To appear in *Proc. 6th ESA*, Venice, Italy, 1998.

A Functional Approach to External Graph Algorithms

James Abello, Adam L. Buchsbaum, and Jeffery R. Westbrook

AT&T Labs, 180 Park Ave., Florham Park, NJ 07932, USA,
{abello,alb,jeffw}@research.att.com,
http://www.research.att.com/info/{abello,alb,jeffw}.

Abstract. We present a new approach for designing external graph algorithms and use it to design simple external algorithms for computing connected components, minimum spanning trees, bottleneck minimum spanning trees, and maximal matchings in undirected graphs and multi-graphs. Our I/O bounds compete with those of previous approaches. Unlike previous approaches, ours is purely functional—without side effects—and is thus amenable to standard checkpointing and programming language optimization techniques. This is an important practical consideration for applications that may take hours to run.

1 Introduction

We present a divide-and-conquer approach for designing external graph algorithms, i.e., algorithms on graphs that are too large to fit in main memory. Our approach is simple to describe and implement: it builds a succession of graph transformations that reduce to sorting, selection, and a recursive bucketing technique. No sophisticated data structures are needed. We apply our techniques to devise external algorithms for computing connected components, minimum spanning trees (MSTs), bottleneck minimum spanning trees (BMSTs), and maximal matchings in undirected graphs and multi-graphs.

We focus on producing algorithms that are purely functional. That is, each algorithm is specified as a sequence of functions applied to input data and producing output data, with the property that information, once written, remains unchanged. The function is then said to have no "side effects." A functional approach has several benefits. External memory algorithms may run for hours or days in practice. The lack of side effects on the external data allows standard checkpointing techniques to be applied [16,19], increasing the reliability of any real application. A functional approach is also amenable to general purpose programming language transformations that can reduce running time. (See, e.g., Wadler [22].) We formally define the *functional I/O model* in Section 1.1.

The key measure of external memory graph algorithms is the disk I/O complexity. For the problems mentioned above, our algorithms perform $O(\frac{E}{B}\log_{M/B}\frac{E}{B}\log_2\frac{V}{M})$ I/Os, where E is the number of edges, V the number of vertices, M the size of main memory, and B the disk block size. The BMST and maximal matching results are new. The asymptotic I/O complexities of our connected components and MST algorithms match those of Chiang et al. [8]. Kumar and Schwabe [15] give algorithms for breadth-first search (which can compute connected components) and MSTs that perform $O(V+\frac{E}{B}\log_2\frac{E}{B})$ and $O(\frac{E}{B}\log_{M/B}\frac{E}{B}\log_2 B+\frac{E}{B}\log_2 V)$ I/Os, respectively. Our connected components algorithm is asymptotically better when $V < M^2/B$, and our

MST algorithm is asymptotically better when $V < MB$. While the above algorithms of Chiang et al. [8] are functional, those of Kumar and Schwabe [15] are not. Compared to either previous approach, our algorithms are simpler to describe and implement.

We also consider a *semi-external* model for graph problems, in which the vertices but not the edges fit in memory. This is not uncommon in practice, and when vertices can be kept in memory, significantly more efficient algorithms are possible. We design new algorithms for external grouping and sorting with duplicates and apply them to produce better I/O bounds for the semi-external case of connected components.

We begin below by describing the I/O model. In Section 2, we sketch two previous approaches for designing external graph algorithms. In Section 3, we describe our functional approach and detail a suite of simple graph transformations. In Section 4, we apply our approach to design new, simple algorithms for computing connected components, MSTs, BMSTs, and maximal matchings. In Section 5, we consider semi-external graph problems and give improved I/O bounds for the semi-external case of connected components. We conclude in Section 6.

1.1 The Functional I/O Model

We adapt the I/O model of complexity as defined by Aggarwal and Vitter [1]. For some problem instance, we define N to be the number of items in the instance, M to be the number of items that can fit in main memory, B to be the number of items per disk block, and $b = \lfloor M/B \rfloor$. A typical compute server might have $M \approx 10^9$ and $B \approx 10^3$.

We assume that the input graph is presented as an unordered list of edges, each edge a pair of endpoints plus possibly a weight. We define V to be the number of vertices, E to be the number of edges, and $N = V + E$. (We abuse notation and also use V and E to be the actual vertex and edge sets; the context will clarify any ambiguity.) In general, $1 < B \ll M < N$. Our algorithms work in the single-disk model; we discuss parallel disks in Section 6. For the connected components problem, the output is a *delineated list* of component edges, $\{C_1, C_2, \ldots, C_k\}$, where k is the number of components: each C_i is the list of edges in component i, and the output is the file of C_is catenated together, with a separator record between adjacent components. For the MST and BMST problems the output is a delineated list of edges in each tree in the spanning forest. For matching, the output is the list of edges in the matching.

Following Chiang et al. [1] we define $\mathit{scan}(N) = \lceil N/B \rceil$ to be the number of disk I/Os required to transfer N contiguous items between disk and memory, and we define $\mathit{sort}(N) = \Theta(\mathit{scan}(N) \log_b \frac{N}{B})$ to be the number of I/Os required to sort N items. The I/O model stresses the importance of disk accesses over computation for large problem instances. In particular, time spent computing in main memory is not counted.

The *functional I/O (FIO) model* is as above, but operations can make only functional transformations to data, which do not change the input. Once a disk cell, representing some piece of state, is allocated and written, its contents cannot be changed. This imposes a sequential, write-once discipline on disk writes, which allows the use of standard checkpointing techniques [16, 19], increasing the reliability of our algorithms. When results of intermediate computations are no longer needed, space is reclaimed, e.g., through garbage collection. The maximum disk space active at any one time is used to measure the space complexity. All of our algorithms use only linear space.

2 Previous Approaches

2.1 PRAM Simulation

Chiang et al. [8] show how to simulate a CRCW PRAM algorithm using one processor and an external disk, thus giving a general method for constructing external graph algorithms from PRAM graph algorithms. Given a PRAM algorithm, the simulation maintains on disk arrays A, which contains the contents of main memory, and T, which contains the current state for each processor. Each step of the algorithm is simulated by constructing an array, D, which contains for each processor the memory address to be read. Sorting A and D by memory address and scanning them in tandem suffices to update T. A similar procedure is used to write updated values back to memory.

Each step thus requires a constant number of scans and sorts of arrays of size $|T|$ and a few scans of A. Typically, therefore, a PRAM algorithm using N processors and N space to solve a problem of size N in time t can be simulated in external memory by one processor using $O(t \cdot \mathit{sort}(N))$ I/Os. Better bounds are possible if the number of active processors and memory cells decreases linearly over the course of the PRAM algorithm. These techniques can, for example, simulate the PRAM maximal matching algorithm of Kelsen [14] in $O(\mathit{sort}(E) \log_2^4 V)$ I/Os and the PRAM connected components and MST algorithms of Chin, Lam, and Chen [9] in $O(\mathit{sort}(E) \log_2 \frac{V}{M})$ I/Os.

The PRAM simulation works in the FIO model, if each step writes new copies of T and A. To the best of our knowledge, however, no algorithm based on the simulation has been implemented. Such an implementation would require not only a practical PRAM algorithm but also either a meticulous direct implementation of the corresponding external memory simulation or a suitable low-level machine description of the PRAM algorithm together with a general simulation tool. In contrast, a major goal of our work is to provide simple, direct, and implementable algorithms.

2.2 Buffering Data Structures

Another recent approach is based on external variants of classical internal data structures. Arge [3] introduces *buffer trees*, which support sequences of insert, delete, and deletemin operations on N elements in $O(\frac{1}{B} \log_b \frac{N}{B})$ amortized I/Os each. Kumar and Schwabe [15] introduce a variant of the buffer tree, achieving the same heap bounds as Arge. These bounds are optimal, since the heaps can be used to sort externally. Kumar and Schwabe [15] also introduce external *tournament trees*. The tournament tree maintains the elements 1 to N, each with a key, subject to the operations delete, deletemin, and update: deletemin returns the element of minimum key, and update reduces the key of a given element. Each tournament tree operation takes $O(\frac{1}{B} \log_2 \frac{N}{B})$ amortized I/Os.

These data structures work by buffering operations in nodes and performing updates in batches. The maintenance procedures on the data structures are intuitively simple but involve many implementation details. The data structures also are not functional. The node-copying techniques of Driscoll et al. [10] could be used to make them functional, but at the cost of significant extra I/O overhead.

Finally, while such data structures excel in computational geometry applications, they are hard to apply to external graph algorithms. Consider computing an MST. The

classical greedy algorithm repeatedly performs deletemins on a heap to find the next vertex v to attach to the tree, T, and then performs updates for each neighbor w of v that becomes closer to T by way of v. In the external version of the algorithm, however, finding the neighbors of v is non-trivial, requiring a separate adjacency list. Furthermore, determining the current key of a given neighbor w (distance of w to T) is problematic, yet the key is required to decide whether to perform the update operation.

Intuitively, while the update operations on the external data structures can be buffered, yielding efficient amortized I/O complexity, standard applications of these data structures in graph algorithms require certain queries (key finding, e.g.) to be performed on-line. Current applications of these data structures thus require ancillary data structures to obviate this problem, increasing the I/O and implementation complexity.

3 Functional Graph Transformations

In this paper, we utilize a divide-and-conquer paradigm based on a few graph transformations that, for many problems, preserve certain critical properties. We implement each stage in our approach using simple and efficient techniques: sorting, selection, and bucketing. We illustrate our approach with connected components. Let $G = (V, E)$ be a graph. Let $f(G) \subseteq V \times V$ be a forest of rooted stars (trees of height one) representing the connected components of G. That is, if $r_G(v)$ is the root of the star containing v in $f(G)$, then $r_G(v) = r_G(u)$ if and only if v and u are in the same connected component in G; $f(G)$ is presented as a delineated edge list.

Consider $E' \subseteq V \times V$, and let $G' = G/E'$ denote the result of contracting all vertex pairs in E'. (This generalizes the usual notion of contraction, by allowing the contraction of vertices that are not adjacent in G.) For any $x \in V$, let $s(x)$ be the supervertex in G' into which x is contracted. It is easy to prove that if each pair in E' contains two vertices in the same connected component, then $r_{G'}(s(v)) = r_{G'}(s(u))$ if and only if $r_G(v) = r_G(u)$; i.e., contraction preserves connected components. Thus, given procedures to contract a list of components of a graph and to re-expand the result, we derive the following simple algorithm to compute $f(G)$.

1. Let E_1 be any half of the edges of G; let $G_1 = (V, E_1)$.
2. Compute $f(G_1)$ recursively.
3. Let $G' = G/f(G_1)$.
4. Compute $f(G')$ recursively.
5. $f(G) = f(G') \cup R(f(G'), f(G_1))$, where $R(X, Y)$ relabels edge list Y by forest X: each vertex u occurring in edges in Y is replaced by its parent in X if it exists.

Our approach is functional if selection, relabeling, and contraction can be implemented without side effects on their arguments. We show how to do this below. In the following section, we use these tools to design functional external algorithms for computing connected components, MSTs, BMSTs, and maximal matchings.

3.1 Transformations

Selection. Let I be a list of items with totally ordered keys. $\texttt{Select}(I, k)$ returns the kth biggest element from I; i.e., $|\{x \in I \mid x < \texttt{Select}(I, k)\}| = k - 1$. We adapt the classical algorithm for $\texttt{Select}(I, k)$ [2].

1. Partition I into j-element subsets, for some $j \approx M$.
2. Sort each subset in main memory. Let S be the set of medians of the subsets.
3. $m \leftarrow \texttt{Select}(S, \lceil S/2 \rceil)$.
4. Let I_1, I_2, I_3 be those elements less than, equal to, and greater than m, respectively.
5. If $|I_1| \geq k$, then return $\texttt{Select}(I_1, k)$.
6. Else if $|I_1| + |I_2| \geq k$, then return m.
7. Else return $\texttt{Select}(I_3, k - |I_1| - |I_2|)$.

Lemma 1. $\texttt{Select}(I, k)$ *can be performed in* $O(\mathit{scan}(|I|))$ *I/Os in the FIO model.*

Relabeling. Given a forest F as an unordered sequence of tree edges $\{(p(v), v), \ldots\}$, and an edge set I, *relabeling* produces a new edge set $I' = \{\{r(u), r(v)\} \mid \{u, v\} \in I\}$, where $r(x) = p(x)$ if $(p(x), x) \in F$, and $r(x) = x$ otherwise. That is, for each edge $\{u, v\} \in I$, each of u and v is replaced by its respective parent, if it exists, in F. We implement relabeling as follows.

1. Sort F by source vertex, v.
2. Sort I by second component.
3. Process F and I in tandem.
   (a) Let $\{s, h\} \in I$ be the current edge to be relabeled.
   (b) Scan F starting from the current edge until finding $(p(v), v)$ such that $v \geq h$.
   (c) If $v = h$, then add $\{s, p(v)\}$ to I''; otherwise, add $\{s, h\}$ to I''.
4. Repeat Steps 2 and 3, relabeling first components of edges in I'' to construct I'.

Relabeling is related to pointer jumping, a technique widely applied in parallel graph algorithms [11]. Given a forest $F = \{(p(v), v), \ldots\}$, *pointer jumping* produces a new forest $F' = \{(p(p(v)), v) \mid (p(v), v) \in F\}$; i.e., each v of depth two or greater in F points in F' to its grandparent in F. (Define $p(v) = v$ if v is a root in F.) Our implementation of relabeling is similar to Chiang's [7] implementation of pointer jumping.

Lemma 2. *Relabeling an edge list* I *by a forest* F *can be performed in* $O(\mathit{sort}(|I|) + \mathit{sort}(|F|))$ *I/Os in the FIO model.*

Contraction. Given an edge list I and a list $C = \{C_1, C_2, \ldots\}$ of delineated components, we can *contract* each component C_i in I into a supervertex by constructing and applying an appropriate relabeling to I.

1. For each $C_i = \{\{u_1, v_1\}, \ldots\}$:
   (a) $R_i \leftarrow \emptyset$.
   (b) Pick u_1 to be the canonical vertex.
   (c) For each $\{x, y\} \in C_i$, add (u_1, x) and (u_1, y) to relabeling R_i.
2. Apply relabeling $\bigcup_i R_i$ to I, yielding the contracted edge list I'.

Lemma 3. *Contracting an edge list* I *by a list of delineated components* $C = \{C_1, C_2, \ldots\}$ *can be performed in* $O(\mathit{sort}(|I|) + \mathit{sort}(\sum_i |C_i|))$ *I/Os in the FIO model.*

Deletion. Given edge lists I and D, it is straightforward to construct $I' = I \setminus D$: simply sort I and D lexicographically, and process them in tandem to construct I' from the edges in I but not D. If D is a vertex list, we can similarly construct $I'' = \{\{u, v\} \in I \mid u, v \notin D\}$, which we also denote by $I \setminus D$.

Lemma 4. *Deleting a vertex or edge set D from an edge set I can be performed in $O(\mathit{sort}(|I|) + \mathit{sort}(|D|))$ I/Os in the FIO model.*

4 Applying the Techniques

In this section, we devise efficient, functional external algorithms for computing connected components, MSTs, BMSTs, and maximal matchings of undirected graphs. Each of our algorithms needs $O(\mathit{sort}(E) \log_2 \frac{V}{M})$ I/Os. Our algorithms extend to multigraphs, by sorting the edges and removing duplicates in a preprocessing pass.

4.1 Deterministic Algorithms

Connected Components.

Theorem 1. *The delineated edge list of components of a graph can be computed in the FIO model in $O(\mathit{sort}(E) \log_2 \frac{V}{M})$ I/Os.*

Proof (Sketch). Let $T(E)$ be the number of I/Os required to compute $f(G)$, the forest of rooted stars corresponding to the connected components of G, presented as a delineated edge list. Recall from Section 3 the algorithm for computing $f(G)$. We can easily select half the edges in E in $\mathit{scan}(E)$ I/Os. Contraction takes $O(\mathit{sort}(E))$ I/Os, by Lemma 3. We use the forest $f(G')$ to relabel the forest $f(G_1)$. Combining the two forests and sorting the result by target vertex then yields the desired result. Thus, $T(E) \leq O(\mathit{sort}(E)) + 2T(E/2)$. We stop the recursion when a subproblem fits in internal memory, so $T(E) = O(\mathit{sort}(E) \log_2 \frac{V}{M})$.

Given $f(G)$, we can label each edge in E with its component in $O(\mathit{sort}(E))$ I/Os, by sorting E (by, say, first vertex) and $f(G)$ (by source vertex) and processing them in tandem to assign component labels to the edges. This creates a new, labeled edge list, E''. We then sort E'' by label, creating the desired output E'.

Minimum Spanning Trees. We use our approach to design a top-down variant of Borůvka's MST algorithm [4]. Let $G = (V, E)$ be a weighted, undirected graph, and let $f(G)$ be the delineated list of edges in a minimum spanning forest (MSF) of G.

1. Let $m = \texttt{Select}(E, |E|/2)$ be the edge of median weight.
2. Let $S(G) \subset E$ be the edges of weight less than m and half the edges of weight m.
3. Let G_2 be the contraction $G/f(S(G))$.
4. Then $f(G) = f(S(G)) \cup R_2(f(G_2))$, presented as a delineated edge list, where $R_2(\cdot)$ re-expands edges in G_2 that are incident on supervertices created by the contraction $G/f(S(G))$ to be incident on their original endpoints in $S(G)$.

Theorem 2. *A minimum spanning forest of a graph can be computed in the FIO model in $O(\mathit{sort}(E)\log_2 \frac{V}{M})$ I/Os.*

Proof (Sketch). Correctness follows from the analysis of the standard greedy approach. The I/O analysis uses the same recursion as in the proof of Theorem 1. Selection incurs $O(\mathit{scan}(E))$ I/Os, by Lemma 1. The input to $f(\cdot)$ is an annotated edge list: each edge has a weight and a label. Contraction incurs $O(\mathit{sort}(E))$ I/Os, by Lemma 3, and installs the corresponding original edge as the label of the contracted edge. The labels allow us to "re-expand" contracted edges by a reverse relabeling. To produce the final edge list, we apply a procedure similar to that used to delineate connected components.

Because our contraction procedure requires a list of delineated components, we cannot implement the classical, bottom-up Borůvka MST algorithm [4], in which each vertex selects the incident edge of minimum weight, the selected edges are contracted, and the process repeats until one supervertex remains. An efficient procedure to contract an arbitrary edge list could thus be used to construct a faster external MST algorithm.

Bottleneck Minimum Spanning Trees. Given a graph $G = (V, E)$, a *bottleneck minimum spanning tree* (*BMST*, or *forest*, *BMSF*) is a spanning tree (or forest) of G that minimizes the maximum weight of an edge. Camerini [5] shows how to compute a BMST of an undirected graph in $O(E)$ time, using a recursive procedure similar to that for MSTs.

1. Let $S(G)$ be the lower-weighted half of the edges of E.
2. If $S(G)$ spans G, then compute a BMST of $S(G)$.
3. Otherwise, contract $S(G)$, and compute a BMST of the remaining graph.

We design a functional external variant of Camerini's algorithm analogously to the MST algorithm above. In the BMST algorithm, $f(G)$ returns a BMSF of G and a bit indicating whether or not it is connected (a BMST). If $f(S(G))$ is a tree, then $f(G) = f(S(G))$; otherwise, we contract $f(S(G))$ and recurse on the upper-weighted half.

Theorem 3. *A bottleneck minimum spanning tree can be computed in the FIO model in $O(\mathit{sort}(E)\log_2 \frac{V}{M})$ I/Os.*

Whether BMSTs can be computed externally more efficiently than MSTs is an open problem. If we could determine whether or not a subset $E' \subseteq E$ spans a graph in $g(E')$ I/Os, then we can use that procedure to limit the recursion to one half of the edges of E, as in the classical BMST algorithm. This would reduce the I/O complexity of finding a BMST to $O(g(E) + \mathit{sort}(E))$ ($\mathit{sort}(E)$ to perform the contraction).

Maximal Matching. Let $f(G)$ be a maximal matching of a graph $G = (V, E)$, and let $V(f(G))$ be the vertices matched by $f(G)$. We find $f(G)$ functionally as follows.

1. Let $S(G)$ be any half of the edges of E.
2. Let $G_2 = E \setminus V(f(S(G)))$.
3. Then $f(G) = f(S(G)) \cup f(G_2)$.

Theorem 4. *A maximal matching of a graph can be computed in the FIO model in* $O(\mathit{sort}(E) \log_2 \frac{V}{M})$ *I/Os.*

Proof (Sketch). Selecting half the edges takes $\mathit{scan}(E)$ I/Os. Deletion takes $O(\mathit{sort}(E))$ I/Os, by Lemma 4. Deleting all edges incident on matched vertices must remove all edges in E_1 from G by the assumption of maximality. Hence $|E(G_2)| \leq |E|/2$.

Finding a maximum matching externally (efficiently) remains an open problem.

4.2 Randomized Algorithms

Using the random vertex selection technique of Karger, Klein, and Tarjan [12], we can reduce by a constant fraction the number of vertices as well as edges at each step. This leads to randomized functional algorithms for connected components, MSTs, and BMSTs that incur $O(\mathit{sort}(E))$ I/Os with high probability. We can also implement a functional external version of Luby's randomized maximal independent set algorithm [17] that uses $O(\mathit{sort}(E))$ I/Os with high probability, yielding the same bounds for maximal matching.

Similar approaches were suggested by Chiang et al. [8] and Mehlhorn [18]. We leave details of these randomized algorithms to the full paper.

5 Semi-External Problems

We now consider *semi-external* graph problems, when $V \leq M$ but $E > M$. These cases often have practical applications, e.g., in graphs induced by monitoring long-term traffic patterns among relatively few nodes in a network. The ability to maintain information about the vertices in memory often simplifies the problems.

For example, the semi-external MST problem can be solved with $O(\mathit{sort}(E))$ I/Os: we simply scan the sorted edge list, using a disjoint set union (DSU) data structure [21] in memory to maintain the forest. We can even solve the problem in $\mathit{scan}(E)$ I/Os, using dynamic trees [20] to maintain the forest. For each edge, we delete the maximum weight edge on any cycle created. The total internal computation time becomes $O(E \log V)$.

Semi-external BMSTs are similarly simplified, because we can check internally if an edge subset spans a graph. Semi-external maximal matching is possible in one edge-list scan, simply by maintaining a matching internally.

If $V \leq M$, we can compute the forest of rooted stars corresponding to the connected components of a graph in one scan, using DSU to maintain a forest internally. We can label the edges by their components in another scan, and we can then sort the edge list to arrange edges contiguously by component. The sorting bound is pessimistic for computing connected components, however, if the number of components is small. Below we give an algorithm to group N records with K distinct keys, so that records with distinct keys appear contiguously, in $O(\mathit{scan}(N) \log_b K)$ I/Os. Therefore, if $V \leq M$ we can compute connected components on a graph $G = (V, E)$ in $O(\mathit{scan}(E) \log_b C(G))$ I/Os, where $C(G) \leq V$ is the number of components of G. In many applications, $C(G) \ll V$, so this approach significantly improves upon sorting.

5.1 Grouping

Let $\mathcal{I} = (x_1, \ldots, x_N)$ be a list of N records. We use x_i to mean the key of record x_i as well as the record itself. The *grouping problem* is to permute $\mathcal{I}$ into a new list $\mathcal{I}' = (x_{\pi_1}, \ldots, x_{\pi_N})$, such that $x_{\pi_i} = x_{\pi_j},\ i < j \Longrightarrow x_{\pi_i} = x_{\pi_{i+1}}$; i.e., all records with equal keys appear contiguously in $\mathcal{I}'$. Grouping differs from sorting in that the order among elements of different keys is arbitrary.

We first assume that the keys of the elements of $\mathcal{I}$ are integers in the range $[1, G]$; we can scan $\mathcal{I}$ once to find G, if necessary. Our grouping algorithm, which we call *quickscan*, recursively re-partitions the elements in $\mathcal{I}$ as follows.

The first pass permutes $\mathcal{I}$ so that elements in the first G/b groups appear contiguously, elements in the second G/b groups appear contiguously, etc. The second pass refines the permutation so that elements in the first G/b^2 groups appear contiguously, elements in the second G/b^2 groups appear contiguously, etc. In general, the output of the kth pass is a permutation $\mathcal{I}_k = (x_{\pi_1^k}, \ldots, x_{\pi_N^k})$, such that

$$\left\lfloor \frac{x_{\pi_i^k}}{\lceil G/b^k \rceil} \right\rfloor = \left\lfloor \frac{x_{\pi_j^k}}{\lceil G/b^k \rceil} \right\rfloor,\ i < j \Longrightarrow \left\lfloor \frac{x_{\pi_i^k}}{\lceil G/b^k \rceil} \right\rfloor = \left\lfloor \frac{x_{\pi_{i+1}^k}}{\lceil G/b^k \rceil} \right\rfloor.$$

Let $\Delta_k = \lceil G/b^k \rceil$. Then keys in the range $[1 + j\Delta_k, 1 + (j+1)\Delta_k - 1]$ appear contiguously in $\mathcal{I}_k$, for all k and $0 \le j < b^k$. $\mathcal{I}_0 = \mathcal{I}$ is the initial input, and $\mathcal{I}' = \mathcal{I}_k$ is the desired final output when $k = \lceil \log_b G \rceil$, bounding the number of passes.

Refining a Partition. We first describe a procedure that permutes a list $\mathcal{L}$ of records with keys in a given integral range $[K, K']$. Let $\delta = \left\lceil \frac{K'-K+1}{b} \right\rceil$. The procedure produces a permutation $\mathcal{L}'$ such that records with keys in the range $[K + j\delta, K + (j+1)\delta - 1]$ occur contiguously, for $0 \le j < b$. Initially, each memory block is empty, and a pointer P points to the first disk block available for the output. As blocks of $\mathcal{L}$ are scanned, each record x is assigned to memory block $m = \left\lfloor \frac{x-K}{\delta} \right\rfloor$. Memory block m will thus be assigned keys in the range $[K + m\delta, K + (m+1)\delta - 1]$. When block m becomes full, it is output to disk block P, and P is updated to point to the next empty disk block.

Since we do not know where the boundaries will be between groups of contiguous records, we must construct a singly linked list of disk blocks to contain the output. We assume that we can reference a disk block with $O(1)$ memory cells; each disk block can thus store a pointer to its successor. Additionally, each memory block m will store pointers to the first and last disk blocks to which it has been written. An output to disk block P thus requires one disk read and two disk writes: to find and update the pointers in the disk block preceding P in $\mathcal{L}'$ and to write P itself.

After processing the last record from $\mathcal{L}$, each of the b memory blocks is empty or partially full. Let M_1 be the subset of memory blocks, each of which was never filled and written to disk; let M_2 be the remaining memory blocks. We compact all the records in blocks in M_1 into full memory blocks, leaving at most one memory block, m_0, partially filled. We then write these compacted blocks (including m_0) to $\mathcal{L}'$.

Let $\{m_1, \ldots, m_\ell\} = M_2$. We wish to write the remaining records in M_2 to disk so that at completion, there will be at most one partial block in $\mathcal{L}'$. Let L_i be the last disk block written from m_i; among all the L_is we will maintain the invariant that there

is at most one partial block. At the beginning, L_0 can be the only partial block. When considering each m_i in turn, for $1 \leq i \leq \ell$, only L_{i-1} may be partial.

We combine in turn the records in L_{i-1} with those in m_i and possibly some from L_i to *percolate* the partial block from L_{i-1} to L_i. Let $|X|$ be the number of records stored in a block X. If $n_i = |L_{i-1}| + |m_i| \geq B$, we add $B - |L_{i-1}|$ records from m_i to L_{i-1}, and we store the remaining $|m_i| - B + |L_{i-1}|$ records from m_i in the new partial block, L'_i, which we insert into $\mathcal{L}'$ after L_i; we then set $L_i \leftarrow L'_i$. Otherwise, $n_i < B$, and we add all the records in m_i as well as move $B - n_i$ records from L_i to L_{i-1}; this again causes L_i to become the partial block.

Quickscan. We now describe phase i of quickscan, given input $\mathcal{I}_{i-1}$. $\Delta_{i-1} = \lceil G/b^{i-1} \rceil$, and keys in the range $[1 + j\Delta_{i-1}, 1 + (j+1)\Delta_{i-1} - 1]$ appear contiguously in $\mathcal{I}_{i-1}$, for $0 \leq j < b^{i-1}$. While scanning $\mathcal{I}_{i-1}$, therefore, we always know the range of keys in the current region, so we can iteratively apply the refinement procedure.

We maintain a value S, which denotes the lowest key in the current range. Initially, $S = \infty$; the first record scanned will assign a proper value to S. We scan $\mathcal{I}_{i-1}$, considering each record x in each block in turn. If $x \notin [S, S + \Delta_{i-1} - 1]$, then x is the first record in a new group of keys, each of which is in the integral range $[S', S' + \Delta_{i-1} - 1]$, for $S' = 1 + \left\lfloor \frac{x-1}{\Delta_{i-1}} \right\rfloor \Delta_{i-1}$. Furthermore, all keys within this range appear contiguously in $\mathcal{I}_{i-1}$. The record read previously to x was therefore the last record in the old range, and so we finish the refinement procedure underway and start a new one to refine the records in the new range, $[S', S' + \Delta_{i-1} - 1]$, starting with x.

We use percolation to combine partial blocks remaining from successive refinements. The space usage of the algorithm is thus optimal: quickscan can be implemented using $2 \cdot \mathit{scan}(N)$ extra blocks, to store the input and output of each phase in an alternating fashion. A non-functional quickscan can be implemented in place, because the ith block output is not written to disk until after the ith block in the input is scanned.

In either case, grouping N items with keys in the integral range $[1, G]$ takes $O(\mathit{scan}(N) \log_b G)$ I/Os. Note that grouping solves the *proximate neighbors* problem [8]: given N records on $N/2$ distinct keys, such that each key is assigned to exactly two records, permute the records so that records with identical keys reside in the same disk block. Chiang et al. [8] show a lower bound of $\Omega(\min\{N, \mathit{sort}(N)\})$ I/Os to solve proximate neighbors in the single-disk I/O model. Our grouping result does not violate this bound, since in the proximate neighbors problem, $G = N/2$.

5.2 Sorting with Duplicates

Because of the compaction of partially filled blocks at the end of a refinement, quickscan cannot sort the output. By using a constant factor extra space, however, quickscan can produce sorted output. This is *sorting with duplicates*.

Consider the refinement procedure applied to a list $\mathcal{L}$. When the last block of $\mathcal{L}$ has been processed, the b memory blocks are partitioned into sets M_1 and M_2. The procedure optimizes space by compacting and outputting the records in blocks in M_1 and then writing out blocks in M_2 in turn, percolating the (at most) one partially filled block on disk. Call the blocks in M_1 *short blocks* and the blocks in M_2 *long blocks*. The global compaction of the short blocks results in the output being grouped but not sorted.

The short blocks can be partitioned into subsets of M_1 that occur contiguously in memory, corresponding to contiguous ranges of keys in the input. We restrict compaction to contiguous subsets of short blocks, which we sort internally. Then we write these subsets into place in the output $\mathcal{L}'$. $\mathcal{L}'$ is thus partitioned into ranges that alternatively correspond to long blocks and contiguous subsets of short blocks. Since a long block and the following contiguous subset of short blocks together produce at most two partially filled blocks in $\mathcal{L}'$, the number of partially filled blocks in $\mathcal{L}'$ is bounded by twice the number of ranges produced by long blocks. Each range produced by long blocks contains at least one full block, so the number of blocks required to store $\mathcal{L}'$ is at most $3 \cdot \mathit{scan}(|\mathcal{L}|)$. The output $\mathcal{L}'$, however, is sorted, not just grouped.

We apply this modified refinement procedure iteratively, as above, to sort the input list $\mathcal{I}$. Each sorted, contiguous region of short blocks produced in phase i, however, is skipped in succeeding phases: we need not refine it further. Linking regions of short blocks produced at the boundaries of the outputs of various phases is straightforward, so in general the blocks in any phase alternate between long and short. Therefore, skipping the regions of short blocks does not affect the I/O complexity.

Sorting N records with G distinct keys in the integral range $[1, G]$ thus takes $\Theta(\mathit{scan}(N) \log_b G)$ I/Os. The algorithm is stable, assuming a stable internal sort.

5.3 Grouping with Arbitrary Keys

If the G distinct keys in $\mathcal{I}$ span an arbitrary range, we can implement a randomized quickscan to group them. Let H be a family of universal hash functions [6] that map $\mathcal{I}$ to the integral range $[1, b]$. During each phase i in randomized quickscan, we pick an h_i uniformly at random from H and consider $h_i(x)$ to be the key of $x \in \mathcal{I}$.

Each bucket of records output in phase i is thus refined in phase $i + 1$ by hashing its records into one of b new buckets, using function h_{i+1}. Let η be the number of distinct keys in some bucket. If $\eta \leq b$, we can group the records in one additional scan. Linking the records in the buckets output in the first phase T in which each bucket has no more than b distinct keys therefore produces the desired grouped output.

Let η' be the number of distinct keys hashed into some new bucket from a bucket with η distinct keys. The properties of universal hashing [6] show that $E[\eta'] = \eta/b$. Theorem 1.1 of Karp [13] thus shows that $\Pr[T \geq \lfloor \log_b G \rfloor + c + 1] \leq G/b^{\lfloor \log_b G \rfloor + c}$ for any positive integer c. Therefore, with high probability, $O(\mathit{scan}(N) \log_b G)$ I/Os suffice to group $\mathcal{I}$. We use global compaction and percolation to optimize space usage.

6 Conclusion

Our functional approach produces external graph algorithms that compete with the I/O performance of the best previous algorithms but that are simpler to describe and implement. Our algorithms are conducive to standard checkpointing and programming language optimization tools. An interesting open question is to devise incremental and dynamic algorithms for external graph problems. The data-structural approach of Arge [3] and Kumar and Schwabe [15] holds promise for this area. Designing external graph algorithms that exploit parallel disks also remains open. Treating P disks as one with

a block size of PB extends standard algorithms only when $P = O((M/B)^\alpha)$ for $0 \leq \alpha < 1$. In this case, the $\log_{M/B}$ terms degrade by only a constant factor.

Acknowledgements. We thank Ken Church, Kathleen Fisher, David Johnson, Haim Kaplan, David Karger, Kurt Mehlhorn, and Anne Rogers for useful discussions.

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *C. ACM*, 31(8):1116–27, 1988.
2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
3. L. Arge. The buffer tree: A new technique for optimal I/O algorithms. In *Proc. 4th WADS*, volume 955 of *LNCS*, pages 334–45. Springer-Verlag, 1995.
4. O. Borůvka. O jistém problému minimálním. *Práce Mor. Přírodověd. Spol. v Brně*, 3:37–58, 1926.
5. P. M. Camerini. The min-max spanning tree problem and some extensions. *IPL*, 7:10–4, 1978.
6. J. L. Carter and M. N. Wegman. Universal classes of hash functions. *JCSS*, 18:143–54, 1979.
7. Y.-J. Chiang. *Dynamic and I/O-Efficient Algorithms for Computational Geometry and Graph Problems: Theoretical and Experimental Results*. PhD thesis, Dept. of Comp. Sci., Brown Univ., 1995.
8. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. 6th ACM-SIAM SODA*, pages 139–49, 1995.
9. F. Y. Chin, J. Lam, and I.-N. Chen. Efficient parallel algorithms for some graph problems. *C. ACM*, 25(9):659–65, 1982.
10. J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *JCSS*, 38(1):86–124, February 1989.
11. J. Ja'Ja. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.
12. D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–28, 1995.
13. R. M. Karp. Probabilistic recurrence relations. *J. ACM*, 41(6):1136–50, 1994.
14. P. Kelsen. An optimal parallel algorithm for maximal matching. *IPL*, 52(4):223–8, 1994.
15. V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. 8th IEEE SPDP*, pages 169–76, 1996.
16. M. J. Litzkow and M. Livny. Making workstations a friendly environment for batch jobs. In *Proc. 3rd Wks. on Work. Oper. Sys.*, pages 62–7, April 1992.
17. M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comp.*, 15:1036–53, 1986.
18. K. Mehlhorn. Personal communication. `http://www.mpi-sb.mpg.de/~crauser-/courses.html`, 1998.
19. J. S. Plank, M. Beck, G. Kingsley, and K. Li. **Libckpt**: Transparent checkpointing under UNIX. In *Proc. USENIX Winter 1995 Tech. Conf.*, pages 213–23, 1995.
20. D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *JCSS*, 26(3):362–91, 1983.
21. R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–81, 1984.
22. P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comp. Sci.*, 73:231–48, 1990.