A Parallel Priority Data Structure with Applications[*]

Gerth Stølting Brodal[†]
BRICS[‡] Dept. of Computer Science, University of Aarhus, DK-8000 Århus C, Denmark
Email: gerth@brics.dk

Jesper Larsson Träff   Christos D. Zaroliagis
Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany
Email: {traff,zaro}@mpi-sb.mpg.de

Abstract

We present a parallel priority data structure that improves the running time of certain algorithms for problems that lack a fast and work-efficient parallel solution. As a main application, we give a parallel implementation of Dijkstra's algorithm which runs in $O(n)$ time while performing $O(m \log n)$ work on a CREW PRAM. This is a logarithmic factor improvement for the running time compared with previous approaches. The main feature of our data structure is that the operations needed in each iteration of Dijkstra's algorithm can be supported in $O(1)$ time.

1 Introduction

Developing work-efficient parallel algorithms for graph and network optimization problems continues to be an important area of research in parallel computing. Despite much effort a number of basic problems have tenaciously resisted a very fast (i.e., NC) parallel solution that is simultaneously work-efficient. A notorious example is the single-source shortest path problem.

The best sequential algorithm for the single-source shortest path problem on directed graphs with non-negative real-valued edge weights is Dijkstra's algorithm [5]. For a given digraph $G = (V, E)$ the algorithm iteratively steps through the set of vertices, in each iteration fixing the distance of a vertex for which a shortest path has been found, while maintaining in the process, for each of the remaining vertices, a tentative distance from the source. For an n-vertex, m-edge digraph the algorithm can be implemented to run in $O(m + n \log n)$ operations by using efficient priority queues like Fibonacci heaps [7] for maintaining tentative distances, or other priority queue implementations supporting deletion of the minimum key element in amortized or worst-case logarithmic time, and decrease key in amortized or worst-case constant time [3, 6, 10].

The single-source shortest path problem is in NC (by virtue of the all-pairs shortest path problem being in NC), and thus a fast parallel algorithm exists, but for general digraphs no *work-efficient* algorithm in NC is known. (The best NC algorithm runs in $O(\log^2 n)$ time and performs $O(n^3(\log\log n/\log n)^{1/3})$ work on an EREW PRAM [9].) Moreover, work-efficient algorithms which are (at least) sublinearly fast are also not known for general digraphs.

Dijkstra's algorithm is highly sequential, and can probably not be used as a basis for a fast (NC) parallel algorithm. However, it is easy to give a parallel implementation of the algorithm that runs in $O(n \log n)$ time [11]. The idea is to perform the distance updates within each iteration in parallel by associating a local priority queue with each processor. The vertex of minimum distance for the next iteration is determined (in parallel) as the minimum of the minima in the local priority queues. For this parallelization it is important that the priority queue operations have worst-case running time, and therefore the original Fibonacci heap cannot be used to implement the local queues. This was first observed in [6] where a new data structure, called relaxed heaps, was developed to overcome this problem. Using relaxed heaps, an $O(n \log n)$ time and $O(m + n \log n)$ work(-optimal) parallel implementation of Dijkstra's algorithm is obtained. This seems to be the currently fastest work-efficient paral-

[*]This work was partially supported by the EU ESPRIT LTR Project No. 20244 (ALCOM-IT), and by the DFG project SFB 124-D6 (VLSI Entwurfsmethoden und Parallelität).

[†]Supported by the Danish Natural Science Research Council (Grant No. 9400044).

[‡]Basic Research in Computer Science, a Centre of the Danish National Research Foundation.

lel algorithm for the single-source shortest path problem. The parallel time spent in each iteration of the above implementation of Dijkstra's algorithm is determined by the (processor local) priority queue operations of finding a vertex of minimum distance and deleting an arbitrary vertex, plus the time to find and broadcast a global minimum among the local minima. Either or both of the priority queue operations take $O(\log n)$ time, as does the parallel minimum computation; for the latter $\Omega(\log n)$ time is required, even on a CREW PRAM. Hence, the approach with processor local priority queues does not seem to make it possible to improve the running time beyond $O(n \log n)$ without resorting to a more powerful PRAM model. This was considered in [11] where two faster (but not work-efficient) implementations of Dijkstra's algorithm were given on a CRCW PRAM: the first (resp. second) algorithm runs in $O(n \log \log n)$ (resp. $O(n)$) time, and performs $O(n^2)$ (resp. $O(n^{2+\epsilon})$, $\forall 0 < \epsilon < 1$) work.

An alternative approach would be to use a parallel global priority queue supporting some form of multi-decrease key operation. Unfortunately, no known parallel priority queues support such an operation; they only support a multi-delete operation which assumes that the k elements to be deleted are the k elements with smallest priority in the priority queue (see e.g., [2] and the references in that paper). A different idea is required to improve upon the running time.

We present a parallel priority data structure that speeds up the parallel implementation of Dijkstra's algorithm, by supporting the operations required at each iteration in $O(1)$ time. Using this data structure we give an alternative implementation of Dijkstra's algorithm that runs in $O(n)$ time and performs $O(m \log n)$ work on a CREW PRAM. More specifically, by sorting the adjacency lists (after weight) it is possible in constant time both to determine a vertex of minimum distance, as well as to add (in parallel) any number of new vertices and/or update the distance of vertices maintained by the priority data structure. It should also be mentioned that the PRAM implementation of the data structure requires concurrent read only for broadcasting constant size information to all processors in constant time.

The idea of the parallel priority data structure is to perform a *pipelined merging* of keys. We illustrate the idea by first giving a simple implementation using a linear pipeline, which requires $O(n^2 + m \log n)$ work (Sec. 2). We then sketch how the pipeline can be dynamically restructured in a tree like fashion such that only $O(m \log n)$ operations are required (Sec. 3). Further applications are discussed in Sec. 4. Due to space limitations many details and proofs are omitted.

2 A parallel priority data structure

In this section we introduce our new parallel priority data structure, and show how to use it to give an alternative, parallel implementation of Dijkstra's algorithm. Let $G = (V, E)$ be an n-vertex, m-edge directed graph with edge weights $c : E \to \mathbb{R}_0^+$, represented as a collection of adjacency lists. For a set $S \subseteq V$ of vertices, define $\Gamma(S)$ to be the neighbors of the vertices in S, excluding vertices in S, i.e., $\Gamma(S) = \{w \in V \setminus S | \exists\ v \in S, (v, w) \in E\}$. We associate with each vertex $v \in S$ a (fixed) real-valued label Δ_v. For a vertex $w \in \Gamma(S)$, define the *distance from S to w* as $\text{dist}(S, w) = \min_{u \in S}\{\Delta_u + c(u, w)\}$. The distance has the property that $\text{dist}(S \cup \{v\}, w) = \min\{\text{dist}(S, w), \Delta_v + c(v, w)\}$. We define the *vertex closest to S* to be the vertex $z \in \Gamma(S)$ that attains the minimum $\min_{w \in \Gamma(S)}\{\text{dist}(S, w)\}$ (with ties broken arbitrarily).

Assume that a processor P_v is associated with each vertex $v \in V$ of G. Among the processors associated with vertices in S at any given instant one will be designated as the *master processor*. Our data structure supports the following four operations:

- Init: initializes the priority data structure.
- Eject(S): deletes the vertex v of $\Gamma(S)$ that is closest to S, and returns the pair (v, D_v) to the master processor, where $D_v = \text{dist}(S, v)$.
- Extend(S, v, Δ, P_v): adds a vertex v associated with processor P_v to S, and assigns it label Δ. Processor P_v becomes the new master processor.
- Empty(S): returns **true** to the master processor of S if $\Gamma(S) = \emptyset$.

Performing $|\Gamma(S)|$ successive Eject-operations on a set S ejects the vertices in $\Gamma(S)$ in non-decreasing order of closeness, and leaves the priority data structure empty. Each vertex of $\Gamma(S)$ is ejected once. Note also that there is no operation to change the labels associated with vertices in S.

These operations suffice for an alternative, parallel implementation of Dijkstra's algorithm. Let $s \in V$ be a distinguished *source vertex*. The algorithm computes for each vertex $v \in V$ the length of a shortest path from s to v, where the length of a path is the sum of the weights of the edges on the path. Dijkstra's algorithm maintains a set S of vertices for which a shortest path have been found, in each iteration adding one more vertex to S. Each vertex $w \in V \setminus S$ has a tentative distance which is equal to $\text{dist}(S, w)$ as defined above. Hence, instead of the usual priority queue with DeleteMin to select the vertex closest to S, and DecreaseKey operations to update tentative distances for the vertices in $V \setminus S$, we use the priority data structure above to determine in each iteration a vertex closest to the current set S of correct vertices. The Extend-operation replaces the updating of tentative distances. Let P_v be the processor associated with vertex v.

```
Algorithm New-Parallel-Dijkstra
/* Initialization */
Init; $d(s) \leftarrow 0$; $S \leftarrow \emptyset$;
Extend($S, s, d(s), P_s$);
/* Main loop */
while ¬Empty($S$) do
    $(v, D_v) \leftarrow$ Eject($S$); /* instead of DeleteMin */
    $d(v) \leftarrow D_v$;
    Extend($S, v, d(v), P_v$);
    /* replaces the update step */
od
```

Our main result in this section is that the New-Parallel-Dijkstra algorithm runs in linear time in parallel.

Theorem 1 *Dijkstra's algorithm can be implemented to run in $O(n)$ time and $O(n^2 + m \log n)$ work using $O(n + m)$ space on a CREW PRAM.*

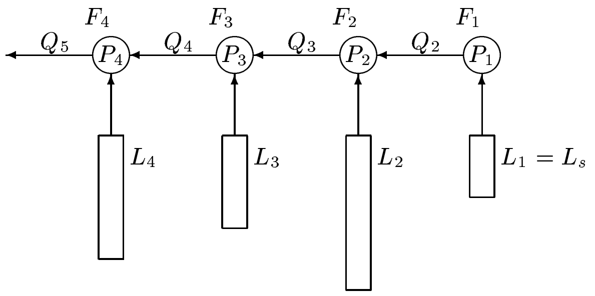
The proof of Theorem 1 is based on the following.

Lemma 1 *Operation* Init *takes $O(m \log n)$ work and $O(\log n)$ time. After initialization, each* Eject(S)*-operation takes constant time using $|S|$ processors, and each* Extend(S, v, Δ, P_v)*-operation takes constant time using $|S| + \deg_{\text{in}}(v)$ processors, where $\deg_{\text{in}}(v)$ is the in-degree of v. The* Empty(S)*-operation takes constant time per processor. The space required per processor is $O(n)$.*

The remainder of this section will be devoted to provide a sketch of a proof for Lemma 1.

In the Init-operation the adjacency lists of G are sorted in non-decreasing order after edge weight, i.e., on the adjacency list of v vertex w_1 appears before w_2 if $c(v, w_1) \leq c(v, w_2)$ (with ties broken arbitrarily). The adjacency lists are assumed to be implemented as doubly linked lists, such that any vertex w on v's adjacency list can be removed in constant time. For each vertex v we also associate an array of vertices u_j to which v is adjacent, i.e., vertices u_j for which $(u_j, v) \in E$. In the array of v we store for each such u_j a pointer to the position of v in the adjacency list of u_j. This enables us to delete all occurrences of v in adjacency lists of such vertices $u_j \in \overline{S} = V \setminus S$ in constant time. Sorting of the adjacency lists takes $O(\log n)$ time and $O(m \log n)$ work [4]. Constructing links and building the required arrays can then be done in constant time using $O(m)$ operations. This completes the description of the Init operation.

The processors associated with vertices in S at any given instant are organized in a linear pipeline. Let v_i be the ith vertex added to S, let S_i denote S after the ith Extend(S, v_i, Δ_i, P_i)-operation where Δ_i is the label to be associated with v_i, and let P_i be the processor assigned to v_i (in the implementation of Dijkstra's algorithm the label Δ_i to be associated with vertex v_i was $d(v)$). Let finally L_i be the sorted, doubly linked adjacency list of v_i. Processor P_i which was assigned at the ith Extend-operation receives input from P_{i-1}, and, after the $(i+1)$th Extend-operation, will send output to P_{i+1}. The last processor assigned to S will be the master processor, and the output from this processor will be the result of the next Eject-operation, i.e., the vertex closest to S. The pipeline for $i = 4$ is shown below. The input queue Q_1 of processor P_1 is empty and not shown.



Assume now that Eject(S_{i-1}) can be performed in constant time by the processors assigned to the vertices in S_{i-1}, and returns to the master processor of S_{i-1} the vertex in $\Gamma(S_{i-1})$ that is closest to S_{i-1}. We show how to maintain this property after an Extend-operation; more specifically, that the vertex v ejected by Eject(S_i), immediately after Extend($S_{i-1}, v_i, \Delta_i, P_i$), is produced in constant time, is indeed the vertex closest to S_i, and that each vertex in $\Gamma(S_i)$ is ejected exactly once.

Performing an Eject(S_{i-1}) returns the vertex u closest to S_{i-1} with value $D_u = \text{dist}(S_{i-1}, u)$. Either this vertex, or the vertex closest to v_i is the vertex to be ejected from S_i. Let w be the first vertex on the sorted adjacency list L_i. If $\Delta_i + c(v_i, w) \leq D_u$, then the result of Eject(S_i) is w with value $D_w = \Delta_i + c(v_i, w)$; otherwise, the result is u with value D_u. In the first case, w is ejected and simply removed from L_i, but the ejected vertex of S_{i-1} must be saved for a later Eject-operation. For this purpose we associate an *input queue* Q_i with each P_i which stores the vertices ejected from S_{i-1} by processor P_{i-1}. The Eject-operation of P_i thus consists in selecting the smaller value from either the input queue Q_i or the adjacency list L_i of v_i. In other words, P_i performs one merging step of the two ordered lists Q_i and L_i. In case P_i exhausts its own adjacency list L_i, it always ejects from Q_i. It can be shown that Q_i never gets empty, unless all vertices of $\Gamma(S_{i-1})$ have been ejected, in which case processor P_i may terminate. The EMPTY(S_i) thus has to return **true** when both adjacency list L_i and input queue Q_i of the master processor are empty.

In order to ensure that a vertex output by P_i is never output at a later EJECT-operation (i.e., inserted into Q_{i+1} with different priorities), we associate a set F_i of *forbidden* vertices with each P_i. Each F_i set is implemented as a Boolean array (i.e., $F_i[w]$ = **true** iff w has been ejected from L_i). When a vertex w is removed from L_i and ejected, w is put into F_i and removed from Q_i (if it is there). A vertex ejected from S_{i-1} is only put into the input queue Q_i of P_i if it is *not* in the forbidden set F_i of P_i. In the case where a vertex u at the head of Q_i (previously ejected from S_{i-1}) "wins" at P_i and is ejected, it is removed from L_i (in case u is adjacent to v_i), and is made forbidden for P_i by putting it into F_i. In order to be able to remove vertices from Q_i in constant time, each P_i has an array of pointers into Q_i, which is updated whenever P_{i-1} outputs a vertex into Q_i. The complete EJECT-operation looks as follows:

```
Function EJECT($S$)
for all $v_i \in S$ do in parallel
        /* processor $P_i$ is associated with vertex $v_i$ */
        $(v', D') \leftarrow$ HEAD$(Q_i)$;
        $v'' \leftarrow$ HEAD$(L_i)$; $D'' \leftarrow c(v_i, v'') + \Delta_i$;
        if $D'' < D'$ then $(v', D') \leftarrow (v'', D'')$ fi;
        remove $v'$ from $L_i$ and $Q_i$ if present;
        insert $v'$ into $F_i$;
        if $v' \notin F_{i+1}$ then append $(v', D')$ to $Q_{i+1}$ fi
od;
if $P_i$ is the master processor return HEAD$(Q_{i+1})$
```

An EXTEND($S_{i-1}, v_i, \Delta_i, P_i$)-operation must first perform an EJECT(S_{i-1}) in order to get an element into the input queue Q_i of P_i. Since we must prevent that a vertex already in S is ever ejected (as $\Gamma(S)$ excludes S), once a vertex is appended to S it must be removed from the adjacency lists of all vertices in $\overline{S}$. This can be done in parallel in constant time using the array of pointers constructed by the INIT-operation (since v occurs at most once in any adjacency list), if concurrent read is allowed: a pointer to the list of vertices u_j to which v is adjacent must be made available to all processors. In parallel they remove v from the adjacency lists of the u_j's, which takes constant time using $\deg_{\text{in}}(v)$ processors, $\deg_{\text{in}}(v)$ being the *in-degree* of v. The required concurrent read is of the restricted sort of broadcasting the same constant size information to all processors. The EXTEND-operation looks as follows.

```
Function EXTEND($S, v, \Delta, P$)
connect the master processor of $S$ to $P$;
make $P$ the (new) master processor;
$(u, D') \leftarrow$ EJECT$(S)$;
append $(u, D')$ to the input queue $Q$ of $P$;
$\Delta_v \leftarrow \Delta$; $S \leftarrow S \cup \{v\}$;
remove $v$ from $\overline{S}$ using pointers constructed by INIT
```

The $O(n^2)$ space due to the forbidden sets and the arrays of pointers into the input queues can be reduced to $O(n+m)$. Instead of maintaining the forbidden sets F_i explicitly, we let each occurrence of each vertex in the priority data structure carry information about whether it has been forbidden and if so, by which processor. Maintaining for each vertex $v \in V$ a doubly linked list of its occurrences in the data structure makes it possible for processor P_i to determine in constant time whether a given vertex v has been forbidden for processor P_{i+1}, and to remove v in constant time from Q_i whenever it is ejected from L_i.

This concludes the sketch of the proof of Lemma 1, Theorem 1 and the basic implementation of the priority data structure.

3 A dynamic tree pipeline

We now briefly describe how to decrease the amount of work required by the algorithm in Sec. 2. Before doing so, we first make an observation about the merging part of the algorithm. The work done by processor P_i is intuitively to output vertices by incrementally merging its adjacency list L_i with the incoming stream Q_i of vertices output by processor P_{i-1}. Processor P_i terminates when it has nothing left to merge. An alternative bound on the real work done by this algorithm is then the sum of the distance each vertex v from an adjacency list L_i travels, where the distance is the number of processors that output v. Because each vertex v from L_i can at most be output by a prefix of the processors $P_i, P_{i+1}, \ldots, P_n$, the distance v travels is at most $n - i + 1$. This gives a total bound on the work done by the processors of $O(mn)$. That the real work can actually be bounded by $O(n^2)$ is due to the fact that vertices get annihilated by forbidden sets.

Using this view of the work done by the algorithm during merging, we sketch now a variation of the data structure that basically bounds the distance a vertex can travel by $O(\log n)$, i.e., bounds the work by $O(m \log n)$. The main idea is to replace the sequential pipeline of processors by a binary tree pipeline of processors of height $O(\log n)$. Each processor P_i still maintains an adjacency list L_i and a set of forbidden vertices F_i. The output of processor P_i is still inserted into an input queue of a processor P_j, but P_i can now receive input from two processors instead of one.

The basic organization of the processor connections are perfect binary trees. Each node corresponds to a processor and the unique outgoing edge of a node corresponds to the output queue of the node (and an input queue to the successor node). The rank of a node is the height of the node in the perfect binary tree and the rank of a tree is the rank of the root. The nodes are connected such that the incoming edges of a node v come from the left child of v and the sibling of v.

The processors are organized in a sequence of trees of rank $r_k, r_{k-1} \ldots, r_1$, where the ith root is connected to the $i+1$st root. We maintain the invariant that

$$r_k \leq r_{k-1} < r_{k-2} < \cdots < r_2 < r_1. \qquad (1)$$

When performing an EXTEND-operation a new processor is initialized. If $r_k < r_{k-1}$ the new processor is inserted as a new rank one tree at the front of the list of trees (as in the sequential pipeline case). That (1) is satisfied follows from $1 \leq r_k < r_{k-1} < \cdots < r_1$. If $r_k = r_{k-1}$ we link the kth and $k-1$st tree with the new node to form a tree of rank $1 + r_{k-1}$. That (1) is satisfied follows from $1 + r_{k-1} \leq r_{k-2} < r_{k-3} < \cdots < r_1$.

For the tree pipeline we can show that all non-terminated processors have the next vertex to be output in one of its input queues. What remains is to divide the work among the available processors. Assuming that $O(\frac{m \log n}{n})$ processors are available, the idea is to simulate the tree structured pipeline for $O(\log n)$ time steps, after which we stop the simulation and in $O(\log n)$ time eliminate the (simulated) terminated processors, and reschedule. By this scheme a terminated processor is kept alive for only $O(\log n)$ time steps, and hence no superfluous work is done. In total the simulation takes linear time. Thus, we have:

Theorem 2 *Dijkstra's algorithm can be implemented to run in $O(n)$ time and $O(m \log n)$ work on a CREW PRAM.*

4 Further applications

The improved single-source shortest path algorithm immediately gives rise to corresponding improvements in algorithms in which the single-source shortest path problem occurs as a subproblem. We mention here the assignment problem, the minimum cost flow problem, (for definitions see [1]), and the single-source shortest path problem in planar digraphs. For example, the minimum cost flow problem (which is P-complete [8]) can be solved by $O(m \log n)$ calls to Dijkstra's algorithm (see e.g. [1, Sec. 10.7]). Using our implementation, we obtain a parallel algorithm that runs in $O(nm \log n)$ time and performs $O(m^2 \log^2 n)$ work. Our bounds are strongly polynomial and speed up the best previous ones [6] by a logarithmic factor. (Similar improvements hold for the assignment problem.)

Greater parallelism for the single-source shortest path problem in the case of planar digraphs can be achieved by plugging our implementation of Dijkstra's algorithm into the algorithm of [12] resulting in an algorithm which runs $O(n^{2\epsilon} + n^{1-\epsilon})$ time and performs $O(n^{1+\epsilon})$ work on a CREW PRAM. With respect to work, this gives the best (deterministic) parallel algorithm known for the single-source shortest path problem in planar digraphs that runs in sublinear time.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows.* Prentice-Hall, 1993.

[2] G. Brodal. Priority queues on parallel machines. In *Algorithm Theory – SWAT'96*, volume 1097 of *Lecture Notes in Computer Science*, pages 416–427, 1996.

[3] G. Brodal. Worst-case efficient priority queues. In *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 52–58, 1996.

[4] R. Cole. Parallel merge sort. *SIAM Journal of Computing*, 17(4):770–785, 1988.

[5] E. W. Dijkstra. A note on two problems in connexion with graphs. *Num. Mathematik*, 1:269–271, 1959.

[6] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

[7] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[8] L. M. Goldschlager, R. A. Shaw, and J. Staples. The Maximum Flow Problem is Log Space Complete for P. *Theoretical Computer Science*, 21:105–111, 1982.

[9] Y. Han, V. Pan, and J. Reif. Efficient parallel algorithms for computing all pair shortest paths in directed graphs. In *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, pages 353–362, 1992.

[10] P. Høyer. A general technique for implementation of efficient priority queues. In *Proc. 3rd Israel Symp. on Theory of Comp. and Systems (ISTCS'95)*, pages 57–66, 1995.

[11] R. C. Paige and C. P. Kruskal. Parallel algorithms for shortest path problems. In *International Conference on Parallel Processing*, pages 14–20, 1985.

[12] J. L. Träff and C. D. Zaroliagis. A simple parallel algorithm for the single-source shortest path problem on planar digraphs. In *Parallel Algorithms for Irregularly Structured Problems*, volume 1117 of *Lecture Notes in Computer Science*, pages 183–194, 1996.