Journal of Computer Science and Information Technology
June 2014, Vol. 2, No. 2, pp. 09-20
ISSN: 2334-2366 (Print), 2334-2374 (Online)
Copyright © The Author(s). 2014. All Rights Reserved.
Published by American Research Institute for Policy Development

Three Versions of Clique Search Parallelization

Bogdan Zavalnij[1]

Abstract

In our paper we present three parallel versions for maximum clique finding algorithms. The parallelization algorithm by quasi coloring was proposed by S. Szabo. In our paper we present the actual implementation of the algorithm and actual results measured on a large scale supercomputer up to 512 cores. We also implemented the proposed tuning of the algorithm and present here the measurements and results. Apart for the implementation we propose another method of tuning the original algorithm, which is based on Las Vegas randomization method. In our paper we compare the measured results of the three versions of the algorithm.

Keywords: maximum clique, parallelization, quasi coloring, Las Vegas method

1. Introduction

Because the problem of maximum clique search proved to be useful in different applied problems different methods were introduced to propose efficient algorithm to this problem. See (Hasselberg 1993), (Pardalos 1998), (Bomze 1999), (Kumlander 2006). Also, because the problem itself is an NP-hard problem some authors proposed parallel algorithms to speed up the computations (Pardalos 1998), (Thimm 2006), (Eblen 2010) (Depolli 2013). In our paper we try to explore the possibilities of clique search parallelization based on S. Szabo's proposition in this paper "Parallel algorithms for finding cliques in a graph" (Szabo 2011).

First, we implemented the original proposal for parallel computing using MPI and made extensive measurements of running times and other information relevant. Second, we introduce a modified version of the original proposal, and alike program it, and make measurements.

[1] PhD, Institute of Mathematics and Informatics, University of Pecs, Hungary. H-7624, Pecs, Ifjusag utja 6. E-mail: bogdan@ttk.pte.hu

Third, we introduce a modification based on the idea of Las Vegas randomization, and again alike program it, and make extensive measurements. All measurements took place on the Finish supercomputer Sisu, at the CSC, IT Center for Science. The measurements used 1+4, 16, 64 and 512 cores, based on, and compared to the sequential program written by S. Szabo. Actually the parallelization is not dependent on this k-clique program, it is only a frame system, and any k-clique program can be used instead. (In the third version a minor modification of the program is involved, so to be more precise mostly any k-clique program can be used.) The measurements proved the parallelization method to be successful for some quite hard clique problems.

The problem we solve is the maximum clique problem. To be more specific, the algorithms described in this paper answer the question: "Is there a maximum clique of size k in a given graph?"

2. Definitions

Let *G=(V,E)* be a simple, finite, undirected graph. A *C* induced sub-graph of *G* is called a clique if the nodes of *C* are pairwise connected to each other for all nodes. The size of a clique is the number of nodes in the clique. A maximum clique of a graph *G* is a clique which size is at least as big as the size of any other clique's in the graph. We call the clique number of a graph *G* the size of its maximum clique.

We call a partitioning of the nodes of a given graph *G* a coloring, if there is no two nodes in any partition such that those two nodes would be connected by an edge. In other words the partitions, or the so called color classes are independent sets of the given graph. The smallest number of the color classes to which the graph can be properly partitioned in a legal coloring is called the chromatic number of the graph.

The neighborhood of a given node *a*, denoted by *N(a)*, is the set of nodes that are connected to *a*. The neighborhood of two nodes *a* and *b*, denoted by *N(a,b)*, are the set of nodes that are connected to both *a* and *b*.

Proposition 1: the chromatic number of a graph is an upper limit to its clique number. **Proof:** as the nodes of any clique are pairwise connected they must be in different color classes in any legal coloring.

Proposition 2: the number of color classes of a given graph by any coloring is an upper limit to its clique number. **Proof:** from Proposition 1. and the definition of the chromatic number.

3. Quasi Coloring

By proposition from S. Szabo we introduce the notion of quasi coloring. This would mean a partitioning of the nodes of a graph which do not fulfill the requirements of a proper coloring. We call these partitions the quasi color classes. That would mean, that there are still some edges inside of a quasi color class. We will call these edges "disturbing", because they prohibit us to make a proper upper limit for the maximum clique problem. We will try to remove those edges, and by removing all of them get a real coloring, which will provide us an upper limit. Because the "removal" will be a hard work, the proposed algorithm suggests us to construct a quasi coloring with as few disturbing edges, as possible.

As the question we should answer is whether there is a clique of size *k* in the graph, we will construct a quasi coloring of *k*-1 partitions. So the algorithm first divides the nodes into *k*-1 partitions, then if the number of the disturbing edges can be lowered by removing a particular node from a partition and placing it in an other one it moves that node. This preconditioning algorithm runs till it finds such a node. This is a greedy algorithm, and obviously will find a sub-optimal solution only.

4. The Original Algorithm

As we search cliques of size k and partitioned the graph into *k*-1 partitions that means that any clique of size *k* must have at least two nodes from the same partition. As all nodes of a clique are connected, thus these two nodes are connected as well. The edge between these two nodes is a disturbing edge, as we call the edges between the nodes of a given partition. A clique of size k with a given edge *(a,b)* exists if and only if there is a clique of size *k*-2 in the neighborhood of a and b – *N(a,b)*.

So our strategy is to take all the disturbing edges, construct the sub-graphs spanned by the endpoints of those edges, and search for *k*-2 clique in these spanned sub-graphs. If we find one, than our answer to the question is: Yes, there is; if we do not find any, than the answer will be: No, there is none.

The property above can be examined in another way. If we examine the neighborhood of the endpoints of a disturbing edge and find no *k*-2 clique, that means that there can be no *k* clique in the graph which includes this edge. If so, we can freely remove this edge from the graph without changing the answer to the original question. By examining all the disturbing edges and finding no *k* clique including those edges, and by the end removing all these disturbing edges, we will get a proper coloring with *k*-1 colors, which concludes that there is no k clique in the remaining graph.

The parallelization of the proposed scheme is strait forward. We construct the spanned sub-graphs of neighborhoods of the endpoints of the disturbing edges and solve these problems independently on different processors using the same *k*-clique program by S. Szabo with which the parallel running times are compared. We use a producer-consumer scheduling (Dijkstra 1968, pp. 31-34.) or in other words the PO Box scheduling, which means that the problems are constructed by a master and the slaves are asking for a new problem from him, and solving them one by one. We hope for more even distribution by this method, as the sub-problems can have huge differences in run times, and the PO Box scheduler will give the slaves more smaller problem to solve and possibly only one for those which get a problem of huge running time.

The results for the running times of the program are indicated in the table by "nopt" (non-optimal).

5. Az Optimized Modification

If we forget about parallelization for a while we can optimize the above described algorithm. Sequentially going along the disturbing edges we can make the deletion of them not only at the end, but even alongside the running of the sequential algorithm. By examining any disturbing edge and finding no *k*-2 clique in the spanned sub-graph of the neighborhood of its endpoints, that means we can exclude this edge from the problems lying ahead. In other words we can delete this disturbing edge right away, and construct the further sub-problems based on the reduced graph.

This will give us hopefully problems in reduced complexity, in front of the queue will stand the harder problems, and at the end the quite easy ones.

The interesting property of this serialization is that the resulting problems are not dependent on this given sequence of the disturbing edges. That is because on one hand if any sub-question would return a “Yes” answer we would not be interested in the other problem's solution, as this is also the global answer to the original question. On the other hand if a sub-question would lead to a “No” answer, that would mean that the disturbing edge of this sub-problem can be deleted from all the other sub-problem, and this deletion can be made at any time, even in the very beginning. So we can solve the problems in other sequence than the sequence of problems that corresponds to the deletion of the edges. This means, that the problems can be solved in parallel meaner as well, as sub-problems are totally independent.

The results for the running times of this version of the program are indicated in the table by "opt" (optimal).

6. Las Vegas Parallelization

A version of the Monte Carlo random method was proposed by Laszlo Babai (Babai 1979), and named as Las Vegas Method. Based on the run-time differences of algorithm appliances the parallelization mostly for discrete optimization problems based on the Las Vegas randomization was proposed and researched by several authors (Luby 1993), (Luby 1994), (Alt 1996), (Truchet 2012).

A randomized algorithm of Las Vegas type has two properties:

1. If for a given problem instance the algorithm A terminates returning a solution s, s is guaranteed to be a correct solution of the problem.
2. For any given problem instance the run-time of A applied to the problem is a random variable.

As we can easily see, the sub-problems of the first method correspond tho this definition, as we always get the right answer to the question of the *k*-2 clique existence, but we cannot determine the running times.

We can say even more: by experience the running times of the sub-problems vary greatly, usually they differ in quite a few orders of magnitude. The modified optimized algorithm sometimes can help with this, as it reduces the problems in the sequence of the disturbing edge removal.

But the measurements show us clearly, that in some cases it does not help, as in the case when the hardest sub-problems are in the beginning of the edge removal sequence.

We propose an alternative method for the edge removal based on the actual hardness of the problems. For this we need to slightly modify the *k*-clique searching program which lies in the core of our parallelization.

First, we start the parallel program the same way as it is described in the first method: no edge removal applied. But when a sub-problem is solved we immediately know, that that very edge cannot be a part of any *k*-2 clique, and we could have removed it from all the problems yet unsolved. We will do it in two ways. First, any problem asked after this point will be constructed without this edge. Second, the already running *k*-clique searching programs are notified about this edge and they delete it from their graph representation. It seems like changing tiers on a car while driving on a highway, but if the algorithm has a strict data structure (e.g. the algorithm does not reorder the nodes and we can point to a certain edge) where the edges are stored than it is usually safe to remove that edge while the algorithm is running. Quite a few algorithms are constructed that way, and the one we use is one of them.

Let us now examine the consequences of this method. This way we do not force some sub-problems to be smaller in the preconditioning part of the algorithm, but rather start them without edge deletions. The problems will be usually harder to solve, but still the solution time will vary in orders of magnitude. The easier problems will be solved fast, and the information gained from this solutions can be used to make the other problems easier in running time. So this way we hope to make the edge elimination in that order, which is more close to an optimal one, where the harder problems have more edges deleted and so get more help.

The results for the running times of this version of the program is indicated in the table by "lv" (Las Vegas).

7. Results

The attached table shows running results from different types of clique search problems. The first part consists of random graphs with different edge density.

The second part is taken from the Second DIMACS Implementation Challenge (DIMACS), and the third part contains some hard problems: monotonic matrices (Stein 1994,, p 95.) and deletion codes problems from Neil Sloane site (Sloan) based on research of (Bogdanova 2001). We included only some of the DIMACS graphs, because parallelization is only interesting for those problems, where the base problem is long running enough, at least a minute, but rather hours. We certainly excluded all problems with sub second running time. The start up time of the parallel environment, especially for hundred of processors, and the first communications are measurable in seconds. So we clearly are interested in problems which are very hard, and especially in those, which would be unfeasible for many algorithms.

The sequential program was executed on the same supercomputer – for problems comparing running times on different computers see the paper by P. Prosser (Prosser). The parallel program run with 5, 16 64 and 512 processors (5 means 1+4, as there is a master thread and 4 slaves who do the actual computation, so the results should be better to compare.) The time limits for the sequential computation and the 64 and 512 process runs were 12 hours, for the 5 and 16 processors 30 minutes. For the extremely hard problems we make measurements with time limit of 3 days. All the problems were stated so to find a clique one bigger than the actual known clique size, so all the instances answered “No” to this question. We do this as the finding of the maximum clique depends on sheer luck, but the proof of the fact that there is no bigger clique (asking the question for clique size +1) does not depend on any lucky configuration, as the search tree of the problem will be explored fully. Obviously we run test on clique size as well, to check the rightness of the program. All such test completed with a right answer and found a clique. As the running times of those instances of less importance we did not included them in the result's table.

As it is demonstrated the table most of the problems were divided to reasonable number of sub-problems. Two much of them result in degradation, as communication time and the preconditioning time of the sub-problems will enormously rise the time of the whole computation.

As an example for this see the p_hat1500-1 graph which suffers from this extremely and slowdown with 4 slaves can be observed.

On the other hand most of the problems achieved good speedup from more and more processors which led to many hard problems to be solved in reasonable amount of time, and some very hard problems to be solved in feasible time limit. We are extremely happy with this little result, as the framework of the parallelization through which we achieved this results is very simple.

Obviously we can observe some caps to speedup, as the most hard subproblem time limits the whole running time: the whole computation cannot be done in shorter time as the longest subprocess. Good example for this are the monoton-8 and monoton-9 problems for 64 and 512 processors by the non-optimal and optimal version of the algorithm.

Of course the most interesting result came from the Las Vegas algorithm. For many graphs (mostly random ones) the result of it lies between the non-optimal and optimal versions, mostly near to the optimal. This is a good result, as it indicates, that this algorithm should not be too badly different from an optimal one in any case – although extreme examples may occur. In none of the cases it is worse than non-optimal, which means that we are not making the problem worse by this approach. And there are some cases, when the Las Vegas approach is far better than the optimal version. Even more important for us is that this improvement is achieved on extremely hard problem instances. It seems that this approach starts the simplification of the sub-problems more conservatively as the optimal version, but if the program runs for a long time it helps more and more. In the end achieving more help in those problems, which prove to be really hard.

By taking a look at the detailed run outputs (not included in this paper) we can see the clockwork of the algorithms through the actual running times of the sub-problems. The hard instances tend to have extremely different running times of sub-problems. The original two versions of the algorithm consume many easy problems in the beginning, and leave a few hard ones to the end. The problems become exponentially harder (figuratively speaking). If we compare this to the Las Vegas version, we find the same configuration, but the tail of the long running algorithms rise more conservatively. The degradation of the algorithm on harder and harder problems is much slower than in the original version.

	N	%	clique	parts	seq	5-nopt	5-opt	5-lv	16-nopt	16-opt	16-lv	64-nopt	64-opt	64-lv	512-nopt	512-opt	512-lv
rand 200 p=0.9	200	90	40	152	623	376	109	126	123	33	66	49	27	39	38	27	39
rand 300 p=0.7	300	70	21	688	17	53	52	52	15	15	15	4	4	4	1	1	1
rand 300 p=0.8	300	80	29	540	898	466	231	242	135	64	71	45	16	23	23	15	23
rand 300 p=0.9	300	90	47	341	*	*	*	*	*	*	*	*	*	*	*	*	22k
rand 500 p=0.5	500	50	13	2780	20	167	158	159	46	44	45	11	11	11	2	2	2
rand 500 p=0.6	500	60	17	2478	67	431	420	424	119	116	118	29	28	29	4	4	4
rand 500 p=0.7	500	70	22	2231	3453	*	1401	1444	584	387	407	142	93	99	25	14	19
rand 500 p=0.8	500	80	32	1664	*	*	*	*	*	*	*	*	18k	21k	14k	6595	4189
rand 800 p=0.3	800	30	10	5709	52	45	41	41	12	11	12	3	3	3	1	1	1
rand 800 p=0.4	800	40	11	7466	68	439	398	402	121	110	113	29	26	27	4	4	4
rand 800 p=0.5	800	50	14	7296	147	*	*	*	520	488	501	125	117	120	16	15	16
rand 800 p=0.6	800	60	19	6345	2658	*	*	*	*	*	*	356	326	335	47	42	44
rand 800 p=0.7	800	70	25	5587	*	*	*	*	*	*	*	16k	7154	7900	2391	961	1143
rand 900 p=0.3	900	30	9	8614	74	94	84	85	26	23	24	6	6	6	1	1	1
rand 900 p=0.4	900	40	12	8694	97	732	669	676	202	185	190	49	44	46	7	6	6
rand 900 p=0.5	900	50	15	8729	244	*	*	*	905	850	873	217	204	210	28	27	27
rand 900 p=0.6	900	60	19	8215	7109	*	*	*	*	*	*	705	620	643	92	80	85
rand 1000 p=0.2	1000	20	8	7550	77	8	8	8	2	2	2	1	1	1	1	1	1
rand 1000 p=0.3	1000	30	9	10986	101	174	153	155	48	42	44	12	10	11	2	2	2
rand 1000 p=0.4	1000	40	12	10918	136	1268	1156	1168	350	319	329	84	76	79	11	10	10
rand 1000 p=0.5	1000	50	15	10955	447	*	*	*	*	*	*	368	345	355	48	45	46
rand 1000 p=0.6	1000	60	20	9823	15k	*	*	*	*	*	*	1236	1064	1100	158	135	142
brock800_3	800	65	25	4888	7302	*	*	*	*	*	*	472	413	425	64	55	59
brock800_4	800	65	26	4592	5621	*	*	*	*	1570	1604	418	377	387	56	50	53
latin_square_10	900	76	90	380	4902	*	1423	1504	531	403	430	150	105	128	82	62	83
keller5	776	75	27	420	4531	*	*	*	986	672	686	318	173	228	138	137	138
MANN_a45	1035	99	345	45	3666	1340	719	1051	402	205	388	183	140	174	183	140	183
sanr200_0.9	200	90	42	128	387	181	60	68	61	19	31	38	17	44	38	17	38
sanr400_0.7	400	7	21	1408	398	386	316	322	107	88	90	27	21	22	5	3	4

		0																		
p_hat1000-1	1000	24	10	7308	79	102	92	93	29	26	27	7	6	7	1	1	1
p_hat1500-1	1500	25	12	14918	278	894	814	824	247	225	232	60	54	56	9	8	8
p_hat500-2	500	50	36	484	29	101	99	100	29	28	29	8	8	8	3	3	3
p_hat700-2	700	50	45	826	935	1015	560	577	358	159	178	164	42	78	121	37	101
p_hat300-3	300	74	36	297	242	91	52	56	29	15	21	19	10	18	19	9	19
p_hat500-3	500	75	50	657	*	*	*	*	*	*	*	*	11k	7165	41k	11k	5300
monoton-7	343	79	19	313	7	76	74	74	23	21	22	8	6	6	4	2	4
monoton-8	512	82	23	590	2347	*	1282	1292	959	408	385	475	409	195	405	409	243
monoton-9	729	84	28	932	*	-	-	-	-	-	-	150k	150k	44k	150k	150k	31k
deletion-9	512	93	52	375	*	-	-	-	-	-	-	-	-	-	*	*	255k

* denotes run time over time limits

8. Some Notes

It is quite obvious, that given the order of the edge removal in the Las Vegas run we could had make the sequence of the edge removal in the optimal version exactly this way. In this case, as problems can be paired up, we get each sub-problem easier, or at least not harder, than in the Las Vegas method. Which means shorter (or at least not longer) running time. Alas for this sequence we should first run the Las Vegas version of the problem. But it can mean, that with some clever measurements of sub-problem hardness we can construct such a sequence for the removal of the disturbing edges, that will result in faster running time, or even for much faster as measured in this paper. This question remains open for now.

Also remains for the future research to use the quasi coloring for the clique search, as this quasi coloring gets closer and closer to real coloring during the algorithm, and this property can perhaps help a lot to the programs used in the core of the method.

We used in the last, Las Vegas method the property of the *k*-clique search program that the graph can be reduced during the search. For programs that cannot be treated this way (Patric Ostergard's cliquer is clearly one example) the method still can be used. In this cases we can restart the clique search after some reduction from the beginning, and thus gain advantage from the altered problem. This approach is similar to some SAT solvers, which "learn" and restart to achieve faster solution.

Acknowledgements

Author would like to thank the HPC Europe grant for the fruitful visit to Helsinki, to the Finish Computer Science Center which hosts the supercomputer Sisu on which the computations was performed, and student Annamaria Kis, who wrote the base program for her BSc thesis that was altered for the purposes of this paper.

This research was realized in the frames of TÁMOP 4.2.4. A/2-11-1-2012-0001 „National Excellence Program – Elaborating and operating an inland student and researcher personal support system" The project was subsidized by the European Union and co-financed by the European Social Fund.

References

Alt, H., Guibas, L., Mehlhorn, H., Karp, R. and Wigderson, A. (1996). A Method for Obtaining Randomized Algorithms with Small Tail Probabilities. Algorithmica, October/November 1996, Volume 16, Issue 4-5, pp 543-547.

Babai, L. (1979). Monte-Carlo algorithms in graph isomorphism testing. Université de Montréal Technical Report, DMS, Citeseer. http://people.cs.uchicago.edu/~laci/lasvegas79.pdf

Bogdanova, G.T., Brouwer, A.E., Kapralov, S.N. and Ostergard, P.R.J. (2001). Error-Correcting Codes over an Alphabet of Four Elements. Designs, Codes and Cryptography, August 2001, Volume 23, Issue 3, pp 333-342.

Bomze, I.M., Budinich, M., Pardalos, P.M. and Pelillo, M. (1999). The Maximum Clique Problem. In D.-Z. Du and P.M. Pardalos (Eds.) Handbook of Combinatorial Optimization. (pp. 1—74.) Kluwer Academic Publishers.

Depolli, M., Konc, J. Rozman, K. Trobec, R. and Janežič, D. (2013) Exact Parallel Maximum Clique Algorithm for General and Protein Graphs. J. Chem. Inf. Model., 2013, 53 (9), pp 2217–2228 DOI: 10.1021/ci4002525

Dijkstra, E.W.D. (1968). Cooperating sequential processes. http://www.cs.utexas.edu/~EWD/ewd01xx/EWD123.PDF DIMACS. ftp://dimacs.rutgers.edu/pub/challenge/graph/ (May 30, 2014)

Eblen, J.D. (2010) The Maximum Clique Problem: Algorithms, Applications, and Implementations PhD diss., University of Tennessee, 2010. http://trace.tennessee.edu/utk_graddiss/793

Hasselberg, J., Pardalos, P.M. and Vairaktarakis, G. (1993) Test case generators and computational results for the maximum clique problem. Journal of Global Optimization, Winter 1993, Volume 3, Issue 4, pp 463-482.

Kumlander, D. (2006) A Simple and Efficient Algorithm for the Maximum Clique Finding Reusing A Heuristic Vertex Colouring. IADIS International Journal on Computer Science and Information Systems, Vol. 1, No. 2, pp. 32-49.

Luby, M. and Ertel, W. (1994). Optimal Parallelization of Las Vegas Algorithms. In Enjalbert, P at all. (Eds.), Lecture Notes in Computer Science. (pp. 461—474.) Springer Berlin Heidelberg.

Luby, M., Sinclair, A. and Zuckerman, D. (1993) Optimal Speedup of Las Vegas Algorithms. In: Proceedings of the 2nd Israel Symposium on Theory of Computing and Systems, Jerusalem, Israel, June 1993.

Szabo, S. (2011). Parallel algorithms for finding cliques in a graph. Journal of Physics: Conference Series Volume 268, Number 1. 2011 J. Phys.: Conf. Ser. 268 012030 doi:10.1088/1742-6596/268/1/012030

Pardalos, P.M., Rappe, J., Mauricio, M. and Resende, G.C. (1998). An Exact Parallel Algorithm For The Maximum Clique Problem. In High Performance and Software in Nonlinear Optimization. (pp. 279—300.) Kluwer Academic Publishers.

Prosser. P. Exact Algorithms for Maximum Clique – A Computational Study. http://www.dcs.gla.ac.uk/~pat/maxClique/distribution/TR-2012-333.pdf (May 30, 2014)

Stein, S.K. and Szabo, S. (1994) Algebra and Tiling. MAA Carus Monograph 25. Sloan, N. http://neilsloane.com/doc/graphs.html (May 30, 2014)

Thimm, L.R., Kreher, D.L. and Merkey, P. (2006) A parallel implementation for the maximum clique problem. Journal of Combinatorial Mathematics and Combinatorial Computing. http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=041F42A052E680B7ABF69191F2055DAD?doi=10.1.1.117.7013&rep=rep1&type=pdf (May 30, 2014)

Truchet, C., Richoux, F. and Codognet, P. (2012) Prediction of Parallel Speed-ups for Las Vegas Algorithms. http://arxiv.org/pdf/1212.4287v1.pdf