FROM PARALLEL TO SEQUENTIAL: KEEPING OPTIMALITY IN ALGORITHMS

Panagiotis Metaxas* Michael Mytilinaios†
Wellesley College‡ Athens University of Economics§

Abstract

In this paper we first give a survey that describes the reductions between several fundamental parallel graph problems. Then, we examine the possibility of designing simple and practical sequential algorithms from parallel ones. Interestingly enough, we show how to design fast (in fact *optimal*) sequential algorithms from the optimal parallel algorithms for a graph theoretic problem that arizes often in many areas including Economics and Operations Research. In particular, we give two optimal algorithms that solve the *vertex updating for a minimum spanning tree* problem. Finally, we show how these algorithms can be used to obtain incremental algorithms for computing the minimum spanning tree (MST) of a graph. The MST algorithms are optimal for dense graphs.

1 Introduction

Parallelizing Sequential Algorithms. Ever since the first attempt of building parallel machines, researchers have looked into the existing sequential algorithms in an effort to parallelize them efficiently. Unfortunately, most of the time, this proved to be a difficult task, because some of the basic sequential algorithms turned out to be inefficient in their parallel versions. Breadth-first search of a graph, one of the basic graph searching techniques, has a fast parallel version which runs in $O(\log^2 n)$ time, where n is the number of graph vertices. However, this algorithm uses a large number (almost n^3) of processors, making the solution impractical. Others, like the well known depth-first search algorithm, do not seem to have a corresponding fast parallel version at all. (By "fast" we mean an algorithm running in polylogarithmic parallel time using a polynomial number of processors.)

Reductions. These observations suggest that the reductions between parallel graph algorithms do not resemble those of sequential algorithms. The reduction graph of Figure 1

*Email address: takis@bambam.wellesley.edu
†Email address: mmit@isosun.ARIADNE-t.gr
‡Department of Computer Science, 106 Central Street, Wellesley, MA 02181, USA.
§Department of Informatics, 76 Patission Street, Athens, Greece 104 34.

outlines a survey that depicts this situation. In this graph, an arc from problem X to problem Y implies that Y uses as subroutine an algorithm that solves X. A label on the graph refers to the paper in the bibliography that establishes the reduction. A preliminary version of this survey appeared in [Met91].

Optimality and Efficiency. Before we proceed, we need a few definitions. We say that a sequential algorithm for some problem of size n is *optimal* if it runs in time that matches a lower bound for the problem to within a constant factor. Let $t(n)$ denote the parallel running time for some parallel algorithm, and $p(n)$ denote the number of processors employed by the algorithm. Then, $w(n) = t(n) \times p(n)$ denotes the *work* performed by the algorithm. A parallel algorithm for some problem is said to be *optimal* if it has polylogarithmic parallel running time and the work $w(n)$ performed by the algorithm is $O(T(n))$, where $T(n)$ is the running time of the best sequential algorithm for the same problem. If, instead, $w(n)$ is within a polylogarithmic factor of the optimal speedup, the algorithm is called *efficient*.

It is well known that a parallel algorithm that performs work $w(n)$ can be simulated on a sequential machine in time $O(w(n))$. Such simulations, however, have usually a large overhead because the uni-processor machine has to switch context very often in order to simulate a single step on each processor. Thus, these simulations are considered impractical and complicated. In our approach, we will be looking for algorithms for which we can separate the *computing* and *scheduling* parts. We can think of the computing part as the "heart" of the algorithm, i.e. the part of the code that the solution is computed. On the other hand, the scheduling part is the code that "directs" the solution. In the algorithms we present, the computing part is identical in both the parallel and the sequential algorithms.

In the remaining of this paper we examine the possibility of designing simple and practical sequential algorithms from parallel ones. Interestingly enough, we show how to design fast (in fact optimal) sequential algorithms from the optimal parallel algorithms for a graph theoretic problem that arizes often in many areas including Economics and Operations Research. In particular, we give two optimal algorithms that solve the vertex updating for a MST problem. Finally, we show how these algorithms can be used to obtain incremental algorithms for computing the minimum spanning tree (MST) of a graph. The MST algorithms are optimal for dense graphs.

1.1 Definition of the Problem.

The vertex updating problem for a minimum spanning tree is defined as follows: We are given a weighted graph $G = (V, E_G)$ with $n = |V|$ vertices and $m = |E_G|$ edges, along with a MST $T = (V, E)$. The graph is augmented by a new vertex z and n weighted edges connecting z to every vertex in V. We want to compute a new MST $T' = (V \cup \{z\}, E')$.

The updating MST problem arizes often in practice in very diverse areas that vary from VLSI design and Artificial Intelligence to Operations Research and Economics. The problem has been addressed in the past in both the parallel [PR86, VD86, JM88, JM92a] and sequential settings. Optimal sequential algorithms have been given in [SP75, CH78].

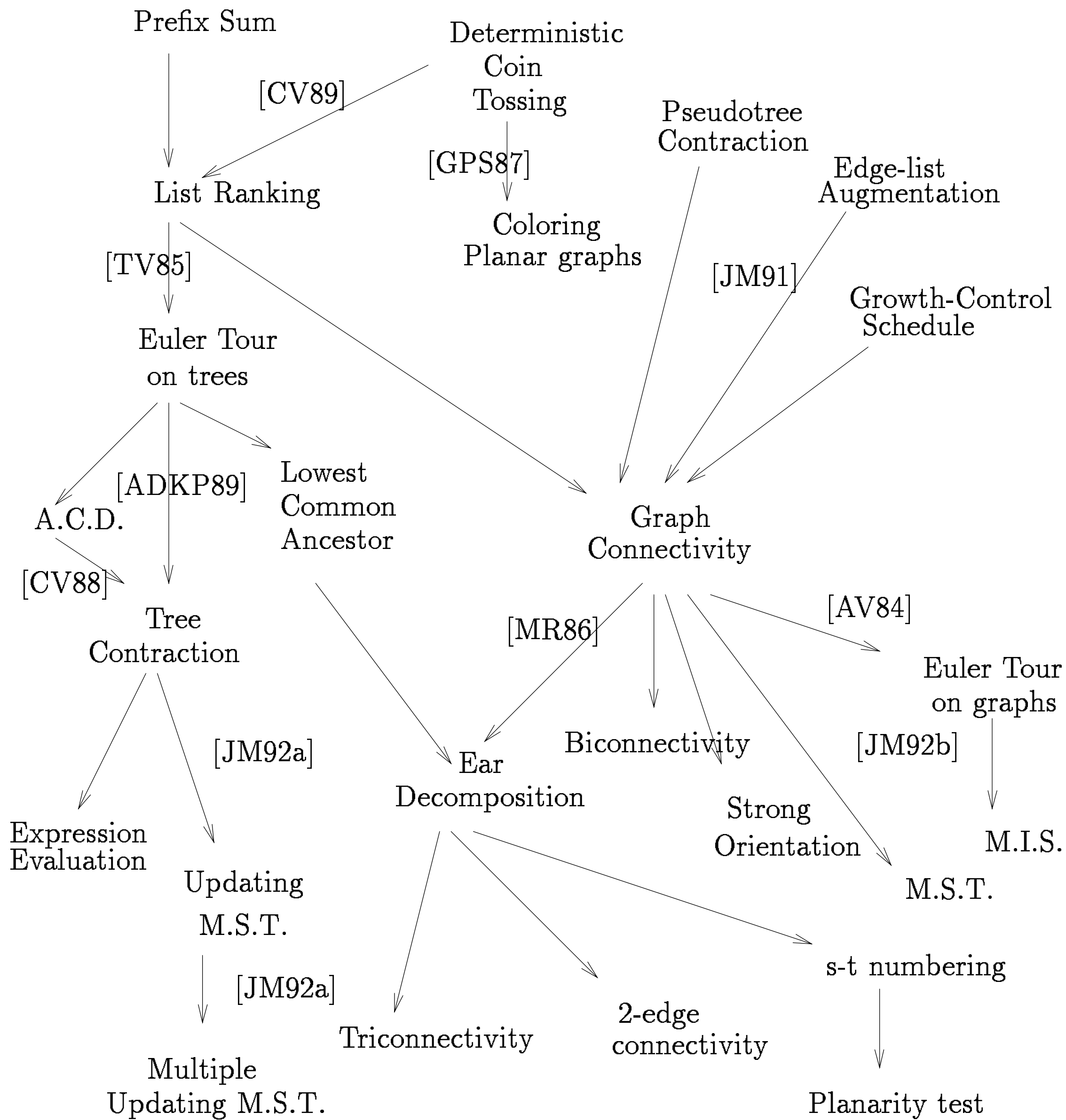

Figure 1: The reduction graph of parallel algorithms for some basic graph problems.

In [JM92a] a set of *rules* is given which is used in conjunction with *parallel tree contraction* methods to develop optimal parallel algorithms for the EREW PRAM (exclusive-read exclusive-write parallel random access machine) model, the weakest of the PRAM models. These algorithms run in $t(n) = O(\lg n)$ time (where $\lg n$ denotes $\log_2 n$) using $p(n) = n/\lg n$ processors, thus achieving work optimality of $w(n) = t(n) \times p(n) = O(n)$. Note that the vertex updating problem has a lower bound of $\Omega(n)$ sequential time. To see that, consider the time it takes to find the maximum weighted edge on a tree composed of a single path of length $n-1$, and whose ends are connected to the newly introduced vertex z.

Two Useful Observations We will use the rules of [JM92a] to develop simple and practical optimal sequential algorithms for the updating MST problem. These rules try to break *small* cycles (i.e. cycles of length 3 or 4) and are based on the following two observations:

1. The edge with minimum weight incident to some vertex will always be included into the MST. In fact, many sequential and parallel algorithms are based on this observation (Prim's algorithm and [SP75, CLC82] actually use this fact). In our algorithms, edge *inclusion* makes use of this observation.

2. Whenever some edge is found to correspond to the maximum weighted edge (MaxWE) of some cycle it can be removed from the tree without affecting the computation of the remaining graph. (Kruskal's MST algorithm makes use of this fact.) In the algorithms we describe, edge *exclusion* is based on this observation.

2 The Algorithms

Representation. Upon introducing the new vertex z along with n weighted edges, $\binom{n}{2}$ cycles are created. [1] If we break these cycles by deleting the maximum weight edge (MaxWE) that appears in each cycle, the resulting tree will be the new MST T'. It is easy to see that at most n of these $2n-1$ weighted edges will be included into T'. No non-tree edge of the old graph can be included because all of them are already MaxWE on some existing cycle in the original graph, so we need not consider any such edge. For this reason we may take the input to be a tree T with $n-1$ weighted edges (corresponding to the given MST) and n weighted nodes (corresponding to weights of the newly introduced edges to z). We will call this object a *weighted tree.*

Without loss of generality we may assume that the weighted tree is binary. If this is not the case, there is an easy transformation which, given a weighted tree having vertices of unbounded degree, transforms it into an equivalent binary weighted tree $T = (V_T, E_T)$. This transformation is as follows: Each node $v = u_0$ with $k > 2$ children is augmented with $k-2$ fake nodes u_i to form a right path. Each of the children v_j of node v is attached as a left child of u_{j-1}, while u_{k-2} has a right child as well. The edges connecting the u_i's

[1] If, in an instance of the problem, z is not connected to some vertex x, we can assume an edge (z, x) having maximum weight.

have weights $-\infty$ which makes them unremovable by the algorithm. The fake nodes do not have weights. They are introduced only to fix the order of processing of v's children. At the end of the algorithm the right path is always included in the MST of the binarized problem, giving a unique obvious solution to the general problem.

We assume that for each node of the binary tree the following pointers are defined: `par(v)` and `ch(v)` denote the parent and the child of node v. Note that whenever we refer to a child of some node v, there will be no ambiguity because v will only have one child. Also, we have a function $weight : V_T \cup E_T \rightarrow R$. `weight(v,w)` represents the weight of edge $(v, w) \in G$, while `weight(edge(v))` is the weight of weighted node v. By that we mean the weight of the newly introduced edge (z, v) or, during the execution of the algorithm, the weight of the MaxWE on the path from z to v via examined nodes.

The rules, presented in the following two subroutines, apply on tree nodes by "pruning" nodes with degree 1 (leaves of the tree or roots with one child) and "shortcutting" nodes of degree 2 (roots or nodes with only one child).

```
procedure prune(v)
   a := weight(v, par(v));
   b := weight(edge(v));
   c := weight(par(v));
   if a = max{a,b,c} then
      exclude(v, par(v)); include(edge(v));
   else if b = max{a,b,c} then
      exclude(edge(v)); include(v, par(v));
   else if c = max{a,b,c} then
      exclude(edge(par(v))); include(min{(v, par(v)), edge(v)});
      edge(par(v)) := max{(v, par(v)), edge(v)};

procedure shortcut(v)
   a := weight(edge(v));
   b:= weight(v, ch(v));
   c:= weight(edge(ch(v)));
   d:= weight(v, par(v));
   e := weight(par(v));
   if a = max{a,b,c} or a = max{a,d,e} then
      exclude(edge(v)); include(min{(v, ch(v)), (v, par(v))});
      (par(v), ch(v)) := max{(v, ch(v)), (v, par(v))};
   else if b := max{a,b,c} then
      exclude(v, ch(v)); prune(v);
   else if d = max{a,d,e} then
      exclude(v, par(v)); prune(v);
   else if c = max{a,b,c} and e = max{a,d,e} then
      exclude(edge(par(v))); exclude(edge(ch(v)));
      include(min{edge(v), (v, ch(v)), (v, par(v))});
```

```
edge(par(v)) := max{edge(v), (v, par(v))};
edge(ch(v)):= max{edge(v), (v, ch(v))};
```

The algorithms are based on the existence of a valid tree-contraction schedule. A *valid tree-contraction schedule* is one which schedules the nodes of the binary tree for pruning and shortcutting in such a way that (i) when a node is operated upon it has degree one or two, and (ii) neighboring nodes are not operated upon simultaneously. For the sequential algorithms only the first condition needs to be satisfied.

We now present two optimal sequential algorithms for the updating MST problem.

Postorder. The simpler algorithm to implement is the following: Visit the nodes of the weighted tree in postorder. A node is processed after all its children have been processed, therefore only the pruning rules are needed. Since each node is processed at most once, we have an $O(n)$ sequential algorithm.

Remove on the fly. Use depth-first-search to visit the nodes of the tree. Every time a node of degree 1 or 2 is encountered, process it using a pruning or a shortcutting rule, respectively. Each node will be visited at most twice (one on the way down the tree and one on the way up the tree), so its running time is $O(n)$.

Let us remark here that more algorithms can be derived by using other valid tree-contraction schedules like the ones reported in [ADKP89, CV88].

3 Incremental MST Algorithms

Using the previously described updating algorithms we can obtain incremental algorithms that compute the minimum spanning tree of a graph G in time $\sum_{1\leq i\leq n} i = O(n^2)$. These algorithms are optimal for dense graphs, i.e. for graphs having $\Theta(n^2)$ edges.

In the remaining part of this section we give the MST algorithm. In brief, the algorithm starts from the empty graph G_0 and computes the MST of the given graph $G_n = G$ in n steps by augmenting in each step G_{i-1} to G_i. Each augmentation introduces a new vertex v_i along with the edges that connect it to vertices already in G_{i-1}, and then calls the vertex updating routine.

Algorithm Incr-MST

Let $v_1, \ldots, v_n$ be a numbering of the vertices.
$G_0 = (V_0, E_0) \leftarrow (\emptyset, \emptyset)$;
$MST(G_0) \leftarrow \emptyset$;
for $i \leftarrow 1$ to n **do**
    Compute $MST(G_i)$ where $G_i = (V_{i-1} \cup v_i, E_{i-1} \cup (v_i, x))$
        and $x \in V_{i-1}$

4 Conclusions

In this paper we have presented an efficient way of designing sequential algorithms from existing parallel ones. Our approach requires that the scheduling and computing parts of the parallel algorithm are well defined and can be separated. This is the case of algorithms that use a valid tree-contraction schedule. We have shown that in this case, optimal, simple algorithms can be derived.

Our work shows that the design of sequential algorithms can be enriched form the research in parallel algorithms. Usually the research has followed the opposite direction.

References

[ADKP89] K. Abrahamson, N. Dadoun, D.G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10:287–302, 1989.

[AV84] M. Atallah and U. Vishkin. Finding Euler tours in parallel. *Journal of Computer and System Sciences*, 29:330–337, 1984.

[CH78] F. Chin and D. Houck. Algorithms for updating minimal spanning trees. *Journal of Computer and System Sciences*, 16(12):333–344, 1978.

[CLC82] F.Y. Chin, J. Lam, and I-N. Chen. Efficient parallel algorithms for some graph problems. *Communications of ACM*, 25(9):659–665, September 1982.

[CV88] R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algor.*, 3:329–346, 1988.

[CV89] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 81:334–352, 1989.

[GPS87] A. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry-breaking in sparce graphs. In *Proc. 19th Annual ACM Symp. on Th. of Comp.*, pages 315–324, 1987.

[JM88] H. Jung and K. Mehlhorn. Parallel algorithms for computing maximal independent sets in trees and for updating minimum spanning trees. *Information Processing Letters*, 27(5):227–236, April, 28 1988.

[JM91] D.B. Johnson and P. Metaxas. Connected components in $O(\log^{3/2} n)$ parallel time for the CREW PRAM. In *Proc. of 32nd IEEE Symposium on the Foundations of Computer Science (FOCS'91)*, October 1991.

[JM92a] D.B. Johnson and P. Metaxas. Optimal algorithms for the vertex updating problem of a minimum spanning tree. In *Proc. of the 6th Intl Parallel Proccessing Symposium (IPPS '92)*, pages 306–314, March 1992.

[JM92b] D.B. Johnson and P. Metaxas. A parallel algorithm for computing minimum spanning trees. In *Proc. of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'92)*, June 1992.

[Met91] P. Metaxas. *Parallel Algorithms for Graph Problems*. PhD thesis, Dept. of Math. and Computer Science, Dartmouth College, Hanover, NH, July 1991.

[MR86] G.L. Miller and V. Ramachandran. Efficient parallel ear decomposition with applications. Technical report, MSRI, Berkeley, CA, 1986.

[PR86] S. Pawagi and I.V. Ramakrishnan. An $O(\log n)$ algorithm for parallel update of minimum spanning trees. *Infor. Proc. Lett.*, 22(5):223–229, April, 28 1986.

[SP75] P.M. Spira and A. Pan. On finding and updating spanning trees and shortest paths. *SIAM Journal on Computing*, 4(3):375–380, September 1975.

[TV85] R.E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal of Computing*, 14(4):862–874, 1985.

[VD86] P. Varman and K. Doshi. A parallel vertex insertion algorithm for minimum spanning trees. In *13th ICALP, Lect. Not.*, volume 226, pages 424–433, 1986.