Brief Announcement: Efficient Graph Algorithms without Synchronization

Johannes Schneider
Computer Engineering and Networks Laboratory
ETH Zurich
8092 Zurich, Switzerland
jschneid@tik.ee.ethz.ch

Roger Wattenhofer
Computer Engineering and Networks Laboratory
ETH Zurich
8092 Zurich, Switzerland
wattenhofer@tik.ee.ethz.ch

Abstract

We give a graph decomposition technique that creates entirely independent subproblems for graph problems such as coloring and dominating sets that can be solved without synchronization on a distributed memory system. For coloring, evaluation shows a performance gain of a factor 3 to 5 at the price of using more colors.

Categories and Subject Descriptors: E.1 Data structures, F.2.3 Tradeoffs among Complexity Measures
General terms: graphs, algorithms, performance
Key Words: parallel algorithms, concurrent data structures, coloring, dominating sets

1. INTRODUCTION

Synchronization operations needed for shared data access cause a large performance penalty for parallel algorithms. When solving graph problems the task of parallelization boils down to split a graph into subgraphs keeping several goals in mind: First, the decomposition process itself should be efficient and simple. Second, if different threads work on different subgraphs, synchronization among these threads should be simple and cause only little overhead. Third, the solution quality, e.g. approximation ratio, should be as good as possible. For example, for coloring the number of employed colors should be minimized. But even coarsely approximating an optimal solution is NP-hard. Thus, in basically any application one is willing to settle for much more than the minimum number of colors and save on computing time. In particular, in fast changing environments, e.g. all kinds of highly dynamic graphs/networks, it is necessary to compute an approximate solution as quickly as possible, since it might be valid only for a short time span. In such situations one is well willing to trade solution quality for computation time. Even checking the correctness of a coloring requires looking at (the colors of) all neighbors of a node, i.e. run time $\Omega(|E|)$. We match this lower bound without relying on synchronization.

2. RELATED WORK

One way to parallelize operations on graphs is to decompose a graph into subgraphs and let each processor compute a solution on a subgraph. Unfortunately, these subgraphs are usually not independent and thus boundary constraints stay in place, which require slow synchronization primitives such as locks. For instance, [2] proposes to iteratively compute maximal matchings. Any pair u, v of matched nodes in a graph G is merged together to form a new node v' in G'. Then a matching is computed in G'. This process is repeated until only p nodes are left. Each node corresponds to a (collapsed) subgraph and is assigned to a processor/core. Thus, for a simple ring with n nodes $\log(n/p)$ matchings must be computed. Every matched node requires an access to a synchronization primitive, since several processors might try to access the same node. For other techniques such as spectral partitioning see [2] for related work.

Coloring has been studied extensively in a local setting, where each node in the graph corresponds to a processor[4]. The fastest algorithm requires no communication to compute an $O(\Delta^2 \log^2 n)$ coloring if a node knows its neighbors. Computing an $O(\Delta)$ coloring needs $O(\log \log n)$ rounds,. i.e. synchronization steps of all processors. In comparison, our algorithm requires no synchronized rounds at all in its simplest version. In the worst-case it might use up to $O(\Delta \cdot p)$ colors, where p is the number of processors. From a theoretical point of view, using that many colors is not too bad given that coloring is hard problem to approximate and the number of processors is often negligible (i.e. constant) compared to the size of the maximum degree. Still, in many cases heuristics allow to compute colorings that allow for less colors than Δ. Unfortunately, the previously mentioned (constant) time algorithm in general requires colors from the entire set of available colors, e.g. to color a tree might require $\Omega(\Delta^2 \log^2 n)$ colors, though two colors suffice. A more effective sequential heuristic is the following greedy strategy: Color node after node, such that any node gets the smallest color not taken by its neighbor. [1] uses this approach. The graph is partitioned into disjoint clusters (obtained by some algorithm, e.g. [2]). Each node is either a boundary node, i.e. it has a neighbor in another cluster, or a non-boundary node. The algorithm iterates two phases. In the first phase each processor speculatively colors the (uncolored) vertices assigned to it in parallel using the greedy sequential heuristic. The second phase consists of a color conflict-detection (for the boundary nodes).

For minimum dominating sets [3] gives a algorithm in a local setting for an $O(k\Delta^{2/k} \log \Delta)$ approximation using $O(k^2)$ synchronization steps. Alternatively, one might employ [2] to compute a decomposition using synchronization and then greedily compute a $\log \Delta$ approximation. We achieve a $p \log \Delta$ approximation without synchronization.

PODC'10, July 25–28, 2010, Zurich, Switzerland.
ACM 978-1-60558-888-9/10/07.

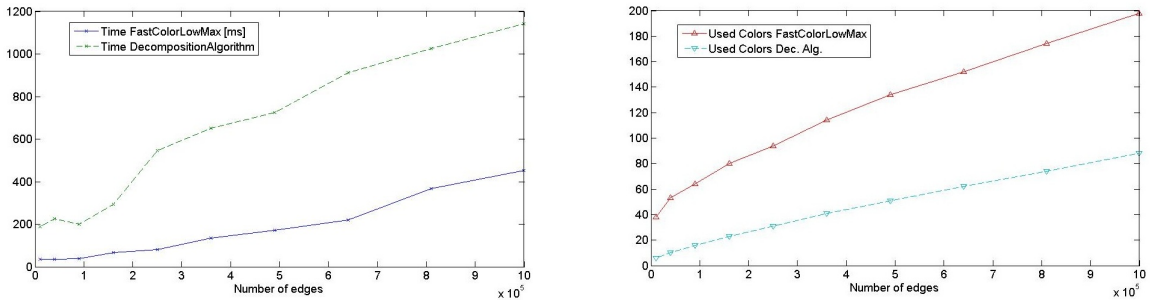

Figure 1: Used colors and time in milliseconds for Algorithm *FastColorLowMax* compared to [1], where random matchings [2] were used for graph decomposition. The number of nodes are fixed to 5000, and the number of edges are varied from 10^4 to 10^6.

3. TECHNIQUE AND APPLICATIONS

We assign a distinct solution space to each processor, e.g. for the coloring problem, processor one can use colors $[0, \Delta]$ to color all nodes assigned to it, processor two can use colors $[\Delta + 1, 2\Delta + 1]$ and so forth. Nodes are assigned in an arbitrary manner (e.g. randomly) to processors. Therefore, we do not have any boundary constraints, since nodes assigned to different processors get different colors. Additionally, since any graph can be colored with $\Delta+1$ colors, every processor can compute a solution, independently of all others. Unfortunately, this algorithm uses up to $(\Delta+1)\cdot p$ colors and even if fewer colors are used they will be distributed in the range $[0, (\Delta+1)\cdot p]$. We overcome this issue by first computing a coloring using up to $(\Delta + 1) \cdot p$ and then only keep the color ranges used by the processors, e.g. consider a system with two processors. If processor 1 has assigned colors $[0, c_{p1}]$, where c_{p1} is the largest assigned color, and processor 2 colors from $[\Delta + 1, \Delta + c_{p2}]$, then we set a node assigned by processor 2 having color c to color $c - (\Delta + 1) + c_{p1}$. In other words, we do not simply use the offset of $i \cdot (\Delta + 1)$ for the colored nodes of processor i but rather compute the offset based on the number of actually used colors of processors j with $j < i$. We refer to this approach by Algorithm *FastColorLowMax*. By slightly modifying our algorithm, we can give a tradeoff between synchronization effort and the number of used colors. Say in Algorithm *FastColorLowMax* a processor p can use only $(\Delta + 1)/r$ colors instead of $\Delta + 1$ for some parameter r. If processor i lacks sufficient colors to color all its assigned nodes, it passes its uncolored nodes to the next processor $i + 1 \bmod p$, which tries to color them. If the total number of colors $(\Delta + 1)/r \cdot p \geq \Delta + 1$, i.e. $p \geq r$, where r is the number of times colors are passed to the next processor, we can be sure that a correct coloring is computed using at most $p/r(\Delta + 1)$ colors.

Though, Algorithm *FastColorLowMax* using $\Delta + 1$ colors per processor requires little coordination among processors, as a drawback more colors are likely to be needed. Apart from the upper bound of $(\Delta + 1) \cdot p$, the number of colors will be in $\Omega(p)$ even for graphs of lower degree than p. This happens because usually we have many more nodes than processors and thus, if we assign nodes randomly to processors, almost surely, every processor gets assigned at least one node and must use one color for it. For illustration of the approximation quality, assume we have a disconnected graph consisting of cliques of size $\Delta+1$. Thus, using a (non-parallel) sequential greedy strategy for the whole graph will result in a coloring with $\Delta + 1$ colors. For the parallel algorithm *FastColorLowMax*, we have that the probability that a processor gets assigned an entire clique is $1/p^{\Delta+1}$. Assume we have $p \ll n$ and $p^{\Delta+1}$ cliques all of small degree, i.e. $n = (\Delta + 1) \cdot p^{\Delta+1}$ or roughly, $\Delta \approx \log n / \log p$. Then, the probability for a processor to be assigned a whole clique becomes $1 - (1 - 1/p^{\Delta+1})^{p^{\Delta+1}} \approx 1 - 1/e \approx 1/2$. Using a Markov bound, the probability that more than $3/4$ of all p processors are not associated with an entire clique is at most $2/3$, thus we expect indeed an approximation factor of at least $3/4 \cdot 2/3 \cdot p = \Omega(p)$. However, if the maximum degree is larger, say $\Delta = n^c - 1$ for some constant $c < 1$, then using a Chernoff bound the probability that a processor gets assigned more than $(1 + \log n/n^{c/2}) \cdot n^c/p$ or less than $(1-\log n/n^{c/2})\cdot n^c/p$ nodes is smaller than $1-1/n^{\log n}$, i.e. roughly $(1 - 1/n^{\log n})^p > 1 - 1/n^{\log n-1}$ for all processors since by assumption $p < n$. Thus, most likely the total number of used colors is only $(1 + \log n/n^{c/2}) \cdot n^c/p \cdot p = (1 + \log n/n^{c/2}) \cdot n^c$ compared to n^c by an optimal algorithm, yielding an approximation ratio converging to 1 as n increases, i.e. $1 + \log n/n^{c/2}$.

Similar thoughts apply for computing a dominating set. After assigning nodes randomly to processors, each processor iteratively picks an (assigned) node with the maximum number of non-dominated neighbors. Since a greedy approach yields an $\log \Delta$ approximation of a minimum dominating set, the overall approximation ratio is at most $p \log \Delta$.

For evaluation we used Java on a system with four quad-core Opteron 8350 processors. Figure 1 shows that for dense graphs our algorithm uses twice as many colors as [1], since for every node its neighbors are distributed relatively evenly among the processor. For sparser graphs it gets worse. But it is always 3 to 5 times faster.

4. REFERENCES

[1] D. Bozdag, A. H. Gebremedhin, F. Manne, E. G. Boman, and Ü. V. Çatalyürek. A framework for scalable greedy coloring on distributed-memory parallel computers. *J. Parallel Distrib. Comp.*, 2008.

[2] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comp.*, 48(1), 1998.

[3] F. Kuhn and R. Wattenhofer. Constant-Time Distributed Dominating Set Approximation. In *J. Distrib. Comp.*, 2005.

[4] J. Schneider and R. Wattenhofer. A New Technique For Distributed Symmetry Breaking. In *Symp. on Principles of Distributed Computing*, 2010.