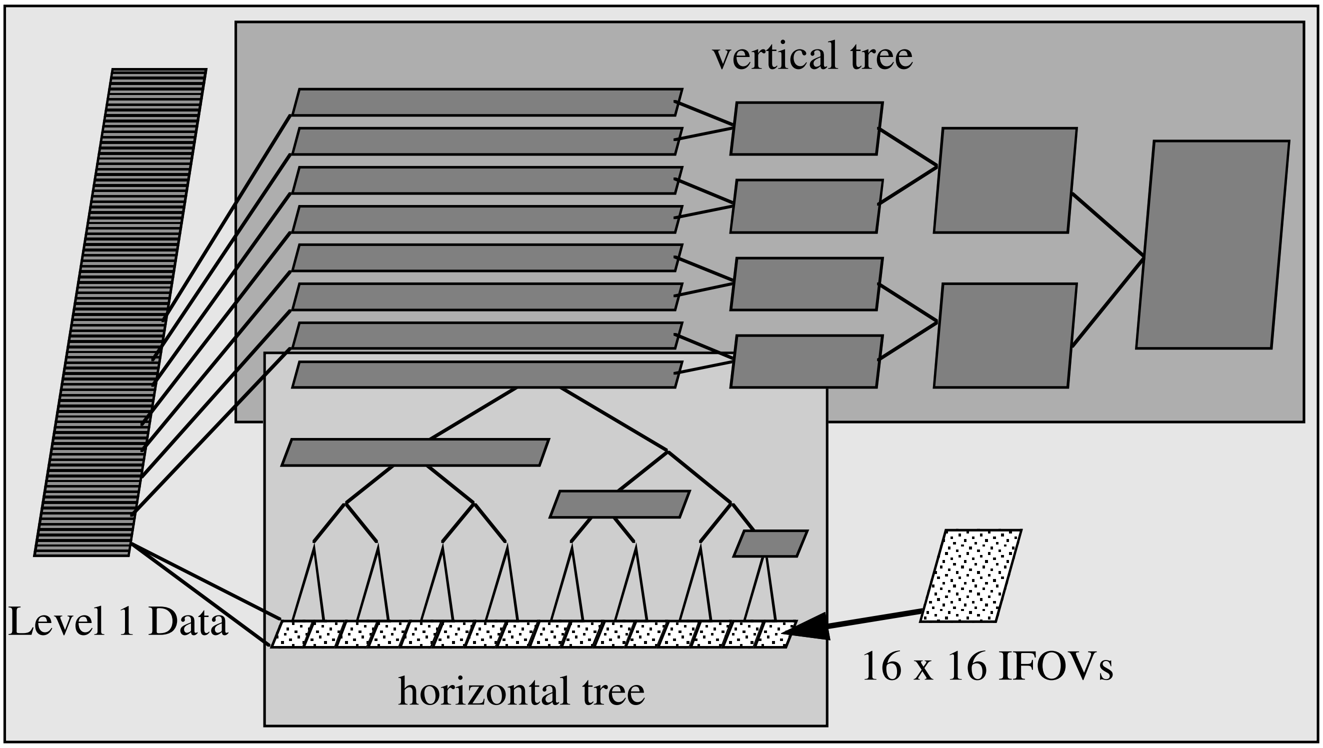

Figure 1: Hierarchy of block trees

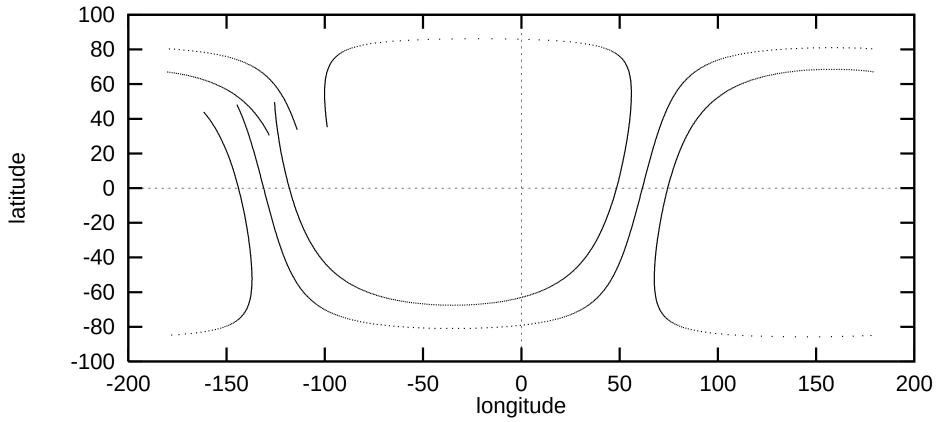

Figure 2: IFOVs in a Level 1B File

Local indices can be generated for entire Level 1B files, or for any portion of a Level 1B file. By partitioning AVHRR GAC Level 1B data files into several local indices it is possible to achieve a much better fit. Figure 2 shows a plot of left-most, right-most and center IFOVs for every sixteenth scan line in an entire AVHRR GAC Level 1B data file. Generating a bounding quadrilateral for this data is difficult due to the extreme curvature of the data near the poles and the range of the data extending over one orbit. By dividing the original file into eight separate files, each representing approximately 1/8 of an orbit, bounding quadrilaterals are easier to generate and yield a much closer fit to the data. Simple division of the file does not, however, address the severe curvature in the data encountered at the earth's poles.

Figure 3 is a plot of the bounding quadrilaterals for 16x16 IFOV blocks from a portion of an AVHRR GAC Level 1B file. To achieve a better fit, it would help to "square up" the shape of the data. The problem occurs because spherical angular data (latitudes and longitudes) is being projected into a planar frame of reference. The solution we employ is to project the bounding quadrilaterals using a *Lambert Conformal Conic Projection* [19]. Figure 4 shows the same blocks from Figure 3 projected using a Lambert projection. Projecting the index yields a much better fit. Projecting the index does, however, require the system to project a *query* into the same coordinate system as the index, but experiments have shown that the computational cost of projecting queries is significantly less than the cost of unnecessary disk accesses from null queries. It is important to note that only the four corners of its bounding quadrilateral of the index are projected, not the underlying sensor data. The original latitude and longitude of each IFOV is preserved. Figure 3 is displayed in a so-called equal angle projection, with a scale in degrees of latitude/longitude. On the other hand, the scale in Figure 4 is in kilometers; the Lambert projection preserves area, not angle.

2.4 Resolving a Query.

When a query (as defined in §2.2) is received by the system, an empty output image is created based on the spatial extent and grid parameters of the query. This output image and the query are then passed to the global index. The global index determines which, if any, of the local indices intersect the query, and passes the query and the intersecting portions of the output image to the corresponding local indices.

Each local index projects the query and output image into its local frame of reference and then searches the *block trees* for all 16x16 blocks of IFOVs that intersect the query space. As the query descends the trees, the output image is pruned to just those pixels that intersect the tree node's bounding quadrilateral. When the query reaches a leaf node in a *horizontal tree*, the 16x16 block of IFOVs referenced by the leaf node is retrieved from disk. Finally, the 256 IFOVs in the retrieved block are iterated through to determine which, if any, IFOVs are included in the output image.

2.5 High Performance Implementation.

The prototype for the system was developed using C++ in a sequential environment. However, AVHRR GAC Level 1B data volume is so large that efficient indexing methods are not enough to ensure timely processing

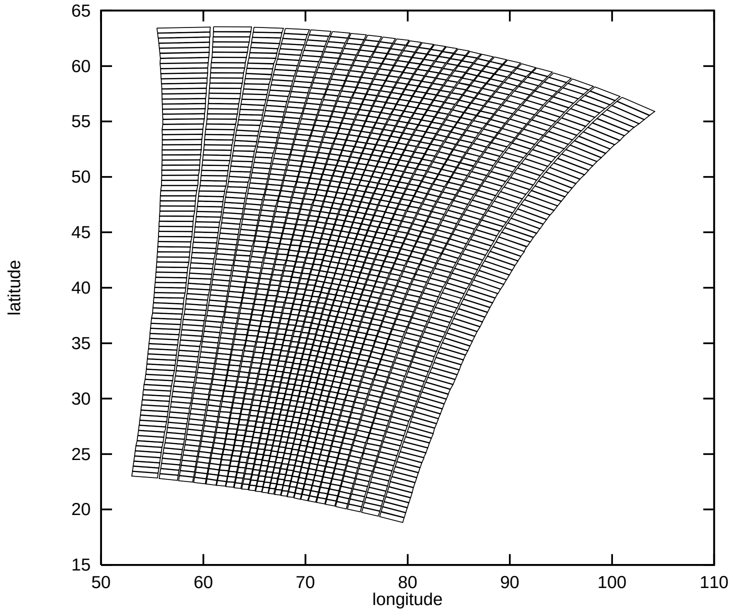

Figure 3: Unprojected 16x16 IFOV Blocks

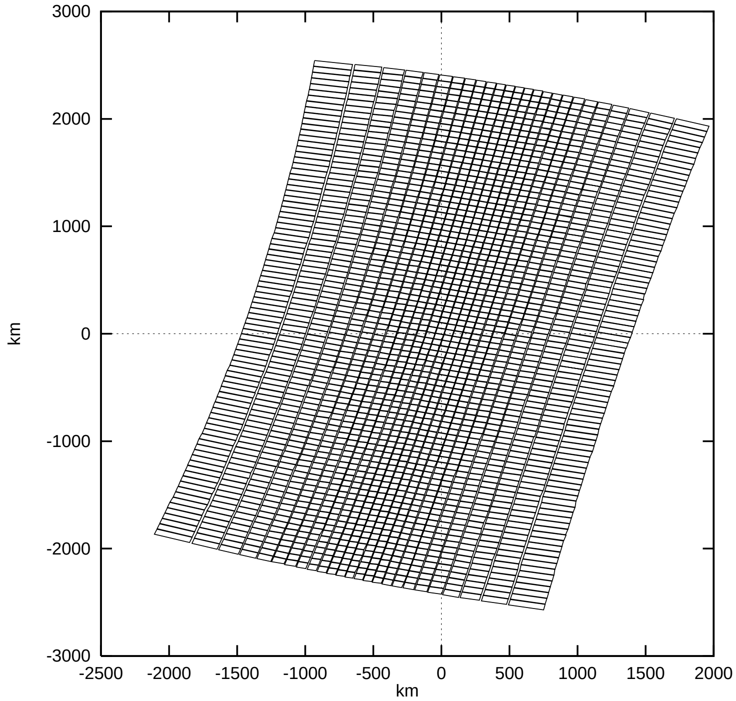

Figure 4: Lambert Projection of 16x16 IFOV Blocks

of queries. The system parallelization is based on the use of objects and high performance computing tools specifically designed to facilitate parallelization of sequential codes. In particular, we use the *CHAOS++* runtime library [3, 4].

CHAOS++ is a runtime library targeted at parallelizing object-oriented applications with dynamic communication patterns. *CHAOS++* provides support for both array and pointer-based data structures, and allows flexible and efficient data exchange of complex data objects among processors. *CHAOS++* is implemented as a C++ class library, and can be used directly by application programmers to parallelize applications with adaptive and/or irregular data access patterns. The design of the library is architecture independent and assumes no special support from C++ compilers. *CHAOS++* uses message passing as its transport layer and is implemented on several distributed memory machines, including the Intel Paragon and IBM SP-2, and on networks of workstations using either PVM[10] or MPI [7, 18].

Parallelization of the system is achieved by replicating the global index across all available processors, and partitioning an incoming query equally amongst available processors using the following scheme:

- The result of any given query is an image, as defined in §2.2. The extent of the image is defined in the query's *grid parameter*.
- A *sub-query* is generated for each available processor. The union of all of the sub-query grid parameters is the grid specified in the original query. All other query parameters are identical on all processors.
- Each sub-query is submitted to a processor for resolution. Each processor returns results independently to the querying process.
- Non-local I/O (to data stored on other processors) is done via NFS on the Ethernet connecting the SP2 processors.

In summary, the original query is broken down into multiple sub-queries, each of which is handled by one processor.

3 Performance Results

We designed experiments for the implementation of the two-level indexing scheme to answer several questions:

General Performance Given a query, how long does it take to get an answer? How much work is done by the system to achieve that answer?

Scalability How does the system perform on different numbers of processors and for different queries.

Partitioning Schemes What is the best way to divide a query among processors to achieve a balanced workload?

I/O Performance The sheer size of remotely sensed data sets suggests an I/O bound problem. We have attempted to quantify I/O load for the system to both prove this hypothesis and suggest possible solutions.

Preprocessing How long does it take to create the database from a set of raw data files? How difficult is it to add new data?

3.1 System Configuration

All experiments were performed using the parallel implementation running on the University of Maryland IBM SP2. The system contains sixteen RS6000/390 processing nodes (so-called **thin** nodes) running AIX version 3.2.5. The nodes are interconnected via the IBM high-performance communications switch, which provides each node with up to 40MB/second communication bandwidth. Each node of the SP2 has 12GB of online disk storage, and the disk configuration makes the 16 node machine capable of an aggregate peak I/O bandwidth exceeding 500 MB/second. The parallel implementation used IBM's MPL for message passing between processors.

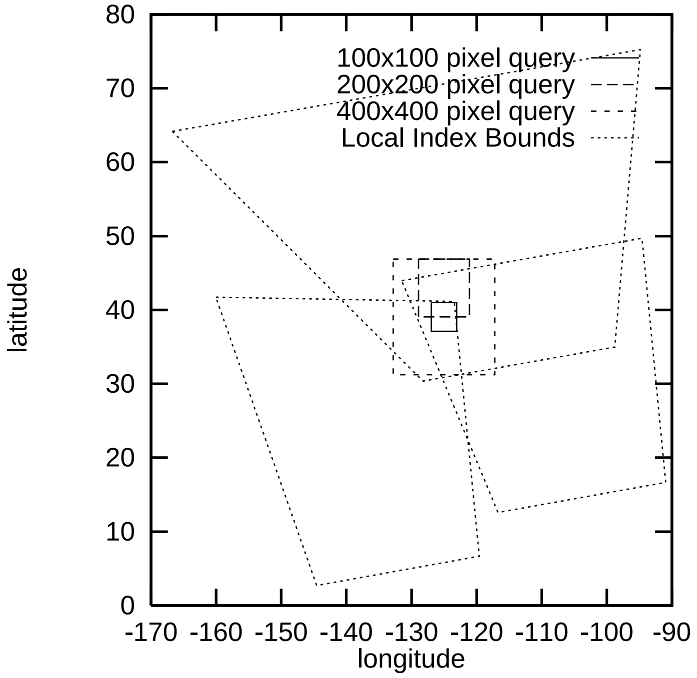

Figure 5: Query Bounds

3.2 Experimental Parameters

Each experiment was conducted on the prototype server loaded with 15 local indices covering the west coast of North America. By default, the database dynamically loads and caches a number of local indices (the *horizontal* and *vertical* trees), using a least recently used replacement scheme. For these experiments, we did not measure the time to load the local indices. Therefore, all 15 local indices were pre-loaded into the cache for each experiment on the database.

Figure 5 shows the image bounds for the three test queries used, and the bounds for the local indices that the images intersect.

The three queries were, with the exception of image bounds, identical. The parameters of the query (see §2.2) were:

- Temporal Bounds: unbounded
- Sensor Type: AVHRR GAC, bands 1 and 2
- Grid Parameters: 5x5 (explained below)
- Result Type: multi-attribute image

The minimum spatial resolution in AVHRR GAC data is 1/128 degrees of latitude or longitude. The grid parameters are therefore specified in increments of 1/128 of a degree. Therefore for these queries, each pixel in the output image was 5/128 degrees square, or approximately 4 kilometers (the same size as a single on-nadir AVHRR GAC IFOV). The spatial extent of the three queries was adjusted to yield 100x100, 200x200 and 400x400 pixel images. Queries were intentionally placed at the intersection of three local indices to measure the load balance characteristics of the system. The grid origin for all queries was 0 degrees latitude and longitude.

The parallel implementation of the database uses an *output partitioning* scheme to distribute the workload over multiple processors. Output partitioning divides the output image as evenly as possible along pixel boundaries based on the number of processors used by the system. Both the partitioning scheme and the queries were held constant for all experiments.

For each experiment we present results for:

IFOVs treated as	Points		Boxes	
I/O status	enabled	disabled	enabled	disabled
sum of all processor times (min:sec)	18:11	15:01	77:53	77:34
longest individual processor time (min:sec)	1:42	1:35	6:34	6:34
polygon-polygon tests (in millions)	6.38	6.38	31.04	31.04
line-line tests (in millions)	64.57	64.57	454.8	438.6
point-in-polygon tests (in millions)	47.30	47.30	145.6	145.9

Table 1: Statistics for 200x200 Pixel Image on 16 processors

- Time to complete a query (also including time spent in I/O tasks).
- Number of polygon-polygon intersection tests performed.
- Number of line-line intersection tests performed.
- Number of point-in-polygon intersection tests performed.
- Number of disk read() system calls.
- Number of bytes read in read() system calls.

For experiments run on multiple processors, separate statistics were collected for each processor. Statistics from all processors were used to generate total processor time, total number of intersection operations, earliest finish and latest finish. We present data on intersection routines because this is the critical code that performs computational geometry within the spatial index. Counts of intersection routine calls are a measure of work done and therefore an indicator of load balance. Finally, intersection calls are not independent. For example, the polygon-polygon intersection test may call both the line-line and point-in-polygon intersection routines.

3.3 Experimental Results

The first set of experiments attempts to determine the performance of the spatial indexing scheme in the absence of I/O. When the query process reached a leaf node in a local index's *horizontal tree*, read() and seek() system calls to retrieve real data were replaced with null spatial objects (points and quadrilaterals). The use of null spatial objects somewhat skews the results. Optimizations in the computational geometry code in the system allow for early exits from intersection tests. However, these results provide only a rough indicator of the time required to find the appropriate data in the spatial index. By comparing statistics for calls to intersection routines with I/O enabled, we can get an estimate of how much work was avoided by using null spatial objects. The test results show 200x200 pixel queries on from 1 to 16 processors. Additional results on 100x100 and 400x400 pixel queries are shown on enough combinations of processors to plot representative curves for these queries. We also present results with I/O enabled.

Table 1 compares execution time and work load for the 200x200 pixel query on 16 processors, both with and without I/O enabled. For all the measurements, enabling I/O did not significantly increase run times or work loads. This result was surprising, to say the least. The reason has to do with the cost of the algorithm used for mapping IFOVs into pixels in the output image. As was discussed in §2.3, once a 16x16 block of IFOVs is retrieved from disk, each pixel's bounds are intersected with each of the 256 IFOVs to determine which IFOVs in the block fall within each pixel. The first prototype implementation treated each IFOV as a simple point and used a point-in-polygon routine to test for intersection with a pixel. Unfortunately, it was possible to pose queries such that an individual pixel in the output image was smaller than the distance between adjacent IFOVs, which can occasionally yield blank pixels in the image. The solution implemented to solve this problem was to construct a bounding quadrilateral using the midpoint distances between an IFOV and its neighbors for each IFOV. The resulting quadrilaterals were then tested against the pixel using a polygon-polygon intersection routine. The more accurate scheme resulted in much more expensive query processing, as is shown in Table 1. The table shows workloads with IFOVs as both points and quadrilateral, for the 200x200 pixel query on 16 processors.

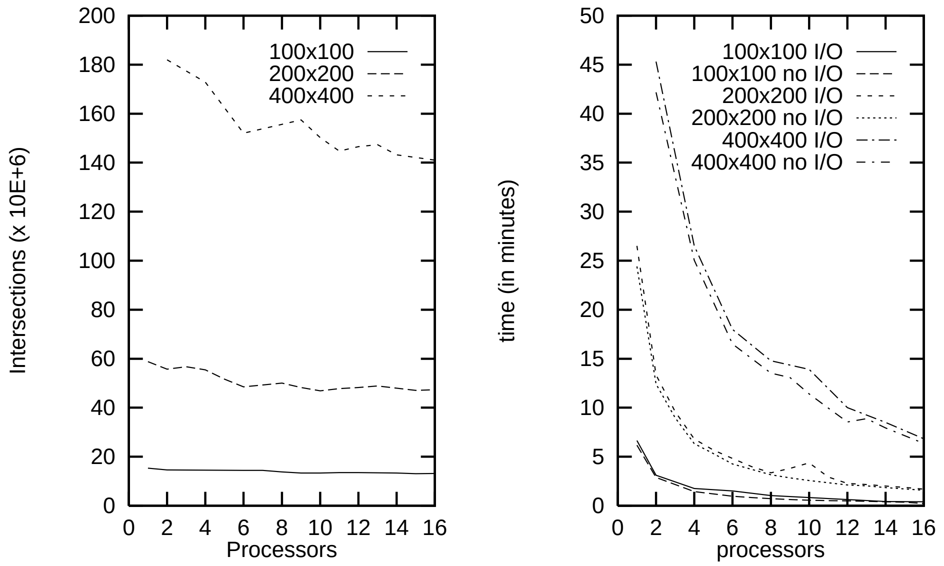

Figure 6: Line-Line Intersection vs. Query Size (left) and Query Completion Time (right).

Figure 6 compares work loads (expressed as number of spatial intersections performed) for the different query sizes, and also completion times for the different queries on different numbers of processors. Spatial intersection counts were identical between similar tests with and without I/O enabled. The results show that the total work load (the sum of all spatial intersections performed by all processors) varied somewhat as more processors were used to process a query, especially for queries with more pixels. In addition, individual processor work loads varied greatly, depending on how the query was partitioned. We also note that as pixel counts in output images quadrupled, work load and time to complete a query approximately tripled, suggesting that the indexing scheme becomes more efficient as the number of pixels in the output image grows.

Next we attempt to quantify the I/O behavior of the image database server. Figure 7 shows processing times for a 200x200 pixel query with and without I/O enabled. I/O appears to account for, on average, approximately 8% of the total processing time for a query. We note, however, that for larger numbers of processors I/O accounted for over one quarter of the time spent on a query.

Finally, we examine the issue of preprocessing. Building the database requires segmenting raw data files, building local indices for each segment, and inserting a description of each local index into the global index. For two full months AVHRR GAC data, or approximately 24GB after preprocessing, the entire preprocessing stage takes about 128 node-hours on the Maryland SP2 (the preprocessing is trivial to run in parallel, so can use as many nodes as are available). Broken down further, it takes about 10 minutes for one node to preprocess one AVHRR GAC level 1B file. This estimate assumes that all raw data files read, and all local indices written, are local to the node doing the preprocessing. Preprocessing data on remote disks over NFS takes significantly longer, since the preprocessing is an I/O intensive task. Each local index is an encapsulated object as described in §2.3. Therefore, inserting new data into the system is relatively quick and straightforward. Once the local index is constructed, a record describing the index is inserted into the global index. For efficiency, the global index can be rebuilt occasionally to maintain a *packed* structure.

4 Evaluation

The performance problems identified in the image database system, as presented in §3 are attributable to several factors. The first set of factors relates to the behavior of the two-level indexing scheme and the mapping of IFOVs into the output image. These factors lead to performance problems in the computations performed on each processor in the parallel implementation, so would also be problems for a strictly sequential implementation. The second set of factors relates to the behavior of the database system in a parallel execution environment. The parallel implementation of the original system performed a straightforward parallelization of an essentially

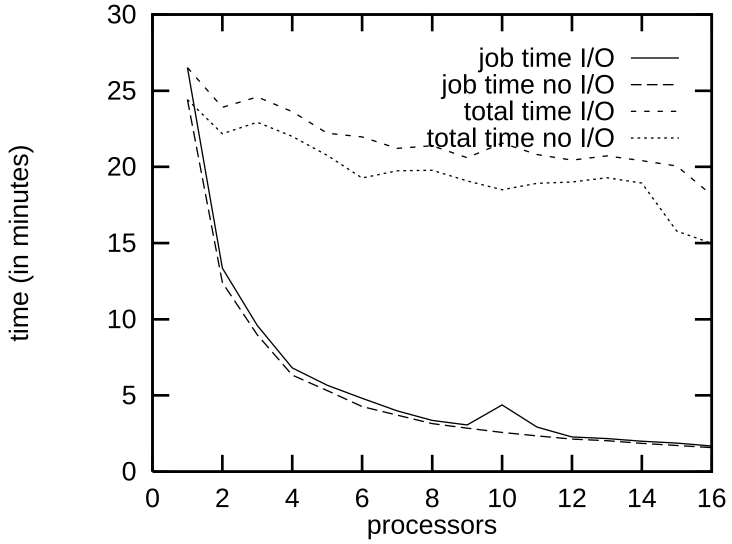

Figure 7: Processing times for 200x200 pixel query.

sequential design, and did not make a serious attempt to optimize computational load balance and I/O and communication performance. We now discuss these issues in more detail, before presenting a redesign of the system that addresses these issues in §5.

4.1 Index and Mapping Performance

For a given query, the image database system we have described performs, with respect to a given local index, what we call *forward mapping* to determine, for each pixel in an output image, which IFOVs map into the pixel. Unfortunately, forward mapping suffers from the non-uniformity of the spatial characteristics within an array of IFOVs, thereby requiring the system to effectively search for IFOVs. However, the result of any query is an image, as described in §2.2, and the *grid parameters* for the image define the spatial extent of any pixel in the image. The pixels of the image are spatially uniform. *Inverse mapping* takes advantage of this uniformity in the output image by determining, for each IFOV that maps into the output image, which pixel(s) contain that IFOV.

Given the geographic location of an IFOV, a simple computation yields the grid cell (or pixel) that contains the IFOV. If the grid is extended past the edges of the actual output image, the same computation defines where the IFOV falls with respect to the entire image (inside, above, below, to the left or right). This information can be used to implement a simple quadrature technique to map entire local indices to images. The basic idea is to map the four *corner* IFOVs of a local index into the output image. If all four corner IFOVs fall inside the image, all the IFOVs in the local index are mapped into the output image. If any corner is outside the image, the local index is partitioned into four approximately equal-sized partial indices and the same procedure can be performed recursively, quickly rejecting any portions of the local index that are wholly outside the image.

Inverse mapping provides two significant advantages compared to forward mapping. First, searching in local indices is unnecessary, as is the requirement to build hierarchical data structures for local indices. In short, a local index becomes a simple array of IFOVs. Second, output image resolution no longer affects the time to process a query. Higher image resolution creates more pixels for a fixed spatial extent of a query, and therefore for the output image to be produced. Forward mapping requires a local index search for each *pixel* in the image. Inverse mapping instead requires a mapping operation for each *IFOV* incident on the image, making the time to process a query resolved using this method independent of the number of output pixels.

4.2 Load Balancing and I/O Performance

The design of the system must take into account both load balancing and I/O and communication performance to effectively use a parallel machine. Load balance problems in the system, as shown in Table 1, are quite significant, as can be seen from the difference between the average time for a processor to complete the query vs. the time for the last processor to complete the query. As was noted in §3, as much as a quarter of the time to resolve a query was spent on I/O for the queries tested. For queries with larger output images, I/O would become even more of a bottleneck. Related to the I/O bottleneck is the issue of interprocessor communication. For the system to make effective use of available disk bandwidth in the parallel machine, the high performance internal network of the machine should be used to move blocks of data from the processor that has the disk containing the required data to the processor requesting the data.

These considerations have led to another *output partitioning* scheme for distributing queries. The first step in improving I/O performance is to intelligently distribute data blocks (local indices) across the disks available on all processors (*declustering*). Data sets within a processing node can also be arranged on disk for efficient spatial searching (*clustering*). Queries are partitioned as in the original system, with each processor assigned to produce a contiguous portion of the output image. However, I/O for a given data block is always done by the processor connected to the disk containing the block, which then uses interprocessor communication to forward the block to each processor that maps any part of the block into its portion of the output image. The new scheme should result in more efficient I/O because all disk accesses are local (rather than through the network file system). Load balance should also improve, especially for large output images, since a good declustering should allow any query (that is big enough) to access approximately equal amounts of data from the disks on all processors. In addition, the three parts of the new scheme, namely I/O, interprocessor communication and the computation to perform mapping and compositing operations should be able to occur concurrently on each processor in the parallel machine.

The next section describes the redesigned system in more detail, and presents some preliminary performance results.

5 New System Design

This section focuses on three aspects of the redesign and implementation of the image database system on a distributed memory parallel machine: data placement on the disk farm, query partitioning and coordination of data retrieval, computation and communication over the entire machine. The data layout decisions for the new system were motivated by the format of AVHRR data and the common query patterns identified by NASA researchers and collaborators in the University of Maryland Geography Department. For declustering, we have adapted the *minimax* scheme, originally proposed by Moon et al. [14] for declustering spatial data structures such as grid files and R-trees on large disk farms. This scheme has been adapted to generate an effective declustering of the coarse-grained remote sensing data, as will be shown in §5.3. We also describe the mechanisms used to coordinate data retrieval, computation and communication over all the processors. The goal of these mechanisms is to overlap all three operations as much as possible while maintaining computation, communication and I/O balance over the entire machine.

5.1 New System Overview

The new system consists of two parts:

- a front end, to interact with querying clients, do initial query processing and partition data retrieval and computation; and
- a back end, to retrieve the data and perform the post-processing and composition operations.

The front end is a single machine which can be located anywhere on the network, though a configuration where it shares a dedicated network with the back end is preferred (even after data reduction from compositing, the result image is usually quite large). The back end consists of a set of nodes that store the data and do the computation. Given the volume of data to be transferred, it is preferable if these nodes share a dedicated network. The new system implementation uses one node of the 16-processor IBM SP2 as the front end and the remaining 15 nodes as the back end. There is no data stored on the disks of the front end node.

The new system partitions its sensor data into coarse-grained chunks, as did the original system. However, while AVHRR data is organized as one file per satellite orbit and the original system simply partitioned the file into several smaller ones, the new system utilizes information both about common access patterns and parallel system disk I/O properties to optimize access to the data. First, while the AVHRR files provided by NASA [2] are organized in a band-interleaved form (i.e., all the values for a single pixel are stored together), the satellite data processing programs that we are aware of process either bands one and two or bands three, four and five [8, 9, 15, 17]. To take advantage of these query patterns, the new system stores AVHRR data in two groups, one containing data for bands one and two and the other containing data for bands three, four and five. Second, AVHRR data is used both for local and global land cover queries. Using individual files as units of data storage and retrieval is likely to result in much more I/O than is required by the query. On the other hand, retrieving the data in very small units (e.g. individual scan lines) does not provide good I/O performance. A good compromise can be achieved by using the smallest data unit that can be retrieved efficiently from disk. Third, a factor that should to be taken into consideration while selecting the data layout is the geometry of individual data units – square groups of pixels provide better indexing and *fit* than elongated groups. From [1], we know that for the Maryland SP2 configuration, the best I/O performance is achieved for blocks larger than 128KB. The new system tiles the data in blocks of 204x204 pixels. For the tiles containing data from bands one and two, this totals about 187 KB; for tiles containing data from bands three, four and five, the tile size is about 270 KB.

These data blocks are analogous to the original system's local indices, and therefore become the atoms in the global index. Blocks are still treated as encapsulated objects for query processing, however the relatively large number of blocks to be stored requires a more robust global index than the simple array used in the original system. The new system uses a binary tree whose keys are quadrilaterals to index data blocks. The root node is a bounding quadrilateral for all data blocks in the tree. The structure is built from a list of data block bounds that has been sorted spatially. Sorting the leaves using a space filling curve [16] allows access to the global index as a range tree. This structure is similar in behavior to that of R-trees [13]. The entry for each data block contains spatial bounds, temporal bounds, meta-data such as sensor type and satellite number and the position of the data block on disk.

The global index is used by the front end to identify data blocks needed for resolving the given query, and to partition the work load for efficient parallel execution. The size of the global index for the new system, indexing about 24GB of Level 1B data, is 11.6MB, which is small enough for the front end to store in its main memory.

Queries into the database are the same as for the original system, consisting of four kinds of constraints:

- temporal bounds,
- spatial bounds - a quadrilateral on the geoid,
- sensor name and type, and
- resolution of the grid for the output image.

As for the original system, the result of a query is a composited multi-band image. The value of each pixel in the result image is generated by composition over all of the IFOVs corresponding to that pixel. The new system currently supports `max` and `min` composition functions, but could be extended to accept a user-defined function that is both commutative and associative, as described in §2.2.

Once it receives a query, the front end searches the index for data blocks that intersect with the query window. The result of this search returns a list of requests for data blocks. Using the position information stored in the index, the front end partitions the list of data blocks among the back end nodes – the list corresponding to each node contains the data blocks stored on that node. In addition, the front end partitions the result image among all back end nodes. It then distributes this information to all back end nodes.

When a back end node receives this information, it computes a communication schedule. With a schedule, a back end node is able to move data blocks residing on its local disks to all back end nodes that need the data blocks, to resolve their parts of the query in a pipelined fashion. Each back end node can thus process data blocks as they arrive. More details are provided in § 5.2.

Once a data block is available (either retrieved from local disk or forwarded by another node), the simple quadrature scheme described in §4.1 is used to search for IFOVs that intersect with the query window. The composition operation can then be applied to each IFOV that maps into an output pixel.

Finally, after all data blocks have been processed, the result image can be either returned to the front end for forwarding to the querying client, or stored in a file.

while (not done)

- issue as many asynchronous disk reads for blocks as possible
- check all pending block sends, freeing send buffer for completed ones
- check pending block receives, and for each completed one:
  - add the receive buffer to the list of buffers that must be processed locally
  - **if** more non-local blocks must be obtained, issue another asynchronous receive
- check pending disk reads that retrieve blocks to be processed only by remote compute nodes, and for each completed one generate asynchronous sends to the remote nodes
- Process a block:

  if a block is already available for processing

  - process the block - perform inverse mapping and compositing operations for all IFOVs in the block

  else

  - check pending disk reads for blocks used by the local compute node, and possibly by remote compute nodes as well

    for the first completed read found (if any):

    * generate asynchronous sends to the remote nodes
    * process the block
    * go to the top of the loop

Figure 8: Main loop for overlapping computation, I/O and communication

5.2 Overlapping I/O, communication and computation

The implementation of the new system is motivated by the observation that to answer a query under the data declustering and query partitioning schemes, a significant amount of data is read from disks and transferred across the network. To hide the enormous latency incurred from I/O accesses and communication, back end nodes issue multiple asynchronous operations to both the file system and the network interface, so that I/O, communication, and computation are all overlapped. By keeping track of various pending operations, and issuing more asynchronous operations when necessary, the back end nodes can move data blocks from their disks to the memory of the consuming back end nodes in a pipelined fashion. In addition, while I/O and communication are both happening, each back end node can process data blocks as they arrive from either its own local disks or the network.

Once a schedule is generated, each back end node asynchronously executes a loop whose body consists of four phases – a disk read phase, a block send phase, a block receive phase, and a block consume phase – as shown in Figure 8.

During the disk read phase, a back end node issues as many asynchronous reads to its disks as possible. An asynchronous read requires pre-allocated buffer space to hold the returned data, and a control block for the back end node to track the status of the asynchronous operation. The number of asynchronous reads that can be issued during each execution of this phase is thus limited by the number of available buffers and free control blocks. The disks also provide an upper bound on the number of outstanding requests that are allowed, since little or no benefit is gained from too many outstanding asynchronous reads per disk.

In the block send phase, the back end node checks for completion of asynchronous sends issued in the block consume phase, which will be described shortly. The sends are for blocks that reside on disks on this node, but are required for computation on other nodes. When all asynchronous sends for a data block complete, the buffer space for the block is released. In the block receive phase, the back end node posts multiple asynchronous receives

Query	Execution time (sec)	Total data read from disk - # blocks	Total data communicated - # messages	Data blocks read per node (min/max)	Data blocks processed per node (min/max)
Africa	5.9	1087 (203MB)	2005 (375MB)	66/79	117/151
Global	78.0	9263 (1732MB)	13138 (2457MB)	599/632	634/1095

Table 2: New system performance results

to receive data blocks from remote back end nodes, for blocks to be processed on that node but stored on other nodes. A buffer needs to be reserved for each pending receive.

All asynchronous reads and receives issued during the disk read phase and the block receive phase are checked for completion in the block consume phase. When an asynchronous disk read completes, the system issues an asynchronous send for each of the consumers of the block (except the local back end node). These asynchronous sends will be checked for completion during the block send phase. When an asynchronous receive completes, or when the local back end node is also a consumer for a completed read, processing of the data in a block occurs, as described earlier, consisting of eliminating portions of the block that lie outside the query region, mapping of each IFOV into an output pixel, and applying the composition function to the current values of the pixel and the values of the IFOV. If an asynchronous receive completes, and the back end node is still expecting more data blocks from remote back end nodes, another asynchronous receive is posted immediately. Buffer space for a processed data block consumed only by the local back end node is released. In the new system implementation, the block consume phase terminates either when none of the outstanding asynchronous operations completes, or as soon as one available data block is processed. This policy prevents the system from failing to monitor the various outstanding asynchronous operations, while many available data blocks are processed.

5.3 Preliminary results

We have produced some preliminary results for the new system, and they are quite promising. The current database stores approximately two months of AVHRR Level 1B GAC data, containing about 24GB of data stored on 60 disks distributed across 15 nodes of the SP2. The data was declustered as described earlier, resulting in a uniform distribution of blocks across the disks. The results shown are for queries that consist of a ten day compositing operation on band 1 and 2 data, on both the African continent and the entire globe. The results in Table 2 show that the new system performs quite well, performing an entire global query into a much larger data set in approximately the same amount of time (less than 1.5 minutes) as the original system took to perform a query into a much smaller data set. The ten-day composite for Africa took only six seconds, showing that local queries can be processed interactively, with reasonable response time. For the original system, the queries shown in §3 were into a data set consisting of 15 local indices (i.e. the global index had 15 leaf nodes), for a total of about 100MB of data, of which data from only three local indices were accessed by the query. On the other hand, the two queries shown in Table 2 were into a data set with a global index containing over 55,000 entries (or about 111,000 data blocks, since each global index entry indexes two blocks, one for sensor bands 1 and 2 and one for bands 3, 4 and 5). The global query accessed about 1/6th of the total data stored, or about 4GB.

As for building the database, with the need to construct *local* hierarchical structures eliminated, preprocessing is greatly simplified. Preprocessing includes segmenting raw AVHRR GAC files into blocks, building a global index from the bounds of those blocks, determining the location of blocks onto disks using the minimax declustering algorithm, and writing the blocks to the appropriate locations on disk. For the sample 24GB data set, all processing takes about four hours on the SP2. Raw file segmenting, global index construction and running the declustering algorithm are tasks that were performed using a single node of the Maryland SP2. Of these tasks, the majority of the time was spent in the declustering algorithm, which took about three hours to run. The final step that moves data from the raw AVHRR GAC files to blocks in disk locations specified by the declustering algorithm takes about five node-hours. In other words, each node in the server takes about twenty minutes to find, process, and write out the blocks that have been assigned to its disks.

The results show that, on average, each data block retrieved from disk on one back end node is sent to about two other nodes for the local Africa query, and about 1.5 other nodes for the global query. This is because of the relatively large extent of AVHRR data sets, even after they are partitioned for use in the new system (a scan line at the equator is about 1700km wide). The declustering algorithm also works well, as shown by the good load balance across back end nodes shown for retrieving data blocks from disk. On the other hand, the processing load is not evenly balanced, even for the global query. We are still investigating the precise reasons for the load imbalance for these queries, but in general, an equal partitioning of the output image across the processors does not correspond to an equal partitioning of the input data to be processed on each processor.

6 Conclusions and Future Work

We have presented the design and evaluation of two versions of a high performance image database for efficiently accessing remotely sensed data with spatial non-uniformity. While the original system addressed many of the problems inherent in accessing data with non-uniform spatial characteristics, the design did not make effective use of the resources available in a parallel machine with multiple disks on every node. As was shown by the experimental results, the original system had significant performance bottlenecks due to both the image production algorithm and a naive parallelization strategy.

The second version of the system addresses many of the weaknesses of the first version. The new system takes advantage of uniformity in the output image to efficiently perform queries using *inverse mapping*. In addition, the new system addresses load balancing, I/O and communication issues in the design of a complete parallel system. Preliminary results show that the new system both provides reasonable load balance across processors and significantly speeds up image production relative to the original system.

One additional problem with the new system that is not apparent from the results presented in §5.3 is that communication of blocks between the nodes that store them and the nodes that process them for a particular query is a major performance issue. The problem stems from a conflict between the declustering strategy for partitioning the blocks of satellite data across processors and the strategy for partitioning the workload, which is based on a straightforward partitioning of the pixels in the output image. These choices force almost all of the blocks a processor requires to compute its portion of the output image to be retrieved from disks on other processors. This requires large amounts of interprocessor communication. The problem is amplified on the IBM SP2 where the experiments were run. Although the algorithm shown in Figure 8 should allow for overlap between communication and other activities (namely I/O and computation), little overlap is seen in practice.

We are currently working on a new scheme for partitioning workload based on the declustering of the input image. The proposed scheme has each processor work on all blocks retrieved from a local disk in response to a query. This effectively requires the output image to be replicated, and then combined across all processors in a final step after all blocks are processed. If the output image is too large to be replicated in the memories of all processors, the output image can be partitioned and produced one piece at a time, with all processors working on the same part of the output image at any given time. This scheme should significantly reduce interprocessor communication; the only communication required is to combine the results in the output image. In addition, such an *input partitioning* scheme should also improve computational load balance, assuming input data is uniformly distributed across the spatial extent of a query, since the workload of each processor will be directly proportional to the amount of data that is stored on local disk. Since the input data declustering works well, as was shown in Table 2, this scheme should produce good computational load balance.

We are also currently investigating techniques for efficiently handling multiple concurrent queries into the image database. Multiple users should be able to access the system simultaneously, so some method for handling multiple queries must be developed. The issues that must be addressed include resource management and data reuse. Resource management issues arise from trying to optimize use of the limited amount of buffering space available on a processing node for various activities, mainly consisting of I/O, interprocessor communication and local computation. Data reuse refers to scheduling processing of queries that overlap in space and/or time. Since retrieving a data block from disk is an expensive operation, once a block is retrieved it should be used for all available queries that intersect the block. We are investigating both off-line algorithms, where the entire set of queries is known ahead of time, and on-line algorithms, where the queries arrive at random intervals.

We are also in the process of building an integrated system for processing the output of the database. The main difficulty lies in providing efficient communication between the various steps in a processing chain, to eventually produce a high-level data product, such as a complete classification of all the pixels in a composite image produced by the database. Since each of the programs in the processing chain may be a parallel program, and the image database is also a parallel program, coordination and communication can be a major performance bottleneck.

References

[1] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. Hollingsworth, J. Saltz, and A. Sussman. Tuning the performance of I/O-intensive parallel applications. In *To appear in the Proceedings of the Fourth ACM Workshop on I/O in Parallel and Distributed Systems*, May 1996.

[2] Web site for AVHRR data from the NASA Goddard distributed active archive center. *http://daac.gsfc.nasa.gov/CAMPAIGN_DOCS/FTP_SITE/readmes/pal.html.*

[3] Chialin Chang, Alan Sussman, and Joel Saltz. Support for distributed dynamic data structures in C++. Technical Report CS-TR-3416 and UMIACS-TR-95-19, University of Maryland, Department of Computer Science and UMIACS, January 1995.

[4] Chialin Chang, Alan Sussman, and Joel Saltz. CHAOS++: A runtime library for supporting distributed dynamic data structures. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*. MIT Press, 1996. To appear.

[5] David J. DeWitt, Navin Kabra, Jun Luo, Jignesh M. Patel, and Jie-Bing Yu. Client–server Paradise. In *Proceedings of the 20th VLDB Conference*, 1994.

[6] K. Doan, C. Plaisant, and B. Shneiderman. Query previews in networked information systems. Technical Report CS-TR-3524, Department of Computer Science, University of Maryland, Oct 1995. Also available as CAR-TR-788, ISR-TR-95-90.

[7] Jack J. Dongarra, Steve W. Otto, Marc Snir, and David Walker. A message passing standard for MPP and workstations. *Communications of the ACM*, 39(7):84–90, July 1996.

[8] Jeff Eidenshink and Jim Fenno. Source code for LAS, ADAPS and XID, 1995. Eros Data Center, Sioux Falls.

[9] Gene Feldman. Source code for the SeaWIFS ocean data processing system, 1995. SeaWIFS group (NASA Goddard).

[10] Al Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[11] Michael F. Goodchild, Yang Shiren, and Geoffrey Dutton. Spatial data representation and basic operations for a triangular heirarchical data structure. Technical Report 91-8, National Center for Geographic Information and Analysis, University of California at Santa Barbara, April 1991.

[12] S. N. Goward and J. Townsend et. al. Toward rational global-scale remote sensing databases, March 1994. Presented at Second Inter-Pathfinder Conference, Washington, D.C.

[13] Antonin Guttman. R-Trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM-SIGMOD Conference*, pages 47–57, Boston, MA, June 1984.

[14] Bongki Moon, Anurag Acharya, and Joel Saltz. Study of scalable declustering algorithms for parallel grid files. In *Proceedings of the Tenth International Parallel Processing Symposium*, pages 434–440, Honolulu, Hawaii, April 1996. Extended version is available as CS-TR-3589 and UMIACS-TR-96-4.

[15] Nazmi El Saleous. Source code for the *gaps* system, 1995. GIMMS group (NASA Goddard).

[16] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison - Wesley, Reading, Massachusetts, 1990.

[17] Peter Smith and Bin-Bin Ding. Source code for the AVHRR Pathfinder system, 1995. Main program of the AVHRR Land Pathfinder effort (NASA Goddard).

[18] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. Scientific and Engineering Computation Series. MIT Press, 1996.

[19] John P. Snyder. *Map Projections - A Working Manual*. U. S. Geological Survey, 1987. U. S. Geological Survey Professional Paper 1395.

[20] Michael Stonebraker. Sequoia 2000: A reflection on the first three years. Technical Report S2K-94-58, EECS Dept., University of California, Berkeley, 1994.

[21] Michael Stonebraker, Jim Frew, Kenn Gardels, and Jeff Meredith. The Sequoia 2000 storage benchmark. In *Proceedings of the 1993 ACM SIGMOD Conference*, May 1993.

[22] The ARC/INFO Geographic Information System. S. morehouse. *Computers and Geosciences: An International Journal*, 18(4):435–41, August 1992.