# Titan: a High-Performance Remote-sensing Database *

Chialin Chang, Bongki Moon, Anurag Acharya, Carter Shock
Alan Sussman, Joel Saltz
Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland, College Park 20742

**Abstract**

There are two major challenges for a high-performance remote-sensing database. First, it must provide low-latency retrieval of very large volumes of spatio-temporal data. This requires effective declustering and placement of a multi-dimensional dataset onto a large disk farm. Second, the order of magnitude reduction in data-size due to post-processing makes it imperative, from a performance perspective, that the postprocessing be done on the machine that holds the data. This requires careful coordination of computation and data retrieval. This paper describes the design, implementation and evaluation of *Titan*, a parallel shared-nothing database designed for handling remote-sensing data. The computational platform for Titan is a 16-processor IBM SP-2 with four fast disks attached to each processor. Titan is currently operational and contains about 24 GB of data from the Advanced Very High Resolution Radiometer (AVHRR) on the NOAA-7 satellite. The experimental results show that Titan provides good performance for global queries, and interactive response times for local queries.

## 1 Introduction

Remotely-sensed data acquired from satellite-based sensors is widely used in geographical, meteorological and environmental studies. A typical analysis processes satellite data for ten days to a year and generates one or more images of the area under study. Data volume has been one of the major limiting factors for studies involving remotely-sensed data. Coarse-grained satellite data (4.4 km per pixel) for a global query that spans the shortest period of interest (ten days) is about 4.1 GB; a finer-grained

*This research was supported by the National Science Foundation under Grant #ASC 9318183, NASA under Grant #NAG 11485 (ARPA Project #8874), and ARPA under Grant #DABT 63-94-C-0049 (Caltech Subcontract #9503).

version of the same data (1.1 km per pixel) is about 65.6 GB. The output images are usually significantly smaller than the input data. For example, a multi-band full-globe image corresponding to the 4.4 km dataset mentioned above is 228 MB. This data reduction is achieved by composition of information corresponding to different days. Before it can be used for composition, individual data has to be processed for correcting the effects of various distortions including instrument drift and atmospheric effects.

These characteristics 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. This requires effective declustering and placement of a multi-dimensional dataset onto a large disk farm. Furthermore, it is necessary that the disk farm be suitably configured so as to provide the high I/O bandwidth needed for rapid retrieval of large data volumes. Second, the order of magnitude reduction in data size makes it imperative, from a performance perspective, that correction and composition operations be performed on the same machine that the data is stored on. This requires careful coordination of computation and data retrieval to avoid slowing down either process.

Several database systems have been designed for handling geographic datasets [4, 24, 8, 31, 32]. These systems are capable of handling map-based raster images as well as geographic entities such as map points (e.g. cities) and line segments (e.g. rivers, roads). The systems provide powerful query operations, including various forms of spatial joins. However, these systems are not suitable for storing raw remote-sensing data that has not been processed to a map-based coordinate system. This means that they are suitable for the output images we have described, rather than the original data. Maintaining remote-sensing data in its raw form is necessary for two reasons [28]: (1) a significant amount of earth science research is devoted to developing correlations between raw sensor readings at the satellite and various properties of the earth's surface; once the composition operation is performed it is no longer possible to retrieve the original data and (2) the process of generating a map-based image requires projection of a globe onto a two-dimensional grid; this results 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. Finally, all of these systems have been implemented on uniprocessor platforms. None of them have been targeted for configurations with large disk farms, which we believe are important given the volume of data retrieved for each query.

This paper describes the design, implementation and evaluation of *Titan*, a parallel shared-nothing database designed for handling remote-sensing data. The computational platform for Titan is a 16-

processor IBM SP-2 with four fast disks (IBM Starfire 7200) attached to each processor. A widely used micro-benchmark indicated the maximum aggregate application-level disk-I/O bandwidth of this configuration to be 170 MB/s using the Unix filesystem interface. [1] Titan is currently operational and contains about 24 GB of data from the Advanced Very High Resolution Radiometer (AVHRR) sensor on the National Oceanic and Atmospheric Administration NOAA-7 satellite.

The paper focuses on three aspects of the design and implementation of Titan: data placement on the disk farm, query partitioning and coordination of data retrieval, computation and communication over the entire machine. Section 2 provides an overview of the system. Section 3 describes the declustering and data placement techniques used in Titan. The data layout decisions in Titan were motivated by the format of AVHRR data and the common query patterns identified by NASA researchers and our collaborators in the University of Maryland Geography Department. Section 4 describes the indexing scheme used by Titan. Section 5 describes the mechanisms used to coordinate data retrieval, computation and interprocessor 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. Section 6 describes the experiments performed to evaluate the system and analyzes the results. Section 7 describes the lessons learned in this endeavor. We believe our experiences suggest useful guidelines that go beyond remote-sensing databases in their scope. In particular, we expect our techniques for coordinating and balancing computation, communication and I/O are useful for other unconventional databases that need to perform substantial post-processing. Similarly, we believe our results provide additional evidence for the utility of the *minimax* algorithm for declustering multidimensional datasets over large disk farms. Finally, Section 8 provides a summary of our results and describes ongoing work.

## 2 System Overview

Titan consists of two parts:

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

[1] The bandwidths reported in [1] were measured under AIX 3.5. The system has recently been upgraded to AIX 4.1, but we have not yet been able to configure the system to again achieve the reported bandwidth.

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 processing 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 current implementation of Titan uses one node of the 16-processor IBM SP-2 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.

Titan partitions its entire sensor data set into coarse-grained chunks. Spatio-temporal keys to search for and retrieve these chunks are stored in a simplified R-tree. The partitioning and indexing schemes will be described more fully in Sections 3.1 and 4, respectively. The index is used by the front-end to identify data blocks needed for resolving a given query, and to partition the work load for efficient parallel execution. The size of Titan's index for 24 GB of sensor data is 11.6 MB, which is small enough for the front-end to retain in main memory.

Titan queries specify four kinds of constraints: (1) temporal bounds, which are specified in Universal Coordinated Time (UTC), (2) spatial bounds, which are specified as a quadrilateral on the surface of the globe, (3) sensor type and number, and (4) resolution of the grid that will be used for the image generated as the result.

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 the sensor readings corresponding to that pixel. The current implementation supports `max` and `min` composition functions.

When the front-end receives a query, it searches the index for all data blocks that intersect with the query window in space and time. The search returns a list of requests for data blocks. Using the location information for each block stored in the index, the front-end partitions the list of data block requests among the back-end nodes – the list corresponding to each node contains the requests for data blocks stored on that node. In addition, the front-end partitions the result image among all back-end nodes. Currently the result image is evenly partitioned by blocks of rows and columns, assigning each back-end node approximately the same number of output pixels. The front-end then distributes data block requests and output partition information to all back-end nodes.

When a back-end node receives its query information, it computes a schedule for retrieving blocks from its local disks. As data blocks are retrieved from the disks, the back-end node generates the communication necessary to move the blocks to all back-end nodes that need them to resolve their parts of the output image, in a pipelined fashion. Each back-end node can thus process the data blocks it requires, as they arrive either from local disks or across the communication network. More details about scheduling the various back-end operations are described in Section 5.

Once a data block is available on a back-end node (either retrieved from a local disk or forwarded by another back-end node), a simple quadrature scheme is used to search for sensor readings that intersect with the local part of the partitioned output image. First, the locations of the four "corner" sensor readings in a data block are mapped to the output image. If all the locations fall inside the image, all readings in the data block are mapped to pixels in the output image and composed with the existing value of the pixel. If any corner location is outside the local part of the output image, the data block is divided into four approximately equal sub-blocks and searched recursively, rejecting any portions that are wholly outside the image.

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

# 3 Data Placement

Titan addresses the problem of low-latency retrieval of very large volumes of data in three ways. First, Titan takes advantage of the AVHRR data format, and of common query patterns identified by earth science researchers, to partition the entire dataset into coarse-grained chunks that allow the hardware configuration to deliver good disk bandwidth. Second, Titan tries to maximize disk parallelism by declustering the set of chunks onto a large disk farm. Finally, Titan attempts to minimize seek time on individual disks by clustering the chunks assigned to each disk. The following subsections describe each of these techniques used to provide low-latency data retrieval.

## 3.1 Data Partitioning

The data partitioning decisions in Titan were motivated by the format of AVHRR data and the common query patterns identified by NASA researchers [7, 29] and our collaborators in the University of Maryland Geography Department [13, 18].

AVHRR data is organized as one file per satellite orbit. The NOAA satellites orbit the earth approximately 14 times a day. Each file consists of a sequence of scan lines, each line containing 409 pixels. Each pixel consists of five readings, each in a different band of the electromagnetic spectrum.

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 consecutively). However, the satellite data processing programs that we are aware of process either band one and two data or band three, four and five data [7, 29]. This grouping is due to the properties of the bands: the first two bands provide information to estimate the amount of chlorophyll in a region [12] whereas the last three bands can be used to estimate cloud cover and surface temperature [19]. To take advantage of these query patterns, Titan stores AVHRR data in two parts, one containing data for bands one and two and the other containing data for bands three, four and five.

The primary use of AVHRR data is to determine the land cover in a region on the ground, so common queries correspond to geo-political regions of world (*e.g.*India, Taiwan, Korea, Africa). Using individual files as units of data storage and retrieval is likely to result in much more I/O than necessary. On the other hand, retrieving the data in very small units (*e.g.*, individual scan lines) is not efficient. A good compromise can be achieved by using the smallest data unit that can be retrieved from disk efficiently. Another factor that should be taken into consideration for selecting the data layout is the geometry of individual data units; square groups of pixels provide better indexing than more elongated data units.

From [1], we know that for our SP-2 configuration the best I/O performance is achieved for blocks larger than 128 KB. We chose to partition the AVHRR data in tiles of 204x204 pixels. The tiles containing data from band one and two contain about 187 KB of data, including about 21 KB of geo-location data, used for navigating the pixels. For tiles containing data from band three, four and five, the tile size is about 270 KB, also including the geo-location data. To minimize disk seek time, all tiles with band 1-2 data are stored contiguously on disk, as are all tiles with band 3-4-5 data. This scheme allows queries with the common patterns described earlier to access multiple tiles sequentially from disk.

### 3.2 Declustering

The key motivation for declustering is to exploit disk parallelism by distributing database files across multiple processors and/or disks, aiming to minimize the number of data blocks fetched from a single disk and thereby minimizing query processing time. Numerous declustering methods have been reported in the literature. For multidimensional datasets, such declustering methods can be classified into two approaches: *grid-based* [3, 6, 9] and *graph-theoretic* [10, 20, 21]. Grid-based methods have been developed to decluster Cartesian product files, while graph-theoretic methods are aimed at declustering more general spatial access structures such as grid files [25] and R-trees [14]. A survey of declustering methods can be found in [23].

Since the Titan index is similar in behavior to an R-tree (see Section 4), we have adopted a graph-theoretic algorithm – Moon *et al.*'s *Minimax spanning tree* algorithm [21]. This algorithm was originally proposed for declustering grid files on a large disk farm, and has been shown to outperform Fang *et al.*'s Short Spanning Path algorithm [10] for that task. Other graph-theoretic algorithms such as Fang *et al.*'s Minimal Spanning Tree (MST) [10] and Liu *et al.*'s iterative partitioning (ITP) algorithms were not selected because (1) MST does not guarantee that the partitions are balanced in size, which means some partitions may be impractically large, and (2) ITP is based on a multi-pass Kernighan-Lin algorithm [17], which requires no fewer than $\mathcal{O}(N^2 \times p)$ operations, where $N$ is the number of data blocks and $p$ is the number of passes. Even though the number of passes, $p$, is usually low, the complexity is not polynomial bounded. The Minimax declustering algorithm requires $\mathcal{O}(N^2)$ operations and achieves perfectly balanced partitions (*i.e.*, each disk is assigned at most $\lceil N/M \rceil$ data blocks where $M$ is the number of disks).

To apply the Minimax algorithm, a complete graph is generated with data blocks as the vertices. Every edge has an associated cost, which represents the probability that two vertices (*i.e.*, data blocks) will be accessed together in a single query. To generate the edge costs, we have chosen the *proximity index* proposed by Kamel and Faloutsos [16]. The alternative we considered, *Euclidean distance* is suitable for point objects that occupy zero area in the problem space but does not capture the distinction among pairs of *partially overlapped* spatial objects of non-zero area or volume[2].

The key idea of the Minimax declustering algorithm is to extend Prim's minimal spanning tree algorithm [27] to construct as many spanning trees as there are disks in the disk farm, and then assign all the blocks associated with a single spanning tree to a single disk.

Prim's algorithm expands a minimal spanning tree by incrementally selecting the minimum cost edge between the vertices already in the tree and the vertices not yet in the tree. This selection policy does not ensure that the increment in the aggregate cost (*i.e.*, the sum of all edge weights inclusive to the group of vertices associated with the spanning tree) due to a newly selected vertex is minimized. Instead, the Minimax spanning tree algorithm uses a *minimum of maximum costs* policy. For every

[2]By partially overlapped objects we mean two disjoint $d$-dimensional objects whose projected images on at least any one of $d$ dimensions intersect.

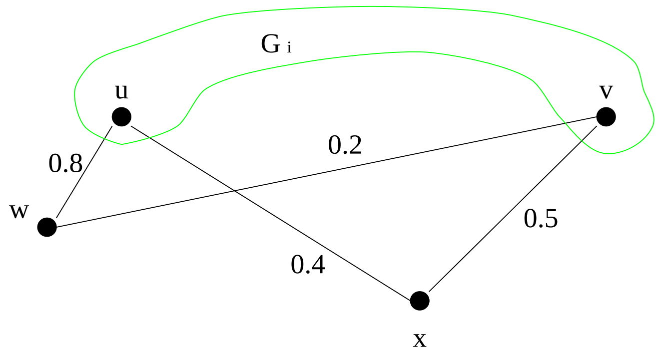


Figure 1: Illustration of expanding spanning trees

vertex that has not yet been selected, the algorithm computes a *maximum* of all edge weights between the vertex and the vertices already selected. The selection procedure picks the vertex that has the smallest such value.

For example, in Figure 1, the *minimum of minimum costs* policy will pick up the vertex $w$ and add it to the spanning tree $G_i$, because the weight of the edge $(w, v)$ is the minimum. However, this decision leads to putting the vertices $w$ and $u$, which are connected by an edge with a very heavy weight, in the same vertex group represented by the spanning tree $G_i$. On the other hand, the *minimum of maximum costs* policy will pick up the vertex $x$ and add it to the spanning tree $G_i$, because the weight of the edge $(x, v)$ is the minimum of maximum costs.

In summary, the Minimax algorithm:

- Seeds $M$ spanning trees by choosing $M$ vertices instead of choosing a single vertex.
- Expands $M$ spanning trees in round-robin fashion.
- Uses a *minimum of maximum costs* policy for edge selection, instead of a *minimum of minimum costs* policy.

A detailed description of the Minimax declustering algorithm is given in [21].

### 3.3 Clustering

In addition to maximizing disk parallelism by declustering, it is important to reduce the number of disk seeks by suitably ordering the data blocks assigned to a single disk. To achieve a good clustering of multidimensional datasets, we have considered clustering techniques based on Hilbert space-filling curves [22] and the Short Spanning Path (SSP) algorithm [10]. Both methods can be used to map multidimensional objects onto the one-dimensional disk space.

It is widely believed that the Hilbert curve achieves the best clustering among space-filling curves [15, 22]. The advantage of using space-filling curves is that the linearization mapping is of linear cost in the number of objects, while SSP is a quadratic algorithm. In [21], however, we have empirically shown that the SSP algorithm achieves better declustering than a Hilbert curve-based algorithm. Given that clustering is the dual of declustering, we conjecture that the SSP algorithm will achieve better clustering. In addition, since the remotely sensed image database is static and Minimax declustering is also a quadratic algorithm, the SSP algorithm was the selected as the clustering method.

Finding the shortest spanning path is NP-complete [11]. Therefore, the SSP algorithm employs a heuristic to generate a path that is short, but not necessarily the shortest. The algorithm works by first picking a vertex randomly from a given set of $N$ vertices. Now suppose we have generated a partial path covering $k$ vertices $v1, \ldots, v_k$, where $k < N$. Then pick another vertex $u$ randomly from the vertex set, and find a position in the path at which the vertex $u$ should be placed by trying the $k+1$ positions in the path, such that the length of the resulting path is minimized. Once again the *proximity index* is used to measure the distance between vertices.

## 4 Indexing Scheme

The Titan index contains spatio-temporal bounds and retrieval keys for the coarse-grained data blocks described in Section 3. For expediency and due to the relatively small number of blocks in the database, we have implemented the index as a simplified R-tree [14]. The index is a binary tree whose interior nodes are bounding quadrilaterals for their children. Leaf nodes in the index correspond to data blocks and contain spatial and temporal extent, meta-data such as sensor type and satellite number, and the position on disk for each data block. The position of a data block is described by a [disk,offset] pair.

The leaf nodes are arranged in a z-ordering [26] before the index is built. Sorting the leaves spatially allows access to the index as a range tree. Furthermore, it allows interior node keys in the index to better approximate the spatial extent of their children, and reduces the overlap between different interior node keys at the same level in the tree. As a result, searching the index becomes more efficient.

We use quadrilaterals instead of rectangles to achieve a better fit for spatial bounds. We chose a binary tree for its simplicity and because the entire index is held in memory, making disk efficiency in the index unimportant. This scheme provides adequate performance for our system, however data block indexing should be more closely examined for a large-scale implementation. We report no performance results for the index.

Using a coarse-grained index has several advantages. First, the index supports efficient retrieval of data from disks. Second, the index supports quick winnowing of large portions of the data base when presented with localized queries. Third, the index allows query previews that enable users to quickly refine their queries, without forcing large volumes of data to be retrieved from disks [5]. Finally, the index is extensible – it is easy to include data from other sensors without re-engineering the indexing scheme or re-indexing existing data.

## 5 Query Processing

As described in Section 2, Titan consists of a front-end and a set of back-end nodes. The front-end uses the query from a client to search the index described in Section 4, and identifies all data blocks that intersect with the spatial and temporal extent of the given query. The front-end also partitions the given query evenly among all back-end nodes.

For each of the data blocks selected by the index, the front-end generates a block request. A block request consists of a disk id and an offset, both obtained from the position information stored in the index, and a list of all back-end nodes whose assigned part of the result image intersects with the spatial extent of the associated data block. We refer to those back-end nodes as the *consumers* of the data block. The consumers of each data block can be easily identified, since the front-end has complete partitioning information for the result image.

The front-end then computes, for each back-end node, the number of data blocks the node is expecting to receive from each of the other back-end nodes. This information, along with a description of the assigned part of the result image and the set of associated block requests, is communicated to each back-end node.

When a back-end node receives all the information from the front-end, it computes a schedule. A schedule is a set of lists of block requests. Each list is identified by a [local-disk-id, back-end-node-id] pair. For every block request that a back-end node receives from the front-end, one of the consumers is chosen as its *representative*. The disk id of the block request, together with the node id of its representative, identifies the list in the schedule that the block request joins. The representative of a block request is chosen as follows. If the block request has only one consumer, that consumer is the representative. If the block request has multiple consumers, there are two cases:

1. if the local node is one of the consumers, the local back-end node is the representative;

2. otherwise, a back-end node is chosen randomly from the consumers of the data block to be the representative.

The schedule for a back-end node effectively partitions the set of block requests among the [disk, representative consumer] queues. This allows the local node to issue read requests to all its local disks in a balanced fashion, to make full use of the available disk bandwidth. The scheme also allows a back-end node to schedule read requests for each local disk, so that all back-end nodes get a fair share of the disk bandwidth, allowing the entire system to make global progress.

Once a schedule is generated, each back-end node asynchronously executes a loop whose body consists of five phases – a disk read phase, a block send phase, a block receive phase, a remote block read phase, and a block consume phase – as shown in Figure 2. Each back-end node also generates an empty result image for its assigned partition.

Our implementation is motivated by the observation that to answer a query under the Titan data declustering and query partitioning schemes, a significant amount of data is read from disks and transferred across the network. To hide the large latency incurred from I/O accesses and interprocessor communication, Titan back-end nodes issue multiple asynchronous operations to both the file system and the network interface, so that I/O, communication, and computation may all be 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 proceeding, each back-end node can process data blocks as they arrive from either its own local disks or the network.

During the disk read phase, the back-end issues as many asynchronous reads to the disks as possible. An asynchronous read requires pre-allocated buffer space to hold the returned data. The number of asynchronous reads that can be issued during each execution of this phase is thus limited by the number of available buffers. 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 checks for completion of asynchronous sends issued in the block consume phase, which will be described shortly. The sends are for data blocks that reside on local disks, but are required for processing on other nodes. When all asynchronous sends for a data block complete, the buffer space is released. In the block receive phase, the back-end node posts multiple asynchronous receives to receive data blocks to be processed on the local node but stored on other nodes. A buffer needs to be reserved for each pending receive.

```
while (not all activities are done)
    /* disk read phase */
    issue as many asynchronous disk reads for blocks as possible;
    /* block send phase */
    check all pending block sends, freeing send buffers for completed ones;
    /* block receive phase */
    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;
    /* remote block read phase */
    check pending disk reads that retrieve blocks to be processed only by remote back-end nodes, and
    for each completed one, generate asynchronous sends to the remote nodes;
    /* block consume phase */
    if (a block is already available for processing)
        process the block - perform mapping and compositing operations for all readings in the block;
    else
        check pending disk reads for blocks used by the local node, and possibly by remote back-end
        nodes as well, and for the first completed read found, if any :
            generate asynchronous sends to the remote consuming nodes;
            process the block;
    endif
endwhile
```

Figure 2: Main loop for overlapping computation, I/O and communication.

In the remote block read phase, disk reads issued for data blocks that reside on the local disks but are only processed by remote nodes are checked for completion. Asynchronous sends are generated for all such disk reads that have completed.

In the block consume phase, the back-end node processes a ready data block that is obtained either from the network or from a local disk. If the ready data block arrived from the network, the node performs the required mapping and composition operations for the block, and exits the block consume phase. Otherwise, the node polls the pending disk reads issued for blocks to be processed locally, and possibly by remote back-end nodes as well. If a pending disk read has completed, an asynchronous send is posted for each remote consumer of the data block. In addition, the ready data block is processed, and the block consume phase is exited. At most one ready data block is processed within the block consume phase per iteration. This policy prevents the system from failing to monitor the various outstanding asynchronous operations, while processing many ready data blocks.

# 6 Experimental Results

Titan currently runs on an IBM SP-2 at the University of Maryland. The SP-2 consists of 16 RS6000/390 processing nodes (so-called **thin** nodes), running AIX 4.1.4. Titan dedicates one node for the front-end and uses the other 15 nodes as the back-end. The current database stores approximately two months of AVHRR Level 1B Global Area Coverage data, containing about 24 GB of data stored on 60 disks distributed across the 15 back-end nodes. Titan uses the IBM MPL library for interprocessor communication, and is compiled with the IBM C and C++ compilers, mpcc and mpCC, version 3.1, with optimization level -O2.

## 6.1 Declustering and Clustering

In this section, we evaluate the performance of the Minimax spanning tree algorithm and the Short spanning path algorithm used as declustering and clustering methods, respectively. We compare these methods with random block assignment using both static measurements and simulation experiments with synthetically generated query sets.

### 6.1.1 Static measurements

We have selected the following static measures to evaluate the declustering and clustering methods:

- The number of $k$-nearest neighbor blocks placed on the same disk, for declustering.
- The aggregate probability that any pair of adjacent data blocks are fetched together, for clustering.

These measures depend only on the actual data placement and are independent of the distribution, sizes and shapes of queries.

| $k$-nearest neighbors | 1 | 5 | 15 | 30 | 59 |
|---|---|---|---|---|---|
| Random assignment | 923 | 4643 | 13969 | 27933 | 55190 |
| Minimax declustering | 280 | 1848 | 6434 | 13586 | 28832 |
| Improvement (%) | 70 | 60 | 54 | 51 | 48 |

Table 1: The number of $k$-nearest neighbor blocks assigned to the same disk

We counted the number of $k$-nearest neighbor data blocks assigned to the same disk unit, varying $k$ from 1 to 59, since the total of 55,602 data blocks were distributed over the 60 disks on the 15 SP-2 nodes that make up the back-end of Titan. The results are summarized in Table 1 for some of the $k$ values. The closer a pair of blocks are to each other, the higher the chance that they are accessed together. Therefore, the reduction of 48 to 70 percent in this measure indicates a potential substantial performance improvement through declustering.

We also computed the aggregate probability that any pair of adjacent data blocks on the same disk unit are fetched together. More precisely, this value is measured as $\sum_{i=1}^{N-1} proximity\_index(Block_i, Block_{i+1})$, where $N$ is the number of blocks assigned to a disk and $Block_i$ and $Block_{i+1}$ are a pair of adjacent blocks on the disk. We call this measure the *probability path length* of $\{Block_1, \ldots, Block_N\}$ on the disk. A high probability path length indicates that data blocks are clustered well on the disk and hence will require a small number of disk seeks for retrieval. When the Short spanning path algorithm was used to cluster data blocks on each disk, the average probability path length was 23.7, which is almost twice as high as the value of 13.3 when random block assignment was used.

#### 6.1.2 Simulation with synthetic queries

### Metrics of Interest

The metrics of interest are block transfer time and seek time. The models of these metrics are formally defined as follows, and we call the metrics *model block transfer time* and *model seek time*, respectively.

DEFINITION 1 *The* **model block transfer time** *of a query $q$ is defined as* $\max_{i=1}^{M}\{N_i(q)\}$, *where $M$ is the number of disks used and $N_i(q)$ is the number of data blocks fetched from disk $i$ to answer the query $q$.*

Since the disks are assumed to be independently accessible, Definition 1 implies that the time required to fetch the blocks in the answer set of the query $q$ is $\max_{i=1}^{M}\{N_i(q)\}$ units, with each unit being the

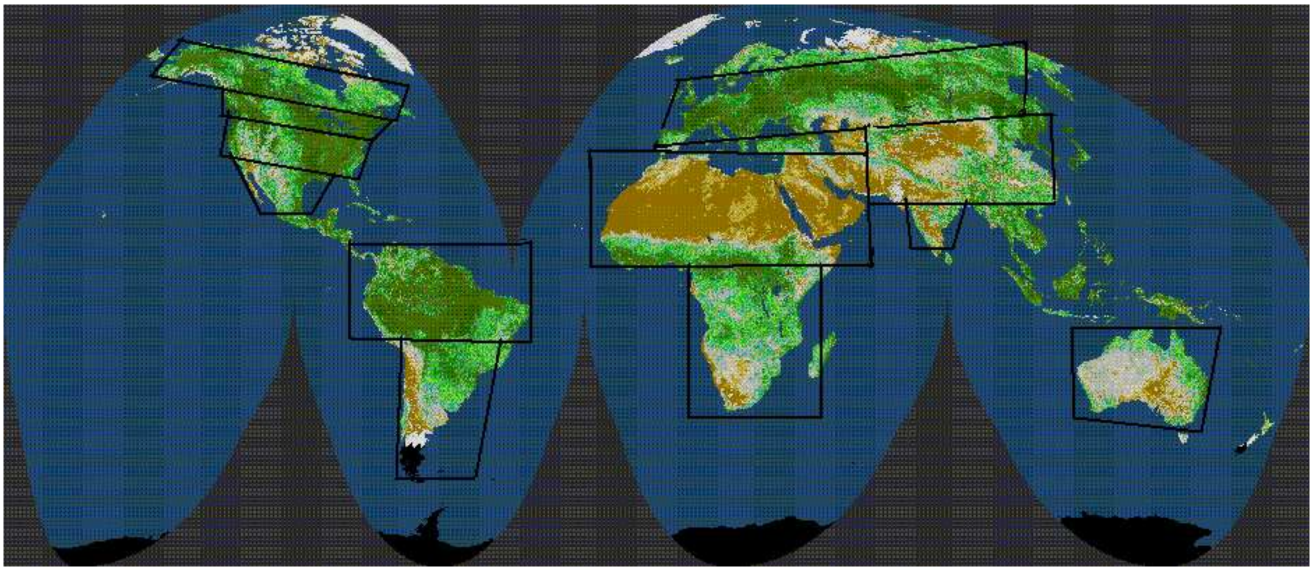

Figure 3: Bounding boxes covering 12 land masses

time required for one disk access to retrieve a block. Ignoring the effects of disk caching, the maximum number of data blocks fetched from the same disk (*i.e.*, $\max_{i=1}^{M}\{N_i(q)\}$) is considered a plausible measure of the actual block transfer time [6, 21].

DEFINITION 2 *A cluster of blocks is a group of data blocks that are contiguous on disk.* [3] *Then, given a query $q$, the* **model seek time** *is defined to be the number of clusters in the answer set of $q$.*

The metric given in Definition 2 was originally proposed in [15] and used to analyze the clustering properties of space-filling curves [15, 22]. As was pointed out in [15], small gaps between fetched blocks are likely to be immaterial. Therefore, later in this section we use the total distance to be traveled by the disk arm, as well as the model seek time, to evaluate the clustering scheme.

**Experimental results**

Based on the common query patterns identified by earth science researchers [7, 12, 18], we generated synthetic queries that uniformly cover the land masses of the world. We divided the land masses into 12 disjoint regions as shown in Figure 3. The synthetic queries are all 3-dimensional, including a temporal dimension (*e.g.*, 60 days), and the sizes of queries are governed by a selectivity factor ($0 < r < 1$). The selectivity factor $r$ denotes the percentage of the total area (in space and time) that the synthetic query

[3] Contiguous data blocks may be considered to have contiguous logical block numbers, assuming that logical block numbers represent the relative locations of physical data blocks.

| Selectivity | 1 percent | | | 10 percent | | | 20 percent | | |
|---|---|---|---|---|---|---|---|---|---|
| Declustering | Random | Minimax | Impr.(%) | Random | Minimax | Impr.(%) | Random | Minimax | Impr.(%) |
| Land region A | 10.9 | 7.9 | 28 | 27.6 | 20.6 | 25 | 38.3 | 28.5 | 26 |
| Land region B | 10.4 | 7.6 | 27 | 23.4 | 18.7 | 20 | 31.6 | 25.8 | 19 |
| Land region C | 11.2 | 8.1 | 27 | 26.8 | 22.0 | 18 | 36.5 | 31.0 | 15 |
| Land region D | 8.6 | 6.1 | 29 | 19.1 | 14.7 | 23 | 25.4 | 20.1 | 21 |
| Land region E | 55.3 | 46.3 | 16 | 119.9 | 105.0 | 12 | 150.4 | 136.0 | 10 |
| Land region F | 65.5 | 56.3 | 14 | 145.1 | 129.1 | 11 | 182.8 | 168.3 | 8 |

Table 2: Model block transfer time

| Selectivity | 1 percent | | | 10 percent | | | 20 percent | | |
|---|---|---|---|---|---|---|---|---|---|
| Clustering | Random | SSP | Impr.(%) | Random | SSP | Impr.(%) | Random | SSP | Impr.(%) |
| Land Region A | 7.8 | 7.5 | 4 | 19.9 | 18.2 | 9 | 27.4 | 24.0 | 12 |
| Land Region B | 7.6 | 7.3 | 3 | 18.4 | 16.8 | 9 | 25.1 | 21.7 | 13 |
| Land Region C | 8.1 | 7.7 | 4 | 21.3 | 18.7 | 12 | 29.5 | 24.5 | 17 |
| Land Region D | 6.0 | 5.8 | 4 | 14.3 | 13.1 | 9 | 19.8 | 17.2 | 13 |
| Land Region E | 43.7 | 35.5 | 19 | 93.5 | 51.7 | 45 | 117.1 | 49.8 | 57 |
| Land Region F | 52.4 | 44.2 | 16 | 110.7 | 64.2 | 42 | 136.4 | 61.9 | 55 |

Table 3: Model seek time

covers from the total area of the land region (also in space and time). For example, a spatio-temporal query into a region of size $L_{Lat} \times L_{Long} \times L_{Time}$ requires a query of size $L_{Lat}\sqrt[3]{r} \times L_{Long}\sqrt[3]{r} \times L_{Time}\sqrt[3]{r}$ to achieve a query selectivity of $r$. To simulate processing the synthetic queries, we accessed the Titan index described in Section 4 and computed the model block transfer time and model seek time for each of the queries, without retrieving the data blocks. In the experiments, $r$ was varied from 1 to 10 to 20 percent.

Table 2 and Table 3 show the experimental results from some of the individual land regions, with three query selectivities. For each land region and each query selectivity, we averaged the model block transfer time and model seek time over 500 synthetic range queries. In Table 2, for each of three selectivities, the first two columns show the average model transfer time with random block assignment and the Minimax declustering method, respectively. The third column displays the improvement shown by the Minimax method. The model transfer time for Minimax was always less than that of random assignment, and we observed 8 to 33 percent performance improvements for all the experiments.

In Table 3, for each of three selectivities, the three columns show the average model seek times and improvement ratio. Minimax declustering was applied for both the cases to isolate the effects of clustering blocks on each disk unit. That is, in both the cases, data blocks were declustered over 60 disk units by the Minimax algorithm. Then, for the *Random* case, data blocks were randomly arranged on each disk, while data blocks were clustered using the Short spanning path algorithm for the *SSP* case. SSP clustering achieved a 3 to 57 percent improvement (*i.e.*, reduction in disk seeks) relative to random assignment in the experiments. We also measured the average distance over which the disk arm needs to move for each synthetic query. The *disk arm travel distance* is modeled by *(highest_offset - lowest_offset)* among the blocks in the answer set of a given query. For all the experiments, we observed an 11 to 97 percent improvement in the disk arm travel distance for SSP clustering relative to random block assignment.

### 6.2 Preprocessing

A preprocessing phase must be executed once to load the entire database. Preprocessing includes segmenting raw AVHRR GAC files into blocks, building an index from the bounding boxes of those blocks, determining the placement of blocks onto disks using the declustering and clustering algorithms, and writing the blocks to the appropriate locations on disks. For the sample 24 GB data set, all preprocessing takes about four hours on the SP-2. Raw file segmenting, index construction, and running the declustering/clustering algorithm are tasks that were performed using a single node of the SP-2. Of these tasks, the majority of the time was spent in the declustering/clustering algorithm, which took about three hours to process over 55,000 data blocks. 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.

### 6.3 Query Processing

To evaluate the end-to-end performance of Titan, we ran a set of sample queries. Each of these queries generates a 10-day composite image, using the sensor measurements from bands 1 and 2. For simplicity, these sample queries are specified as rectangular boxes that cover several land masses, namely the United Kingdom and Ireland, Australia, Africa, North America, and South America. In addition, we also ran a global query, spatially covering the entire globe. The resolution of the output images for all the sample queries was 100/128 degrees of longitude by 100/128 degrees of latitude.

The sample queries were run under several different system configurations. The configurations allowed us to selectively disable one or more of the three system components, namely I/O, interprocessor communication, and computation, to observe how those components interact with each other.

First, we disabled all but one of the 15 back-end nodes, and resolved the global query with only one back-end node. Only data stored on the local disks of the one active back-end node is retrieved to answer the global query, so this effectively disables the interprocessor communication component of Titan. We then ran the global query with the computation component both turned on and off, in two separate experiments. With the computation turned on, the back-end node performs the mapping and compositing operations for the data blocks read from its local disks, as described in Section 2. With the computation turned off, the back-end throws away retrieved data blocks without performing any computation. Turning off the computation allowed us to measure the effective disk bandwidth seen by a back-end node. Comparing this effective disk bandwidth to the disk bandwidth measured by a disk benchmark program should reveal the amount of software overhead for I/O added by Titan. However, even with the computation turned on, the output images generated with this configuration are not complete, because only about 1/15th of the data stored in the database is retrieved.

Figure 4 shows the results for answering the global query on one node. The left and right bars show the execution times with the computation turned off and on, respectively. With the computation turned on, the total time spent in performing computation is measured separately, and is shown in the right bar as the area labeled computation. The remaining time represents non-overlapped I/O.

To resolve the global query, the back-end node retrieved 619 data blocks, or 116.2 MB, and the effective disk bandwidth observed by the back-end node is 9.2 MB/sec. Comparing this value to the aggregate application-level disk bandwidth of 10.6 MB/sec achievable on an SP-2 node, as measured by a micro-benchmark using the UNIX filesystem interface [1], the results show that very little I/O overhead is incurred by Titan. The difference between the heights of the left bar and the I/O part of the right bar indicates the amount of overlap between disk operations and computation on the node. The overlap is about 8 sec, which means that about 65% of the time spent doing asynchronous disk reads overlaps with computation.

For the next experiment, we ran all the queries on all the back-end nodes. Table 4 shows the total amount of data read from disk and communicated between processors to resolve the sample queries.

To use all 15 back-end nodes, data blocks read from disks must be forwarded to their consumer back-end nodes, as described in Section 5. This effectively enables the interprocessor communication component of Titan. We ran the sample queries, with the computation turned on and off, as for the previous experiment. With the computation turned on, Titan back-end nodes perform the mapping and

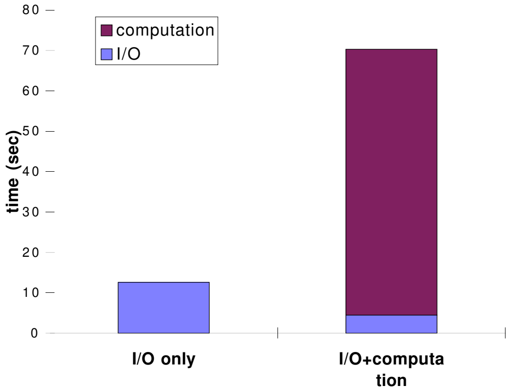


Figure 4: Resolving the global query using a single back-end node.

| sample query | total data read (# data blocks) | total data communicated (# messages) |
|---|---|---|
| global | 9263 (1700 MB) | 13138 (2500 MB) |
| Africa | 1087 (203.3 MB) | 2005 (374.9 MB) |
| North America | 2147 (401.5 MB) | 4313 (806.5 MB) |
| South America | 896 (167.6 MB) | 1740 (325.4 MB) |
| Australia | 375 (69.9 MB) | 997 (186.4 MB) |
| United Kingdom | 97 (18.1 MB) | 602 (112.6 MB) |

Table 4: The total number of data blocks read and communicated to resolve the sample queries.

compositing operations for the data blocks read from disks. Since all the data blocks needed to resolve the queries are retrieved and processed, Titan returns complete result images for that configuration. Therefore, the execution times obtained with the computation turned on are the end-to-end results for completely resolving the sample queries. On the other hand, with the computation turned off, Titan back-end nodes read the necessary data blocks from disks and forward the data blocks to the consumer back-end nodes, but the data blocks are discarded by the consumers.

Figure 5 shows the execution times for resolving the sample queries using all 15 back-end nodes, with the computation turned on and off. In the figure, each query is shown with three bars. For comparison, the disk read time, using an estimated bandwidth of 10 MB/sec per back-end node, is plotted for each query in the leftmost bar. The middle and right bars show, respectively, the execution times with the computation turned off and on, broken down into I/O, interprocessor communication and computation components.

By comparing the heights of the I/O parts of the two leftmost bars for each sample query, we see

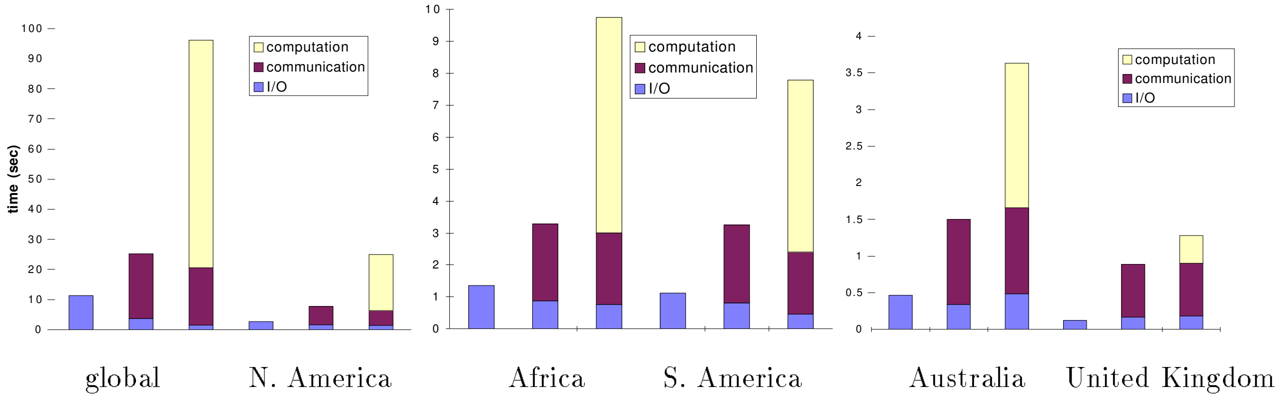


Figure 5: Resolving the sample queries with 15 back-end nodes.

that when the query is large enough a significant part of the asynchronous disk read time is overlapped with interprocessor communication. When the query is small, however, each back-end node only reads a few data blocks from its disks, so cannot achieve maximum disk bandwidth. That is why the estimated I/O times (at 10 MB/sec) for the Australia and United Kingdom queries are less than the measured times left for performing the non-overlapped I/O.

The rightmost bars in Figure 5 also show that the computation component does not overlap well with the other components. However, since the previous experiment showed that disk reads overlap quite well with the computation, we can conclude that the interprocessor communication component does not overlap well with the computation. We have found that the lack of overlap between communication and computation on an SP-2 processor is mainly because of the current implementation of the IBM MPL communication library. In MPL, the processor participates in the communication protocol to copy messages between user space and the network adapter. This memory copy operation is a major memory bandwidth bottleneck on a thin SP-2 node, leaving no time for the processor to perform computation while interprocessor communication takes place [30]. Snir *et al.* did, however, report that better overlap between computation and communication can be achieved with wide SP-2 nodes [30], which have higher memory bandwidth.

## 7 Evaluation

The experimental results presented in Section 6 show that Titan delivers good performance for both small and large queries. In particular, Titan provides interactive response times for local queries. The declustering and clustering schemes allow Titan to effectively utilize the high aggregate I/O bandwidth available from the disk farm. Good data placement coupled with a post-processing algorithm that overlaps I/O, interprocessor communication,and computation causes Titan query processing to become mainly a computation-bound task. However, there is still room for improvement.

Our current implementation uses an equal partitioning of the output image to partition the work load among all back-end nodes. Data blocks retrieved by a back-end node must be forwarded to all consuming back-end nodes, resulting in a large amount of interprocessor communication, which has become a major performance bottleneck. The communication problem becomes more pronounced when small queries are resolved, or if more back-end nodes are to be employed by the system. The problem worsens in those cases because when the spatial extent of each sub-image is comparable to, or smaller than, that of a single data block, each data block is very likely to intersect with more than one sub-image, and therefore must be forwarded to multiple back-end nodes. As can be seen in Table 4, on average each data block retrieved from disks on one back-end node is sent to about 1.4 other nodes for the global query, about 1.8 other nodes for the Africa query, and about 6.2 other nodes for the United Kingdom query.

The communication 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 data blocks a back-end node requires to compute its portion of the output image to be retrieved from disks on other back-end nodes. The problem is amplified on the IBM SP-2 where the experiments were run, because of lack of overlap between communication and computation.

Another performance issue is that an equal partitioning of the output image across the back-end nodes does not correspond to an equal partitioning of the input data to be processed on each back-end node. This is particularly true when the data blocks are not uniformly distributed within the entire spatial extent of the database. The AVHRR satellite data that Titan currently stores has this property. Table 5 shows, for each of the sample queries from Section 6.3, the minimum and maximum numbers of data blocks read and processed by a back-end node. As the table shows, the declustering algorithm achieves reasonable I/O load balance across the back-end nodes, but the workload partitioning scheme results in a very imbalanced computational workload among the back-end nodes. The back-end node with the most data blocks to process is therefore likely to becomes a bottleneck during query processing.

We are currently working on a new scheme for partitioning computational workload, based on the declustering of the input image. The proposed scheme has each processor process all data blocks retrieved from its local disks in response to a query. This effectively requires the output image to be

| query | data blocks read | | data blocks processed | |
|---|---|---|---|---|
| | min | max | min | max |
| global | 599 | 632 | 634 | 1095 |
| Africa | 65 | 79 | 117 | 151 |
| North America | 137 | 151 | 172 | 430 |
| South America | 50 | 66 | 89 | 148 |
| Australia | 19 | 28 | 56 | 80 |
| United Kingdom | 5 | 9 | 35 | 48 |

Table 5: The number of data blocks read and processed by the back-end nodes.

replicated, and then combined across all processors in a final step after all data 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 a partitioning scheme should also improve computational load balance, since the workload of each processor will be directly proportional to the amount of data that is stored on its local disk. With the help of a good declustering algorithm, such as Minimax, the new scheme should produce good computational load balance.

## 8 Conclusions and Future Work

We have presented the design and evaluation of Titan, a high performance image database for efficiently accessing remotely sensed data with spatial non-uniformity. Titan partitions the data into coarse-grained chunks, and distributes the chunks across a disk farm using declustering and clustering algorithms. The system consists of both a front-end, for doing query partitioning, and a back-end, for performing data retrieval and post-processing. The back-end runs in parallel, and coordinates I/O, interprocessor communication and computation over all processors to allow for overlap among all three operations. The experimental results show that Titan provides good performance for queries of widely varying sizes.

We are currently investigating techniques for efficiently handling multiple concurrent queries into the image database. 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. Data reuse refers to scheduling processing of queries that overlap in space and/or time to achieve good system throughput.

We also plan to add support for tertiary storage systems. In practice, terabytes of data would be stored in the database, far exceeding the capacity of a disk farm of any reasonable size. Data would need to be placed on tapes for permanent storage, but staged onto disks for query processing. To provide good performance, we will have to extend all the optimization techniques currently used to provide high performance in Titan to include support for a tertiary storage system.

## 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 *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] Ling Tony Chen and Doron Rotem. Declustering objects for visualization. In *Proceedings of the 19th VLDB Conference*, pages 85–96, Dublin, Ireland, 1993.

[4] D. DeWitt, N. Kabra, J. Luo, J. Patel, and J-B Yu. Client-server Paradise. In *Proceedings of the* $20^{th}$ *VLDB Conference*, 1994.

[5] 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.

[6] H. C. Du and J. S. Sobolewski. Disk allocation for Cartesian product files on multiple-disk systems. *ACM Transactions on Database Systems*, 7(1):82–101, March 1982.

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

[8] Michael Shapiro et al. *GRASS 4.0 Programmer's Manual.* US Army Construction Engineering Research Laboratory, 1992.

[9] Christos Faloutsos and Pravin Bhagwat. Declustering using fractals. In *the 2nd International Conference on Parallel and Distributed Information Systems*, pages 18–25, San Diego, CA, January 1993.

[10] M. T. Fang, R. C. T. Lee, and C. C. Chang. The idea of de-clustering and its applications. In *Proceedings of the 12th VLDB Conference*, pages 181–188, Kyoto, Japan, 1986.

[11] Michael R. Garey and David S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, New York, 1979.

[12] S. Goward, B. Markham, D. Dye, W. Dulaney, and J. Yang. Normalized difference vegetation index measurements from the Advanced Very High Resolution Radiometer. *Remote Sensing of Environment*, 35:257–77, 1991.

[13] NSF/ARPA Grand Challenge Project at the University of Maryland for Land Cover Dynamics, 1995. *http://www.umiacs.umd.edu:80/research/GC/.*

[14] 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.

[15] H. V. Jagadish. Linear clustering of objects with multiple attributes. In *Proceedings of the 1990 ACM-SIGMOD Conference*, pages 332–342, Atlantic City, NJ, May 1990.

[16] Ibrahim Kamel and Christos Faloutsos. Parallel R-trees. In *Proceedings of the 1992 ACM-SIGMOD Conference*, pages 195–204, San Diego, CA, June 1992.

[17] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, February 1970.

[18] S. Liang, L. Davis, J. Townshend, R. Chellappa, R. Dubayah, S. Goward, J. JaJa, S. Krishnamachari, N. Roussopoulos, J. Saltz, H. Samet, T. Shock, and M. Srinivasan. Land cover dynamics investigation using parallel computers. In *Proceedings of the 1995 International Geoscience and Remote Sensing Symposium, Quantitative Remote Sensing for Science and Applications.*, pages 332–4, July 1995.

[19] S. Liang, S. Goward, J. Ranson, R. Dubayah, and S. Kalluri. Retrieval of atmospheric water vapor and land surface temperature from AVHRR thermal imagery. In *Proceedings of the 1995 International Geoscience and Remote Sensing Symposium, Quantitative Remote Sensing for Science and Applications.*, pages 1959–61, July 1995.

[20] Duen-Ren Liu and Shashi Shekhar. A similarity graph-based approach to declustering problems and its application towards parallelizing grid files. In *the 11th Inter. Conference on Data Engineering*, pages 373–381, Taipei, Taiwan, March 1995.

[21] 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.

[22] Bongki Moon, H.V. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of Hilbert space-filling curve. Technical Report CS-TR-3611 and UMIACS-TR-96-20, University of Maryland, College Park, MD, March 1996. Submitted to IEEE Transactions on Knowledge and Data Engineering.

[23] Bongki Moon and Joel H. Saltz. Scalability analysis of declustering methods for Cartesian product files. Technical Report CS-TR-3590 and UMIACS-TR-96-5, University of Maryland, College Park, MD, April 1996. Submitted to IEEE Transactions on Knowledge and Data Engineering.

[24] S. Morehouse. The ARC/INFO geographic information system. *Computers and Geosciences: An International Journal*, 18(4):435–41, August 1992.

[25] J. Nievergelt and H. Hinterberger. The Grid File: An adaptive, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.

[26] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 181–190, Waterloo, Canada, April 1984.

[27] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(11):1389–1401, November 1957.

[28] C.T. Shock, C. Chang, L. Davis, S. Goward, J. Saltz, and A. Sussman. A high performance image database system for remote sensing. In *24th AIPR Workshop on Tools and Techniques for Modeling and Simulation*, Washington, D.C., October 1995.

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

[30] M. Snir, P. Hochschild, D.D. Frye, and K.J. Gildea. The communication software and parallel environment of the IBM SP2. *IBM Systems Journal*, 34(2):205–221, 1995.

[31] M. Ubell. The Montage extensible datablade architecture. In *Proceedings of the 1994 ACM-SIGMOD Conference*, page 482, May 1994.

[32] T. Vijlbrief and P. van Oosterom. The GEO++ system: An extensible GIS. In *Proceedings of the Fifth International Symposium on Spatial Data Handling*, pages 40–50, August 1992.