Workshop #8
Parallel DSP and Image Processing

A High Performance Image Database System for Remotely Sensed Imagery *

Carter T. Shock, UMIACS†
Chialin Chang, UMIACS and Department of Computer Science‡
Larry Davis, UMIACS and Department of Computer Science
Samuel Goward, Department of Geography
Joel Saltz, UMIACS and Department of Computer Science
Alan Sussman, UMIACS and Department of Computer Science

University of Maryland Institute for Advanced Computer Studies
University of Maryland at College Park
A.V. Williams Building
College Park, MD 20742
voice: (301) 405-6722
fax: (301) 314-9658

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.

*This research was supported by the National Science Foundation under Grant #ASC 9318183.
†ctso@umiacs.umd.edu
‡chialin@cs.umd.edu

1 INTRODUCTION

Earth scientists have encountered a variety of obstacles in their attempts to use remotely sensed imagery to study the earth. They have been thwarted by the staggering volume of remotely sensed data and by limitations associated with currently available data products. In this paper, we present a database design for indexing 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 of *Advanced Very High Resolution Radiometer* (AVHRR) *Global Area Coverage* (GAC) Level 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 [1]. 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. The shape of this region is an ellipsoid [2]. On nadir [3] GAC IFOV's are approximately a circle with a diameter of four kilometers.

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 worth of scan lines, or approximately 12,000 scan lines. Each record in GAC Level 1B data 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

Before discussing an indexing scheme, we must describe the classes of queries it must support, as well as the form of answers to the queries.

A *query* is 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 Simply 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.

[1] First run AVHRR sensors only record four thermal bands. However, all AVHRR data includes five bands of thermal data. Sensors lacking the fifth thermal sensor duplicate band 4 data for band 5

[2] The exact shape of an IFOV is the intersection of an oblate spheroid (the earth) and a cone (the aperture of the thermal detector)

[3] *On nadir* refers to measurements directly beneath the orbital path of the sensor. Off nadir IFOV's are elongated ellipsoids whose spatial extent is proportional to their distance from nadir.

The result of a query is one of two types of *image*. For simplicity we will 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 by a user supplied function.

List Images are the simplest, and potentially the largest image type. Each pixel contains the entire list 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. For example, one 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

Our search for an indexing scheme considered a simple matrix, R-trees, quadtrees, spherical methods, and custom designs.

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.

R-trees[5] and quadtrees[6] are inherently 2-dimensional and as such have a general bounding box for the entire structure. When used to index the surface of a sphere, it is difficult to represent the fact that the two sides and the top and bottom of the bounding box are spatially adjacent to each other. Although IFOV's can be represented by a point in two dimensions (latitude and longitude), large collections of IFOV's are spherical in nature. Quadtrees suffer from the additional drawback that they index space using powers of two. While this would work well for other data sets such as AVHRR *local area coverage* with 2048 IFOV's per scan line, it yields unacceptable data growth for 409 IFOV's per scan line[7]. Both R-trees and quadtrees can be implemented in three or more dimensions, however, all of our data lies on the surface of a spheroid. Quadtrees and R-trees in 3 dimensions would require us to index the interior of the sphere as well, resulting in a larger index with much of the space wasted.

Goodchild et. al.[8] suggest a hierarchical tessellation of triangles over the globe as a global spatial index. While this approach overcomes the limitations of inherently 2-dimensional structures, it too has serious

[4] This co-registration is accurate only to within the spatial accuracy of the level 1B data
[5] [Gut84]
[6] [Sam90]
[7] A quadtree's index would have to be expanded to 512, a growth of almost 25%.
[8] [GSD91]

drawbacks. Neighbor finding is very difficult and the use of a global index for all IFOV's would require replication of large amounts of meta-data for each IFOV.

Our solution is a hierarchical indexing scheme based on *global* and *local* indices.

2.3.1 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 us a very fast way to reject large portions of the data base when presented with localized queries.

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 a matrix. While the distance between IFOV's is not uniform, adjacency is. 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 [9]. We assume that all IFOV's within a scan line were collected at the same instant. Spatial indexing is significantly more complicated.

Before we discuss our spatial indexing scheme for local indices, some preprocessing issues must be mentioned. Requirements capture for the system revealed that user queries to the system will not reference all five bands of AVHRR GAC data. 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 bands of data must be retrieved. The file of geographic data is an implicit key to the band data files. One latitude/longitude pair is stored for each IFOV. The file offset to the geographic location of an IFOV can be used to find the offset to the IFOV's radiometry in the band files.

We also had to choose a granularity for the local indices. In other words, should the index discern between individual IFOV's or small collections of IFOV's? We chose the latter to reduce the overall size of the index. Furthermore, requirements capture has revealed 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. 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.

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 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. Because each node's key must encompass all children, the root of each block tree is also 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

[9] We arbitrarily restrict minimum temporal resolution to sixteen scan lines.

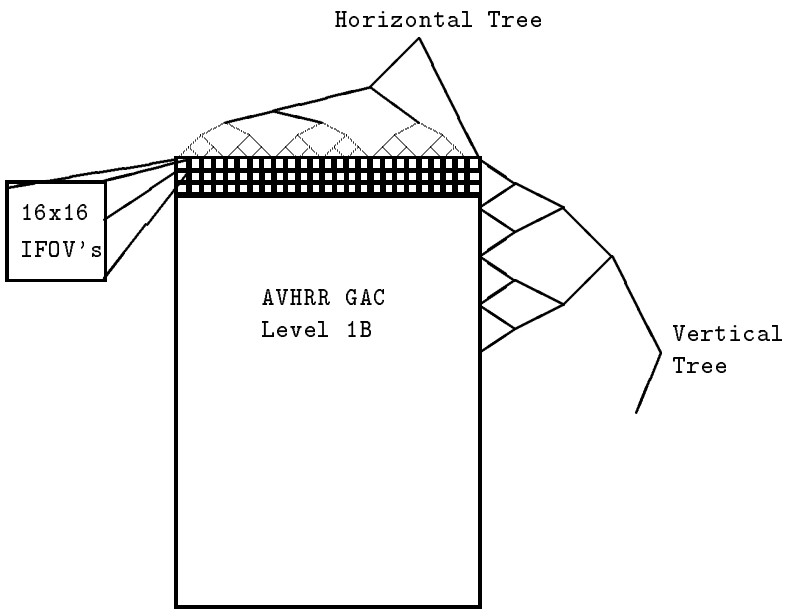

Figure 1: Hierarchy of *Block Trees*

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.

2.3.3 Local Index Optimizations

The efficiency of spatial indexing based on bounding quadrilaterals 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 data within that bounding quadrilateral. Our optimizations strive to achieve good fit at low cost.

Local indices as described 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 both 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.

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, of course, 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*[10]. Figure 4 shows the same blocks from figure 3 projected using a Lambert projection. Projecting the index yields a much better fit and fewer null queries. Projecting the index does, however, require us to project the query for use with the index, but we have found the cost of projecting queries to be significantly less than the cost of null queries[11]. It is important to note that we project the index only, not the underlying data. The

[10] [Sny87]

[11] Projecting the query into the index's frame of reference is a relatively simple computation while null queries can result in unnecessary disk accesses.

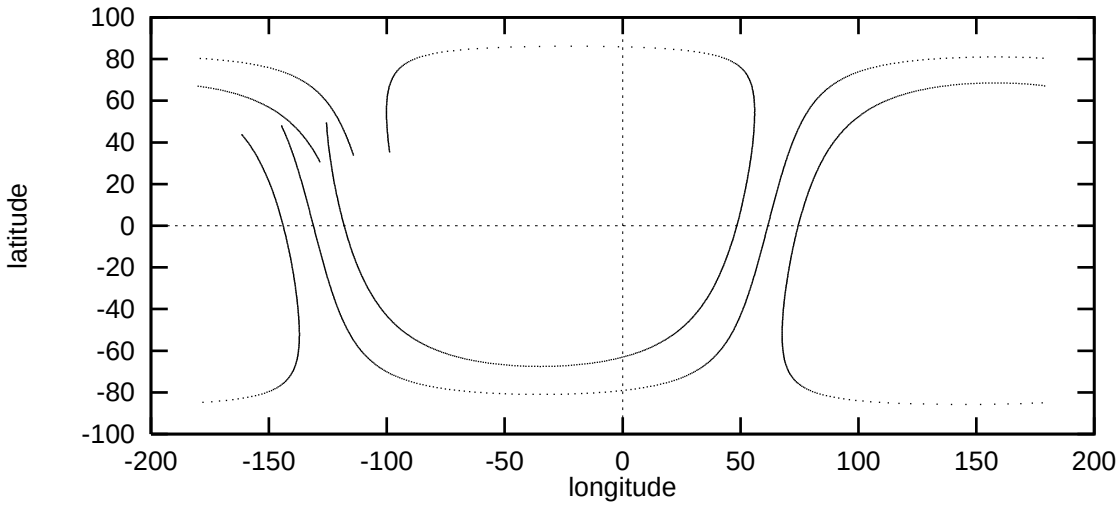

Figure 2: Whole Level 1B File

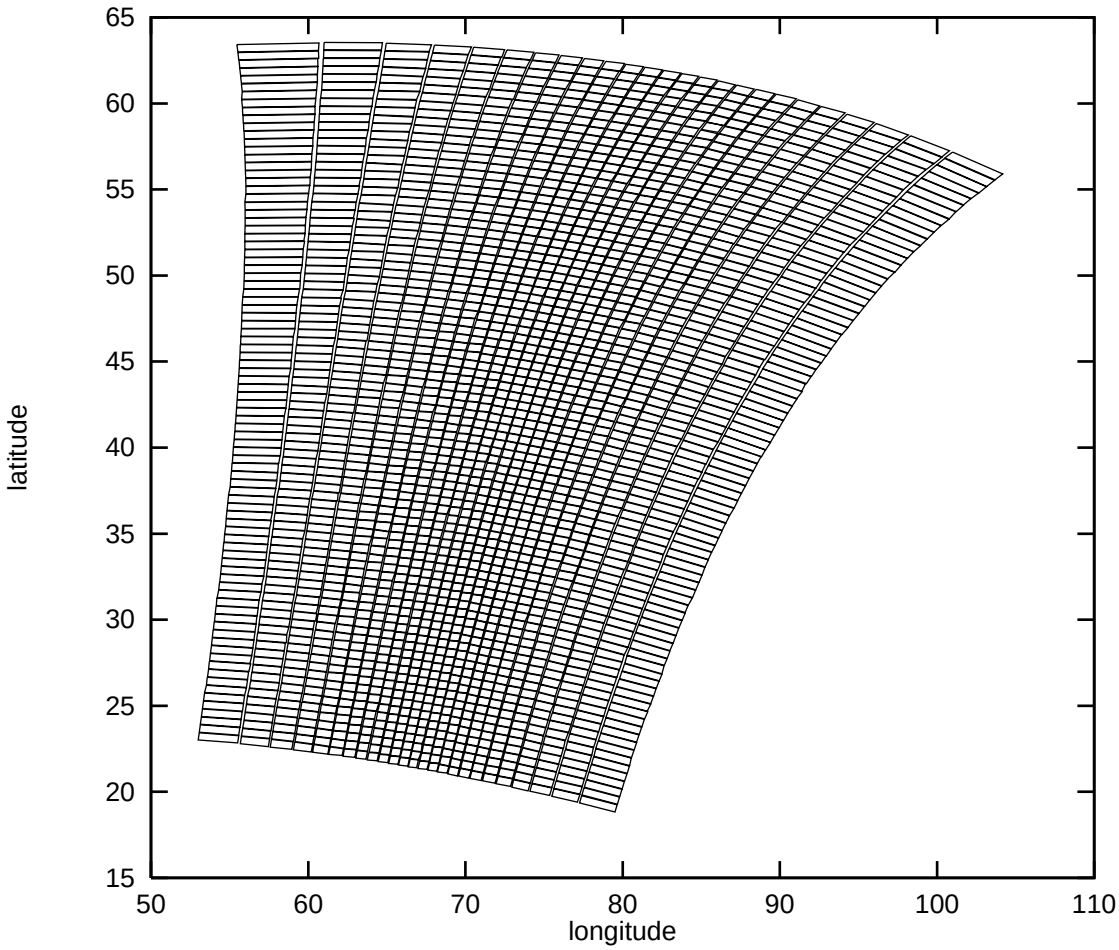

Figure 3: Unprojected 16x16 IFOV Blocks

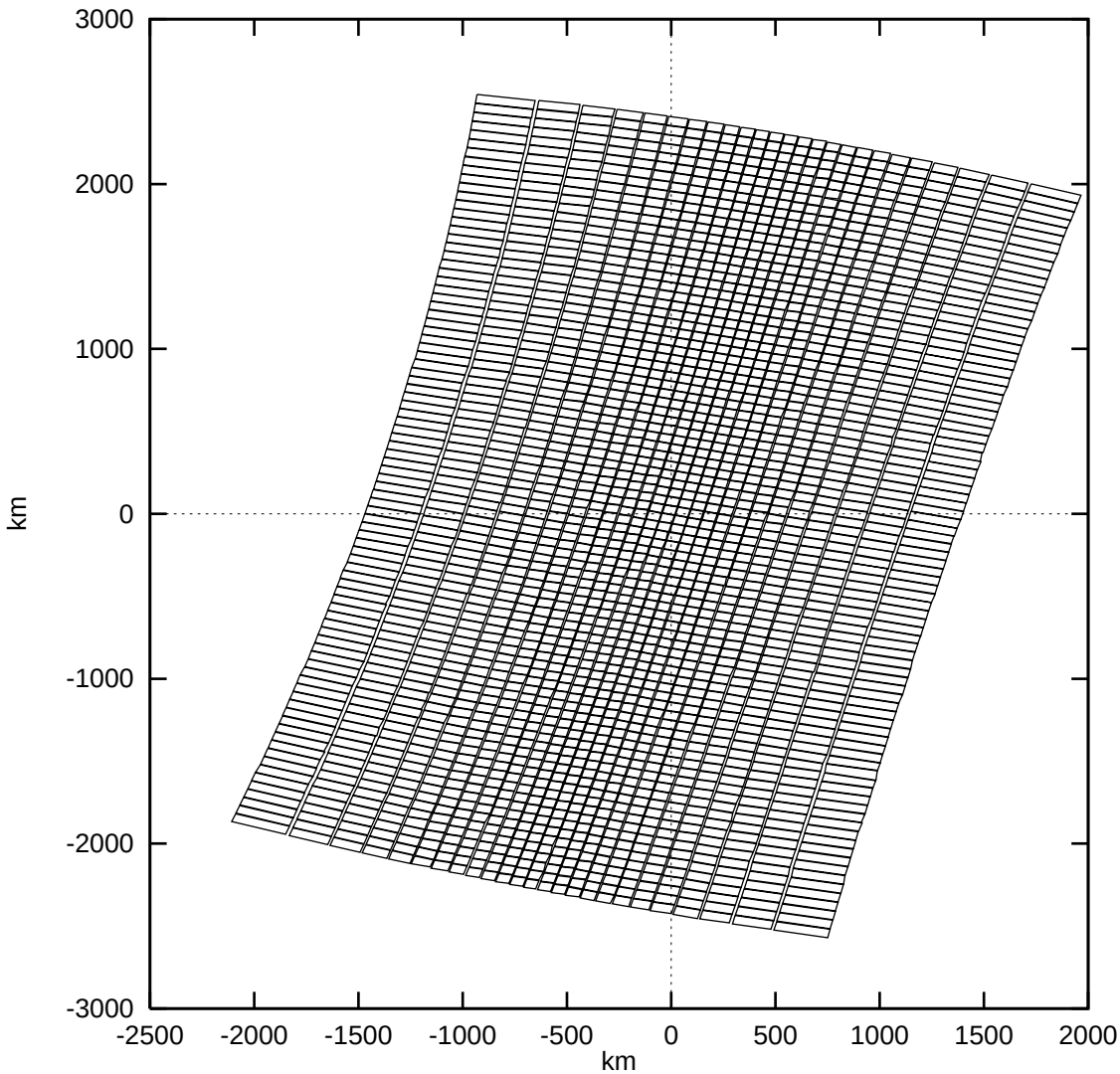

Figure 4: Lambert Projection of 16x16 IFOV Blocks

original latitude and longitude of each IFOV is preserved.

2.3.4 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 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.

2.3.5 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 was 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++*[12] 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 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 sum off 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 it's 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?

Scaleability How does the system perform given different numbers of processors and different queries.

[12] [CSS95]

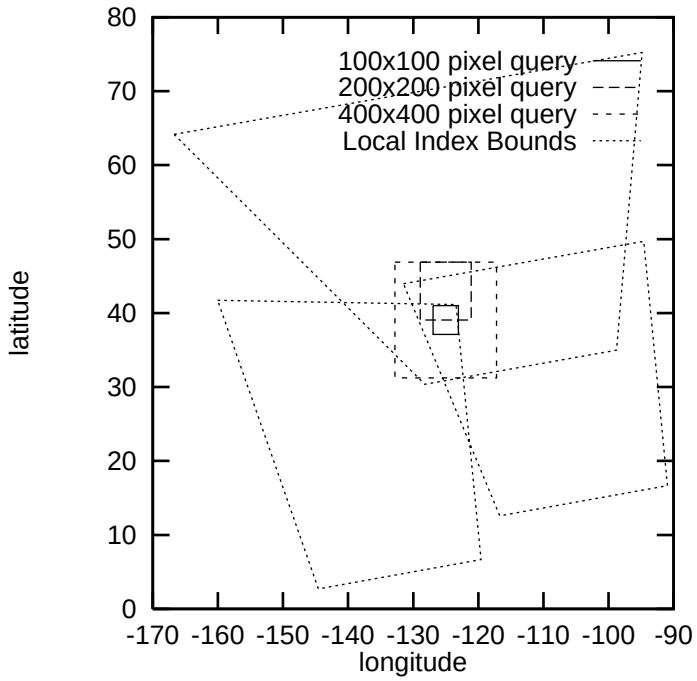

Figure 5: Query Bounds

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 machine was acquired in November 1994 with grants from the National Science Foundation and IBM. This system contains sixteen RS6000/390 processing nodes running AIX. These nodes are interconnected via a proprietary interprocessor communications switch providing each node with up to 40 Mbytes per second bandwidth. In May of 1995, the SP-2 was expanded to include 244 Gbytes of online disk storage capable of an aggregate I/O bandwidth exceeding 500 Mbytes/sec. The SP2 has a a RS6000/590 compile server front-end.

3.2 Test Parameters

Each test suite was conducted on our prototype server loaded with 15 local indices. 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 run of 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 database processed a query as if it had been received from a user and then terminated.

The three static queries were, with the exception of image bounds, identical. Query parameters[13] were:

- Temporal Bounds: unbounded.

[13] See §2.2

- 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 are specified in increments of 1/128 of a degree. Therefore, for all 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.

Our database uses an *output partitioning* technique to distribute the workload over multiple processors. In short, 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 test suites.

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 is 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*, no actual data was retrieved from disk[14]. Null or random spatial objects (points and quadrilaterals) were substituted for the real data. 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 1,2,...,15, and 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 timing and work load for the 200x200 pixel query on 16 processors with and without I/O enabled. In all cases, 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 of the database 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

[14] The read() and seek() system calls were simply commented out.

IFOV's treated as	Points		Boxes	
I/O status	enabled	disabled	enabled	disabled
τ_{total} [a]	18:11	15:01	77:53	77:34
τ_{job} [b]	1:42	1:35	6:34	6:34
ι_{pp} [c]	6,389,194	6,389,194	31,042,506	31,042,506
ι_{ll} [d]	64,579,998	64,579,998	454,889,555	438,697,786
ι_{pip} [e]	47,304,362	47,304,362	145,645,054	145,917,520

[a] τ_{total} = total processor time
[b] τ_{job} = time to complete the query
[c] ι_{pp} = total polygon-polygon tests
[d] ι_{ll} = total line-line tests
[e] ι_{pip} = total point-in-polygon tests

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

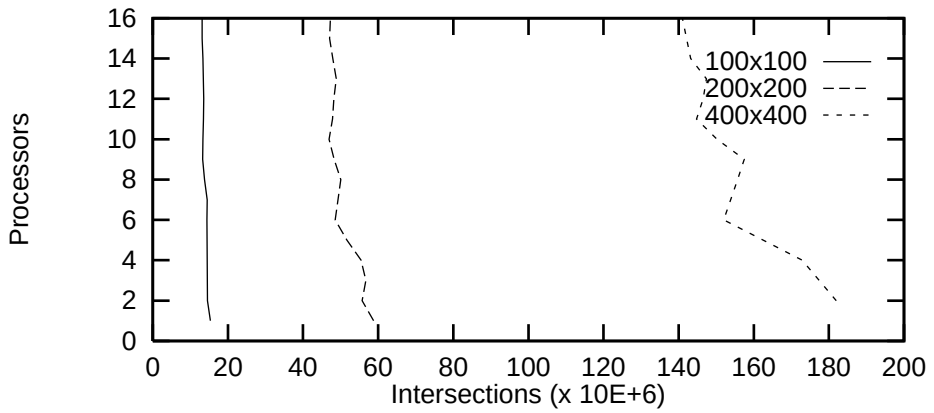

Figure 6: Line-Line Intersection vs. Query Size

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.

Figure 6 compares work loads for the different query sizes. We use the number of spatial intersections performed as our measure of work load. Spatial intersection counts were identical between similar tests with and without I/O enabled. Figure 7 shows job completion times for different queries on different numbers of processors. 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 count grows.

Table 2 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.

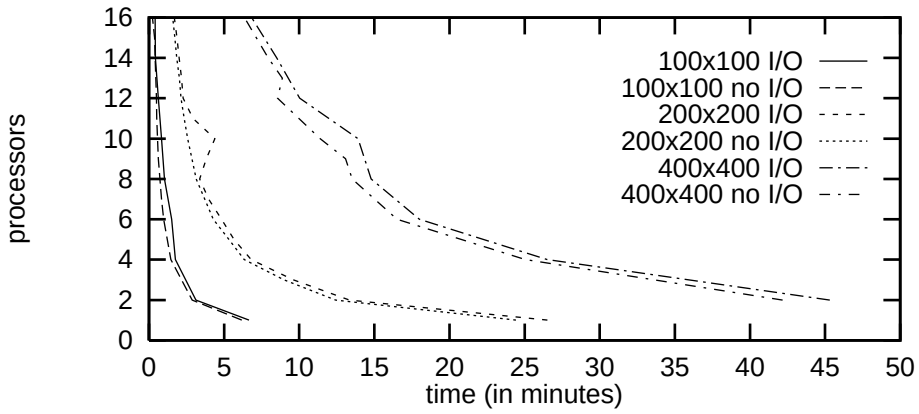

Figure 7: Query Completion Time

	Job Time			Total Time [a]				
n^b	i/o on	i/o off	diff	i/o on	i/o off	diff	ρ^c	β^d
1	26:31	24:25	8.6%	26:31	24:25	8.6%	335655	171.85
2	13:21	12:25	7.5%	23:55	22:11	7.8%	335655	171.85
3	9:36	8:59	6.9%	24:35	22:56	7.2%	335655	171.85
4	6:48	6:20	7.4%	23:38	22:00	7.4%	335655	171.85
5	5:40	5:19	6.6%	22:13	20:45	7.1%	309501	158.46
6	4:49	4:16	12.9%	21:58	19:16	14%	293643	150.34
7	3:59	3:43	7.2%	21:13	19:45	7.4%	299895	153.54
8	3:22	3:09	6.9%	21:23	19:47	8.1%	303888	155.59
9	3:04	2:51	7.6%	20:36	19:05	7.9%	293112	150.07
10	4:23	2:34	70.8%	21:34	18:30	16.6%	285963	146.41
11	2:55	2:21	24.1%	20:49	18:55	10%	290913	148.94
12	2:17	2:08	7%	20:27	19:01	7.5%	294237	150.64
13	2:10	2:02	6.6%	20:43	19:17	7.4%	297690	152.41
14	2:00	1:51	8.1%	20:25	18:56	7.8%	292485	149.75
15	1:52	1:43	8.7%	20:04	15:47	27.1%	286515	146.69
16	1:41	1:35	6.3%	18:09	14:59	21.1%	288906	147.91

[a] Sum of all individual processor times
[b] n = number of processors
[c] ρ = system calls to `read()`
[d] β = megabytes read

Table 2: Processing times for 200x200 pixel query with and without I/O enabled.

4 Conclusions and Future Work

4.1 Spatial properties of IFOV's in local indices

As seen in table 1, arbitrarily treating all IFOV's as quadrilaterals significantly increases the workload on the system. In our tests this treatment of IFOV's slowed it by a factor of 4. We plan to investigate adaptive schemes where only those IFOV's that would be returned erroneously blank will be converted into quadrilaterals for intersection tests. We have rejected as futile any attempt to store pre-calculated bounds for IFOV's since the majority of the time seems to be spent in performing intersection tests, not in generating bounding quadrilaterals.

4.2 Scaleability

Our results demonstrate that query processing in our database does scale. However, scaleability is highly query dependent. This information should prove useful for tuning the database, especially in a multi-user environment. Our tests indicate that for some queries, it might be advantageous to distribute the query over only a portion of the available processors, leaving other processors free to handle queries from other users.

4.3 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 an *output partitioning* scheme for distributing queries.

Our hope is that *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*). Then data sets local to a processing node are arranged on disk for efficient spatial searching (*clustering*). Next, queries will be partitioned such that each processor will only resolve those pixels in an output image for which it has local data. Finally, an aggregation step takes place where pixels with partial solutions from multiple processors will be aggregated. Our plan should result in more efficient I/O because all disk accesses will be local (rather than through notoriously slow NFS). Load balance should also improve, especially for large output images.

4.4 Spatial Engine

In general, we are pleased with the performance of our computational geometry codes. We perform several million intersections per minute within the system. However, our experience with treatment of IFOV's reinforces just how critical this portion of the database code is. Efforts to optimize these codes even further are underway.

References

[CSS95] 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.

[GSD91] 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.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the SIGMOD Conference*, pages 47–57, Boston, June 1984.

[Sam90] Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Addison - Wesley, Reading, Massachusetts, 1990.

[SDGS95] 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.

[Sny87] John P. Snyder. *Map Projections - A Working Manual.* U. S. Geological Survey, 1987. U. S. Geological Survey Professional Paper 1395.