Three-coloring a directed cycle

Paul Hubbard

EECE 595

Professor David A. Bader

11/12/1999

Introduction

The problem addressed in this work is the implementation of an optimal, parallel, 3-coloring algorithm.

We wish to color a directed graph, with exactly one cycle, with three colors, utilizing a shared-memory multiprocessor machine. Applications of this algorithm are varied, e.g. VLSI design, adaptive mesh generation, and so forth.

The problem was coded as both a serial and SMP algorithm. Results are presented and discussed, with suggestions for further work.

Sequential Algorithm

The sequential program was written first, which turned out to be very useful. To test the JáJá [1] algorithms, we need to have correctly generated graphs, and the data structures and subroutines to create them. Also written are routines to check data sets, count colors, and other debugging aids.

Generating a correct single-cycle graph turned out to be an interesting subproblem in its own right. JáJá uses a data structure called a *successor* or S-graph, where the content of each vertex is the index of its successor. This is delightful later on where we need the colors of our neighbors, but it's difficult to generate. The final algorithm used, a simple O(n) loop, was written by mathematician Chris Hurlburt: to generate a data set, we create a sequentially numbered array, use a Bader subroutine to randomly permute its contents, and then run the Hurlburt algorithm. This works for any size array, and is overall O(n).

To color the graph, a simple O(n) loop is run, alternately coloring vertices with colors zero and one. The last vertex is given color two, to ensure that colors at the end and beginning

of the graph are not the same. Obviously, as JáJá notes, this algorithm is optimal, since it visits each vertex exactly once.

One item of interest is that the loop jumps around the memory space quite a lot, since it is following successor pointers. This yields bad cache utilization, but is inherent in the design of a successor data structure.

Source code for the serial version is integrated into the main program, which can be found in Appendix A.

Analysis of Sequential Implementation

Following the standard conventions, data creation and validation were not timed, on the assumption that we start with a valid data set. Similarly, the ending code that counts colors and does other validation testing was not timed.

The test machine had significant load from other users, yielding rather noisy data. Despite that, the O(n) behavior is quite clear, and indeed this simple serial algorithm is faster than the parallel version until we reach very large data sets.

Graphs are presented in the comparison section, and the numeric data is in Appendix B.

Parallel Algorithm for PRAM

For the definitive explanation and analysis, the reader is referred to JáJá [1]. For this write-up, I will summarize: On an CREW PRAM, 3-coloring can be done in O(logn) time with O(n) storage.

Of course, this is an approximation embodying many simplistic assumptions about the size of the machine, memory bandwidth, etc, etc. Of particular note is the assumption of O(n) processors to obtain the O(log n) runtime – we only have eight processors, so we should obtain something closer to linear runtime growth.

Parallel Algorithm for SMP

There are a couple of noteworthy changes from the PRAM algorithm: The use of counting sort, and the *predecessor* or P-array. JáJá does not detail how to sort or compute the color indices; after much assistance from D. Bader it was determined that both could be accomplished via a modified counting sort. One exists in SIMPLE [2], and this was used in modified form. Via small modifications, we can now sort our structures, and save the final probability density function (PDF) for later use.

One difference from the stock JáJá algorithm is the generation and use of a *predecessor* array, or P-array. This yields direct lookups of neighbors, without link-dragging or other such methods. This is also O(n) in time and storage to generate, O(1) in use, and so does not alter the asymptotic behavior of the code. This array could be removed if one is willing to spend the additional computational time to find the predecessor.

As noted above, the P-array is still useful as a fast lookup method. It's creation is now done via pardo, for O(n/p) time and O(n) storage, but the cache behavior is almost certainly suboptimal.

Algorithm 2.9 [1]is done exactly as given, via parallel do (pardo) loops. Again, this could probably be improved by tuning it for cache effects; this was not investigated.

Algorithm 2.10 [1] is very close to the JáJá version, and differs mostly in housekeeping details. We use the PDF of the sorted list to find the start and end indices of a given color, and then use pardo over all of the instances of that color. How well this works depends on how many of each color there are; this is the load-balancing problem in another guise. Unfortunately, the color distribution depends on how the input graph is structured, thus we have data-dependant load balancing. Improving this would require a new algorithm.

One other difference from the serial algorithm is caused by the implementation of counting sort – in this version, we can only use data sets that are a multiple of the number of CPUs. This is not a large drawback, but would require some care to change.

One performance improvement remaining is the bound on the sort - we sort the least significant eight bits of the color value, because we know that the upper limit is less than 64. We could further tighten the bound, for a linear improvement in runtime; this is noted in the code but has not been done.

Analysis

It should be noted that no tuning has been done to this point; the emphasis was on clarity and correctness. We fully expect that such tuning would yield large improvements in runtime and memory consumption.

At present, the runtime is dominated by the O(n) sorting time, followed by the O(n/p) terms. To achieve the O(logn) runtime of JáJá, we need more processors - O(n) of them, actually. Needless to say, few machines have such large numbers of processors in a SMP configuration.

Experimental Results

Test Machine

All parallel code was run on a DEC/Compaq AlphaServer 8400, with eight 21164A/433MHz CPUs, 4GB of memory, and local UWSCSI storage. We are indebted to Tim Wilcox of Sandia National Labs for allowing us to use this machine, and also thank manager Alan Hurd for approving our usage.

Performance Results

Timings were obtained under heavy user load, so these should be interpreted as indicative but very imprecise. Each graph size was run three times, and the standard deviations shown are as computed from the three timings. Times are in seconds.

For input data, randomized graphs were created via the Hurlburt algorithm, of whatever size was being tested.

Serial results

Graph size	Minimum time	Maximum time	Average time	Sigma
65536	0.018500	0.019500	0.019200	0.00040000
1.3107e+05	0.040000	0.042000	0.040700	0.00080000
2.6214e+05	0.051700	0.051700	0.051700	0.0000

5.2429e+05	0.12790	0.12880	0.12820	0.00040000
1.0486e+06	0.52800	1.2573	0.78950	0.28710
2.0972e+06	1.5852	2.8780	2.0200	0.52540
4.1943e+06	4.2879	6.9463	5.2560	1.0387
8.3886e+06	13.702	16.862	15.393	1.1253

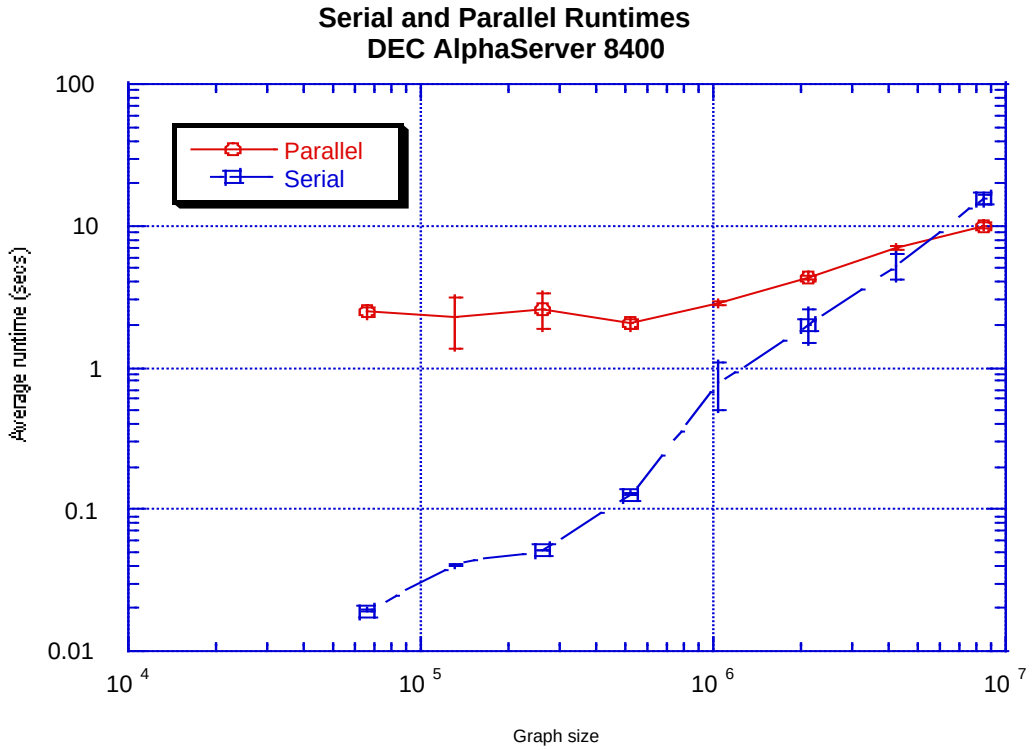
From these, we can see the generally linear growth and small overhead of this algorithm.

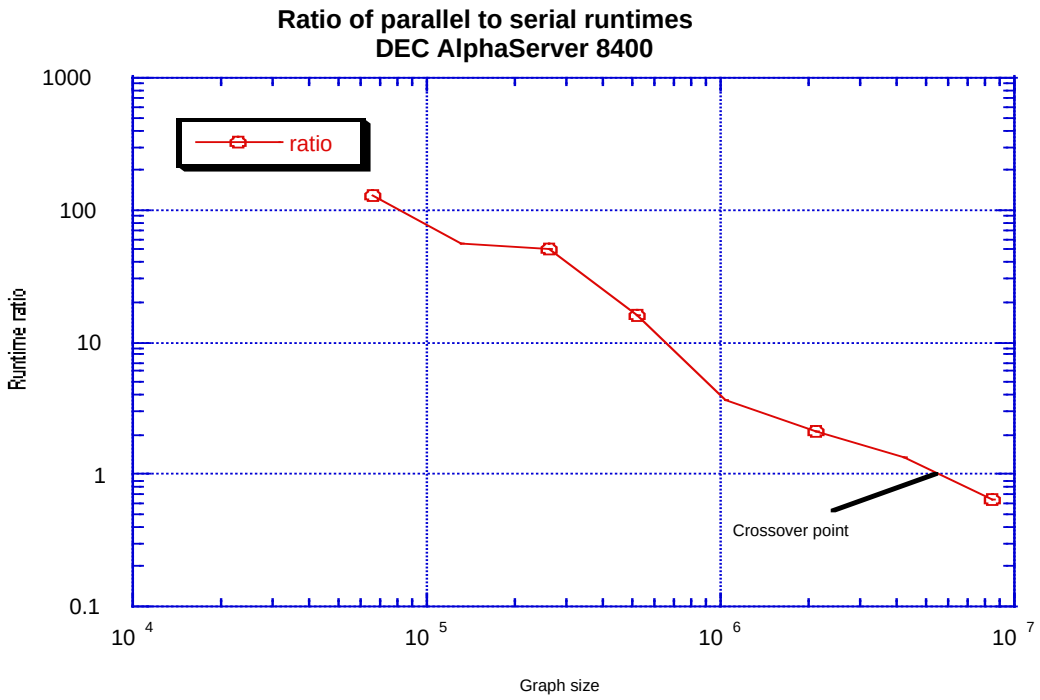
Parallel results

This table is the same as the serial table, with the additional column showing the ratio of serial to parallel average runtimes for that graph size. Clearly, when the ratio drops below unity, the parallel version is faster.

Graph size	Min time	Max time	Avg. time	Sigma	S/P ratio
65536	2.1854	2.7306	2.4776	0.19420	129.08
1.3107e+05	0.81650	3.0625	2.2400	0.87520	55.081
2.6214e+05	1.8887	3.7931	2.6029	0.73370	50.319
5.2429e+05	1.8638	2.3338	2.0590	0.17320	16.063
1.0486e+06	2.7218	3.0283	2.8578	0.11040	3.6198
2.0972e+06	4.0556	4.5423	4.3026	0.17210	2.1300
4.1943e+06	6.7872	7.1005	6.9802	0.11930	1.3280
8.3886e+06	9.5114	10.754	9.9585	0.48870	0.64700

Plots of Results





Discussion of results

The results hew closely to their expected values – the serial version has low overhead and steady growth, and the parallel version has substantial overhead but better growth characteristics. For the load present when these results were obtained, the parallel version begins to win around a size of 7x10^6.

Although the data is noisy due to the collateral user load, the ratio plot shows us that the parallel version is clearly gaining as the graph size increases, which validates our expectations – if the graph is “large enough,” the parallel version is faster.

To a first approximation, both versions are linear in growth, but the parallel version has a smaller slope and a larger intercept – namely, more overhead and better scalability. However, it would require some effort to produce a faster version more suited for smaller data sets.

References

1. J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992

2. D. A. Bader and J. JáJá, "*SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors (SMPs),*" Journal of Parallel and Distributed Computing, 58(1):92-108, 1999.

Appendix A: Source Code

Appendix B: Numerical Results