A High Performance Image Database System for Remotely Sensed Imagery *

Carter T. Shock[1], Chialin Chang[12], Larry Davis[12],
Samuel Goward[3], Joel Saltz[12], Alan Sussman[12]
University of Maryland at College Park

[1] University of Maryland Institute for Advanced Computer Studies
[2] University of Maryland Department of Computer Science
[3] University of Maryland Department of Geography

Abstract. We present the design of and performance results for an image database system for remotely sensed imagery. The system stores and serves level 1B remotely sensed data, providing users with a flexible and efficient means for specifying and obtaining image-like products on either a global or a local scale. We have developed both parallel and sequential versions of the system; the parallel version uses the CHAOS++ library, developed at the University of Maryland as part of an NSF Grand Challenge project, to support parallel object oriented programming.

1 Introduction

Earth scientists have been thwarted by the staggering volume of remotely sensed data in their attempts to study the earth. Currently, they are forced to use the 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. As a result, not only is the original sensor radiometry unrecoverable, but different level 2 images must be re-projected into a common frame of reference for direct comparison. Our goal is to provide a database of level 1 data capable of producing unprojected remotely sensed images that conform to the requirements of individual users[5]. Systems like Sequoia2000[10, 11] and Client-Server Paradise[3] index level 2 products and therefore do not address this goal.

In this paper, we present a database design for indexing Level 1B remotely sensed data on a global scale using parallel methods. Then, we present the results of several test suites using this database to generate remotely sensed image products.

2 System Description

We have implemented a high performance global database for storing *Advanced Very High Resolution Radiometer Global Area Coverage* (AVHRR GAC) Level

* This research was supported by the National Science Foundation under Grant #ASC 9318183. The Maryland IBM SP2 used for the experiments was provided by NSF CISE Institutional Infrastructure Award #CDA9401151 and a grant from IBM.

1B data. The following sections describe AVHRR GAC data, the class of queries that the system is designed to answer, and our data structures and retrieval algorithms for indexing 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 IFOV's directly beneath the orbital path of the sensor (or "on nadir") are approximately a circle with a diameter of four kilometers. Off nadir GAC IFOV's are ellipsoids whose major axis increases with the distance from nadir.

The AVHRR sensor sweeps perpendicular to its orbital track. Each sweep of the detector yields 409 IFOV's and is referred to as a *scan line*. AVHRR GAC Level 1B data sets are 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 IFOV's, 51 latitude/longitude pairs for navigating (geo-locating) the IFOV's, 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 includes meta-data for the collection of scan lines such as temporal bounds, sensor characteristics, etc.

2.2 Queries

A *query* is defined as an an object that minimally specifies:

temporal bounds A start and stop time specified in Greenwich Mean Time. Only data recorded between the start and stop times will be used to answer the query.

spatial bounds The user's *area of interest* specified, currently, as a polygon on the surface of the globe. Only IFOV's incident on this region will be included in the query's result.

sensor types A list of sensors whose data is to be used to answer the query.

grid parameters Grid parameters specify the grid that will be used for the resultant image. This includes spatial extent of a "pixel" (they need not be square) and an origin for the grid.

result type The type of image returned as described below.

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 all image types, the database generates pixel values in the resultant images by finding all of the IFOV's incident on that pixel that satisfy the spatio-temporal bounds of the query.

Multi-attribute Images are images in which each pixel may have 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 IFOV's contained in a pixel with a function supplied by the user as part of the query.

List Images are the simplest, and potentially the largest, image type. Each pixel contains the entire set of IFOV's incident on that pixel.

We use a variety of image types to give users greater flexibility and control over image creation.

- In all cases, the images conform to a user specified grid. Two images with the same grid parameters are co-registered[4], greatly simplifying multi-sensor analysis.
- Traceability is preserved. Queries are repeatable given identical grid parameters. A user could specify a grid that contained exactly one pixel, and, using a *list image*, retrieve the list of IFOV's incident on that pixel.
- Mapping errors can be controlled. The grid used for a query is not necessarily linked to any projection. Thus, a user can choose to *display* the data using any projection desired.

2.3 Indexing Scheme

AVHRR GAC Level 1B data is, in effect, a 6x409x12000 matrix.

- 5 bands and a geographic location for each IFOV.
- 409 IFOV's for each scan line.
- Approximately 12,000 scan lines per data set.

Unfortunately, a simple matrix is not sufficient to capture the spatial attributes of the data. The distance between IFOV's is not uniform. The distance between scan lines varies due to eccentricities in the sensor's orbit. The distance between IFOV's in a single scan line varies with respect to the IFOV's distance from nadir *and* eccentricities in the sensor's orbit.

We considered spatial structures such as R-trees[6] and quadtrees[7]. Goodchild et. al.[4] suggest a hierarchical tessellation of triangles over the globe as a global spatial index. We found that, in general, the use of a global index for all IFOV's would require replication of large amounts of meta-data for each IFOV. Furthermore, a global index whose atoms are IFOV's tends to be prohibitively large. Obviously some bucket method is called for. In order to minimize data growth from replication of meta-data and to provide the greatest flexibility, we use a hierarchical indexing scheme based on *global* and *local* indices.

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

The Global Index. The global index is a coarse spatio-temporal index of 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.

From the global index perspective, a local index is an encapsulated object. The global index only needs to determine if a given query intersects the local index and pass the query to the local index for resolution. By implementing local indices as objects, and by keeping the global index general we can avoid the problems associated with global spherical approaches. Furthermore, implementing local indices as objects allows us to include remotely sensed data from other sensors in the future without re-engineering the global index. Finally, a coarse index allows a very fast way to reject large portions of the data base when presented with localized queries.

The global index is currently implemented as a packed R-Tree, however any general spatio-temporal index would be satisfactory. The root node of the global index is a bounding solid for all local indices in the global index. The packed structure is built from a list of local indices that has been sorted spatially. This allows us to use the global index as a range tree for faster query processing.

The Local Index. Local indices provide fine grained spatio-temporal indexing into AVHRR GAC Level 1B data. Recall that AVHRR GAC data is essentially a matrix. While the distance between IFOV's is not uniform, adjacency in the matrix does correspond to adjacency on the sphere. Therefore, a good solution would preserve the innate matrix qualities of the data while providing spatial indexing into that matrix. If we preserve the matrix, data selection using temporal criteria is trivial. We know the temporal bounds for the entire file and can calculate a time for each scan line. Spatial indexing is significantly more complicated.

Before we discuss our spatial indexing scheme for local indices, some preprocessing issues must be mentioned. Our first step is to divide the raw Level 1B file into seven files: one file for meta-data, one file for geographic locations of IFOV's 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 IFOV's sensor values in the band files.

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

Our index is a hierarchy of binary trees whose keys are quadrilaterals (see Figure 1). For convenience 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 IFOV's for each 16x16 block. Each "row" of 25 blocks (comprising 16 scan lines) gives us 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. This allows us to build a single *vertical tree* for the file whose leaves are 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 spatial extent in the global index. While this local indexing scheme is satisfactory, there are several things we can do to significantly improve spatial performance.

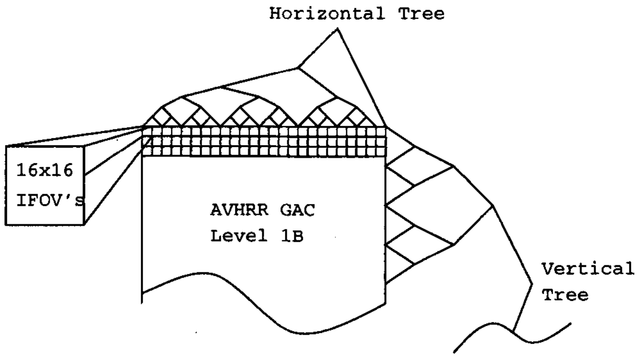

Fig. 1. Hierarchy of *block trees.*

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 chance of a null query in a local index 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 IFOV's within that bounding quadrilateral. Our optimizations strive to achieve good fit at low computational cost.

Local indices can be generated for entire Level 1B files, or for any portion of a Level 1B file. By dividing single AVHRR GAC Level 1B data files into several local indices we achieve a much better fit. Figure 2 shows a plot of left-most, right-most and center IFOV's 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 whole file into eight separate files, each 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.

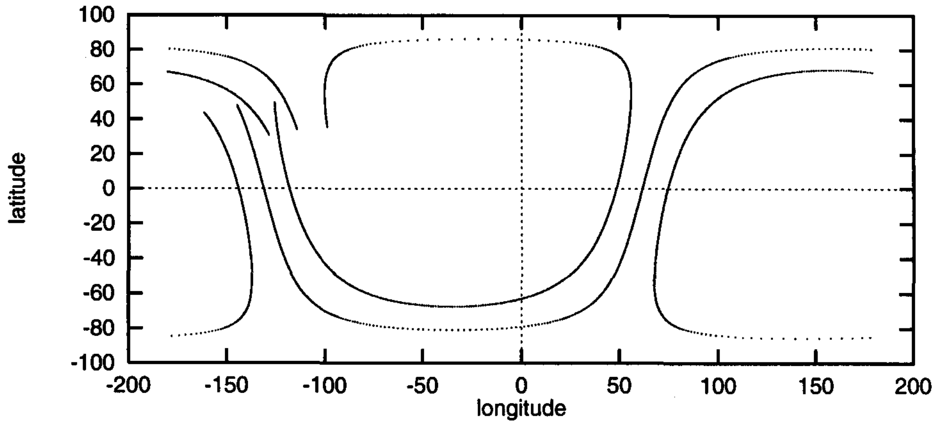

Fig. 2. IFOV's in a level 1B file.

Figure 3 is a plot of the bounding quadrilaterals for 16x16 IFOV blocks from a segment of an AVHRR GAC Level 1B file. To achieve a better fit we need to "square up" our view of the data. The problem is that we are plotting spherical angular data (latitudes and longitudes) in a planar frame of reference. Our solution is to project these bounding quadrilaterals using a *Lambert Conformal Conic Projection*[9]. 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 us to project the query for use with the index, but we have found the computational cost of projecting queries to be significantly less than the cost of unnecessary disk accesses from null queries. It is important to note that only the index is projected, not the underlying sensor data. The original latitude and longitude of each IFOV is preserved.

Resolving a Query. When a query (as defined in §2.2) is received, a blank 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 that portion of the output image that intersects each qualifying local index to the local indices.

Each local index first projects the query and output image into its local frame of reference and then searches its *block trees* for 16x16 blocks of IFOV's

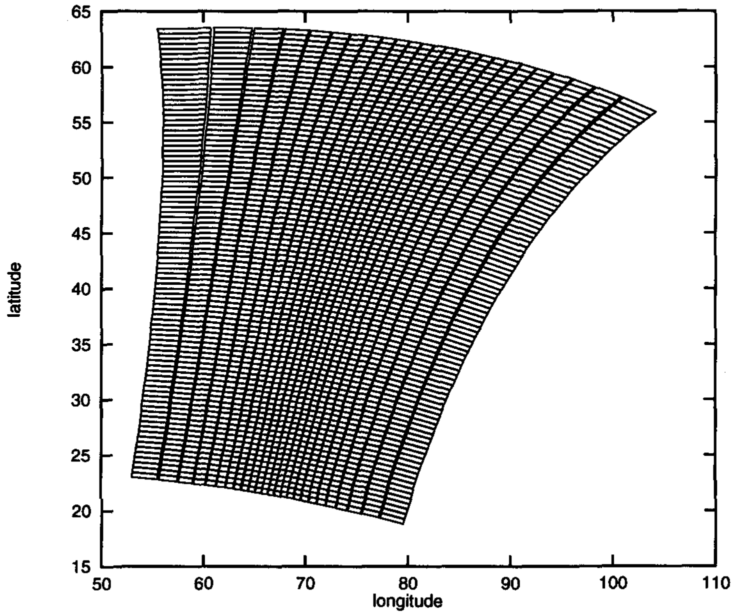

Fig. 3. Unprojected 16x16 IFOV blocks.

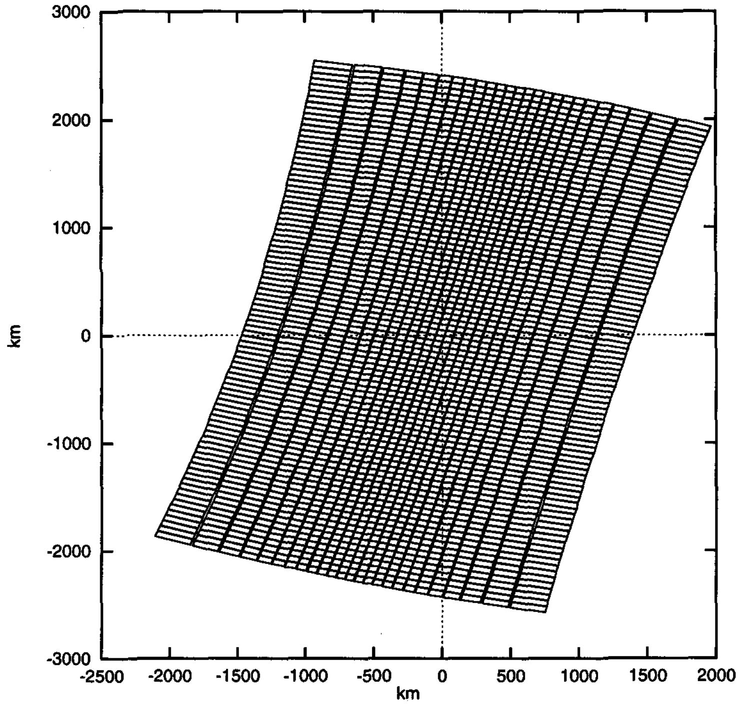

Fig. 4. Lambert projection of 16x16 IFOV blocks.

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 IFOV's referenced by the leaf node is retrieved from disk. Finally, we iterate through the 256 IFOV's in the retrieved block to determine which, if any, IFOV's to include in the output image.

High Performance Implementation. The prototype for our 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 of queries. The obvious solution is to parallelize the system. Unfortunately, parallelization is notoriously difficult and tends to change application codes in fundamental ways. Our solution is based on the use of objects and high-performance tools specifically designed to facilitate parallelization of sequential codes. In particular, we use the *CHAOS++* [1, 2] runtime library.

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. Currently, *CHAOS++* uses message passing as its transport layer and is implemented on several distributed memory machines, including the Intel iPSC/860 and Paragon, the Thinking Machines CM-5, and the IBM SP-1 and SP-2.

Parallelization of the system is achieved by first replicating the global index across all available processors, then dividing any incoming queries 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.
- Each sub-query is submitted to a processor for resolution. Each processor returns its results independently to the querying process.

In short, the original query is broken down into several smaller queries, each of which is handled by one processor.

3 Database Performance

After implementing our indexing scheme we ran test suites to examine:

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 given different numbers of processors and 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.

3.1 System Configuration

All testing was performed using parallel codes on the University of Maryland's IBM SP-2. This system contains sixteen RS6000/390 processing nodes running AIX. The nodes are interconnected via a proprietary interprocessor communications switch providing each node with up to 40 Mbytes per second bandwidth. Each node of the SP-2 has 12 Gbytes of online disk storage, making the 16 node machine capable of an aggregate I/O bandwidth exceeding 500 Mbytes/sec. The parallel implementation used IBM's MPL for message passing between processors.

3.2 Test Parameters

Each test suite was conducted on our prototype server loaded with 15 local indices covering the west coast of North America. Normally, the database dynamically loads and caches a number of local indices using a least recently used replacement scheme. We wished to exclude statistics for work done loading local indices. Therefore, all 15 indices were pre-loaded into the cache for each experiment on the database.

Figure 5 shows the image bounds for our three test queries and the bounds for the local indices these images intersect. The database normally receives queries interactively from users. For the purposes of testing, each query was hard coded into the database.

The three static queries were, with the exception of image bounds, identical. Query parameters (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

Minimum spatial resolution in AVHRR GAC data is 1/128 degrees of latitude or longitude. Grid parameters are therefore specified in increments of 1/128 of a degree. 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 demonstrate load balance characteristics of the system.

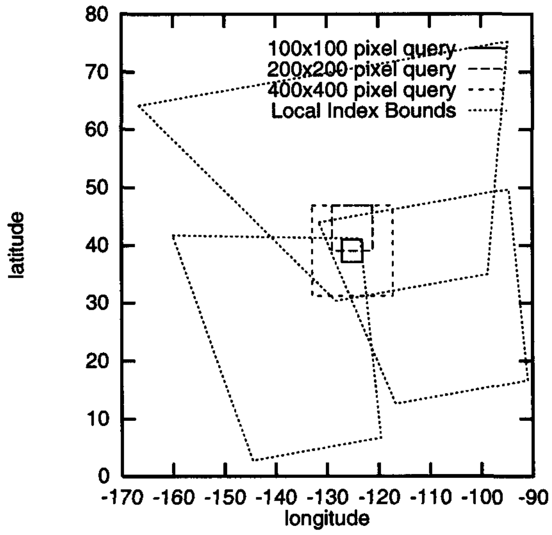

Fig. 5. Query bounds.

Our database uses an *output partitioning* technique to distribute the workload over multiple processors. We attempt to evenly divide the output image along pixel boundaries based on the number of processors available for the task. Both the partitioning scheme and queries were held constant for all tests.

For each test suite we collected data on the following:

- Time to complete a query (includes 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 read() system calls.
- Number of bytes read in read() system calls.

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

3.3 Test Suite Results

Our first test suite was an attempt 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 object (points and quadrilaterals) constructors. The use of null or random spatial objects skews the results. Optimizations in our computational geometry codes allow for early exits from intersection tests. However, we offer these results as only a rough indicator of the time required to find the appropriate data in our 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. We ran 200x200 pixel queries on from 1 to 16 processors. Next we ran 100x100 and 400x400 pixel queries on enough combinations of processors to plot representative curves for these queries. We then enabled I/O and ran identical tests.

Table 1 compares times and work load for the 200x200 pixel query on 16 processors with and without I/O enabled. In this first comparison, enabling I/O did not significantly increase run times or work loads. Needless to say, we were stunned to find similar timings for the two tests. Recall that once a 16x16 block of IFOV's is retrieved from disk, a pixel's bounds are iterated through each of the 256 IFOV's to determine which IFOV's in the block fall within the pixel. Our first prototype treated each IFOV as a simple point and used a point-in-polygon routine to test for intersection with a pixel. Unfortunately, queries could be posed such that an individual pixel in the output image was smaller than the distance between adjacent IFOV's, occasionally yielding blank pixels in the middle of the image. Our solution was to construct a bounding quadrilateral using the midpoint distances between an IFOV and it's neighbors for each IFOV. This quadrilateral was then tested against the pixel using a polygon-polygon intersection routine. We disabled this routine, again treating IFOV's as points, and re-ran the test suites. Table 1 shows our results with IFOV's as points for a 200x200 pixel query on 16 processors as well. When IFOV's are treated as points, I/O becomes a significant contributor to overall run time.

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

IFOV's 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

Figure 6 compares work loads (expressed as number of spatial intersections performed) for the different query sizes and job completion times for different queries on different numbers of processors. Spatial intersection counts were identical between similar tests with and without I/O enabled. We found that total work load (the sum of all spatial intersections performed by all processors) varied slightly as more processors were devoted to a given query. Individual processor work loads varied greatly depending on how the query was partitioned. We also found it interesting that as pixel counts in output images quadrupled, work load and time to complete a query approximately tripled, suggesting that our indexing scheme becomes more efficient as output image pixel counts grow.

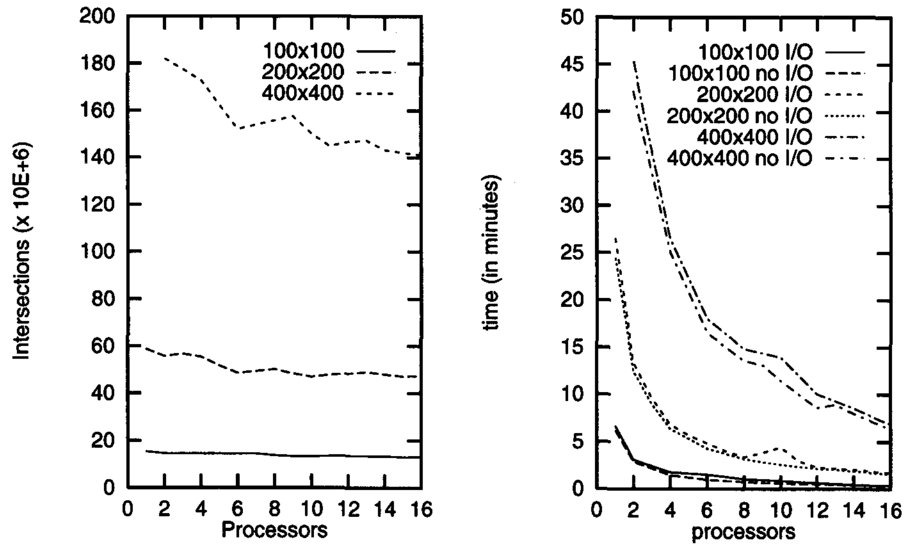

Fig. 6. Line-line intersection vs. query size (left) and query completion time (right).

Figure 7 shows processing times for a 200x200 pixel query with and without I/O enabled. I/O appears to add, on average, approximately 8% to the total processing time for a given query. We note with alarm that in some cases I/O accounted for over one quarter of the time spent on a query.

4 Conclusions and Future Work

4.1 Index Performance

Our initial approach to the problem assumed that any database of this size would be I/O bound. Our experiments demonstrate that, using this indexing scheme, the computational cost of searching local indices can easily match the cost of I/O. Given an image (a collection of pixels) and a local index (a collection of IFOV's), *forward mapping* asks "for each pixel, which IFOV's are in this pixel?" *Inverse mapping* asks "for each IFOV, which pixels contain this IFOV?" The indexing scheme described in this paper uses forward mapping and non-uniformity of spatial characteristics within a matrix of IFOV's forces us to search for IFOV's.

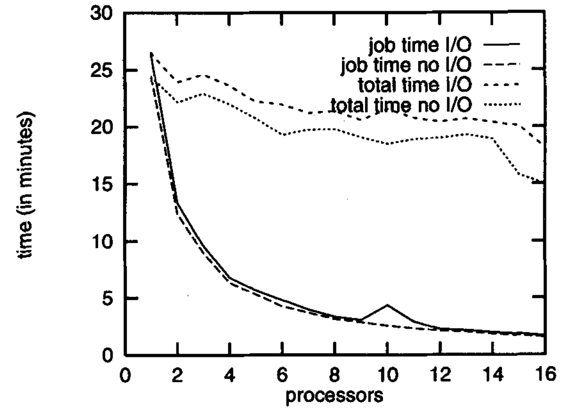

Fig. 7. Processing times for 200x200 pixel query.

However, the result of any query is an image as described in §2.2 with a *grid* that indicates the spatial extent of any pixel in the image. The pixels of the image are spatially uniform. We are developing a new indexing scheme that takes advantage of this uniformity by using inverse mapping.

Inverse mapping should yield two significant advantages. First, searching in local indices is eliminated as is the requirement to build hierarchical data structures for local indices. In short, a local index becomes a simple matrix of IFOV's. Second, image resolution should no longer affect system run time. Higher image resolution produces more pixels in a given image. Forward mapping requires a local index search for each pixel. Inverse mapping requires an operation for each IFOV incident on the image and is unaffected by the number of pixels.

4.2 Load Balancing and I/O Performance

Significant work must be done to improve load balancing and I/O performance for the system. As much as a quarter of the time to resolve a query is being spent on I/O. As we attempt queries with larger output images, we expect the problem to get worse.

We have started work on another *output partitioning* scheme for distributing queries. Our hope is that the new *output partitioning* will improve both load balance and I/O performance. The first step is to intelligently distribute data sets across all available processors' disks (*de-clustering*). Data sets local to a processing node will be arranged on disk for efficient spatial searching (*clustering*). Queries will be partitioned such that each processor will mostly resolve those pixels in an output image for which it has local data. We are investigating partitioning schemes that both provide good load balance and minimize interprocessor communication. This plan should result in more efficient I/O because most disk accesses will be for data to be processed on the local processor, rather than requiring communication across the network.

References

1. 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.
2. 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.
3. 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.
4. 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.
5. 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.
6. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the SIGMOD Conference*, pages 47–57, Boston, June 1984.
7. Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Addison - Wesley, Reading, Massachusetts, 1990.
8. Carter T. Shock, Larry Davis, Samuel Goward, and Alan Sussman. A high performance image database system for remote sensing. In *Proceedings of the Advanced Imagery Pattern Recognition Conference*, Washington, D.C., October 1995.
9. John P. Snyder. *Map Projections - A Working Manual.* U. S. Geological Survey, 1987. U. S. Geological Survey Professional Paper 1395.
10. Michael Stonebraker. Sequoia 2000: A reflection on the first three years. Technical Report S2K-94-58, EECS Dept., University of California, Berkeley, 1994.
11. Michael Stonebraker, Jim Frew, Kenn Gardels, and Jeff Meredith. The Sequoia 2000 storage benchmark. In *Proceedings of the 1993 ACM SIGMOD Conference*, May 1993.