A N G E W A N D T E M A T H E M A T I K

U N D

I N F O R M A T I K

Workshop on Massive Geometric Data Sets (Massive2005)

Lars Arge[1], **Mark de Berg**[2], **Jan Vahrenhold**[3] **(eds.)**

[1]Department of Computer Science, University of Aarhus, Aabogade 34, DK-8200 Aarhus N, Denmark
[2]Department of Mathematics and Computing Science, TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, the Netherlands
[3]Westfälische Wilhelms-Universität Münster, Department of Computer Science, Einsteinstr. 62, D-48149 Münster, Germany

02/05 – I

U N I V E R S I T Ä T M Ü N S T E R

Preface

This booklet contains the abstracts of the 13 talks given at the *Workshop on Massive Geometric Data Sets* (Massive2005). Bringing together researchers from different areas but with a common interest in handling massive data sets, this informal workshop was held in connection with the *Twenty-First Annual ACM Symposium on Computational Geometry* on June 9, 2005, at the National Research Council Campus in Pisa, Italy.

We thank the Netherlands Organization for Scientific Research and the University of Münster for their financial support that made this workshop possible. We are also grateful to the *Istituto di Informatica e Telematica* of the *Consiglio Nazionale delle Ricerche* for their help with the local arrangements and for hosting the workshop.

Lars Arge, Mark de Berg, and Jan Vahrenhold

Organizing Committee

Lars Arge	University of Aarhus / Duke University	(workshop organization)
Mark de Berg	TU Eindhoven	(workshop organization)
Adriana Lazzaroni	IIT-CNR	(local arrangements)
Giuseppe Liotta	University of Perugia	(local arrangements)
Marco Pellegrini	IIT-CNR	(local arrangements)
Jan Vahrenhold	University of Münster	(workshop organization)

Contents

Space Complexity of Hierarchical Heavy Hitters in Multi-Dimensional Data Streams

John Hershberger* Nisheeth Shrivastava † Subhash Suri† Csaba D. Tóth‡

Abstract

Heavy hitters, which are items occurring with frequency above a given threshold, are an important aggregation and summary tool when processing data streams or data warehouses. Hierarchical heavy hitters (HHHs) have been introduced as a natural generalization for hierarchical and multi-dimensional data domains. An important application, for instance, involves inferring patterns in the stream of IP packets in the Internet, for the purpose of traffic engineering or network security. Each IP packet is mapped to a multi-dimensional point using its header fields, and the goal is to extract classification rules that account for a large fraction of the traffic. In geometric terms, the problem involves identifying rectangular boxes that contain a significant fraction of a stream of points, excluding those points counted in a smaller heavy hitter box.

Formally, consider a *stream* $\mathcal{S}$ of d-dimensional points and a family $\mathcal{B}$ of axis-aligned d-dimensional boxes. Both $|\mathcal{S}|$ and $|\mathcal{B}|$ are large—the points of the stream arrive online, and they are too numerous to be stored in memory; the boxes also are too numerous to be maintained explicitly and are defined implicitly by certain rules. The boxes in $\mathcal{B}$ are *partially ordered* by the containment relation: for two boxes B and B', we write $B' \prec B$ if $B' \subset B$. We define the frequency (or, population) of a box B to be the number of points in the stream that lie in B, namely, $|B \cap \mathcal{S}|$; we use the notation $\mathcal{S}$ to also denote the part of the stream seen so far. We are interested in identifying those boxes with frequency greater than $\phi|\mathcal{S}|$, for a given parameter $0 < \phi < 1$. In order to avoid redundancy, however, hierarchical heavy hitters are defined using the discounted frequency of a box. (Otherwise, all boxes containing a heavy box may be flagged as heavy even if they do not contain many additional points.) *Discounted frequencies* and *ϕ-hierarchical heavy hitters* (*ϕ-HHHs*) are defined recursively: the discounted frequency of B counts only those points that lie in B but not in another *ϕ-HHH* B' where $B' \prec B$. A box B is a *ϕ-HHH* if its discounted frequency exceeds $\phi|\mathcal{S}|$.

Hierarchical heavy hitters are natural and powerful constructs, but reliably estimating the discounted frequency of boxes has proved elusive. None of the known space-efficient data stream algorithms offer a worst-case guarantee on the approximation quality of the boxes they flag as *ϕ-HHH*. In this talk, we formalize the difficulty of computing true hierarchical heavy hitters and prove lower bounds on the space complexity of algorithms that compute them.

For streams of 1-dimensional data, we give an $\Omega(1/\phi^2)$ space lower bound for *any* algorithm, using an information-theoretic argument. To prove lower bounds for streams of multi-dimensional data and to establish stronger space bounds, we limit our discussion to a simple model of deterministic algorithms, which we call the *box frequency* model. In this model, an algorithm with space bound s is allowed s distinct counters, and each counter maintains the frequency of a box. We show that any single-pass deterministic scheme that computes *ϕ-HHHs* for d-dimensional data in the box frequency model with any bounded approximation guarantee must use $\Omega(1/\phi^{d+1})$ space. This bound is asymptotically tight as we can show a deterministic data stream algorithm (in the box frequency model) that computes *ϕ-HHHs* with constant approximation error, using $O(1/\phi^{d+1})$ memory.

*Mentor Graphics Corp., 8005 SW Boeckman Road, Wilsonville, OR 97070, USA, john_hershberger@mentor.com and (by courtesy) Department of Computer Science, University of California at Santa Barbara.

†Department of Computer Science, University of California at Santa Barbara, Santa Barbara, CA 93106, USA, {nisheeth, suri}@cs.ucsb.edu.

‡Department of Mathematics, Room 2-336, MIT, Cambridge, MA 02139, USA, toth@math.mit.edu.

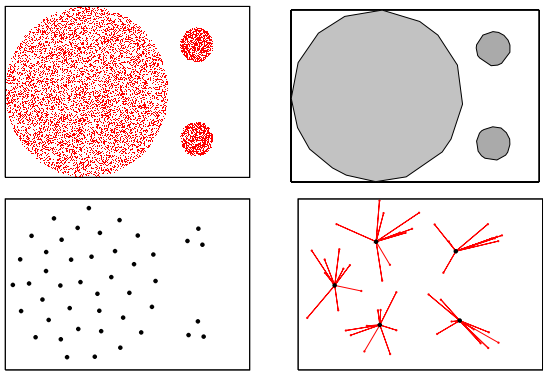
Cluster Hulls:
A Technique for Summarizing Spatial Data Streams

John Hershberger* Nisheeth Shrivastava† Subhash Suri†

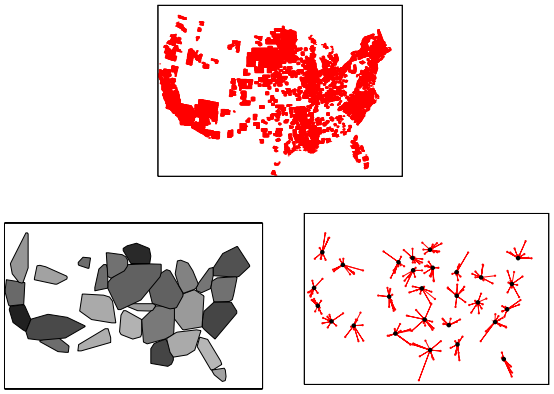
The extraction of meaning from data is perhaps the most important problem in all of science. Algorithms that can aid in this process by identifying useful structure are valuable in many areas of science, engineering, and information management. In particular, finding a simple characterization of a distribution known only through a collection of sample points is fundamental in many settings.

We consider the following problem: given an on-line, possibly unbounded stream of two-dimensional points, how can we summarize its spatial distribution or *shape* using a small, bounded amount of memory? We propose a novel scheme, called *ClusterHulls*, which represents the shape of the stream as a dynamic collection of convex hulls, with a total of at most m vertices, where m is the size of the memory. The algorithm dynamically adjusts both the number of hulls and the number of vertices in each hull to represent the stream using its fixed memory budget. Thus, depending on the input, the algorithm adaptively spends more points on clusters with complex (potentially more interesting) boundaries and fewer on simple clusters. Because each cluster is represented by its convex hull, the ClusterHull summary is particularly useful for preserving such geometric characteristics of each cluster as its boundary shape, orientation, and volume. Furthermore, since hulls are objects with spatial extent, we can also maintain additional information such as the number of input points contained within each hull, or their approximate *data density* (e.g., population divided by the hull volume). By shading the hulls in proportion to their density, we can then compactly convey a simple visual representation of the data distribution.

We implemented ClusterHulls and experimented with both synthetic and real data to evaluate its performance, comparing it against a data stream version of k-means clustering. In all cases, the representation by ClusterHulls appears to be more information-rich than k-means, even when the latter is enhanced with some simple ways to capture cluster shape. See the example figures below.



The top row shows input data consisting of three natural circular clusters (left) and the output of ClusterHulls (right) with $m = 45$. The bottom row shows two different outputs of k-means, with $m = 45$. Left: the result of computing $k = 45$ centers. Right: the result of computing $k = 5$ centers and for each center maintaining a random sample of 9 points to get a rough representation of the cluster geometry. The k-means algorithm does not give an accurate representation of the cluster boundaries in either case.



The data set shown at the top contains about 68,000 points corresponding to the locations of the *West Nile* virus cases reported in the US, as collected by the CDC and the USGS. The lower figures show the results of ClusterHulls (left) and k-means (right) for $m = 256$. The hulls generated by ClusterHulls are shaded according to their densities (darker regions are more dense).

*Mentor Graphics Corp., 8005 SW Boeckman Road, Wilsonville, OR 97070, USA, john_hershberger@mentor.com and (by courtesy) Department of Computer Science, University of California at Santa Barbara.

†Department of Computer Science, University of California at Santa Barbara, Santa Barbara, CA 93106, USA, {nisheeth,suri}@cs.ucsb.edu.

Using data stream algorithms for computing properties of large graphs *

Luciana S. Buriol
Dipartimento di Informatica e Sistemistica
Universitá di Roma "La Sapienza"
Via Salaria 113, 00198 Roma, Italy
buriol@dis.uniroma1.it

Debora Donato
Dipartimento di Informatica e Sistemistica
Universitá di Roma "La Sapienza"
Via Salaria 113, 00198 Roma, Italy
donato@dis.uniroma1.it

Stefano Leonardi
Dipartimento di Informatica e Sistemistica
Universitá di Roma "La Sapienza"
Via Salaria 113, 00198 Roma, Italy
Stefano.Leonardi@dis.uniroma1.it

Tobias Matzner
Fakultät für Informatik
Universität Karlsruhe
Karlsruhe, Germany
tobias.matzner@rechnerpost.de

ABSTRACT

The focus of this paper is on the practical use of data stream algorithms for monitoring statistical and topological properties of large graphs such as the webgraph. By webgraph we mean the directed graph generated from the link structure of webpages: each webpage is a node and each hyperlink is an arc in this graph. We study experimentally the application of the algorithm of Datar and Muthukrishnan [5] for maintaining the indegree rarity distribution and the density of low cardinality bipartite cliques in a graph read in a streamed fashion. The α-rarity of a stream is the ratio between the number of elements that appear exactly α times and the total number of different items in the stream. We present results and show that the approximated values are very close to the optima even when a low precision is requested.

1. INTRODUCTION

Data stream algorithms aim to maintain the underlying information of a stream of data, using small memory space. The data is processed on the fly, as it is generated, or it can also be read from second memory devices. Typical applications of data stream algorithms are originated from massive datasets such as network traffic measurements, telephone call records, biological datasets and atmospheric observations. In these applications is unnecessary or impractical to read data multiple times. In many cases, the data is not even stored. This paper focuses on a "new" natural application for data streams. We are interested in using data stream algorithms for monitoring statistical and topological properties of large graphs such as the webgraph. By webgraph we mean the directed graph generated from the link structure of webpages: each webpage is a node and each hyperlink is an arc in this graph. Likewise, sub-graphs can be generated from specific webpage collections such as blogs, online encyclopedias, online bookstores, the collection of webpages within a domain, and many others. The graph read in a streaming fashion considers each edge as an item and the stream is not required to be structured.

The main advantage of using data streams instead of exact algorithms is that the space used for managing and mining the stream is small, without resorting to external memory algorithms. Furthermore, results can be output anytime during the stream processing, not requiring that the whole data input be processed in advance. On the other hand, data stream algorithms do not provide exact values, but an approximation that depends on the precision required and the amount of resources we are willing to invest.

Several theorical results have been proposed in this new research field, some of them have not yet been implemented and experimented, some of them are not practical. In this paper we observe how a data stream algorithm behaves in practice for computing the indegree rarity distribution of a graph over the arc arrivals. More specifically, we maintain the distribution of the number of nodes that has a given indegree over the total number of different nodes seen in the stream so far. We use the algorithm proposed by Datar and Muthukrishnan [5] and show experimentally that the results are very close to the optima even when a low precision is requested. The original algorithm proposes the use of min-wise hash functions, whereas we use universal hashing [5]. This decision is due to the fact that computing min-wise hashing consumes about two orders of magnitude more time than universal hashing without providing better results in practice for the graphs we have tested.

When considering a specific structure in the data stream, other properties can be computed. For example, reading the stream in an adjacency list fashion, the same rarity algorithm can be used for estimating the density of minors such as small bipartite cliques.

*This work was partially supported by the EU within the 6th Framework Programme under contract 001907 "Dynamically Evolving, Large Scale Information Systems" (DELIS) and the FET Open Project IST-2001-33555 "COevolution and Self-organization in Dynamical Networks".

The indegree of webpages is an important measure of their popularity. The experimental observation of the indegree distribution has been the subject of seminal works aimed to characterize the structure of the webgraph [2, 3]. This study has also revealed a surprising number of dense subgraphs, specifically bipartite cliques, of moderately small size [8], considered as cores of hidden web communities.

In the next section we present the α-rarity algorithm of Datar and Muthukrishnan [5]. Section 3 describes the adaptations of the α-rarity algorithm for computing indegree rarity distribution, as well as for computing minors of small size. In section 4 we present experimental results for rarity of indegree e bipartite cliques of size three `(k3,3)` and for the minors `k(1,3)` and `k(2,3)`. We generalize the set of all minors mentioned above using the term `k(i,3)`, where i denotes the number of nodes in the graph that points to each node of a triple (set of three nodes). Comparison with the results of an optimal computation shows excellent practical results of our implementations. Section 4 also described the optimization of a an implementation of min-wise hash functions [10].

2. ESTIMATING RARITY OVER DATA STREAM WINDOWS

We use the $\alpha - rare$ algorithm of Datar and Muthukrishnan [5] for driving our experiments. Consider a stream of items a_i generated in a universe `U`=[1,..,n]. A stream is a set of m elements $a_1, a_2, ..., a_m$ such that $a_i \in U$. An item i is called $\alpha - rare$ if it appears exactly α times in the stream. Let's call $\#\alpha - rare$ the number of elements that appear exactly α times in the stream. Likewise, $\#distinct$ denotes de number of distinct items in the stream. The α-rarity ρ_α is defined as the ratio $\rho_\alpha = \frac{\#\alpha - rare}{\#distinct}$. In other words, the α-rarity of a stream is the measure of number of items that repeat exactly α times in the stream.

The algorithm proposed by Datar and Muthukrishnan [5] for computing the α-rarity of a stream uses min-wise hash functions. Min-wise independent permutation families are defined in [4]. Let S_n be the set of all permutations π of [1,...,n]. A permutation family `F` (subset of all permutations over [1..n]) is exactly *min-wise independent* if for any subset `X` of [1..n], and any $x \in X$, when π is chosen at random from `F` we have $Pr\{min\{\pi(X)\} = \pi(x)\} = \frac{1}{|X|}$. In other words, it is required that all elements of a given set `X` have an equal chance to become the minimum element of the image of `X` under π.

The referred algorithm uses only *O(log N + log u)* space, and *O(log log N)* per item processing time. It estimates ρ by $\hat{\rho} \in [1 \pm \epsilon]\rho + \epsilon\rho$ for a given fraction ϵ, with hight probability. The algorithm uses $h = 2\epsilon^{-3}p^{-1}log\tau^{-1}$ hash functions and two $|h|$-vectors, `min` and `count`, in main memory. Each position i of the vector `min` contains the minimum value found so far by the min-wise hash h_i, whereas `count` maintains, for each position i, the number of times that the current minimum min-wise value was found. For each value of α, $\hat{\rho}$ is computed as the ration between the number of counters that have exactly value α and h.

A slightly different algorithm is proposed for computing the α-rarity of a windowed stream. E.g, Considering a fix window size equal to `W`, the algorithm maintains the α-rarity of the last `W` items seen in the stream. In this case, to maintain the current minimum for each hash function is a bit more complicated. Instead of the $|h|$-vector `min`, a linked list `LM` is used. For each min-wise hash function, all non dominated minima are maintained, also with indication of the number and the time of the occurrences of that minimum. For non dominated minima we mean the min-wise hash values that are larger than the current minimum, but were generated more recently and belong to the current window. The number of times that each minimum was found is stored in another linked list `LT`, instead of using the $|h|$-vector `count`. Each element in this list contain information about the time the correspondent minimum was found. For each new item processed in the stream, the lists are updated twice. First the items no longer in the lists are removed (checking `LT` info) and next, the lists are update with the new element.

3. COMPUTING THE RARITY DISTRIBUTION OF INDEGREE AND K(I,3)

In this sections we describe how the not-windowed algorithm described in the previous section is adapted to compute the α-rarity algorithm for computing the indegree and `k(i,3)` rarity distributions of a graph.

Considering an arbitrary scan of a digraph G=(V,E), where V is the set of nodes and E is the set of edges of this graph. The items of the stream, in this case, are the list of edges. The α-rarity of the stream can be understood as the percentage of nodes that has indegree α. With the underlying data stored for estimating α, we can compute the α- rarity for any $\alpha_i < \alpha$. So, computing the rarity distribution for an α large enough, we obtain the rarity indegree distribution of the graph considering any value α. The rarity distribution can be computed for a complete stream, or for the window of the last `W` items seen in the stream.

When considering some structure in the stream, other properties can be computed. For example, reading the stream in an adjacency list fashion, the same rarity algorithm can be used for approximating the density of minors, such as small bipartite cliques. Such kind of structured data stream can be found naturally on some applications. For example, during a crawling process, each current fetched page is parsed and all outgoing links of this page identified. It is exactly this kind of order that we are considering here.

Now we describe the adaptation of the α-rarity algorithm for computing the `k(i,3)` rarity distribution on a graph. The digraph G is read in a streaming fashion, e.g., all outgoing links of a node $i \in V$ are read in sequence. The lists of outgoing edges are not required to be in any specific order, as well as the edges intern to each list. So, for each node u, for each outgoing edge $(\overrightarrow{u,a}) \in OUT(u)$, triples are calculated considering node a and all combinations two by two of the head-node of the edges seen so far in $OUT(u)$. E.g, triples `(a,b,c)` are calculated for nodes $b, c \in OUT(u)$ considering edges $(\overrightarrow{u,b})$ and $(\overrightarrow{u,c})$ previously located in $OUT(u)$ than $(\overrightarrow{u,a})$. So, the overall number of triples (T) of the graph is the sum of the combination three by three of head-nodes of the outgoing list of each node $u \in V$, e.g., $T = \sum_{i=1}^{i} \frac{d_i*(d_i-1)*(d_i-2)}{6}$ where $d_i = |OUT(i)|$ is the outdegree of the node i. We require to store in main memory the whole outgoing adjacency list of the current node.

4. EXPERIMENTAL RESULTS

In this section we describe the experimental results we performed using the α-rarity algorithm. The algorithms were coded in g++ version 3.3.2. The experiments were conducted in a Intel Pentium IV, with 1GB RAM, running Mandrake 9.0.

Due to the excessive computational time spent by min-wise hash functions, we use universal hash functions instead. We used the hash function (`hash31`) and the random number generator (`prng_int`) from the online available codes from the MassDAL group of Rutgers (*http://www.cs.rutgers.edu/ muthu/massdal-code-index.html*).

The implementation of all algorithms presented in this section, as well as the optimized version of min-wise hash functions, are available by e-mail request.

We start describing the optimization applied to the online available implementation of min-wise functions. Next, we describe the datasets we used and afterwards present results for the indegree distribution for the entire graph view and for the windowed case. We conclude our experiments with some results for the α-rarity algorithm applied for the `k(i,3)` case.

4.1 Optimization of Min-wise hashing

We use an optimized version of Jerry Zhao's implementation [10] of an approximate restricted min-wise independent permutation family proposed by Alon et al. [1]. In practice, one can allow certain relaxations. One of the relaxed definition is approximate restricted min-wise permutation family. The implementation uses a linear feedback shift register with a irreducible polynomial as feedback rule as described in [1] to generate hash values. A certain number of those hashing functions is then used to compute the permutation value [4]. The time to compute a permutation using this implementation is $O(n(loglog\ n + log\ k + log\frac{1}{\epsilon}))$ using a $\frac{1}{\epsilon}$ away approximate min-wise permutation for $[0..n]$ restricted by 2^k-wise independence. The space cost of the original implementation is $O(loglog\ n + log\ k + log\frac{1}{\epsilon})$. Our modification is to store intermediate results during the calculation of the register values instead of starting over from the beginning every time the hash function is called. For this purpose we memorize blocks of consequent values long enough to be able to resume calculation from this point i.e. of the length of the feedback rule. Thus the space cost is increased but the software remains usable on a normally equipped system. The calculation time however – together with slight changes to avoid expensive functions – was reduced by a factor of orders of magnitude.

4.2 Datasets

We conducted our experiments on streams of Wikipedia graphs. A graph of this type is generated from the link structure of the online and free-content encyclopedia Wikipedia (*www.wikipedia.org*). It started in January 15, 2001 with a few English articles. Four years later, Wikipedia has more than 1 million articles, available in more than 100 languages: The English version is the largest one, with about half million articles. Following the definition of a webgraph, each article is a node, and each hyperlink is a link in the graph. One graph is extracted for each language.

There are a few reasons why we are using wikipedia graphs for tests:

- *independency of external links*: wikipedia articles link mainly to articles on the same dataset.
- *variety of graph sizes*: it can be collected one graph by language, and the graph dimensions vary from a few hundred pages up to half million pages.
- *generation on time*: wikipedia provides time information associated with nodes. Moreover, it provides old information: time information of data of creation and dates of modification for each page on the dataset.
- *available on dumps*: it can be dumped as mysql tables, instead of been crawled. New dumps are provided almost weekly.

We generate streams of edges of the wikipedia graphs following their generation on time. In our experiments we use the graphs `wikiEN`, `wikiDE`, `wikiFR`, `wikiIT`, `wikiPT` from the datasets extracted from the English, German, French, Italian and Portuguese languages, respectively. The graphs were obtained from an old dump of July 2004. Due to space restrictions, we limited the presentation of experimental results in this extended abstract to the `wikiEN` and `wikiPT` graphs. Some comments are added about the experimental results on the other three graphs. Graph `wikiPT` contains 8,131 nodes and 48,168 edges, while graph `wikiEN` is two orders of magnitude larger containing 286,754 nodes and 4,065,530 edges.

4.3 Rarity Indegree Distribution

This subsection describes the results obtained using the α-rarity algorithm for the entire stream (unbounded) and the windowed cases. Figure 1 presents results for the rarity for the unbounded case, using 1000 hash functions. The lines are plot for a logarithmic number of indegree values. The plot omits results for indegree higher than 63 for the sake of clarity of the figure, but a complete plot would present additional lines on the bottom of the figure, appearing on increasing order of the number of edges processed.

For a good approximation, a larger number of hash functions are required. For example, if we set $\epsilon = p = \tau = 0.1$, 10,000 hash function are required. For $\epsilon = p = \tau = 0.2$, just 437 hash functions are needed. But we observed, that even with a small number of hash functions, the results are close to the optima. Figure 2 presents results when using only 100 hash functions.

For the windowed case, similar quality of results were found. Figure 3 presents results for windows of 100,000 items, estimated using 100 hash functions.

We also found good approximation when using the α-rarity algorithm for computing the rarity distribution of `k(i,3)` on the graph. Results for i=1,2,3 are plot in Figure 4. The plot is in log scale to be able to visualize all three distributions clearly on the same plot. Usually the number of `k(1,3)` $\gg$ `k(2,3)` $\gg$ `k(3,3)`. The difference between this values decrease with the increase of i. Observe, for example, the precision on results between the estimated and exact computation of `k(1,3)` and `k(2,3)`. Since `k(1,3)` is found many more times than `k(2,3)`, the results are more accurate. For values of $i > 4$ we did not plot for the sake of clarity of the plot, but the precision on the results decrease

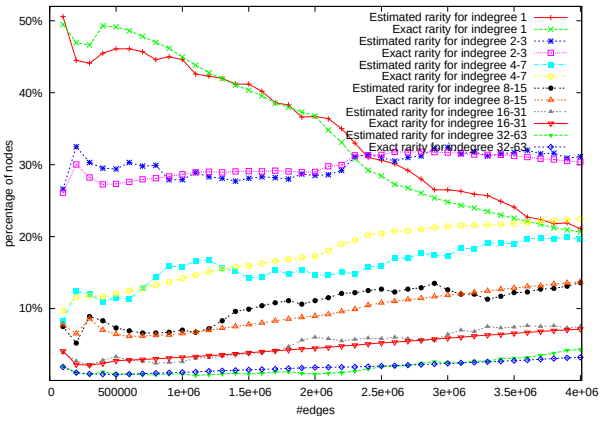


Figure 1. Estimated and exact indegree rarity distributions computed for edges arrivals of graph `wikiEN`. The estimation makes use of 1000 universal hashing functions. Values are presented to α up to 63, presented as $\log_2$ plot. This plot presents the percentage of nodes with a given indegree (y-basis) considering the amount of edges processed so far (x-basis). Results are plot every 100,000 items processed.

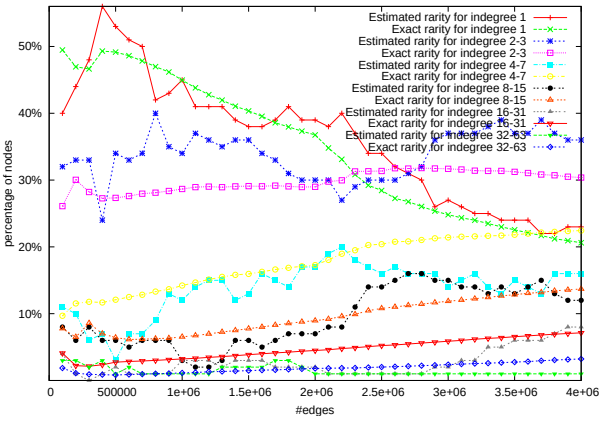


Figure 2. Estimated and exact indegree rarity distribution computed for edges arrivals of graph `wikiEN`. The estimation makes use of 100 universal hashing functions. Values are presented to α up to 63, presented as $\log_2$ plot. This plot presents the percentage of nodes with a given indegree (y-basis) considering the amount of edges processed so far (x-basis). Results are plot every 100,000 items processed.

with the increase of i. As expected, we have less precision for computing $\hat{\rho}$ of α-rare elements that occur less frequently.

We finalize the experimental results section with a time analysis as a function of the number of hash functions used and the number of elements hashed. Table 1 presents the average time spent for computing windows of 1,000, 10,000, 100,000 and 1,000,000 items (`W`). For each value of `W`, the use of 100 and 1000 universal hash functions are considered. The first column (`W`) indicates the number of elements processed for the respective time information presented. It also indicates the size of the windows for the windowed case. The times are an average of the overall times of all windows of `W` items processed. Times are about constant over the windows. The second column, `#h`, indicates the number of universal hash functions used. The last three columns, `I`, `WI` and `k(i,3)`, presents the average time for the indegree rarity distribution, windowed indegree rarity distribution and `k(i,3)` rarity distribution, respectively. For the `k(i,3)` case, a second time value is printed, indicating the time spent to process `W` triples. Similar results for approximation and times are observed for the other 4 graphs (we omit results due to space limitations). For the `k(i,3)` case, values for `W` of triples processed were add in parenthesis. Computational times using the optimized min-wise hash function was omitted due to the excessive time spent. For example, for computing 10,000 items and using 100 hash functions, for the indegree rarity distribution, the algorithm takes on average more than 200 seconds, whereas only 0.03 s. is spent using universal hashing. Before the optimization, it was spending thousands of seconds for the same configuration.

Times presented for the indegree rarity distribution are very small, even when considering the larger windows. For example, using a thousand hash functions just half second is spent on average for processing 1 billion items. Another observation is that the time grows linearly with the increase of the number of hash functions used and with the number of items considered in each window considered. For the windowed case, much higher times were found. That happens

Table 1. Average computation times in seconds for streams of fix size from the `wikiEN` (results for the indegree rarity distribution) and wikiPT (results for the `k(i,3)` rarity distribution) graphs. For each one of the three applications (indegree, windowed indegree and `k(i,3)` rarity distribution), times are printed for each `W` elements hashed, considering the use of 100 and 1000 hash functions. For the `k(i,3)` column, values for `W` of triples processed were add in parenthesis.

`W`	`#h`	`I`	`WI`	`k(i,3)`
100	1,000	0.003	0.03	9.36 (0.015)
1000	1,000	0.03	0.67	118.53 (0.19)
100	10,000	0.03	0.37	131.01 (0.21)
1000	10,000	0.30	10.22	1166.61 (1.87)
100	100,000	0.32	3.90	1272,67 (2.04)
1000	100,000	2.93	141.35	11765,96 (18.86)
100	1,000,000	3.24	40.54	12776,61 (20.48)
1000	1,000,000	29.32	1708.30	117859.27 (188.92)

because each update on the dynamic lists take O(`L`), where `L` represents the size of the list. Again the times increase linearly with increase of the window size and the number of hash functions used. For the `k(i,3)` computation, much more time was spent. The bottleneck of this application is that all triples of nodes within and adjacency list have to be computed. This enumeration takes long time, since often nodes of webgraphs have very large outdegree. For example, for the `wikiPT` graph used for the experiments and average of 624 triples are composed for each edge (his head node is considered for composing triples). Again the times increase linearly with the increase of `W` and `#h`. The graph `wikiPT` was used in this experiments since it is not possible to compute the exact number of `k(3,3)` for large graphs as the `wikiEN` in main memory.

5. CONCLUDING REMARKS

In this paper we use in practice data stream algorithms for computing statistical and topological properties of large

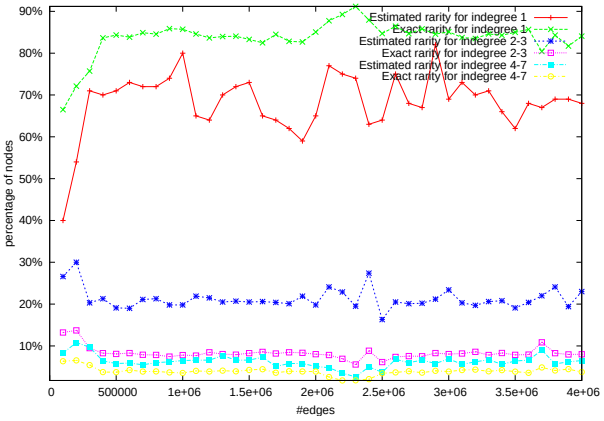


Figure 3. Estimated and exact indegree rarity computed for edges arrivals of graph `wikiEN` under the windowed model. Each window contains 100,000 items. The estimation makes use of 100 universal hashing functions. Values are presented to α up to 7, presented as $\log_2$ plot. This plot presents the percentage of nodes with a given indegree (y-basis) considering the amount of edges processed so far (x-basis).

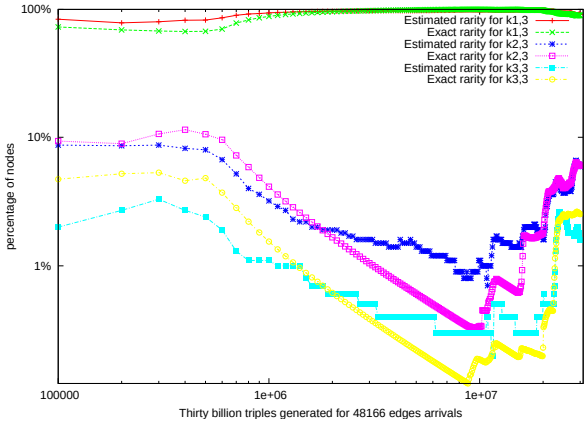


Figure 4. Plot in log scale of the estimated and exact `ki,3` rarity distribution, for i=1,2 and 3, computed for edges arrivals of graph `wikiPT`. The estimation makes use of 1000 universal hash functions. This plot presents the percentage of triples pointed by exactly i nodes (y-basis) considering the amount of triples seen so far (x-basis). The triples are computed accordingly with the edges arrivals. Results are plot every 10,000 triples processed.

graphs. We presented experimental results for the α-rarity algorithm applied on webgraphs for computing the rarity distribution of indegree and `k(i,3)` and obtained very good approximations. For the windowed case, applied for the indegree distribution, we observed again good approximation in a reasonable time. For the `k(i,3)` estimation we obtained good approximations, but spending long time. That happens because, in this case, all triples obtained are hashed by the `#h` hash functions. For the `wikiPT` graph, we observe a total of 624 triples generated for each edge processed.

We conclude that using universal hashing by this algorithm speed up a lot the codes, maintaining good approximations.

As further work, we would like to test other algorithms that estimates interesting statistical and topological properties of webgraphs. Moreover, dynamic aspects of webgraphs also could be explored, as edges being inserted and removed over time. The α-rarity algorithm does not have solution for deletions. But a recent publication of Cormode, Muthukrishnan and Rozenbaum [6] presents an algorithm that maintain results considering also deletions. Likewise, a sampling algorithm was presented by Frahling, Indyk and Sohler [7], also for maintaining distributions under insertions and deletions. Another important issue is on computing minors. The only reference on data streams for computing minors is by Yossef, Kumar and Sivakumar [9], but the bounds are not encouraging. Algorithms for computing minors as triangles and small cliques, with implementable bounds, would be a great contribution of that stream algorithms for our purposes.

6. ACKNOWLEDGEMENTS

We are very thankful to Jerry Zhao for providing the first version of the min-wise hash functions and for the suggestions for its optimization. We also thanks S. Muthukrishnan for several helpful discussions.

References

[1] N. Alon, O. Goldreich, J. Hastad, and R. Peralta, *Simple constructions of almost k-wise independent random variables*, Journal of Random structures and Algorithms **3** (1992), no. 3, 289–304.

[2] A.L. Barabasi and A. Albert, *Emergence of scaling in random networks*, Science (1999), no. 286, 509.

[3] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, S. Stata, A. Tomkins, and J. Wiener, *Graph structure in the web*, Computer Networks **33** (2000), 309–320.

[4] A.Z. Broder, M. Charikar, A.M. Frieze, and M. Mitzenmacher, *Min-wise independent permutations*, Proc. of STOC (1998), 327–336.

[5] M. Datar and S. Muthukrishnan, *Estimating rarity and similarity over data stream windows*, LNCS **2461** (2002), 323–334.

[6] Irina Rozembaum G. Cormogode, S. Muthukrishnan, *Summarizing and mining inverse distributions on data streams via dynamic inverse sampling*, Proccedings of the 31st VLDB Conferenct (2005).

[7] C. Sohler G. Frahling, P. Indyk, *Sampling in dynamic data streams and applications*, 21st Annual Symposium on Computational Geometry (2005).

[8] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins, *Trawling the web for emerging cyber communities*, (1999), 403–416.

[9] D. Sivakumar Z. Bar-Yosseff, R. Kumar, *Reductions in streaming algorithms, with an application to counting triangles in graphs*, Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (2002), 623–632.

[10] J. Zhao, *An implementation of min-wise independent permutation family*, (2005), http://www.icsi.berkeley.edu/ zhao/minwise/.

Coresets in Dynamic Geometric Data Streams

Gereon Frahling
frahling@upb.de

Christian Sohler
csohler@upb.de

Heinz Nixdorf Institute & Computer Science Department
University of Paderborn
Fuerstenallee 11
33102 Paderborn, Germany

ABSTRACT

A dynamic geometric data stream consists of a sequence of m insert/delete operations of points from the discrete space $\{1,\ldots,\Delta\}^d$. We develop streaming $(1+\epsilon)$-approximation algorithms for k-median, k-means, MaxCut, maximum weighted matching (MaxWM), maximum travelling salesperson (MaxTSP), maximum spanning tree (MaxST), and average distance over dynamic geometric data streams. Our algorithms maintain a small weighted set of points (a coreset) that approximates with probability 2/3 the current point set with respect to the considered problem during the m insert/delete operations of the data stream. They use $\mathrm{poly}(\epsilon^{-1}, \log m, \log \Delta)$ space and update time per insert/delete operation for constant k and dimension d.

Having a coreset one only needs a fast approximation algorithm for the weighted problem to compute a solution quickly. In fact, even an exponential algorithm is sometimes feasible as its running time may still be polynomial in n. For example one can compute in $\mathrm{poly}(\log n, \exp(O((1+\log(1/\epsilon)/\epsilon)^{d-1})))$ time a solution to k-median and k-means where n is the size of the current point set and k and d are constants. Finding an implicit solution to MaxCut can be done in $\mathrm{poly}(\log n, \exp((1/\epsilon)^{O(1)}))$ time. For MaxST and average distance we require $\mathrm{poly}(\log n, \epsilon^{-1})$ time and for MaxWM we require $O(n^3)$ time to do this.

Online Data Reconstruction

Bernard Chazelle
Department of Computer Science
Princeton University
35 Olden Street
Princeton, NJ 08544-2087
chazelle@cs.princeton.edu

ABSTRACT

I will discuss open problems and a few preliminary results in the area of online data reconstruction.

Consider a (geometric) dataset that is assumed to satisfy various structural properties. Because of noise and errors, however, an unknown fraction of the data might be violating some of these properties.

Can one enforce the desired structural properties in an online setting?

(Joint work with Nir Ailon, Seshadhri Comandur, and Ding Liu.)

Identifying Geometric Outliers in Massive Data Sets

José H. Dulá
School of Business
Virginia Commonwealth University
1015 Floyd Avenue, Box 84-4000
Richmond, VA 23284-4000, USA

ABSTRACT

Consider a finite point set $\mathcal{A}$ in m-dimensional space and the polyhedral hulls it generates from constrained linear combinations of its elements. A data point, a^{j^*} , is a *geometric outlier* if it belongs to a support set of a hull. There are several interesting problems that are modelled using these point sets and the resulting polyhedral objects. Examples include efficiency/performance evaluation, ranking and ordering schemes, mining for the detection of fraud, etc; all of which offer instances involving massive data sets. These applications require the identification of the extreme elements of the polyhedral sets; i.e., geometric outliers; a computationally intensive task. Traditional approaches solve as many LPs as the cardinality of the point set. A new generation of faster, output-sensitive, algorithms increase dramatically the scale of the applications. We discuss algorithmic and computational issues for massive data sets.

Cache-Oblivious Linear Programming

Sergio Cabello
IMFM, Department of Mathematics
Jadranska 19
SI-1000 Ljubljana, Slovenia
sergio.cabello@imfm.uni-lj.si

Mark de Berg
Department of Mathematics and Computing Science
TU Eindhoven, P.O. Box 513
5600 MB Eindhoven, the Netherlands
M.T.d.Berg@win.tue.nl

Xavier Goaoc
Loria
615 rue du Jardin Botanique, B.P. 101
54602 Villers-lès-Nancy cedex, France
goaoc@loria.fr

Mark Schroders
Department of Mathematics and Computing Science
TU Eindhoven, P.O. Box 513
5600 MB Eindhoven, the Netherlands
M.F.A.Schroders@win.tue.nl

ABSTRACT

Linear programming is a fundamental optimization problem: minimize a linear function in d-dimensional space under linear constraints. In small dimension, several optimal algorithms have been proposed both deterministic or randomized. There exist several simple internal-memory randomized incremental algorithms whose expected running time is optimal.

Most computers use a memory hierarchy, from the fast cache memory to the slow hard-drive, to store and manipulate data. The internal-memory model assumes a constant cost for a basic operation, be it adding two numbers or retrieving a data from memory. However, for large data sets the cost of accessing the memory can be really high and, from an efficiency point of view the bottleneck switches from computation time to memory access latency. The cache-oblivious model addresses this issue by taking into account the paging mechanism inherent to memory hierarchies. This model is theoretically sound as it does not assume any precise knowledge of the characteristics of the machine. Furthermore, several results indicate that it can predict practical behaviors for large data sets: algorithms that are faster in that model have better practical performances.

We study the problem of linear programming in small dimension in the cache oblivious model. We analyze the performance of randomized incremental algorithms and show that they are sub-optimal. We then give an algorithm optimal in both the internal-memory and cache-oblivious models.

Streaming Formats for Geometric Data Sets

Martin Isenburg*
Max-Planck-Institut für Informatik
Saarbrücken

Peter Lindstrom
Lawrence Livermore
National Laboratory

Stefan Gumhold
Max-Planck-Institut
für Informatik

Jack Snoeyink
University of North Carolina
at Chapel Hill

Abstract

Recent years have seen an immense increase in the complexity of geometric data sets. Today's gigabyte-sized 3D models can no longer be completely loaded into the main memory of common desktop PCs. Unfortunately, most storage and exchange formats for geometric data do not account for this. They were designed years ago when models were orders of magnitudes smaller. Using these formats to store and distribute giga-byte sized data sets is inefficient and unduly complicates all subsequent processing.

In this talk we will describe streaming formats for geometric data that are basically as simple as existing formats but more suitable for storing large data sets than all current alternatives. Such formats contain tiny bits of additional information that "finalize" previously read data. This information specifies which elements of a mesh or which areas in space have already been completely traversed. This gives the necessary guarantees to safely process these parts of the data and deallocate the corresponding data structures without first parsing the entire data set. While the focus of this talk is mainly on "topological streaming" of unstructured meshes, we will also motivate "spatial streaming" of meshes and point clouds.

1 Motivation and Overview

Modern scientific technologies enable the creation of digital 3D models of incredible detail and precision. These geometric data sets easily reach sizes of several gigabytes, making subsequent processing a difficult task. The sheer amount of data may not only exhaust the main memory resources of common desktop PCs, but even exceed the address space limit of a 32-bit machine. To process such data sets, one resorts to *out-of-core* algorithms that arrange the data so that it does not need to be kept in memory in its entirety, and adapt their computations to operate mainly on the loaded parts.

But for unstructured surface or volume meshes, already the way the raw input data is stored can turn the simplest pre-processing into a highly IO-inefficient operation. Current mesh formats use an array of floats to specify the vertex properties followed by an array of indices into the vertex array to specify the polygons or polyhedra. Storing large meshes in such a format means that one gigabyte-sized array of data is indexed by another gigabyte-sized block of data. Since the order in which the mesh elements appear in these arrays is left unspecified even simple de-referencing (i.e. resolving all vertex references) can potentially thrash the memory.

The ineffiency of indexed mesh input has been addressed in large mesh papers for the last eight years. [Chiang and Silva 1997] write that "Unfortunately, the datasets are often given in a format that contains indices to vertices. Thus we have to de-reference the indices before actually building the interval tree." and propose to use external sorting for this. Despite requiring large amount of scratch space and multiple passes over the data, this has since become the standard mechanism for dealing with large indexed meshes. Recent works often try to abandon indexed meshes altogether. [Cignoni et al. 2004], for example, assume that "the mesh is represented as a triangle soup, i.e., a list of triangles with direct vertex information". But as most their data sets are originally stored as indexed meshes, like the 3D scans of Michelangelo's statues [Levoy et al. 2000], they still need to de-reference in a pre-processing step.

*isenburg@cs.unc.edu http://www.cs.unc.edu/ ~ isenburg/sm

We will try to convince the audience that *streaming mesh* formats are much better suited for storing and distributing large meshes than current alternatives. First, they do not have the problem of in-efficient dereferencing, second, they are a more "natural" output format for memory-limited applications that generate large meshes, and third, they are an ideal input and ouput format for I/O-efficient algorithms that perform out-of-core stream processing.

The basic idea is to logically interleave vertices and the mesh elements that reference them and to provide explicit information about when vertices are "finalized" or "referenced for the last time". While the required changes to go from existing formats to streaming formats are minimal, the payoff can be substantial. Because the format tells us which of the previously read vertices to keep in memory, we can trivially de-reference such meshes in an IO-optimal manner—the problem of repeated, possibly incoherent look-up of vertex data in a gigantic array does not exist. And because the format tells us which vertices can safely be deallocated because they are no longer needed, we can do this for meshes of practically arbitrary size while requiring only moderate amounts of memory.

But a streaming mesh format is not only a better input format for large meshes—it is also a more natural output format for most mesh generating applications. Given limited memory resources, it is in fact quite *difficult* to output meshes into standard indexed formats. A mesh generating application that can only hold and work on small pieces of the data at any time will need to store vertices and triangle into seperate temporary files and concatenate them later. Memory mapping the vertex and triangle arrays is not possible without knowing the exact size of the vertex array in advance. For example, an out-of-core marching cubes iso-surface implementation that processes the volume layer by layer will naturally output vertices and triangles in the same order. And vertices from the last layer can trivially be finalized before moving on to the next layer.

Furthermore, a streaming mesh format is the ideal input and output for stream processing. In this model, the mesh streams through an in-core buffer, which is large enough to hold all active mesh elements. For straight-forward tasks, such as rendering a flat shaded mesh, a minimal stream buffer is needed. For more elaborate processing tasks, a larger stream buffer may hold as many additional mesh elements as there are memory resources, allowing random access to a localized but continuously changing subset of the mesh.

Streaming meshes allow pipelined processing, where multiple tasks run concurrently on separate pieces of the mesh. One module's output then serves as the input for another module. Envision a scenario where one algorithm extracts an isosurface and pipes it as a streaming mesh to a simplification process, which in turn streams the simplified mesh to a compression engine that encodes it and immediately transmits the resulting bit stream to a remote location where triangles are rendered as they decompress. In fact, we now have all components of this pipeline—and it is the streaming format that makes it possible to pipe them all together.

References

CHIANG, Y.-J., AND SILVA, C. T. 1997. I/O optimal isosurface extraction. In *Visualization '97*, 293–300.

CIGNONI, P., GANOVELLI, F., GOBBETTI, E., MARTON, F., PONCHIO, F., AND SCOPIGNO, R. 2004. Adaptive tetrapuzzles - efficient out-of-core construction and visualization of gigantic polygonal models. In *SIGGRAPH 2004*, 796–803.

LEVOY, M., PULLI, K., CURLESS, B., RUSINKIEWICZ, S., KOLLER, D., PEREIRA, L., GINZTON, M., ANDERSON, S., DAVIS, J., GINSBERG, J., SHADE, J., AND FULK, D. 2000. The Digital Michelangelo Project. In *SIGGRAPH 2000*, 131–144.

A Java-Based System for Large-Scale Rendering

[Extended Abstract]

Sourav Dalal
London South Bank University
103 Borough Road
London, SE1 0AA, UK
dalalsa@lsbu.ac.uk

Frank Dévai
London South Bank University
103 Borough Road
London, SE1 0AA, UK
fl.devai@lsbu.ac.uk

Md Mizanur Rahman
London South Bank University
103 Borough Road
London, SE1 0AA, UK
rahmanml@sbu.ac.uk

ABSTRACT

The geometric aspects of the visualisation of large data sets, in particular, digital models of real or planned solid objects in a heterogeneous distributed environment are investigated. It is demonstrated that image compositing does not scale well on networks of workstations. A distributed system, based on scanline algorithms and using Java technology, is proposed. The proposed system is efficient as servers only need to solve a problem of growth rate of $n \log n$, it is fault tolerant as both lost messages and server failures are tolerated, and it has negligible hardware costs as it runs on existing networks of workstations. The system is also scalable, as data sets sent to servers only need to be sent once regardless of the number of servers, and the amount of data transmitted by servers only depends on the resolution of the final image.

1 Introduction

Applications like flight simulation, medical training, computer aided design, three-dimensional geographic information systems etc, require the visualisation of large data sets. These data sets are usually polygon-mesh models, containing 10–100 million polygons. One of the fundamental problems of rendering these data sets is visibility determination; the process of deciding what parts of the surface of the model can be seen from a possibly moving point. Any algorithm for determining the visibility of a set of polygons in three-dimensional space with a total of N edges takes $\Theta(N^2)$ time in the worst case [5, 13]. This growth rate is a serious difficulty. To overcome it hardware accelerators are used in low-end systems, and parallel algorithms are proposed for high-performance graphics systems. For example, the hidden-line problem can be solved in $\Theta(\log N)$ time either on N^2 EREW or on $N^2/\log N$ CREW PRAM processors, and the hidden-surface problem in the same $\Theta(\log N)$ time on N^2 CREW PRAM processors. The $\Theta(\log N)$ result cannot be further improved even if arbitrarily many processors were available [6, 8, 9]. Algorithms with a smaller number of processors, called *coarse-grain algorithms*, can also be derived from these results, e.g., a hidden-surface algorithm that takes $O(\frac{N^2 \log N}{p} + \log N)$ time in the worst case by using p CREW PRAM processors [9].

Unfortunately, parallel architectures either suffer from the problem of latency or do not scale well, hence are expensive. Using distributed shared memory (DSM) running on a network of workstations [2] seems to be a better alternative. There are some drawbacks, however, e.g., overhead: if a processor needs to access a variable available only on a remote machine, the whole page must be transferred. Experts like Tanenbaum and van Steen [21] are skeptical: *"After almost 15 years of research on distributed shared memory, DSM researchers are still struggling to combine efficiency and programmability. To attain high performance on large-scale multicomputers, programmers resort to message passing despite its higher complexity compared to programming (virtual) shared memory systems. It seems therefore justified to conclude that DSM for high-performance parallel programming cannot fulfill its initial expectations."*

2 Previous work

Early approaches to high-performance rendering can be classified as image parallelism and object parallelism [4]. In *image parallelism* regions of the screen are allocated to processors [1, 10, 16]. In *object parallelism* subsets of objects, typically polygons, are allocated to processors [14, 15, 18, 19]. Different processors may produce pixels at the same screen location, therefore pixels from each processor must be combined into a complete scene. This problem is called the *pixel merging* or *image compositing problem*.

The simplest solution is *binary-tree compositing*. Considering n processors, $1, 2, \ldots, n$, where n is a power of 2, each processor determines the full-screen image of the objects allocated to it. Then each odd-numbered processor i, $i = 1, 3, \ldots, n-1$, passes on its image to processor $i+1$, where the two images are composited pixel by pixel. In the next stage processor 2 passes on its image to processor 4, processor 6 to processor 8, and so on, where again the two images are composited. After $\log n$ stages the final image is contained by processor n.

The disadvantage of binary-tree compositing is that when a processor passed on its image, it becomes idle; the final stage of compositing is performed by only one processor, processor n. To exploit more parallelism *binary-swap compositing* has been proposed [12]. The key idea is that instead of processor i sending its image to processor j, and leaving processor j alone doing the composition, both processor i and j splits their image into two halves, which are swapped, and both processors perform compositing one half of the image. After $\log n$ stages each processor contains $1/2^n$ of the final image.

Considering the hardware base, an important observation is that early systems used special-purpose hardware [1, 10,

14, 15, 16, 18, 19]. In the 1990s high-performance rendering systems gradually moved on to general-purpose high-performance computer systems, such as the Connection Machine CM-5 [12] or hypercubes [11]. The trend continues with the use of clusters of workstations [3, 20].

Implementing high-performance rendering systems on existing clusters of workstations communication costs are becoming a bottleneck. While in a hypercube, for example, each one of n processors are connected to $\log n$ other processors, PC clusters typically use bus-based interconnection networks, such as Gigabit Ethernet. The communication cost of image compositing can be significant on these architectures. For example, binary-swap compositing transmits up to $2.43n^{1/3}p$ pixels on average on n processors, where p is the number of pixels in the final image [12]. Though binary-swap compositing is aimed at massively parallel processing [12], around 100 workstations would transmit an order of magnitude more pixels than necessary for the final image. We propose a system that never transmits more than the number pixels in the final image.

3 The proposed system

The objective of this research is the design and implementation of a light-weight, inexpensive message-passing architecture for the visualisation of large geometric models. This system also uses networks of workstations, typically available in a university environment, but unlike DSM, it is a more efficient approach using scanline algorithms [7]. A scanline is a row of pixels of the image. The pixels of a scanline can be obtained by determining the visibility of the objects in the plane perpendicular to the screen and containing the scanline. The intersection of a polygon-mesh model (i.e., a set of polygons) and a plane is a set of line segments in the plane.

Our system is based on a *client-server architecture*, where a number of servers support typically one client. Each server is responsible for the calculation of a couple of scanlines, and the delivery of the results back to the client which displays the image.

The operation of the system is outlined as follows. The user, sitting in front of a graphics workstation running the client software, selects a geometric model or virtual environment they want to visualize. Once the geometric model has been selected, the client software distributes the corresponding data set on a high-speed network by using *scalable reliable multicast*. All workstations having the server software installed pick up the model, and store it on their local disk. On user interaction, e.g., zoom, translation or rotation of the model, the client broadcasts a 4x4 homogeneous transformation matrix. All the available servers receiving this matrix perform the required transformation on their stored model.

Each server is associated with a unique name, which is an integer in the range $[0, m-1]$, where m is the total number of the available servers at any given point of time. In reply to a client request, each server calculates a number of scanlines. Scanlines are numbered from 0 to $h-1$, where h is the height of the image, i.e., the total number of scanlines on the screen of the client's workstation. Server k calculates scanlines $i = k + jm$, for $j = 0, 1, 2, \ldots$, such that $i < h$, and sends them back to the client.

The client then assembles the scanlines, and displays the next frame. It also calculates a new transformation matrix from user input, gathers server statistics, rename servers if necessary and informs each server involved. Then it goes back to the beginning of the loop broadcasting a new transformation matrix and requesting scanlines.

In the unlikely case of the output of a single server is lost, the scanlines from the previous frame are used. Often there is no difference between the same scanlines of subsequent frames (e.g., when a designer is contemplating a part of the model) and the difference is hardly noticeable on moving images. (Note that the scanlines are delivered by the servers in an interleaved pattern: a scanline from server k is followed by one from server $k+1$, and so on.)

If, however, the client notices from server statistics that the output of a server is consistently lost in the last few frames, the client renames the servers: If, say, server k is not responding, server $m-1$ is renamed as k, and m is reduced by 1. If a server re-appears, it is given the name m, and m is incremented by 1.

From the above it follows that the system is *fault tolerant* in the sense that it tolerates both lost messages and server failures. The latter is particularly important, because in this way the workstations do not have to be dedicated to the system. The servers run on workstations as background processes; when a workstation gets idle, the server joins the system, and it leaves the system when preempted by an interactive process. Thus, the hardware cost of the system is negligible, as organizations like universities or design offices already have extensive workstation networks.

The increase of processor and network speed and memory and disk capacities makes it possible that servers can store the complete data set, hence eliminating image compositing. Also new compression techniques help to distribute models to servers more quickly. For example, the Edgebreaker compression technique [17] can be used when the client multicasts the 3D geometric model to servers. If the 3D model is a triangle mesh, typically there are twice as many triangles as vertices. Each vertex needs typically three floating-point numbers, i.e., $3 \times 4 = 12$ bytes for its x, y and z coordinates. If the triangles are represented either by integer references or pointers to vertices, each triangle needs also 3×4 bytes. This memory requirement is roughly twice as much as the memory requirement of the vertices as there are twice as many triangles. Edgebreaker can compress this connectivity information just to two bits or less than two bits per triangle [17]. The servers can use *run-length encoding* when returning the scanlines.

Another significant advantage of the proposed architecture is that the servers only need to compute a planar visibility problem with a complexity of $\Theta(n \log n)$ rather than the 3D problem of complexity $\Theta(N^2)$, where $n \leq N$ is the number of line segments in the plane of the scanline. The planar visibility problem is well understood, and this research gave us the opportunity to implement 10 scanline algorithms, and analyse their asymptotic resource requirements.

Asymptotic analysis, however, cannot take into consideration constant factors, which can be different in different environments. Therefore a *portable testbed* was developed for the comparative evaluation of the actual performance of the algorithms on the particular hardware-software platform they are used. In this way in a heterogeneous distributed

system the fastest version of the server can be installed in any given machine environment.

Considering the number of possible algorithms together with their variants, the number of time measurements required for conclusive results is substantial, hence using real 3D models would be too expensive and time consuming. Since the input to a scan-line algorithm is only a planar set of line segments, a more efficient test-data generation method based on random line segments was developed.

As some scanline algorithms exploit the fact that the input obtained from real solid models results in a set of line segments that are non-intersecting (except at their endpoints) a test-data generation method was required to produce a planar set of non-intersecting random line segments. Our method takes a total of $4n$ random numbers and $O(n \log n)$ time in the worst case to generate $4n$ coordinates for a set of n non-intersecting random line segments in the plane.

The Java language is used for both the development of a prototype distributed system and the testbed. Since the running time of scanline algorithms is very short, accurate time-measurement techniques are required. These were implemented by reading the time-stamp counters of the processors using Java native methods.

The advantages of using the Java language are that Java is a publicly available software technology with the a support of a large community. It provides large class libraries and an object-oriented development environment. The main disadvantage is that object creation can be too expensive for a high-performance graphics system. Therefore we reuse objects wherever possible. Also some of our scanline algorithms use parallel arrays instead of objects. For example, it would be natural to represent the x_1, y_1, x_2 and y_2 coordinates of a line segment by using a line-segment object, but using four parallel arrays for the four co-ordinates improves server performance.

4 Concluding remarks

Though the planar visibility problem is well researched, we encountered some interesting problems which, we believe, are unsolved. For example, one of our algorithms, called the *priority-queue method*, gave better than expected experimental results. This algorithm uses a heap to maintain an order on line segments. It is known that n elements can be inserted in a heap in $O(n)$ expected time, but deleting the minimum element takes $\Theta(\log n)$ time on average. Our algorithm, however, deletes *arbitrary elements*, which are near to leaf nodes most of the time, thus repairing the heap costs little. So far, however, we could not turn our arguments into a formal proof.

References

[1] K. Akeley and T. Jermoluk. High-performance polygon rendering. *Computer Graphics*, 22(4):239–246, 1988.

[2] C. Amza et al. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 1996.

[3] J. Chhugani et al. vLOD: High-fidelity walkthrough of large virtual environments. *IEEE Transactions on Visualization and Computer Graphics*, 11(1):35–47, 2005.

[4] M. Cox and P. Hanrahan. Pixel merging for object-parallel rendering: a distributed snooping algorithm. In *PRS'93: Proc. 1993 Symposium on Parallel Rendering*, pages 49–56, New York, NY, USA, 1993. ACM Press.

[5] F. Dévai. Quadratic bounds for hidden-line elimination. In *Proc. 2nd Annu. ACM Sympos. Comput. Geom.*, pages 269–275, 1986.

[6] F. Dévai. An $O(\log N)$ parallel time exact hidden-line algorithm. In *Eurographics'87 Workshop on Advances in Computer Graphics Hardware II*, pages 47–63. Springer, 1988.

[7] F. Dévai. Scan-line methods for parallel rendering. In M. Chen et al., editors, *High-Performance Computing for Computer Graphics and Visualisation.*, pages 88–98. Springer, 1995.

[8] F. Dévai. On the computational requirements of virtual reality systems. In *State of the Art Reports,* EUROGRAPHICS'97, pages 59–92, Sep. 1997.

[9] F. Dévai. Parallel algorithms for visibility computations. In *Eurographics UK Chapter Annual Conference*, volume 17, April 1999.

[10] H. Fuchs et al. Pixel-Planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. *Computer Graphics*, 23(3):79–88, 1989.

[11] T. Kurc, C. Aykanat, and B. Ozguc. Object-space parallel polygon rendering on hypercubes. *Computers & Graphics*, 22:487–503, 1998.

[12] K.-L. Ma et al. Parallel volume rendering using binary-swap compositing. *IEEE Comput. Graph. Appl.*, 14(4):59–68, 1994.

[13] M. McKenna. Worst-case optimal hidden-surface removal. *ACM Trans. Graph.*, 6:19–28, 1987.

[14] S. Molnar. Combining z-buffer engines for higher-speed rendering. In *Eurographics'88 Workshop on Advances in Computer Graphics Hardware III*, pages 171–182. Springer, 1991.

[15] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-speed rendering using image composition. *Computer Graphics*, 26:231–240, 1992.

[16] Michael Potmesil and Eric M. Hoffert. The pixel machine: a parallel image computer. *Computer Graphics*, 23(3):69–78, 1989.

[17] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61, January/March 1999.

[18] B.-O. Schneider and U. Claussen. PROOF: An architecture for rendering in object space. In *Eurographics'88 Workshop on Advances in Computer Graphics Hardware III*, pages 121–140. Springer, 1991.

[19] C. D. Shaw, M. Green, and J. Schaeffer. A VLSI architecture for image composition. In *Eurographics'88 Workshop on Advances in Computer Graphics Hardware III*, pages 183–199. Springer, 1991.

[20] A. Takeuchi et al. An improved binary-swap compositing for sort-last parallel rendering on distributed memory multiprocessors. *Parallel Computing*, 29(11–12):1745–1762, 2003.

[21] A. S. Tanenbaum and M. van Steen. *Distributed Systems—Principles and Paradigms.* Prentice Hall, 2002.

Cache-Oblivious Layouts of Polygonal Meshes

Extended Abstract

Sung-Eui Yoon[1*] Peter Lindstrom[2] Valerio Pascucci[2] Dinesh Manocha[1]

[1]University of North Carolina at Chapel Hill [2]Lawrence Livermore National Laboratory
http://gamma.cs.unc.edu/COL

Abstract

We present a novel method for computing cache-oblivious layouts of large meshes that improve the performance of interactive visualization and geometric processing algorithms. Given that the mesh is accessed in a reasonably coherent manner, we assume no particular data access patterns or cache parameters of the memory hierarchy involved in the computation. Furthermore, our formulation extends directly to computing layouts of multiresolution and bounding volume hierarchies of large meshes.

We develop a simple and practical cache-oblivious metric for estimating cache misses. Computing a coherent mesh layout is reduced to a combinatorial optimization problem. We designed and implemented an out-of-core multilevel minimization algorithm and tested its performance on unstructured meshes composed of tens to hundreds of millions of triangles. Our layouts can significantly reduce the number of cache misses. We have observed $2-20$ times speedups in view-dependent rendering, collision detection, and isocontour extraction without any modification of the algorithms or runtime applications.

1 Introduction

Over the last few years, advances in model acquisition, computer-aided design, and simulation technologies have resulted in massive databases of complex geometric models. Meshes composed of tens or hundreds of millions of triangles are frequently used to represent CAD environments, terrains, isosurfaces, and scanned models.

Efficient algorithms for processing large meshes utilize the computational power of CPUs and GPUs for interactive visualization and geometric applications. A major computing trend over the last few decades has been the widening gap between processor speed and main memory speed. As a result, system architectures increasingly use caches and memory hierarchies to avoid memory latency. The access times of different levels of a memory hierarchy typically vary by orders of magnitude. In some cases, the running time of a program is as much a function of its cache access pattern and efficiency as it is of operation count [Frigo et al. 1999; Sen et al. 2002].

Our goal is to design cache efficient algorithms to process large meshes. The two standard techniques to reduce cache misses are:

1. **Computation Reordering**: Reorder the computation to improve program locality. This is performed using compiler optimizations or application specific hand-tuning.
2. **Data Layout Optimization:** Compute a cache-coherent layout of the data in memory according to the access pattern.

In this paper, we focus on data layout optimization of large meshes to improve cache coherence. A triangle mesh is represented by linear sequences of vertices and triangles. Therefore, the problem becomes one of computing a cache efficient layout of the vertices and triangles.

Many layout algorithms and representations have been proposed for optimizing the cache access patterns for specific applications. The representations include *rendering sequences* (e.g. triangle strips) that are used to improve the rendering performance of large meshes on GPUs. Recent extensions include *processing sequences* (e.g. streaming meshes), which work well for applications that can access the data in a fixed order. Some algorithms for image processing and visualization of large datasets use space filling curves as a heuristic to improve cache coherence of a layout. These algorithms work well on models with a regular structure; however, they do not take into account the topological structure of a mesh and are not general enough to handle unstructured datasets.

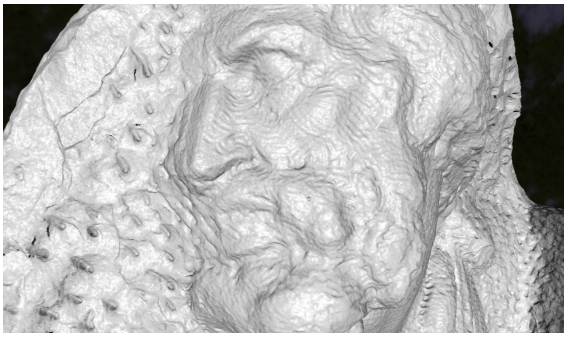


Figure 1: **Scan of Michelangelo's St. Matthew:** We precompute a cache-oblivious layout of this 9.6GB scanned model with 372M triangles. Our novel metric results in a cache-oblivious layout and at runtime reduces the vertex cache misses by more than a factor of four for interactive view-dependent rendering. As a result, we improve the frame rate by almost five times. We achieve a throughput of 106M tri/sec (at 82 fps) on an NVIDIA GeForce 6800 GPU.

Main Results: We present a novel method to compute cache-oblivious layouts of large triangle meshes. Our approach is general in terms of handling all kinds of polygonal models and cache-oblivious as it does not require any knowledge of the cache parameters or block sizes of the memory hierarchy involved in the computation.

We represent the mesh as an undirected graph $G = (V, E)$, where $|V| = n$ is the number of vertices. The mesh layout problem reduces to computing an optimal one-to-one mapping of vertices to positions in the layout, $\varphi : V \rightarrow \{1, ..., n\}$, that reduces the number of cache misses. Our specific contributions include:

1. Deriving a practical cache-oblivious metric that estimates the number of cache misses.
2. Transforming the layout computation to an optimization problem based on our metric.
3. Solving the combinatorial optimization problem using a multilevel minimization algorithm.

We also extend our graph-based formulation to compute cache-oblivious layouts of bounding volume and multiresolution hierarchies of large meshes.

We use cache-oblivious layouts for three applications: view-dependent rendering of massive models, collision detection between complex models, and isocontour extraction. In order to show the generality of our approach, we compute layouts of several kinds of geometric models including CAD environments, scanned models, isosurfaces, and terrains. We use these layouts directly without any

*sungeui@cs.unc.edu

modification to the runtime application. Our layouts significantly reduce the number of cache misses and improve the overall performance. Compared to a variety of popular mesh layouts, we achieve on average:

1. Over an order of magnitude improvement in performance for isocontour extraction.
2. A five time improvement in rendering throughput for view-dependent rendering of multi-resolution meshes.
3. A two time speedup in collision detection queries based on bounding volume hierarchies.

This extended abstract provides summary of our results on cache-oblivious layouts of polygonal meshes. More details on cache-oblivious mesh layouts and layouts of bounding volume hierarchies are available at [Yoon et al. 2005] and [Yoon and Manocha 2005] respectively.

2 Related Work

In this section we briefly review related work on cache-efficient algorithms, out-of-core techniques, mesh sequences, and layouts.

2.1 Cache-Efficient Algorithms

Cache-efficient algorithms have received considerable attention over last two decades in theoretical computer science and compiler literature. These algorithms include theoretical models of cache behavior [Vitter 2001; Sen et al. 2002], and compiler optimizations based on tiling, strip-mining, and loop interchanging; all of these can minimize cache misses [Coleman and McKinley 1995].

At a high level, cache-efficient algorithms can be classified as either cache-aware or cache-oblivious. Cache-aware algorithms utilize knowledge of cache parameters, such as cache block size [Vitter 2001]. On the other hand, cache-oblivious algorithms do not assume any knowledge of cache parameters [Frigo et al. 1999]. There is a considerable amount of literature on developing cache-efficient algorithms for specific problems and applications, including numerical programs, sorting, geometric computations, matrix multiplication, FFT, and graph algorithms. Most of these algorithms reorganize the data structures for the underlying application, i.e., computation reordering. More details are given in recent surveys [Arge et al. 2004; Vitter 2001]. There exists relatively little work on computing cache-coherent layouts for a wide variety of applications.

2.2 Mesh Sequences and Layouts

The order in which a mesh is laid out can affect the performance of algorithms operating on the mesh. Several possibilities have been considered.

Rendering and Processing Sequences: In order to maximize the benefits of vertex buffers for fast rendering, triangle reordering for rendering sequences is necessary [Deering 1995]. Hoppe [1999] casts the triangle reordering as a discrete optimization problem with a cost function relying on a specific vertex buffer size. There are also a few methods to improve cache-coherency of view-dependent simplified meshes [Bogomjakov and Gotsman 2002; Karni et al. 2002]. Isenburg and Gumhold [2003] propose processing sequences as an extension of rendering sequences to large-data processing. A processing sequence represents a mesh as an interleaved ordering of indexed triangles and vertices that can be streamed through main memory [Isenburg and Lindstrom 2004]. However, global mesh access is restricted to a fixed traversal order.

Space Filling Curves: Many algorithms use space filling curves [Sagan 1994] to compute cache-friendly layouts of volumetric grids or height fields. These layouts are widely used to improve performance of image processing [Velho and Gomes 1991] and terrain or volume visualization [Pascucci and Frank 2001; Lindstrom and Pascucci 2001]. A standard method of constructing a layout is to embed the meshes or geometric objects in a uniform structure that contains the space filling curve. Therefore, these algorithms have been used for objects or meshes with a regular structure (e.g. images and height fields). Methods based on space filling curves do not consider the topological structure of meshes. It is unclear whether these approaches would extend to large CAD environments with an irregular distribution of geometric primitives. Moreover, if an application needs to access the mesh primitives based on connectivity information, space filling curves may not be useful. Algorithms have also been proposed to compute paths on constrained, unstructured graphs as well as to generate triangle strips and finite-element mesh layouts [Heber et al. 2000; Oliker et al. 2002; Bartholdi and Goldsman 2004; Gopi and Eppstein 2004].

Sparse Matrix Reordering: There is considerable research on converting sparse matrices into banded ones to improve the performance of various matrix operations [Diaz et al. 2002]. Common graph and matrix reordering algorithms attempt to minimize one of three measures: bandwidth (maximum edge length), profile (sum of maximum per-vertex edge length), and wavefront (maximum front size, as in stream processing). These measures are closely connected with MLA and layouts for streaming, and generally are more applicable to stream layout than cache-oblivious mesh layout.

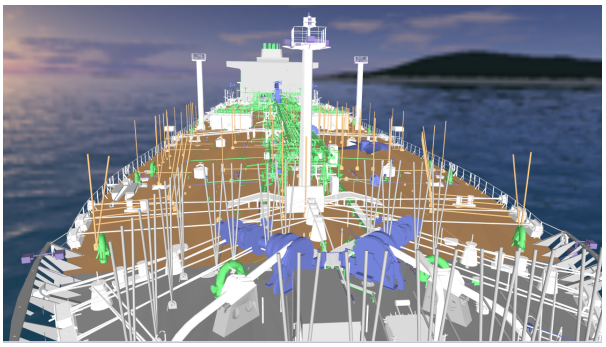


Figure 2: **Double Eagle Tanker:** We compute a cache-oblivious layout of the tanker with 82M triangles and more than 127K different objects. This model has an irregular distribution of primitives. We use our layout to reduce vertex cache misses and to improve the frame rate for interactive view-dependent rendering by a factor of two; we achieve a throughput of 47M tri/sec (at 35 fps) on an NVIDIA GeForce 6800 GPU.

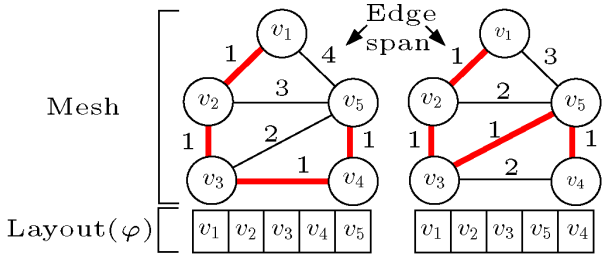


Figure 3: **Vertex layout for a mesh:** A mesh consisting of 5 vertices is shown with two different orderings obtained using a local permutation of v_4 and v_5. We highlight the span of each edge based on the layout. The ordering shown on the right minimizes cache misses according to our cache-oblivious metric.

3 Mesh Layout and Cache Misses

In this section, We represent a mesh as a graph to capture access patterns of applications and extend our approach to layouts of multi-resolution and bounding volume hierarchies of a mesh.

3.1 Mesh Layout

A mesh layout is a linear sequence of vertices and triangles of the mesh. We construct a graph in which each vertex represents a data element of the mesh. An edge exists between two vertices of the graph if their representative data elements are likely to be accessed in succession by an application at runtime.

For single-resolution mesh layout, we map mesh vertices and edges to graph vertices and edges. A vertex layout of an undirected graph $G = (V, E)$ is a one-to-one mapping of vertices to positions,

$\varphi : V \rightarrow \{1, \ldots, n\}$, where $|V| = n$. Our goal is to find a mapping, φ, that minimizes the number of cache misses during accesses to the mesh.

3.2 Layouts of Multiresolution Meshes and Hierarchies

Hierarchical data structures are widely used to speed up computations on large meshes. Two types of hierarchies are used for geometric processing and interactive visualization: bounding volume hierarchies (BVHs) and multi-resolution hierarchies (MRHs). The BVHs use simple bounding shapes (e.g. spheres, AABBs, OBBs) to enclose a group of triangles in a hierarchical manner. MRHs are used to generate a simplification or approximation of the original model based on an error metric. In order to compute a layout of a hierarchy, we construct a graph that captures cache-coherent access patterns to the hierarchy. We add extra edges to our graph that capture the spatial locality and connectivity locality within the hierarchy. More details on parent-child locality and spatial localities of BVHs are also available [Yoon and Manocha 2005].

4 Cache-Oblivious Layouts

In this section we present a novel algorithm for computing a cache-coherent layout of a mesh. We make no assumptions about cache parameters and compute the layout in a cache-oblivious manner.

4.1 Terminology

We use the following terminology in the rest of the paper. The *edge span* of the edge between v_i and v_j in a layout is the absolute difference of the vertex indices, $|i - j|$ (see Fig. 3). We use E_l to denote the set that consists of all the edges of edge span l, where $l \in [1, n-1]$. The *edge span distribution* of a layout is the histogram of spans of all the edges in the layout. The *cache miss ratio* is the ratio of the number of cache misses to the number of accesses. The *cache miss ratio function (CMRF)*, p_l, is a function that relates the cache miss ratio to an edge span, l. The CMRF always lies within the interval $[0, 1]$; it is exactly 0 when there are no cache misses, and equals 1 when every access results in a cache miss. We alter the layouts using a *local permutation* that reorders a small subset of the vertices. The local permutation changes the edge span of edges that are incident to the affected vertices (see Fig. 3).

4.2 Cache-Coherent Access Pattern

If we know the runtime access pattern of a given application a priori and the CMRFs, we can compute the exact number of cache misses. However, we make no assumptions about the application and instead use a probabilistic model to estimate the number of cache misses. Our model approximates the edge span distribution of the runtime access pattern of the vertices with the edge span distribution of the layout. Based on this model, we define the expected number of cache misses of the layout as:

$$ECM = \sum_{i=1}^{n-1} |E_i| p_i \tag{1}$$

where $|E_i|$ is the cardinality of E_i and is a function of the layout, φ.

4.3 Assumptions

Our goal is to compute a layout, φ, that minimizes the expected number of cache misses for all possible cache parameters. We present a metric that is used to check whether a local permutation would reduce cache misses. We make two assumptions with respect to CMRFs: invariance and monotonicity.

Invariance: We assume that the CMRF of a layout is invariant before and after a local permutation. Since a local permutation affects only a small region of a mesh, the changes in CMRF due to a local permutation are very small.

Monotonicity: We assume that the CMRF is a monotonically non-decreasing function of edge span. As we access vertices that are farther away from the current vertex (i.e. the edge spans increase), the probability of having a cache miss increases, until eventually leveling off at 1.

Figure 4: **Puget Sound contour line:** This image shows a contour line (in black) extracted from an unstructured terrain model of the Puget Sound. The terrain is simplified down to 143M triangles. We extracted the largest component (223K edges) of the level set at 500 meters of elevation. Our cache-oblivious layouts improve the performance of the isocontour extraction algorithm by more than an order of magnitude.

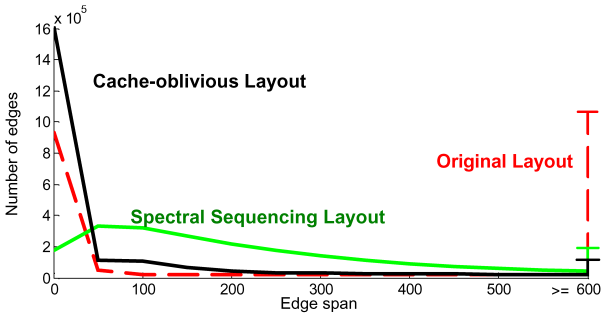


Figure 5: **Edge span distributions:** The edge span histogram of the dragon model with 871K triangles and 437K vertices. We show the histogram of the original model representation (red), spectral sequencing (green), and our cache-oblivious metric (black). In the original layout, a large number of edges have edge spans greater than 600. Intuitively, our cache-oblivious metric favors edges that have small edge spans. Therefore, our layouts reduce cache misses.

4.4 Cache-Oblivious Metric

Our cache-oblivious metric is used to decide whether a local permutation decreases the expected number of cache misses, which due to the invariance of p_i is true if the following inequality holds:

$$\sum_{i=1}^{n-1} (|E_i| + \Delta|E_i|) p_i < \sum_{i=1}^{n-1} |E_i| p_i \Leftrightarrow \sum_{j=1}^{m} \Delta|E_{l(j)}| p_{l(j)} < 0 \tag{2}$$

Here $\Delta|E_i|$ is the signed change in the number of edges with edge span i after a local permutation and $n-1$ is maximum edge span for a mesh with n vertices. Furthermore, we let m denote the number of sets (among $E_1, E_2, \ldots, E_{n-1}$) whose cardinality changes because of the permutation, and let $l(j)$ denote the edge span associated with the jth such set, with $l(j) < l(j+1)$ and $m \ll n - 1$.

Constant Edge Property: The total number of edges in a layout is the same before and after the local permutation. Hence

$$\sum_{j=1}^{m} \Delta|E_{l(j)}| = 0 \tag{3}$$

Parameterization of cache miss ratio: We parameterize each cache miss ratio, $p_{l(j)}$, by introducing a parametric variable, x_j, which due to the monotonicity of $p_{l(j)}$ is monotonically non-decreasing with j. This is represented as:

$$p_{l(j)} = x_j p_{l(1)} \tag{4}$$

where $1 \leq j \leq m$ and

$$1 = x_1 \leq x_2 \leq \cdots \leq x_{m-1} \leq x_m \leq \frac{1}{p_{l(1)}} \quad (5)$$

$p_{l(1)}$ is the cache miss ratio of the first edge, and $0 \leq p_{l(1)} \leq 1$.

The leftmost constraint of Eq. (5) is obvious because $p_{l(1)} = x_1 p_{l(1)}$. The rightmost constraint is computed from $p_{l(m)} = x_m p_{l(1)} \leq 1$, because all the cache miss values are less than or equal to 1.

By substituting the parameterization of cache miss ratios shown in Eq. (4) into Eq. (2) and canceling the constant $p_{l(1)}$, we have:

$$\sum_{j=1}^{m} \Delta |E_{l(j)}| x_j < 0. \quad (6)$$

This is our exact cache-oblivious metric.

4.5 Geometric Formulation

We reduce the computation of the expression in Eq. (6) to a geometric volume computation in an m dimensional hyperspace. Geometrically, the parameterization domain represented in Eq. (5) defines a closed hyperspace in $\mathbb{R}^m$. Eq. (6) defines a closed subspace within the domain of Eq. (5). Moreover, we define V_+ to be the volume of the subspace represented in Eq. (6) and V_- to be the volume of its complement within the closed domain.

Volume Computation: Intuitively speaking, the volume V_+ corresponds to the set of cache configurations parameterized by $\{x_j\}$ for which we expect a reduction in cache misses. Since we assume all configurations to be equally likely, we probabilistically reduce the number of cache misses by accepting a local permutation whenever V_+ is larger than V_-. Unfortunately, the complexity of the volume computation in m dimensions is very high [Lasserre and Zeron 2001].

4.6 Fast and Approximate Metric

Given the complexity of exact volume computation, we use an approximate metric to check whether a local permutation would reduce the expected number of cache misses. In particular, we use a single sample point as an estimate of $\{x_j\}$ and compute an approximate metric with low error.

By using the single sample point, we obtain our fast and approximate metric as the following:

$$\sum_{j=1}^{m} \Delta |E_{l(j)}| j < 0 \quad (7)$$

If inequality (7) holds, we allow the local permutation. Based on this metric, we compute a layout, φ, that minimizes the number of cache misses. More detail on our approximate metric and its derivation is available at [Yoon et al. 2005].

5 Layout Optimization

Given the cache-oblivious metric, our goal is to find the layout, φ, that minimizes the expected number of cache misses, defined in Eq. (1). This is a combinatorial optimization problem for graph layouts [Diaz et al. 2002]. Finding a globally optimal layout is NP-hard [Garey et al. 1976] due to the large number of permutations of the set of vertices. Instead, we use a heuristic based on *multilevel minimization* that performs local permutations to compute a locally optimal layout.

Multilevel Minimization: Our multilevel algorithm consists of three main steps. First, a series of coarsening operations on the graph are computed. For coarsening operations, we perform clustering via a graph partitioning [Karypis and Kumar 1998]. Next, we compute an ordering of vertices of the coarsest graph. We list all possible orderings of the vertices and compute the costs based on the cache-oblivious metric from Eq. (7). We choose a vertex ordering that has the minimum cost. Finally, we recursively expand the graph by reversing the coarsening operations and refine the ordering by performing *local permutations*.

Local Permutation: We compute local permutations of the vertices during the ordering and refinement steps. A local permutation affects only a small number of vertices in the layout and changes the edge spans of those edges that are incident to these vertices. Therefore, we can efficiently recompute the cost associated with the metric. For efficiency we restrict each coarsening operation to merge no more than $k = 5$ vertices at a time.

Model	Type	Vert. (M)	Tri. (M)	Size (MB)	Layout Comp. (min)
Dragon	s	0.4	0.8	33	0.25
Lucy	s	14.0	28.0	520	8
Double Eagle	c	77.7	81.7	3,346	56
Puget Sound	t	67.0	134.0	1,675	58
St. Matthew	s	186.0	372.0	9,611	176

Table 1: **Layout Benchmarks:** Model complexity and time spent on layout computation are shown. Type indicates model type: s for scanned model, i for isosurface, c for CAD model, and t for terrain model. Vert. is the number of vertices and Tri. is the number of triangles of a model. Layout Comp. is time spent on layout computation.

Model	Double Eagle	Isosurface	St. Matthew
PoE	3	5	1
Frame rate	35	30	82
Rendering throughput(million tri./sec.)	47	90	106
Avg. Improvement	2.1	4.5	4.6

Table 2: **View-Dependent Rendering** This table highlights the frame rate and rendering throughput for different models. We improve the rendering throughput and frame rates by $2.1 - 4.6$ times.

6 Implementation and Performance

In this section we describe our implementation and use cache coherent layouts to improve the performance of three applications: view-dependent rendering of massive models, collision detection between complex models, and isocontour extraction.

6.1 View-dependent rendering

View-dependent rendering and simplification are frequently used for interactive display of massive models. These algorithms precompute a multiresolution hierarchy of a large model (e.g. a vertex hierarchy). At runtime, a dynamic simplification of the model is computed by incrementally traversing the hierarchy until the desired pixels of error (PoE) tolerance in image space is met.

We use a clustered hierarchy of progressive meshes (CHPM) representation [Yoon et al. 2004] for view-dependent refinement. We computed layouts for three massive models including a CAD environment of a tanker with 127K separate objects (Fig. 2), a scanned model of St. Matthew (Fig. 1) and an isosurface model . The details of these models are summarized in Table 1.

Results: Table 2 highlights the benefit of COL over the simplification layout (SL), whose vertex layout and triangle layout are computed by the underlying simplification algorithm. We are able to increase the rendering throughput by a factor of of 2.1-4.6 times by precomputing a COL of the CHPM of each model. We obtain a rendering throughout of 106M triangles per second on average, with a peak performance of 145M triangles per second.

6.2 Collision Detection

Many collision detection algorithms use bounding volume hierarchies to accelerate the interference computations [Lin and Manocha 2003]. In particular, we compute layouts of OBB-trees [Gottschalk et al. 1996] and use them to accelerate collision queries within a dynamic simulator. We have tested the performance of our collision detection algorithm in a rigid body simulation where 20 dragons (800K triangles each) drop on the Lucy model (28M triangles). The details of these models are shown in Table 1. Fig. 6 shows a snapshot from our simulation. We compared our cache-oblivious layout with a depth-first layout (DFL) of OBB-trees. We chose DFL because it

Figure 6: **Dynamic Simulation:** Dragons consisting of 800K triangles are dropping on the Lucy model consisting of 28M triangles. We obtain 2 times improvement by using COL on average.

preserves the spatial locality within the bounding volume hierarchy.

Results: We are able to achieve 2 times improvement in performance over the depth-first layout on average. This is mainly due to reduced cache misses, including main memory page faults. We would like to point out that we further improve the performance of collision detection by separately considering two different localities in an algorithm [Yoon and Manocha 2005]. We are able to achieve up to 5 times improvement by using this new algorithm as compared to the DFL layout.

6.3 Isocontour Extraction

The problem of extracting an isocontour from an unstructured dataset frequently arises in geographic information systems and scientific visualization. We use an algorithm based on seeds sets [van Kreveld et al. 1997] to extract the isocontour of a single-resolution mesh. The running time of this algorithm is dominated by the traversal of the triangles intersecting the contour itself. We compare the performance of the isocontouring algorithm with five different layouts including our cache-oblivious layouts, geometric $X/Y/Z$ orders (vertices sorted by their position along the corresponding coordinate axis) and in spectral sequencing order [Diaz et al. 2002].

Comparison with other layouts: The empirical data shows that our cache-oblivious layout minimizes the worst case cost of generic coherent traversals. The three layouts that are sorted by geometric direction along the X, Y, and Z axis show that the worst case performance is at least one order of magnitude slower than the best case, which is achieved by the layout that happens to be perfectly aligned along the query direction. The spectral sequencing layout also does not perform well since the geometric query is unlikely to follow its streaming order. Our cache-oblivious layout consistently exhibits good performance (up to 20 times speedup) compared to other layouts.

6.4 Limitations

Our metric and layout computation algorithm has several limitations. The assumptions we make about invariance and monotonicity of CMRFs may not hold for all applications, and our minimization algorithm does not necessarily compute a globally optimal solution. Our cache-oblivious layouts result in good improvements primarily in applications where the running time is dominated by data access.

Acknowledgments

This work was supported in part by ARO Contracts DAAD19-02-1-0390 and W911NF-04-1-0088, NSF awards 0400134 and 0118743, DARPA/RDECOM Contract N61339-04-C-0043 and Intel. Some of the work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48. The St. Matthew, Lucy, and Atlas models are courtesy of the Digital Michelangelo Project at Stanford University. The isosurface model is courtesy of the LLNL ASCI VIEWS Visualization project and the Double Eagle tanker is courtesy of Newport News Shipbuilding. We would like to thank Martin Isenburg, Brandon Lloyd, Brian Salomon, Avneesh Sud, Dawoon Jung, and the members of UNC Walkthrough and Gamma group for their feedback on an earlier draft of the paper and technical discussions.

References

ARGE, L., BRODAL, G., AND FAGERBERG, R. 2004. Cache oblivious data structures. *Handbook on Data Structures and Applications*.

BARTHOLDI, J., AND GOLDSMAN, P. 2004. Multiresolution indexing of triangulated irregular networks. In *IEEE Transaction on Visualization and Computer Graphics*, 484–495.

BOGOMJAKOV, A., AND GOTSMAN, C. 2002. Universal rendering sequences for transparent vertex caching of progressive meshes. In *Computer Graphics Forum*, 137–148.

COLEMAN, S., AND MCKINLEY, K. 1995. Tile size selection using cache organization and data layout. *SIGPLAN Conference on Programming Language Design and Implementation*, 279–290.

DEERING, M. F. 1995. Geometry compression. In *SIGGRAPH 95 Conference Proceedings*, Addison Wesley, R. Cook, Ed., Annual Conference Series, ACM SIGGRAPH, 13–20. held in Los Angeles, California, 06-11 August 1995.

DIAZ, J., PETIT, J., AND SERNA, M. 2002. A survey on graph layout problems. *ACM Computing Surveys 34*, 313–356.

FRIGO, M., LEISERSON, C., PROKOP, H., AND RAMACHANDRAN, S. 1999. Cache-oblivious algorithms. *Symposium on Foundations of Computer Science*, 285–297.

GAREY, M. R., JOHNSON, D., AND STOCKMEYER, L. 1976. Some simplified np-complete graph problems. *Theoretical Computer Science 1*, 237–267.

GOPI, M., AND EPPSTEIN, D. 2004. Single-strip triangulation of manifolds with arbitrary topology. In *EUROGRAPHICS*, 371–379.

GOTTSCHALK, S., LIN, M., AND MANOCHA, D. 1996. OBB-Tree: A hierarchical structure for rapid interference detection. *Proc. of ACM Siggraph'96*, 171–180.

HEBER, G., BISWAS, R., AND GAO, G. 2000. Self-avoiding walks over adaptive unstructured grids. In *Concurrency: Practice and Experience*, 85–109.

HOPPE, H. 1999. Optimization of mesh locality for transparent vertex caching. *Proc. of ACM SIGGRAPH*, 269–276.

ISENBURG, M., AND GUMHOLD, S. 2003. Out-of-core compression for gigantic polygon meshes. In *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH)*, vol. 22, 935–942.

ISENBURG, M., AND LINDSTROM, P. 2004. Streaming meshes. Tech. Rep. UCRL-CONF-201992, LLNL.

KARNI, Z., BOGOMJAKOV, A., AND GOTSMAN, C. 2002. Efficient compression and rendering of multi-resolution meshes. In *IEEE Visualization*, 347–354.

KARYPIS, G., AND KUMAR, V. 1998. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 96–129.

LASSERRE, J. B., AND ZERON, E. S. 2001. A laplace transform algorithm for the volume of a convex polytope. *Journal of the ACM*, 1126–1140.

LIN, M., AND MANOCHA, D. 2003. Collision and proximity queries. In *Handbook of Discrete and Computational Geometry*.

LINDSTROM, P., AND PASCUCCI, V. 2001. Visualization of large terrains made easy. *IEEE Visualization*, 363–370.

OLIKER, L., LI, X., HUSBANDS, P., AND BISWAS, R. 2002. Effects of ordering strategies and programming paradigms on sparse matrix computations. In *SIAM Review*, 373–393.

PASCUCCI, V., AND FRANK, R. J. 2001. Global static indexing for real-time exploration of very large regular grids. *Super Computing*, 225–241.

SAGAN, H. 1994. *Space-Filling Curves*. Springer-Verlag.

SEN, S., CHATTERJEE, S., AND DUMIR, N. 2002. Towards a theory of cache-efficient algorithms. *Journal of the ACM 49*, 828–858.

VAN KREVELD, M., VAN OOSTRUM, R., BAJAJ, C., PASCUCCI, V., AND SCHIKORE, D. R. 1997. Contour trees and small seed sets for isosurface traversal. In *Proceedings of the 13th International Annual Symposium on Computational Geometry (SCG-97)*, ACM Press, New York, 212–220.

VELHO, L., AND GOMES, J. D. 1991. Digital halftoning with space filling curves. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, T. W. Sederberg, Ed., vol. 25, 81–90.

VITTER, J. 2001. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 209–271.

YOON, S.-E., AND MANOCHA, D. 2005. Cache-Oblivious Layouts of Bounding Volume Hierarchies. Tech. rep., University of North Carolina-Chapel Hill.

YOON, S.-E., SALOMON, B., GAYLE, R., AND MANOCHA, D. 2004. Quick-VDR: Interactive View-dependent Rendering of Massive Models. *IEEE Visualization*, 131–138.

YOON, S.-E., LINDSTROM, P., PASCUCCI, V., AND MANOCHA, D. 2005. Cache-Oblivious Mesh Layouts. *To appear in Proc. of ACM SIGGRAPH*.

Sorting points from $\mathbb{R}^k$ into Hilbert order

Yuanxin Liu
liuy@cs.unc.edu

Ajith Mascarenhas
ajith@cs.unc.edu

Jack Snoeyink
snoeyink@cs.unc.edu

Department of Computer Science
Campus Box 3175, Sitterson Hall
UNC-Chapel Hill
Chapel Hill, NC 27599-3175, USA

ABSTRACT

There are a number of orderings of the entries of a matrix or cells of a regular grid that try to improve locality of reference by keeping spatially nearby cells close in the linear order: examples include Morton or Z order, Gray code order, and Hilbert curve order. Of these, only Hilbert curves preserve adjacency.

There are a couple of ways to transform grid coordinates to Hilbert order number sand vice versa: The basic definition leads to a slow, top-down recursive procedure that maintains state; faster implementations on the web use difficult to understand boolean operations suggested by Butz (1971). In our applications we want to permute large sets of points into Hilbert order by incorporating the generation into an out-of-core sorting method.

We give a notation for Hilbert curves that allows use to explain Butz's algorithm, and to create a fast procedure for permuting large sets of points into Hilbert order. This can serve as the basis for subsequent out-of-core processing based on spatial locality.

Computing Pfafstetter Labelings I/O-Efficiently

(abstract)

Lars Arge*
Department of Computer Science, University of Aarhus
Aabogade 34, DK-8200 Aarhus N, Denmark
large@daimi.au.dk

Andrew Danner†
Department of Computer Science, Duke University
P.O.Box 90129, Durham, NC 27708-0129, USA
adanner@cs.duke.edu

Herman Haverkort‡
Dept. of Computer Science, University of Aarhus
Aabogade 34, DK-8200 Aarhus N, Denmark
cs.herman@haverkort.net

Norbert Zeh§
Faculty of Computer Science, Dalhousie University
6050 University Ave, Halifax, NS B3H 1W5, Canada
nzeh@cs.dal.ca

ABSTRACT

We present an I/O-efficient algorithm that decomposes a grid-based terrain model into a hierarchy of watersheds. Each watershed gets a unique label, a *Pfafstetter* label, and each grid cell is labeled with the labels of all (nested) watersheds it belongs to. The algorithm runs in $\mathcal{O}(\mathrm{sort}(T))$ I/Os, where T is the total length of the computed cell labels. Our algorithm is simple and practical. We substantiate these claims by presenting experimental results that verify the performance of our algorithm.

1. INTRODUCTION

Over millions of years, rainfall has been slowly etching networks of rivers into the terrain. Today, studying these river networks is important for managing drinking water supplies, tracking pollutants, creating flood maps, and more. Hydrologists can use large-scale raster-based digital elevation models, or DEMs, of the terrain along with a Geographic Information System, or GIS, to automate much of these studies. Often it is not necessary to study the entire terrain or river network at once. People are typically interested in regions that are downstream of a particular river, or the upstream areas that contribute flow to a particular river. By decomposing the terrain into a set of disjoint *hydrologic units*—regions where all water flows towards a single, common outlet—users can quickly identify areas of interest without having to examine the entire terrain. The Pfafstetter labeling method described by Verdin and Verdin [10] defines a hierarchical decomposition of a terrain into such units, each with a unique ID, or Pfafstetter label. These labels can be computed automatically, given a network of rivers and their drainage area. Pfafstetter labels encode topological properties such as upstream and downstream neighbors, making it possible to automatically identify hydrological units of interest based on the Pfafstetter ID alone.

Existing algorithms for computing hydrological units on grid-based terrain models typically use either local filters to identify terrain features [7, 9] or model flow over the entire terrain [8] and then extract watersheds. Most of these algorithms are not designed to handle large data sets.

In this paper, we show how to compute Pfafstetter labels efficiently on grid-based DEMs that are too large to fit into the main memory of a computer and must therefore reside on disks, which are larger, but also considerably slower. To our knowledge, this paper provides the first algorithmic analysis and experimental running times of Pfafstetter label computation.

1.1 Pfafstetter labeling on grids

Conceptually, the definition of Pfafstetter labeling is independent of the representation of the terrain, but for concreteness, we give a definition tailored to a grid-based terrain model. A planar orthogonal *grid* or *raster* is a pattern of horizontal and vertical lines that divide the plane into isometric rectangular cells. In geographic information systems, we use grids to model properties of the Earth's surface: we project a grid onto the surface and store the value of the property of interest for each cell (for example, the elevation of the cell's center). The Pfafstetter labeling of a grid-based terrain model is defined by the *flow directions* and the *drainage areas* of the cells. Each cell u in the grid has eight neighbors that share at least one vertex with u. The flow direction of u is a pointer to the neighbor cell to which water that falls on or flows through u is assumed to flow. The grid can thus be seen as a *flow graph* that has a node for each cell in the grid, and in which there is a directed edge (u, v) if and only if the flow direction of u points to neighbor v. A cell w *drains through* u if there is a path in the graph, following the flow directions, from w to u. The *drainage area* of a cell u is the total area of the cells–including u–that drain through u. Flow graphs without cycles and corresponding drainage areas can be computed from digital elevation models using a few easy-to-use GIS tools. The TerraFlow software package [4] in particular computes these grids efficiently on large elevation models.

For simplicity, we assume that our input consists of a grid representing the river basin of a single river. This means that there is one unique cell ρ, the mouth of the river, whose flow direction points to a cell that is not among the cells in our input. The flow graph must therefore be a single tree $\mathcal{T}$

*Supported in part by the US National Science Foundation through RI grant EIA–9972879, CAREER grant CCR–9984099, ITR grant EIA–0112849, and U.S.-Germany Cooperative Research Program grant INT–0129182, by the US Army Research Office through grant W911NF-04-1-0278, and by a Ole Rømer Scholarship from the Danish National Science Research Council. Part of this work was done while the author was at Duke University.
†Supported in part by the US National Science Foundation through grant CCR–9984099.
‡Supported by a grant from the Danish National Science Research Council.
§Supported by the Natural Sciences and Engineering Research Council of Canada and the Canadian Foundation for Innovation.

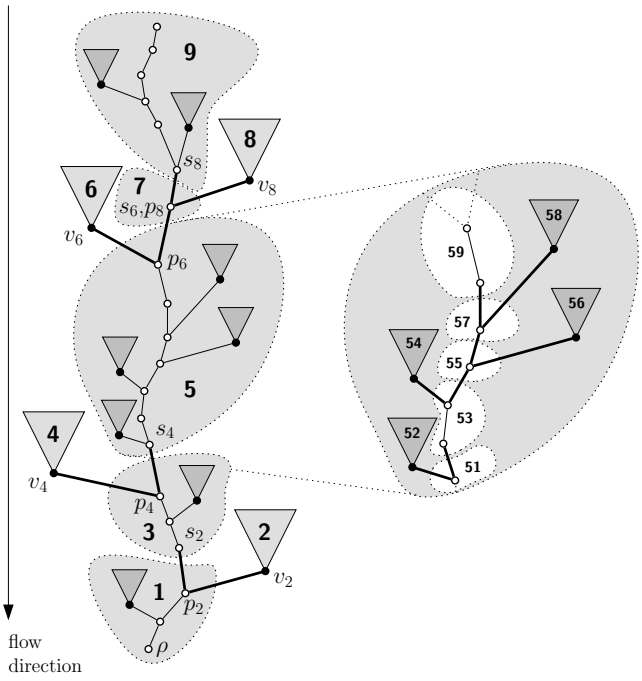


Figure 1. A flow graph $\mathcal{T}$ with the main river path ('blue' cells, shown here as circles) and mouths of tributaries (black dots). Removing the eight bold edges creates nine subtrees, each with the Pfafstetter label shown in bold type. Each of these subtrees will be subdivided and labeled recursively.

with root ρ.[1] The *main river path*, $\mathcal{R}_0$, of $\mathcal{T}$ is a path that starts at the root ρ and, at each cell, continues, against the flow direction, to the child with highest drainage area, until it ends in a leaf. Imagine that all cells of $\mathcal{R}_0$ are colored blue and all other cells are currently black. For simplicity, we assume that each blue cell has at most one black child.[2] We define a subtree of $\mathcal{T}$ to be a *tributary basin* if the root of the subtree, v, is black, but the parent of v is blue. We call v a *tributary mouth*.

Consider the four tributary mouths v_2, v_4, v_6, v_8 with the largest drainage area, where for $i < j$, the mouth v_i flows into the main river downstream of v_j. Let p_i and s_i denote the parent and the sibling of v_i, respectively. Consider the nine subtrees resulting from the removal of the eight edges (v_i, p_i) and (s_i, p_i), for $i \in \{2, 4, 6, 8\}$, from $\mathcal{T}$. Four of these subtrees are tributary basins, they are rooted at v_2, v_4, v_6, and v_8. The Pfafstetter label for a cell in a subtree rooted at v_i is i. The remaining subtrees are called *interbasins* and are rooted at ρ, s_2, s_4, s_6 and s_8. All cells in the subtree rooted at ρ, the root of $\mathcal{T}$, have label 1. For a cell in a subtree rooted at s_i the Pfafstetter label is $i+1$. See Figure 1 for an example decomposition. In the case where a flow graph has $0 < k < 4$ tributary mouths, we proceed as above but do not assign the labels $2k+2$ through 9. Each of the (at most) nine subtrees is labeled recursively by applying the definition just given and appending the resulting labels to the existing label of the subtree—see, for example, interbasin 5 in Figure 1. The recursive labeling stops when each subtree is a single root-leaf path.

1.2 I/O Model

Because on large data sets, the efficiency of an algorithm tends to be dominated by the time spent on transferring data between main memory and disk, we analyse our algorithms under the standard I/O-model proposed by Aggarwal and Vitter [1]. In this model, computation only occurs on data located in a main memory with a capacity of M elements. An I/O transfers a block of B consecutive elements between main memory and a disk of conceptually infinite capacity. The complexity measure of an algorithm in this model is the number of I/Os it performs. Algorithms with low complexity under this model are called *I/O-efficient* and perform well even on large data sets. Trivially, the complexity of scanning N elements is $\text{scan}(N) = \Theta(\frac{N}{B})$. Aggarwal and Vitter showed that the complexity of sorting N elements is $\text{sort}(N) = \Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$. Note that $\text{sort}(N)$ is typically much smaller than N. In the past decades, a number of I/O-efficient data structures have been described, including stacks on which N operations can be performed in $\mathcal{O}(\text{scan}(N))$ I/Os and priority queues on which N insertions and extractions can be performed in $\mathcal{O}(\text{sort}(N))$ I/Os [2, 5].

1.3 Our results

In this paper, we present an I/O-efficient algorithm that computes the Pfafstetter labels of the cells in a grid-based terrain model in $\mathcal{O}(\text{sort}(T))$ I/Os, where T is the total length of the computed labels. In practice, we are only interested in the $\mathcal{O}(1)$ most significant digits of the labels, so that each label can be truncated and encoded in $\mathcal{O}(1)$ bytes. Then the computations take only $\mathcal{O}(\text{sort}(N))$ I/Os, where N is the number of grid cells.

When the input, the output, and some $\mathcal{O}(N)$-size auxiliary data structures fit in internal memory, we can compute the labeling in $\mathcal{O}(T)$ time (or $\mathcal{O}(N)$, if we are only interested in the $\mathcal{O}(1)$ most significant digits of the labels).

The remainder of the paper is structured as follows. As a first step towards a solution, we define a simpler problem in Section 2, namely the computation of Pfafstetter labels on a flow graph that represents a single river whose tributary basins consist of only one cell each. We describe a data structure known as the Cartesian tree and show how to use it to compute Pfafstetter labels on such a river. In Section 3, we discuss how to decompose a grid model of a general river basin with flow directions and drainage areas into a tree of tributaries, each of which can be labeled with a local Pfafstetter label independently using the algorithm in Section 2. We conclude the description and analysis of our algorithm with Section 4, where we describe how to label a complete river basin by combining the local Pfafstetter labels into complete labels for each cell in the river basin. We present some experimental results showing the scalability and performance of our algorithm in Section 5, and give some concluding remarks in Section 6. We omit internal-memory algorithms and the analysis for truncated labels from this abstract.

2. COMPUTING PFAFSTETTER LABELS ON A SINGLE RIVER

In this section, we consider a flow graph as defined in Section 1.1 where each subtree attached to the main river consists of a single leaf. These leaves do not need to have the same drainage areas. Refer to Figure 2 for an example. We will show how to compute the Pfafstetter labels as defined earlier on such a pruned flow graph.

As before, let the cells on the main river be colored blue, while the remaining cells are colored black. We assume that the cells are given as a list L such that the blue cells are ordered from mouth to source, and each black cell is placed between its parent and its sibling. Our goal is to compute

[1] Grids with multiple basins are in fact quite easy to handle, but they would complicate the exposition in this paper unnecessarily.
[2] We could enforce this by expanding each blue cell into a number of consecutive blue nodes, one for each child.

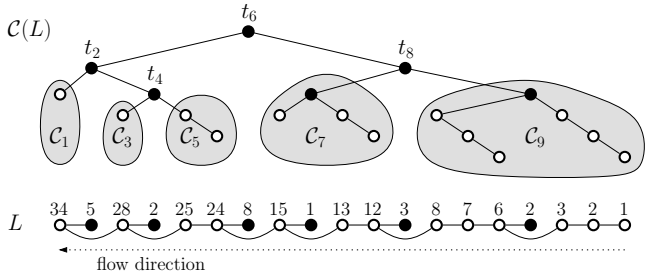


Figure 2. *Bottom figure:* a flow graph that consists of a single river, where each subtree not on the main river is a single leaf. The numbers are the drainage areas of the cells. List L contains the cells of the flow graph from left to right. *Top figure:* the Cartesian tree on L, with its four heaviest nodes and the five subtrees between them.

the Pfafstetter label for each element in L. To this end, we scan L to compute an augmented Cartesian tree on the elements of L, as explained in Section 2.1; then we process this tree recursively to compute the labels for all elements in the tree, as explained in Section 2.2.

2.1 Cartesian Tree

Let $A = (a_1, a_2, \ldots, a_N)$ be a sequence of N distinct weights. The Cartesian tree [6], $\mathcal{C}(A)$, of the sequence A is defined as follows: if A is empty, $\mathcal{C}(A)$ is empty. For $N > 0$, let a_i be the largest element in A. The Cartesian tree of A consists of a root v that contains $a_v := a_i$, a left subtree that is the Cartesian tree $\mathcal{C}((a_1, ..., a_{i-1}))$ of the elements to the left of a_i, and a right subtree that is the Cartesian tree $\mathcal{C}((a_{i+1}, ..., a_N))$ of the elements to the right of a_i. The internal-memory algorithm for the construction of a Cartesian tree takes $\mathcal{O}(N)$ time [6]. When implemented carefully using two stacks that hold the nodes of the tree under construction, the algorithm is I/O-efficient, taking $\mathcal{O}(\text{scan}(N))$ I/Os to output the nodes of the tree in post-order[3].

To be able to label a river with Pfafstetter labels as explained in the next subsection, we store with every node in the Cartesian tree the four heaviest elements among its descendants (including itself). The four heaviest elements below every node can be determined by a straightforward post-order traversal of the tree. Because the tree is constructed incrementally in post-order, we can perform this post-order traversal while the tree is being constructed, without increasing the number of I/Os by more than a constant factor. We omit the details from this abstract.

2.2 Labeling a river

Recall that a single river is represented by a list L consisting of a main river of blue cells and a set of black tributary cells each of which is stored between its blue parent and its blue sibling. We build an augmented Cartesian tree on these cells as described above, where the weight of a cell is defined as follows: A black cell's weight is equal to its drainage area, while every blue cell has weight zero. When cells of equal weight need to be compared, the cell that appears first in the list is considered to have the highest weight. With each cell we store not only its position in the list and its weight, but also its location in the grid.

When the river has at least one tributary (black cell), the root of the tree now stores the tributary t with biggest drainage area, along with the three next-biggest tributaries (if they exist); the left child of the root is a Cartesian tree on the cells that lie in or flow into the main river downstream of t (excluding t itself), while the right child is a Cartesian tree on the cells upstream of t's parent.

Observation 1 *The four weights stored in the root of the augmented Cartesian tree $\mathcal{C}$ are the weights of the nodes in a connected subgraph of $\mathcal{C}$ that includes the root.*

We can now label the complete list L recursively as follows. Each recursive call is parameterized with a node of $\mathcal{C}$ and the Pfafstetter label of an interbasin. The recursion starts by calling the algorithm on the root of the tree with an empty label.

When called on a node v, we find node v in $\mathcal{C}$ and examine v for the four heaviest cells in the tree rooted at v. If all of them have weight zero, the tree rooted at v represents a stretch of river without black cells, that is, without confluences with tributaries. We then label all nodes under v with the given interbasin label from right to left in the order in which they appear in L.

Otherwise, we order the heaviest cells under (and including) v from left to right according to their position in L into a list t_2, t_4, t_6, t_8. Assume for the moment that all four heaviest nodes exist and have positive weight, that is, they represent black cells. Because by Observation 1, these four nodes together form a tree with three edges, and because the Cartesian tree is a binary tree, these nodes together have at most five children other than t_2, t_4, t_6 and t_8. These children are the roots of the five subtrees $\mathcal{C}_1, \mathcal{C}_3, \mathcal{C}_5, \mathcal{C}_7, \mathcal{C}_9$ that would be obtained by removing the nodes of t_2, t_4, t_6 and t_8 from the Cartesian tree—see Figure 2 for an example. $\mathcal{C}_1$ contains all cells that lie on or flow into the main river downstream of t_2 (excluding t_2). For $i \in \{3, 5, 7\}$, subtree $\mathcal{C}_i$ contains all cells that lie on or flow into the main river upstream of the parent of t_{i-1} and downstream of t_{i+1} (excluding t_{i+1}). $\mathcal{C}_9$ contains all cells that lie upstream of t_8's parent. We now proceed as follows. We label $\mathcal{C}_9$ recursively by recursing on the root of $\mathcal{C}_9$ with the interbasin label equal to the given interbasin label plus the digit "9". For $i = 8, 6, 4, 2$ (going in downstream order), we label t_i (which is stored in v) with the given interbasin label plus the digit i, and then recurse on the root of $\mathcal{C}_{i-1}$ with the given interbasin label plus the digit $i - 1$.

If v has $k < 4$ black descendants, we order them by their position in L into a list $t_2, t_4, \ldots, t_{2k}$: subtrees $\mathcal{C}_{2k+3}, \ldots, \mathcal{C}_9$ do not exist and are not labeled.

Lemma 1 *The Pfafstetter labels for the N cells in a list L that represents a single river can be computed and output from right to left in $\mathcal{O}(\text{scan}(T))$ I/Os, where T is the total size of the computed labels.*

Proof. We implement the above algorithm by first computing a post-order listing of a Cartesian tree on L as explained in Section 2.1. This costs $\mathcal{O}(\text{scan}(N))$ I/Os.

When the recursive labeling algorithm visits a node u, it always visits it before any descendants of u, and it visits any descendants in the right subtree of u before any descendants in the left subtree of u. The algorithm thus visits the nodes of the Cartesian tree in the reverse order of left-to-right post-order (skipping nodes that are the second-, third- or fourth-heaviest nodes below nodes that have already been visited). The nodes to recurse on can thus be obtained in $\mathcal{O}(\text{scan}(N))$ I/Os in total by putting the post-order listing of the tree on a stack and popping nodes from it as needed.

Outputting the labels of all cells in L takes $\mathcal{O}(\text{scan}(T))$ I/Os, where T is the total size of the computed labels.

We omit further details from this abstract. □

[3] A post-order listing of a binary tree is a list of its nodes that consists of the post-order listing of the left subtree of the root, followed by the post-order listing of the right subtree of the root, followed by the root.

3. DECOMPOSING A TERRAIN INTO RIVERS

Above we explained how to label a single river $\mathcal{R}_i$ when given as a list L_i of blue cells ordered from mouth to source, with tributary mouths placed as black cells between their parents and their siblings. In this section we show that we can efficiently decompose a grid-based terrain model into a set of rivers and construct such a list for each river. Moreoever, our decomposition constitutes a hierarchy of tributaries, a *tributary tree*, where each vertex stores a river $\mathcal{R}_i$ represented by a list L_i, and where $\mathcal{R}_i$ is a child of $\mathcal{R}_j$ if and only if $\mathcal{R}_i$ flows directly into $\mathcal{R}_j$, that is, the mouth of $\mathcal{R}_i$ is a black cell in L_j. We consider the children of each node $\mathcal{R}_j$ in this tree to be ordered from left to right according to the ordering of their mouths in L_j.

Recall the flow graph $\mathcal{T}$ of Section 1.1 shown in Figure 1. Without loss of generality, we assume that the minimum drainage area of any cell in $\mathcal{T}$ is one. The first river in our decomposition is a root-leaf path, $\mathcal{R}_0$ of $\mathcal{T}$ defined by starting at the root and, at each cell, continuing to the child with highest drainage area. The list L_0 for $\mathcal{R}_0$ is the list of all cells along the path $\mathcal{R}_0$ in order from mouth to source, along with the remaining children of those cells. The remaining rivers are defined by applying the definition recursively to each tributary of $\mathcal{R}_0$, that is, to each subtree rooted at a node v such that the parent of v is on $\mathcal{R}_0$, but v is not. This defines a set of rivers, each represented by a list of cells in that river (blue cells) and mouths of tributaries (black cells). Each cell in $\mathcal{T}$ appears in one or two such lists: once as a blue node in the list of the river that flows through that cell, and, if the cell is the mouth of a river $\mathcal{R}_i \neq \mathcal{R}_0$, once as a black node in the list of the river to which $\mathcal{R}_i$ is a tributary.

The above definition could be translated in a straightforward way into a depth-first traversal of $\mathcal{T}$ that generates all river lists and produces a pre-order listing of the nodes of the tributary tree in $O(N)$ CPU operations. However, when the input does not fit in internal memory and I/O-efficiency determines the running time, we need another approach.

We compute the decomposition into rivers by processing the flow graph $\mathcal{T}$ from the root to the leaves, constructing each river's list incrementally from the mouth to the source. We do not construct the rivers one by one, but in parallel: at each point in the process, there may be several rivers under construction. To organize this process, we assign each river a unique integer ID as soon as the parent of its mouth in $\mathcal{T}$ is visited, and maintain each river's state as a quintuple $(RID, RLen, v, TID^-, TID^+)$, where:

- RID is the ID of the river;
- $RLen$ is the number of elements that were already appended to its list L_{RID};
- v is the next cell to append to L_{RID};
- $\{TID^-, ..., TID^+\}$ are IDs that are reserved to be assigned to tributaries of river R_{RID}.

The river states are kept in a priority queue, where highest priority is given to the river whose next cell has highest drainage area, with ties broken arbitrarily. Initially we set up a priority queue with one river state that is the state of the main river $\mathcal{R}_0$ before any cells have been added to its list—more precisely, this river state is initialized as $(RID = 0, RLen = 0, v = \rho, TID^- = 1, TID^+ = \infty)$, where ρ is the root of $\mathcal{T}$, that is, the mouth of $\mathcal{R}_0$.

We copy the flow graph $\mathcal{T}$ and store with each cell a copy of its children, sort the cells by decreasing drainage area (with ties broken in the same way as in the priority queue), and put them on a stack of cells still to be processed. The cell with highest drainage area, which must be the mouth of the main river, is put on top of stack.

We now repeat the following until the stack is empty. We pop a cell v from the stack of cells to be processed. The first time we do this, v is the mouth of the main river, and the state of the main river is the only river state in the priority queue. Every other time, v has a parent, which has bigger drainage area and therefore must have been popped from the stack and dealt with before. Therefore the mouth of the river $\mathcal{R}$ that contains v must have been found already, and the state of $\mathcal{R}$ must be in the priority queue. Furthermore, since v is the unprocessed cell with highest drainage area, river $\mathcal{R}$ must have highest priority. We extract the river state with highest priority from the priority queue, and thus obtain the ID RID of $\mathcal{R}$, the length $RLen$ of the list L_{RID} of $\mathcal{R}$, and the minimum and maximum ID TID^- and TID^+ available to name tributaries. We now process v as follows.

We first increase $RLen$ by one and append v as element number $RLen$ to L_{RID}, colored blue (we will discuss later how to do this I/O-efficiently). Then we look at v's children.

If v has no children, we are done with v: river $\mathcal{R}$ ends here and there are no tributaries to discover, so we proceed to processing the next cell on the stack.

If v has one child, it must be the next cell v_{blue} upstream on $\mathcal{R}$. The current state of $\mathcal{R}$ is thus described by $(RID, RLen, v_{blue}, TID^-, TID^+)$. We insert that state into the priority queue and proceed to processing the next cell on the stack.

If v has two children, the one with biggest drainage area is, by definition, the next cell v_{blue} upstream on $\mathcal{R}$, and the other one must be the mouth v_{black} of a tributary to $\mathcal{R}$. We increase $RLen$ by one again, and append v_{black} as element number $RLen$ to L_{RID}, colored black. Since the cell v we are visiting is the parent of tributary mouth v_{black}, we must now give that tributary an ID and insert its state into the priority queue. We assign it ID TID^-, initialize its state to $(TID^-, 0, v_{black}, TID^- + 1, TID^- + drainageArea(v_{black}) - 1)$, and insert it into the queue. Thus we reserve IDs $TID^- + 1$ through $TID^- + drainageArea(v_{black}) - 1$ for tributaries of the newly discovered river $\mathcal{R}_{TID^-}$. The state of $\mathcal{R}$ is now described by $(RID, RLen, v_{blue}, TID^- + drainageArea(v_{black}), TID^+)$. We insert that river state into the priority queue, and proceed to processing the next cell on the stack.

Lemma 2 *In $\mathcal{O}(\mathrm{sort}(N))$ I/Os a grid-based elevation model can be decomposed into a pre-order listing of the rivers in the tree of tributaries, such that each $\mathcal{R}_i$ is returned as a list L_i that contains the cells in the river from mouth to source, with tributary mouths placed between their parents and their siblings in the flow graph.*

Proof. We implement the above algorithm as follows. We run TERRAFLOW [4] on the input elevation grid to get a flow direction and a drainage area for each cell in $\mathcal{O}(\mathrm{sort}(N))$ I/Os. We scan the flow direction and drainage area grid with a 3x3 window in $\mathcal{O}(\mathrm{scan}(N))$ I/Os to create a list of all cells in the grid, where each cell stores not only its own drainage area, but also the drainage areas of all of its children in $\mathcal{T}$. Then we sort this list by decreasing drainage area in $\mathcal{O}(\mathrm{sort}(N))$ I/Os and put it on a stack of unprocessed cells, the cell with highest drainage area on top. The processing of each cell v requires one stack operation, one extraction from the priority queue, inspecting the drainage areas of the children of v (which are stored with v), up to two insertions into the priority queue, and up to two additions to a river list. Using I/O-efficient stacks and priority queues, all $\mathcal{O}(N)$ stack and queue operations can be carried out in $\mathcal{O}(\mathrm{sort}(N))$ I/Os [2, 5]. We implement the additions to the river lists by maintaining one big list L^* with elements of the form $(v, RID, ROff, color)$, where v is a grid cell with

its drainage area and grid location, *RID* is the ID of the river that contains v, *ROff* is the position of v in the list L_{RID} of river *RID*, and *color* is the color of the cell in that list (blue or black). When we append a cell v with color *color* as element number *ROff* to list L_{RID}, we simply append $(v, RID, ROff, color)$ to L^*. When the complete algorithm is done, we sort L^* by *RID* and *ROff* in $\mathcal{O}(\text{sort}(N))$ I/Os to obtain the lists per river. Thus the total number of I/Os needed to obtain the lists for all rivers in a given watershed is $\mathcal{O}(\text{sort}(N))$. The way in which the river IDs are assigned guarantees that sorting by river ID automatically gives a pre-order listing of the rivers in the tree of tributaries.

We omit the correctness proof from this abstract. □

4. LABELING A COMPLETE BASIN

Consider the main river $\mathcal{R}_0$ of the flow graph $\mathcal{T}$ represented by a list L_0. Each cell in $\mathcal{T}$ is either a blue cell in L_0 or is part of some subtree whose root r is a black cell in L_0. For each blue cell u in L_0, the Pfafstetter label is simply the label of u in L_0 as assigned by the algorithm of Section 2. The Pfafstetter label for a cell in a subtree rooted at a black cell v in L_0 is the label of v in L_0 concatenated with the recursive labeling of the subtree rooted at v.

We can thus label all cells in the terrain as follows. We decompose the terrain into a pre-order listing of a tree of tributaries, each represented by a list of blue and black cells, as explained in Section 3. We initialize an empty stack of label prefixes and push an empty label on it. Then we process the rivers in the tree of tributaries one by one in pre-order. For each river $\mathcal{R}_i$, we pop a prefix from the stack and label the cells in its list L_i with the algorithm from Section 2, while prefixing all labels with the prefix popped from the stack. We append the labeled blue cells to a list of labeled cells. We push the labels for the black cells on the stack in the reverse order in which they appear in L_i, to be used as prefixes for the child rivers in the tributary tree. When we have labeled all rivers, we sort the labeled blue cells by location to arrange them in a grid. The following now follows in a straightforward way from Lemma 1 and Lemma 2:

Theorem 1 *The Pfafstetter labels of all cells of a grid-based elevation model can be computed in $\mathcal{O}(\text{sort}(T))$ I/Os, where T is the total length of the computed labels.*

5. EXPERIMENTAL RESULTS

We implemented the algorithms in this paper in C++ using TPIE [3], a library that provides support for implementing I/O-efficient algorithms and data structures. In particular, all sorting steps in our algorithm are done by simply calling a TPIE function. For the priority queue, we used the implementation from TERRAFLOW [4], also based on TPIE.

We ran preliminary tests on grids of varying size. The biggest data set contains 396.5 million grid cells. It is an elevation model of the Neuse basin in North Carolina at a resolution of 20 feet, and is publicly available from ncfloodmaps.com. The other data sets come from the National Elevation Dataset (NED) from the United States Geological Survey and model parts of Tennessee at a resolution of one arc second (approximately 30 m). These data are publicly available at seamless.usgs.gov. The experiments were run on a Dell Precision Server 370 running Linux 2.6.11 with 1 GB of physical memory, a Pentium 4 3.40 GHz processor with hyperthreading enabled, and three 400 GB SATA disk drives. We used a single disk for temporary storage and set the software memory limit to 258 MB. We preprocessed all data sets with TERRAFLOW to obtain grids of flow directions and drainage areas, and then ran the algorithm described in this paper. This resulted in the following running times (excluding the running time of TERRAFLOW).

input size (MB)	17	116	150	713	5,819
size (mln cells)	2.7	21.7	30.8	147.0	396.5
running time	0m30	6m51	10m29	58m10	187m43
spent on:					
importing data	16%	9%	8%	7%	16%
sorting input cells	12%	16%	16%	15%	13%
tracing rivers	43%	30%	31%	34%	30%
sorting river lists	9%	19%	19%	20%	19%
computing labels	5%	8%	7%	6%	6%
sorting labeled cells	8%	13%	14%	13%	12%
exporting data	6%	4%	5%	4%	5%

6. CONCLUDING REMARKS

In this paper, we presented an I/O-efficient algorithm that computes the Pfafstetter labeling of a river basin on a grid-based terrain model in $\mathcal{O}(\text{sort}(T))$ I/Os, where T is the total length of the computed labels.

Once the Pfafstetter labeling is computed, the watershed boundaries yield a hierarchical decomposition of ridge lines of the terrain. When overlaid with the stream lines generated by TERRAFLOW, we could get a decomposition of the terrain into hill slopes at multiple levels of detail. This could be a starting point for terrain simplification algorithms that preserve hydrological properties of the terrain.

References

[1] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[3] L. Arge, R. Barve, D. Hutchinson, O. Procopiuc, L. Toma, D. E. Vengroff, and R. Wickremesinghe. *TPIE User Manual and Reference (edition 082902)*. Duke University, 2002. The manual and software distribution are available on the web at `http://www.cs.duke.edu/TPIE/`.

[4] L. Arge, J. Chase, P. Halpin, L. Toma, D. Urban, J. S. Vitter, and R. Wickremesinghe. Flow computation on massive grid terrains. *GeoInformatica*, 7(4):283–313, 2003.

[5] G. S. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. In *Proc. Scandinavian Workshop on Algorithms Theory, LNCS 1432*, pages 107–118, 1998.

[6] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. of 16th ACM Symposium on Theory of Computing*, pages 135–143, 1984.

[7] S. Jenson and J. Domingue. Extracting topographic structure from digital elevation data for geographic information system analysis. *Photogrammetric Engineering and Remote Sensing*, 54(11):1593–1600, 1988.

[8] J. F. O'Callaghan and D. M. Mark. The extraction of drainage networks from digital elevation data. *Computer Vision, Graphics and Image Processing*, 28, 1984.

[9] T. K. Peucker. Detection of surface specific points by local parallel processing of discrete terrain elevation data. *Computer Graphics and Image Processing*, 4:375–387, 1975.

[10] K. L. Verdin and J. P. Verdin. A topological system for delineation and codification of the Earth's river basins. *Journal of Hydrology*, 218:1–12, 1999.

Out-of-Core Multi-Tessellation

Emanuele Danovaro, Leila De Floriani, Enrico Puppo
Department of Computer and Information Sciences (DISI) – University of Genova
Hanan Samet
Department of Computer Science – University of Maryland at College Park

1. Introduction

Thanks to improvement in simulation, high resolution scanning facilities and multidimensional medical imaging, the size of geometrical dataset is rapidly increasing, and huge datasets are commonly available. Such datasets may be hard to manipulate at their full resolution. Level-Of-Detail (LOD) techniques have been developed in the literature to help concentrating the resolution only where and when it is necessary.

Since the size of many datasets easily exceeds primary memory, LOD techniques, in order to be effective, must be implemented out-of-core and data exchange between primary and secondary memory must be optimized.

In [16, 6] we have proposed a LOD model, called Multi-Tessellation, which provides a general framework for many of the models developed in the literature. In this paper, we analyze different techniques for developing an out-of-core implementation of such a model, aimed at handling arbitrary large datasets either with explicit or implicit encoding of modifications.

2. The Multi-Tessellation

The Multi-Tessellation (MT) is a variable-resolution continuous LOD model which can represent multiresolution simplicial meshes in arbitrary dimensions [16, 6].

Let Σ denote a simplicial complex, a *modification* is an operation that replaces a sub-complex u^- of Σ with another sub-complex u^+. The interesting case is when u^- and u^+ are approximations at two different resolutions of the same (portion of a) shape. We represent a modification as a pair $u = (u^-, u^+)$. Given a set of modifications acting on a base complex Σ, we say that a modification u_i *directly depends* on another modification u_j (and, thus, u_j *directly blocks* u_i) if and only if u_i removes some of the simplices introduced by u_j. Figure 1 shows a sequence of modifications and the corresponding dependency relation.

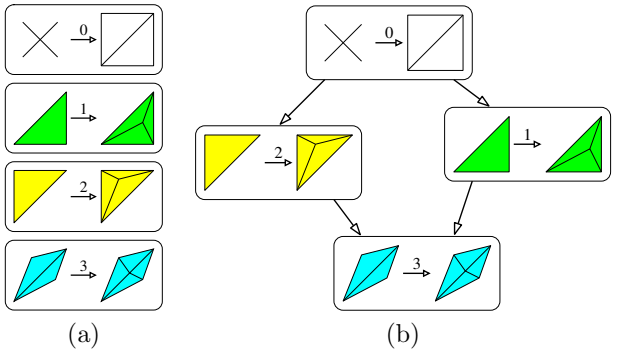


Figure 1. A sequence of modifications (a), and the corresponding MT with direct dependencies represented with arrows (b).

The major issue here is that the relation which defines the dependencies among the modifications in a multi-resolution model is a partial order, which cannot be described by a tree, but rather by a Directed Acyclic Graph (DAG). Some data structures have been proposed in the literature, which can encode dependencies via binary trees plus suitable numbering mechanisms, for specific structures that are special cases of the MT [10]. However the general model needs to be managed by using a DAG.

The queries performed on the DAG are essentially aimed at extracting meshes with a given LOD, which can be variable over the domain. A mesh is associated to a cut in the DAG, and it is obtained by applying all modifications above the cut, in a consistent order, to the base mesh. A mesh is extracted by adjusting the position of the cut in the DAG according to LOD parameters: the cut is moved upwards to coarsen the mesh and downwards to refine it. Corresponding algorithms are not based on classical graph traversal techniques, but they are specific to multi-resolution modeling.

3. Related work

Several techniques have been proposed in the literature for multi-resolution representation of triangle meshes.

Lindstrom and Pascucci [12, 13] present a multi-resolution technique for representing large terrain datasets described as a triangle bintree in external memory. DeCoro and Pajarola in [7] propose a technique to store and query multi-resolution models built through half-edge collapse. El-Sana and Chiang [9] develop a data structure based on the clustering of the view-dependent tree presented in [10], as well as a simplification and a selective refinement algorithms operating in external memory. El-Sana and Bachmat [8] propose an alternative out-of-core multiresolution data structure based on a spatial subdivision imposed over the view-dependent tree. Lindstrom [11] describes an out-of-core multiresolution model based on an octree built through the out-of-core simplification technique developed in [14].

Cignoni et al. [5] describe an out-of-core technique for large multi-resolution triangle meshes based on a domain decomposition as a hierarchy of tetrahedra. Leaves of such hierarchy are associated with portions of the mesh at full resolution, while internal nodes are associated with simplified patches made of triangles. Each patch is subdivided with strips and stored in secondary memory with a compact encoding.

Very recently, an out-of-core technique has been developed based on the Multi-Triangulation [4]. In this approach, an MT with updates of large size is considered. Under this assumption, the DAG describing the MT fits in main memory, while the meshes forming the updates are organized into triangle strips and stored in external memory independently.

4. Out-of-core MT

Our purpose is to maintain and query a two- or three-dimensional MT having a size that exceeds main memory. Here we investigate a general out-of-core approach, independent of the simplification strategy used to generate the LOD model.

Our approach is based on clustering nodes of the DAG and filling disk pages with clusters. We devoted a considerable effort in defining, implementing and experimenting a large number of clustering techniques based on DAG traversal and on spatial partitioning and grouping. Out-of-core representation of the dependency relation can be coupled with different representations for the modifications in the model.

We have first analyzed the I/O operations performed by the basic selective refinement algorithms and designed and implemented a simulation environment which allows us to evaluate a large number of data structures for out-of-core encoding. We have designed and developed more than sixty clustering techniques for the modifications in a multi-resolution model, which take into account their mutual dependency relations and their spatial arrangement. We are currently implementing an out-of-core prototype system for multi-resolution modeling which is somehow independent of the way the single modifications are encoded, and thus will provide a general-purpose tool for out-of-core multi-resolution modeling.

Developing a good clustering policy requires some knowledge about the shape of the multi-resolution model and its behavior with standard queries. Some queries, like point location or other queries based on region of interest, focus on a subset of the domain, usually at a high resolution. Thus, we are interested in space partitioning techniques. On the contrary, other queries are aimed at extracting e mesh at uniform resolution on the whole domain. Thus, they suggest a subdivision of the MT in layers. Each layer should guarantee the reduction of the approximation error. View-dependent queries are still queries based on a region of interest, but in this case the error smoothly decreases according to the distance from the observer. View-dependent queries combine somehow space partitioning and error-driven queries. Queries based on the field value (of a scalar filed) are strongly dependent on the size and on the shape of the iso-contours. They can behave like point-location queries for really small iso-surfaces; on the contrary, they can be compared to uniform queries for large iso-surfaces that span over the whole domain.

For this purpose, we have defined several rules that can be used to sort set of modifications belonging to an MT:

- *random* (`Rnd`): it is implemented just for comparison purposes and is a completely random sorting of the modifications;
- *sequential* (`Seq`): modifications are kept in the same sequence generated by a top-down simplification procedure, or in reversed order in case of bottom-up simplification strategies. Usually the simplification process is error-driven, but there is no guarantee that the approximation error associated with modifications decreases monotonically;
- *approximation error* (`Err`): it sorts modifications according to their approximation error $e(u)$. It can take into account approximation error (in case of

free form surfaces), field or isosurface error (for scalar fields);

- *depth-first* (`DFS`): it sorts modifications according to a depth-first traversal of the DAG. For modifications with more than one child, children are visited in the same order as they are stored;
- *breadth-first* (`BFS`): it sorts modifications according to a breadth-first traversal of the DAG. As in the `DFS` case, in case of modifications with more than one child, children are visited in the same order they are stored;
- *layer* (`Lyr`): modifications are sorted according to the value of their layer: length of the shortest path from the root;
- *level* (`Lev`): modifications are sorted according to the value of their level: length of the longest path from the root;
- *distance* (`Ly2`): modifications are sorted according to the value of their distance: average length of the paths from the root;
- *graph visit - breadth first* (`GrB`): it is similar to a breadth-first traversal, but before adding a modification u it checks if its ancestors are in the current set of modifications. If they are missing, they will be added. It simulates a breadth-first selective refinement algorithm for a query at uniform resolution;
- *graph visit - depth first* (`GrD`): it is a variation of `GrB`, based on a depth-first visit.

All rules above take into account only the topological structure of the DAG.

However, queries that select a certain region of interest suggest to subdivide the MT according to a space partitioning technique. In this perspective, we consider first a common spatial index, that is the *Point Region quadtree* (PR quadtree) [15, 17]. It is a spatial index for points in the two-dimensional Euclidean space, based on the recursive decomposition of a square domain in $\mathbb{E}^2$, which contains the data points, into four quadrants obtained by splitting the square block through the midpoint of the square. The subdivision is performed recursively until a full block that contains just one point. Thus, the data points are in the leaves of a PR-quadtree and the internal nodes are just discriminators. A PR-quadtree can be extended to d-dimensional space. In this case, each recursive decomposition splits a block into 2^d blocks. But a PR-quadtree suffers the curse of dimensionality, since in high dimensional spaces a high percentage of the leaves is empty. In $\mathbb{E}^2$, at least 25% of the leaves are full, while in $\mathbb{E}^4$ this percentage decreases to 6.25%.

Since the Multi-Tessellation is a data structure that can handle multiresolution model in arbitrary dimensions, we have considered also other space partitioning techniques that does not depend on space dimensionality, namely *Point Region k-d trees* (PR k-d trees) [2] and *R* trees* [1].

A PR k-d tree is a spatial index based on the recursive subdivision of the domain containing the data points. At each step the domain is halved. The halving process cycles through different dimensions in a predefined and constant order. At each step a block is subdivided in its mid-point. We associate each modification to its centroid, and we construct the PR k-d tree according to the coordinates of centroids. We have performed several tests in higher dimensions by adding the approximation error, or the layer, or the distance from the root as an additional dimension.

To store a PR k-d tree on disk, we could have used a technique commonly used for quadtrees and octrees, which consist of computing the location codes of the full leaves in the PR k-d tree and storing the location codes in a B-tree [18]. Such technique has the disadvantage that the boundary of the domain covered by leaves stored in a node of the B-tree can be arbitrary complex. In general, the domain might be not connected and, thus, we could loose spatial coherence.

An interesting alternative consists of using a PK-tree as a grouping mechanism. Originally proposed in [19], a PK-tree is clustered by a parameter k, called the *instantiation value*, which is the minimum number of nodes in the tree grouped in a cluster. A PK-tree is constructed by applying a bottom-up grouping process to the nodes of a tree T. Nodes belonging to T are grouped into clusters until the minimum occupancy k has been reached. During the grouping process empty leaves of the tree T are removed. If the tree T is an n-ary tree, each node, except the root, has a minimum of k children or objects and a maximum of $n \cdot (k-1)$. Since the PR k-d tree is a binary tree, an interesting property of the PR k-d tree grouped according to the PK-tree, that we call a *PK PR k-d tree*, is that each node has a minimum of k children and a maximum of $2 \cdot (k-1)$, regardless of the dimensionality of the space [19]. This guarantees that disk blocks are at least half-full.

R-trees offer an object-based technique that performs object aggregation [1]. In an R-tree, each object is defined by its bounding box. The first step aggregates up to k bounding boxes into a box of minimum size that contains them. This process is repeated recursively until there is just one block left. Each box is mapped to a disk block. Several extent-based aggregation techniques have been developed. The main differ-

ences among such techniques are related to coverage and to minimization of overlapping areas. R*-trees [1] have been shown to be the best extent-based aggregation technique. They can handle d-dimensional objects, and minimize domain coverage and rectangle overlap.

We have combined a sorting criteria with a space partitioning techniques. The idea is to combine a technique that subdivides the DAG in subsets of modifications according to their depth, with a technique that subdivides the space. At a lower resolution, we are interested to have clusters that span the whole domain, while, as the resolution increases, we are looking for clusters associated with finer space subdivisions. In order to achieve this goal, we have adopted a technique that interleaves the effect of a sorting rule and the effect of a space partitioning rule similar to a PR k-d tree.

Note that a clustering technique which simply sorts modifications and fills each disk block (or stripe) with a contiguous set of sorted modifications does not introduce any overhead in storage space. A disk block is usually at least 4 KBytes. There is no upper bound in block size, but a block that spans on a whole disk track bests amortize seek times and latency, and, thus, there is no need to create larger blocks. This results in an upper bound equal to 50 to 200 KBytes, according to disk radius and density. In a RAID subsystem with up to 5 disks configured for disk striping, it can result in up to 200 to 1000 KBytes.

If we consider a 3D MT encoded explicitly (i.e., by listing all tetrahedra inside each node of the DAG), each modification requires on average 325 Bytes. This results in a block transfer size B in a range between 12 and 150-600 modification, if we consider a single disk architecture and between 60 to 600-3000 with a large RAID subsystem. The number of modifications per block can dramatically increase by a compact (implicit) encoding for the modifications (e.g., in case of MTs built through vertex removal or edge collapse operations).

5. Experimental results

Our tests focused on MT for volume data sets. The out-of-core MT can be applied to dataset of arbitrary dimension. Unfortunately, up to now, no out-of-core simplification tools is available for tetrahedral datasets. For this reason, we have used the largest model that can be handled by an in-core, high-quality, simplification tool [3]. In order to get reasonable and meaningful results, we have artificially reduced the amount of available core memory, thus forcing the out-of-core prototype to load only a subset of disk blocks.

We have scored the ratio between the number of disk accesses required to perform a set of selective refinement queries and the number of disk blocks required to store the model. This allow us to compare results, independently on model size. Due to the lack of space, we show only one graph, related to results obtained with queries at uniform resolution on the San Fernando dataset (courtesy of Quake Project D.R. OHallaron and J.R. Shewchuk), a 3D scalar field composed of about two million tetrahedra showing a simulation of an earthquake in the region of San Fernando (CA). However, we made similar experiment also on selective refinement queries, both on this dataset and on other 2D and 3D datasets, obtaining similar results.

Figure 2 compares performances obtained with different clustering techniques. Besides some of the techniques defined in the previous section (`Err`, `DFS`, `GrB`, `GrD`) we show results on the following other techniques: space partitioning on coordinates, and `GrB` sorting (`SpaGrB`); space partitioning on approximation error and coordinates, and `GrB` sorting (`SpaErrGrB`); space partitioning on layer and coordinates, and `Err` sorting (`SpaLyrErr`); space partitioning on distance and coordinates, and `Ly2` sorting (`SpaLy2Ly2`); space partitioning on level and coordinates, and `GrB` sorting (`SpaLevGrB`); space partitioning on approximation error and coordinates, and grouping with PK tree (`PK PR kd tree + Error`); space partitioning and grouping with an R*-tree built according with approximation error and coordinates (`R* tree + Error`).

It is easy to notice that `GrB` outperforms other clustering techniques. It is also interesting to note that, even with a really small cache, that is about 1% of the size of the whole model, a clustering technique based on `GrB` exhibits really small overhead, compared to loading the whole model.

References

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD Conference*, pages 322–331, June 1990.

[2] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[3] P. Cignoni, D. Costanza, C. Montani, C. Rocchini, and R. Scopigno. Simplification of tetrahedral volume data with accurate error evaluation. In *Proceedings IEEE Visualization 2000*, pages 85–92. IEEE Computer Society, 2000.

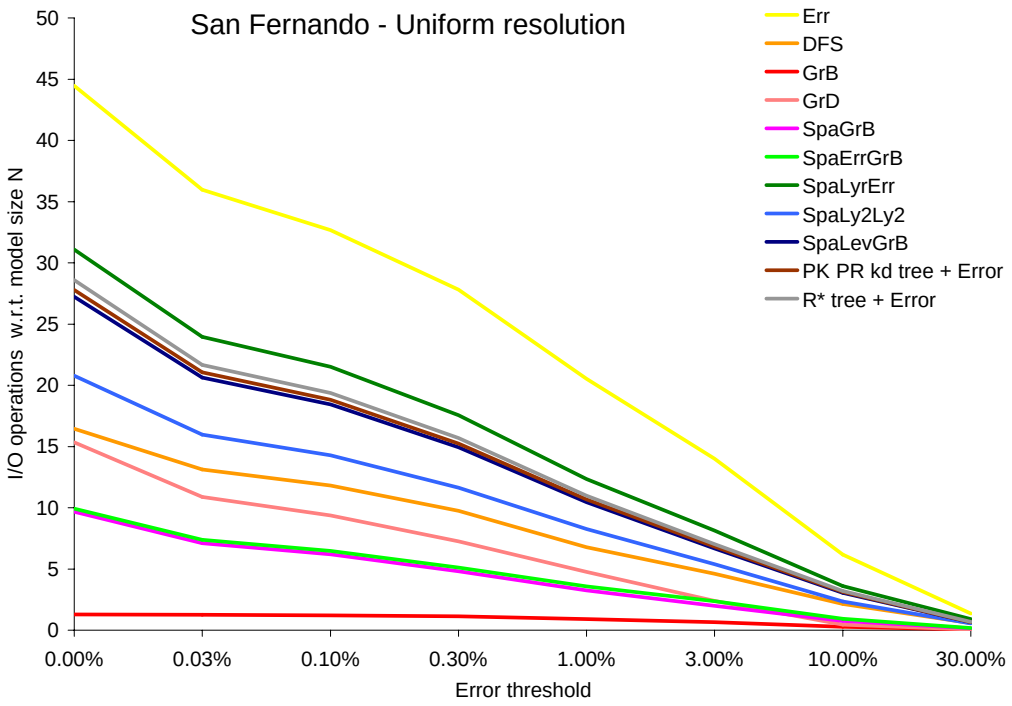


Figure 2. Number of I/O operations performing queries at uniform resolution on San Fernando dataset.

[4] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchi, and R. Scopigno. Gpu-friendly multi-triangulation. Technical report, 2005.

[5] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Adaptive TetraPuzzles – efficient out-of-core construction and visualization of gigantic polygonal models. *ACM Transactions on Graphics*, 23(3), August 2004. Proceedings SIGGRAPH 2004.

[6] L. De Floriani, E. Puppo, and P. Magillo. A formal approach to multi-resolution modeling. In W. Strasser, R. Klein, and R. Rau, editors, *Geometric Modeling: Theory and Practice*, pages 302–323. Springer-Verlag, 1997.

[7] C. DeCoro and R. Pajarola. XFASTMESH: Fast view-dependent meshing from external memory. In *Proceedings IEEE Visualization 2002*, pages 263–270. IEEE Computer Society, October 2002.

[8] J. El-Sana and E. Bachmat. Optimized view-dependent rendering for large polygonal datasets. In *Proceedings IEEE Visualization 2002*, pages 77–84. IEEE Computer Society, October 2002.

[9] J. El-Sana and Y. Chiang. External memory view-dependent simplification. *Computer Graphics Forum*, 19(3):C139–C150, 2000.

[10] J. El-Sana and A. Varshney. Generalized view-dependent simplification. *Computer Graphics Forum*, 18(3):C83–C94, 1999.

[11] P. Lindstrom. Out-of-core construction and visualization of multi-resolution surfaces. In *ACM SIGGRAPH 2003 Symposium on Interactive 3D Graphics*, pages 93–102. ACM Press, April 2003.

[12] P. Lindstrom and V. Pascucci. Visualization of large terrains made easy. In *Proceedings IEEE Visualization 2001*, pages 363–370, October 2001.

[13] P. Lindstrom and V. Pascucci. Terrain simplification simplified: A general framework for view- dependent out-of-core visualization. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):239–254, 2002.

[14] P. Lindstrom and C. T. Silva. A memory insensitive technique for large model simplification. In *IEEE Visualization 2001*, pages 121–126, 550, October 2001.

[15] J. A. Orenstein. Multidimensional tries used for associative searching. *Information Processing Letters*, 14(4):150–157, 1982.

[16] E. Puppo. Variable resolution terrain surfaces. In *Proceedings Eight Canadian Conference on Computational Geometry, Ottawa, Canada*, pages 202–210, August 12-15 1996.

[17] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[18] H. Samet. *Foundations of Multi-Dimensional Data Structures*. Morgan-Kaufmann, to appear, 2005.

[19] W. Wang, J. Yang, and R. Muntz. PK-tree: a spatial index structure for high dimensional point data. In K. Tanaka and S. Ghandeharizadeh, editors, *Proceedings of the 5th International Conference on Foundations of Data Organization and Algorithms (FODO)*, pages 27–36, November 1998.

Preprints
"Angewandte Mathematik und Informatik"

10/03 - S G. Alsmeyer, U. Rösler: A stochastic fixed point equation related to weighted branching with deterministic weights

11/03 - S N. Gantert, M. Löwe, J. Steif: The voter model with antivoter bonds

12/03 - S M. Löwe, F. Vermet: The storage capacity of the Blume-Emery-Griffiths neural network

13/03 - S P. Eichelsbacher, M. Löwe: Moderate Deviations for the overlap parameter in the Hopfield model

14/03 - S H. Knöpfel, M. Löwe: Fluctations in a p-spin interaction model

15/03 - S H. Kösters: Prophetentheorie bei Mehrfachauswahlen: Der allgemeine Fall

01/04 - S M. Wrede: Bewertung von Derivaten in zeitdiskreten Finanzmarktmodellen

02/04 - I H. Blunck, L. Becker, K. Hinrichs, J. Vahrenhold: A Framework for Representing Moving Objects

03/04 - I T. Ropinski, K. Hinrichs: An image-based algorithm for interactive rendering of 3D Magic Lenses

04/04 - I J. Lechtenbörger: Computing Unique Canonical Covers for Simple FDs via Transitive Reduction

05/04 - S A. Janssen, D. Völker: Most powerful conditional tests

06/04 - S D. Völker: Finit optimale nichtparametrische Tests für Lebensdauerdaten

07/04 - I F. Steinicke, T. Ropinski, K. Hinrichs: Improved Virtual Pointer Metaphors for Interactive Object Selection in Virtual Environments

08/04 - S S. Alink, M. Löwe, M. Wüthrich: Analysis of the Expected Shortfall of Aggregate Dependent Risks

09/04 - S M. Löwe, F. Vermet: On the Bit-Error Probability of CDMA Systems with Optimal Hard Decision Interference Cancellation

10/04 - N A. Arnold, E. Dhamo, C. Manzini: The Wigner-Poisson-Fokker-Planck system: Global-in-time solutions and dispersive effects

11/04 - N M. Cheney, F. Natterer: Tomographic Reconstruction from Circles

12/04 - S D. Kuhlbusch: Moment Conditions for Weighted Branching Processes

13/04 - N F. Natterer: Finding the Support of a Scatterer from a Single Incoming Wave

14/04 - N F. Natterer, F. Wübbeling: Scatter Correction in PET Based on Transport Models

01/05 - S G. Alsmeyer, U. Rösler:

02/05 - I L. Arge, M. de Berg, J. Vahrenhold (Eds.): Workshop on Massive Geometric Data Sets (Massive2005)