Using SIMD and Parallelization to Compute Graph Colorings

Jeron Lau, Erik Ayavaca-Tirado, Eduardo Flores Bautista, Aleksandr Lukanen
Computer Science Department
Augsburg University
Minneapolis, MN 55454
Scott Kerlin <kerlin@augsburg.edu>

Abstract

Graph coloring is the process of using different colors to represent data on each edge in a graph. Using graph colorings has many beneficial applications in mathematical calculations. Using graph colorings, we are able to calculate Ramsey numbers. The brute force algorithm using graph coloring to generate the graphs needed to detect Ramsey numbers operates at $O(c^{n^2})$, some constant raised to the number of edges squared. It requires the generation of all of the different colorings of complete graphs of size *n*. SIMD assembly instructions, or single instruction multiple data, are assembly instructions that can perform an operation on multiple registers at once; This is designed to take the same amount of time as is does to perform a single operation on a single register. This paper explores a highly parallelized graph coloring algorithm that uses SIMD assembly instructions to reduce the constant in the run time of calculating Ramsey numbers.

SIMD instructions are extremely useful in computing. Because SIMD instructions perform many operations in the time it usually takes to perform a single operation, time to compute is reduced. Software for multimedia processing, such as video editing, makes use of these instructions to make their software run more efficiently. Making the usage of SIMD instructions is quite advantageous.

Reducing the time it takes to calculate Ramsey numbers is not a simple task. Calculating Ramsey numbers is a NP-complete problem. In addition to this, the only known algorithm to calculate Ramsey numbers is a brute force algorithm. This is why using SIMD instructions to help calculate Ramsey numbers is worth examining. Compounding the reduction of computation time at the assembly code level can lead to reductions in overall computation time. NP-complete problems like the Ramsey numbers need complex solutions in order to reduce calculation time.

1 Graph Theory Background

In Graph Theory, a *graph* is a representation of a number of vertices and their edges or lack thereof. On a computer, we can represent the relation between two vertices with one bit (1 if there is an edge, or 0 if there is no edge between the two vertices). A *complete graph* is a graph where all vertices are connected by edges to all of the other vertices.

1.1 Colorings

In a graph, we often wish to attach data to each edge. The convention is to assign a color to each edge of the graph. If there are two colors, we can represent each color as a bit. In our specific case, we will assign each edge either the color red (1) or the color blue (0). The colors are what we use to visually distinguish between the two variants of edges in the graphs. An *edge coloring* is a specific way in which a graph is colored. For two colors, we have 2^n edge colorings, for a graph with n vertices.

Since a general graph and an edge coloring of two colors on a complete graph both use a binary representation they are represented identically. An edge coloring of a complete graph can represent any other graph with the same number of vertices. So for two colors, we don't need a different specialized data structure; We can just use a graph data structure to represent an edge coloring of a complete graph.

1.2 Subgraphs and Cliques

A *subgraph* is a graph that is derived from another graph by removing zero or more vertices from that graph. In a graph, a *clique* is a subgraph that is itself a complete graph.

1.3 Ramsey's Theorem

In a complete graph, all vertices are connected to each other through an edge, and each edge is assigned a color (red or blue). In such a graph, a *monochromatic clique* is a clique in which all of the edges are the same color. The *Ramsey number* R(r, s) is a number that represents the minimum number of vertices of a complete graph that is needed to ensure that there will be a monochromatic clique of at least size r or size s. Ramsey numbers have symmetry, so R(r, s) = R(s, r).

R(r, s)	s=1	2	3	4	5	6	7	8
r=1	1	1	1	1	1	1	1	1
2	x	2	3	4	5	6	7	8
3	x	x	6	9	14	18	23	28
4	x	x	x	18	25	?	?	?
5	x	x	x	x	?	?	?	?

Table 1: Table of currently known Ramsey Numbers [1]. There is symmetry across the diagonal, so those entries are denoted with an x.

Very few Ramsey numbers are actually known due it's runtime complexity. Certain Ramsey numbers are trivial to calculate. For instance, R(1, *x*) = 1, and R(2, *x*) = *x*. The equations for R(3, *x*) and up are unknown, so R(3, 10) is currently unknown [1].

2 Potential Avenues for Single-Core Optimization

For our particular use case we chose to store the graph as a list of edge colorings in a bit string. Each edge is stored as a bit. Since a graph and a complete graph with two colors can always be mapped to each other by a function that is both one-to-one and onto, we will represent our two-color complete graph with a one-color graph. If the bit is 1 there is an edge (red) between the two vertices, and the bit is 0 if there is not an edge (blue). So each bit is 1 for red edges (an edge in the one-color graph), or 0 for blue edges (lack of an edge in the one-color graph).

We chose to pack the edge information into 64-bit registers, which we read as 256-bit registers with SIMD. This allows us to do operations on a register that is 4 times larger, so theoretically our calculations could be four times faster.

2.1 SIMD

SIMD stands for Single Instruction Multiple Data. These are a set of instructions that are provided by specific extensions to X86 Assembly (sse, avx extensions). Architectures other than X86 also have instructions that work similarly to SIMD. For instance, ARM and MIPS have similar *vector instructions*. The original purpose of these instructions was to make multimedia processing faster, but they have also been found to be useful for other types of algorithms.

These instructions take multiple data types and pack them into a large register. For example, there might be 8 signed 16 bit integers packed into a 128 bit register. The SIMD instructions act on these registers so that you may apply the same operation, say addition, onto the 8 integers at the same time, rather than use a for loop. This theoretically optimizes the algorithm by a factor of 8; now the addition inside of the for loop is +reduced to one assembly instruction.

2.2 Leaving Out Unneeded Edge Colorings

Not all edge colorings need to be checked for getting greater values of R(*r*, *s*). If the graph is all red and *r* is the current size of the graph, we know that we have a clique. This is because we know that we will have exactly one monochromatic clique with number of vertices *r*, for a graph with number of vertices *v* that is a monochromatic graph.

Additionally, there are duplicate edge colorings. For a K_4 graph coloring represented by the bit string 100101, we don't need to check 011010, the bitwise complement of the bit string because then we're just swapping red and blue. Additionally, the graph represented by the bit string 100000 is isomorphic to the graphs represented by the bit strings 010000, 001000, 000100, 000010, 000001. This means we only have to check one of these graphs because they have the same number of cliques on the same colors [5].

2.3 Other Optimizations

Besides possibly using SIMD and leaving out specific edge colorings, it's been shown that in the structure of the Ramsey numbers, there are substructures that can be used to find them in less computation time [2]. Additionally, there are ways to split up the algorithm into less computationally expensive algorithms, which was used to discover R(4, 5) [6].

3 Using SIMD To Solve The Clique Problem

Finding a clique of size *K*, in a graph *G* is NP-complete. If *K* is a constant, the algorithm can be brought down to polynomial time, but we can't do that because we're trying to search for the two biggest cliques of each of the two colors in our graph.

3.1 Algorithm

The algorithm in SIMD to determine the Ramsey number based on the *r* and *s* values begins with generating all different graphs possible starting with the minimum of the two *r* and *s* values. For example, with R(3,3), one would start to generate complete graphs starting with 3 vertices, which is the minimum of *r* and *s*.

After identifying the minimum possible Ramsey number, the algorithm iterates through all of the different edge colorings for the current size complete graph. For each coloring, we create a bit string, which is called *gc* (graph coloring).

Next, the algorithm will generate all variations of the cliques of size *r* or *s* within the current graph. All the potential cliques of size of *r* or *s* in the graph will be represented through a list of bit strings. For each iteration, the potential clique is called *pc* (possible clique).

After identifying values for *pc,* a bitwise AND is utilized to compare each *pc* with the current *gc* of the current graph. This output is called *c*, the mask of *pc* onto *gc*. In order to understand the output we will classify them based on three different cases:

Case 1: If $c = pc$, it indicates that the current graph has a red clique on the edges specified by bit string *pc*.

Case 2: If *c* = 0, it indicates that the current graph has a blue clique on the edges specified by bit string *pc*.

Case 3: If neither case 1 or case 2 is true, then there is no red clique of size *r* or blue clique of size *s* in the current graph coloring.

If case 3 does occur, then an additional vertex will be added and the algorithm repeats itself until all edge colorings of the graph pass either case 1 or case 2. Once all edge coloring pass case 1 or case 2, the algorithm returns the number of vertices in the graph.

3.2 Theoretically Testing Of Algorithm With R(3,3)

The algorithm begins with identifying all the possible cliques created using 3 vertices. Next, the different cliques are searched for on the current graph coloring for the graph with 3 vertices. This is done for all variations. Eventually we come to the example below:

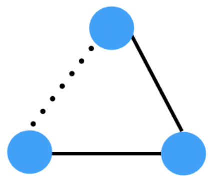

Figure 1: K_3 graph coloring without a 3 vertex clique.

After establishing that a K_3 graph does not satisfy either case 1 or case 2, we move on and add an additional vertex. Now we are looking at graphs of size 4. We then generate the possible cliques with 3 vertices within the graph of 4 vertices. Eventually we come to the example below:

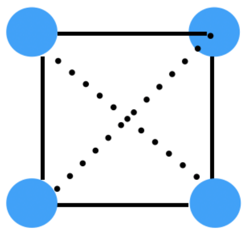

Figure 2: K_4 graph coloring without a 3 vertex clique.

After not meeting either case 1 or case 2, we can conclude R(3,3) does not equal 4, so move on to add another vertex.

Now on to the example of 5 vertices. Most graph colorings for a K_5 have a 3 vertex clique, except for the coloring shown below.

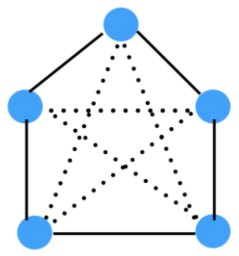

Figure 3: K_5 graph coloring without a 3 vertex clique.

Initially, in the algorithm we would start by giving each vertex a label, *Figure 4* illustrates this in it's vertices which are labeled starting from the the top and going clockwise the vertices are labeled as A, B, C, D, E as shown below:

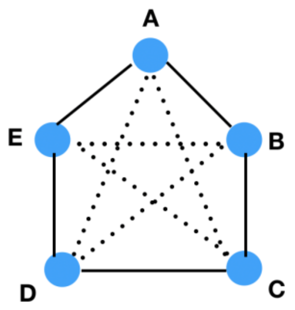

Figure 4: Vertex-labeled K_5 graph coloring without a 3 vertex clique.

Next, we create a table detailing all possible edge combinations; if in the current graph there exists an edge between the vertices then we put a 1, and if there is not an edge then we put a 0.

BA	CA	CB	DA	DB	DC	EA	EB	EC	ED
1	0	1	0	0	1	1	0	0	1

Table 2: An example mapping of a bit string to the edges in a K_5 graph.

The above table contains the bit string representation of the current graph coloring. For this algorithm, we would call this *gc*.

Next, we calculate all of the cliques with *r* vertices; In our case, we look for all of the different cliques with 3 vertices within a 5 vertex complete graph. The number of possible cliques is *n* choose *k*, where *k* is the number of vertices and *n* is the clique size. So in the case of a graph with 5 vertices and finding cliques of size 3, we would have 10 possible combinations due to 5 choose 3 = 10.

ABC	ABE	AED	EDC	DCB	EBC	EBD	ABD	EAC	ACD

Table 3: All of the possible cliques in K_5

Next we decide to represent each possible clique as a bit string. This is what we call our possible cliques or *pc*.

	BA	CA	CB	DA	DB	DC	EA	EB	EC	ED
ABC	1	1	1	0	0	0	0	0	0	0
ABE	1	0	0	0	0	0	1	1	0	0
AED	0	0	0	1	0	0	1	0	0	1
…….	…….	…...	…….	……	……	……	……	……	……	……

Table 4: Bit strings for each possible clique in K_5

Once we have found both our *gc* and *pc* bit strings, we are able to use the bitwise AND operation to compare the bit strings. The output of the bitwise AND is the variable *c*.

We don't get either case 1 or case 2 for any of the possible cliques in this graph coloring, so we have proven that the graph of 5 vertices is not the solution for R(3,3), we move on in the process and add an additional vertex.

The complete graph using 6 vertices pictured below:

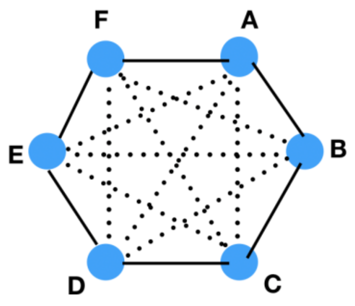

Figure 5: Vertex labeled K_6 graph coloring with multiple 3 vertex cliques.

Using this example and all other colorings of a K_6, we find that all edge colorings of K_6 have a 3 vertex clique.

4 Distributing The Work to Multiple Processors

The algorithm for calculating Ramsey Numbers scales well to multiple processors. In order to actually calculate the numbers, the computer needs to form a proof that there are no graphs that have red and blue cliques with the number of vertices more than *r* or *s* for that value of *v* in equation $R(r, s) = v$.

For calculating Ramsey Numbers, all known algorithms are some variation of a brute force approach. In the brute force approach, all possible colorings of the graph are checked for the sizes of their respective cliques. To split up the work, the graphs can be sent to their own thread and count the cliques. Once this is completed, the main thread can compare it with the current minimum for *r* and *s*, and increment the minimum if needed. For example, $R(3, 3) \neq 5$ can be proved by showing a K_5 graph without either a red clique with 3 vertices or a blue clique with 3 vertices. For $R(3, 3) = 6$, you can prove by calculating that all of the possible combinations of edges in a K_6 have either a red clique of size 3 or a blue clique of size 3.

When the value of $R(r, s)$ gets proved to not be a certain number of vertices, the other threads can be interrupted and the variable for *v* can be incremented and work distributed to the other processors again. The proof can then be recorded in a file, so that the work doesn't have to be repeated on the next run of the code.

4.1 Distributing to Multiple Computers With a Message Passing Interface

Parallel computation on a single computer allows you to perform multiple separate checks at the same time, but this still limits the algorithm to the number of cores on the computer. To improve the speed of finding Ramsey numbers we can simply use more computers to search for cliques. It would be helpful to use some software such as MPI. With MPI all computers are connected to each other on the same network and are able to communicate with one another. Using MPI, we would pass unique graphs to each instance of our algorithm running on each of the networked computers. Alternatively, Dask Distributed could be used to parallelize our computation [4]. The advantage of using Dask Distributed is that we could easily setup a cluster of computers instead of having to create a dedicated cluster of computers running MPI.

We would write a program in python that uses MPI for Python to interface with MPI [3]. For example, the rust program would be running on each of these computers and accept bit strings from the head node where each bit string represents a graph. If any one of the the bit strings does not form a graph with a clique then it would pass a message back to the head node telling the head node to stop all other processes from computing any other bit strings of that length. Then the head node would start passing out the next set of bit strings of one length longer than the previous. If all bit strings form cliques then we have found the Ramsey number for that combination cliques and value for *v*. The python program would then continue on to increment *v* and the get the next combination of possible cliques and repeat this process.

5 Results And Future Work

For our benchmarking, we used Criterion.rs. Criterion.rs is a Rust library typically used for doing benchmarking. It does statistically significant tests for the algorithm, and generates graphs to help visualize the results. For an example that doesn't take to much CPU time, we are using R(2, 3) to test the improvements in the algorithm.

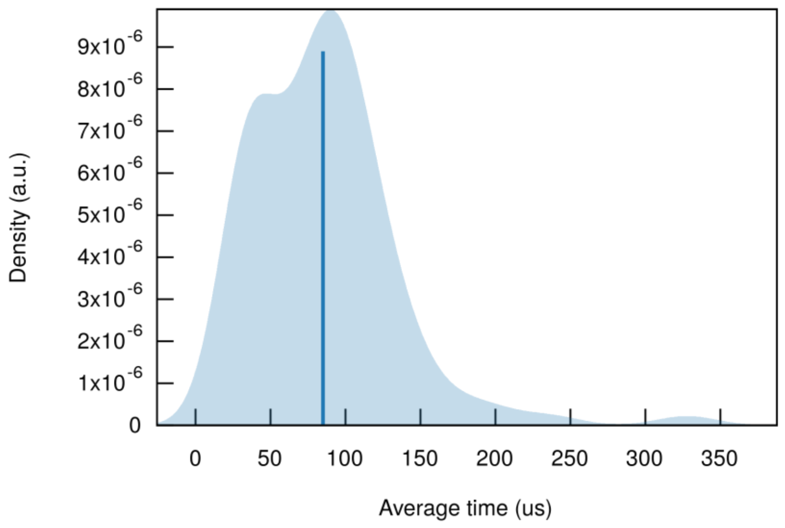

Figure 10: Probability density function generated by Criterion.rs for a SIMD-disabled run on an i3-6100U laptop.

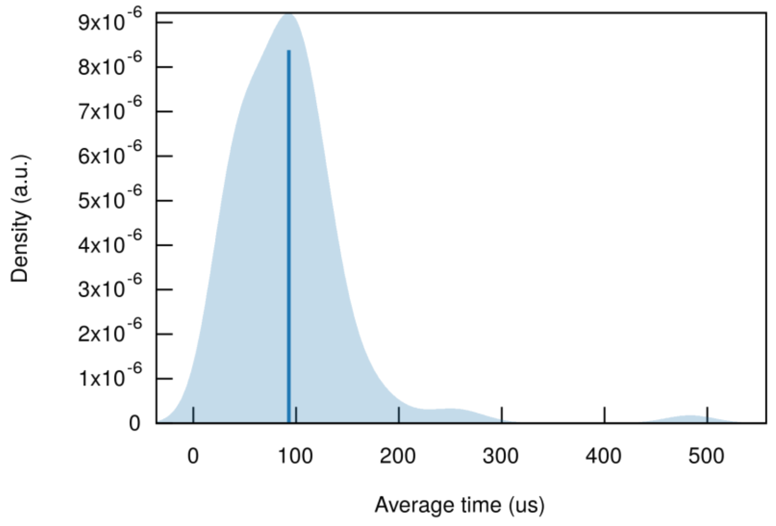

Figure 11: Probability density function generated by Criterion.rs for a SIMD-enabled run on an i3-6100U laptop.

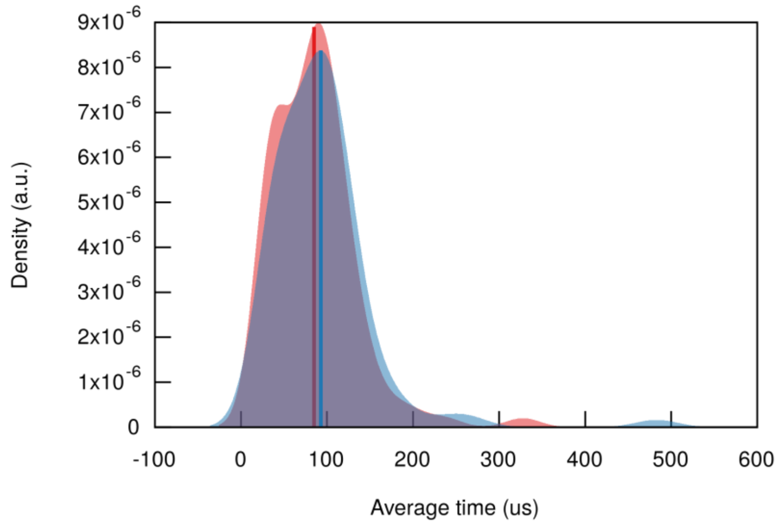

Figure 12: The two probability density functions generated by Criterion.rs for a SIMD-enabled run against a SIMD-disabled run on an i3-6100U laptop.

We found that without SIMD we get 88.774 us, with SIMD we get 93.843 us. This shows that running with SIMD enabled is actually slightly slower. Even though it is slower, the difference is not statistically significant. Since it's slower, this might not be a good case for using SIMD. That being said, the idea of using SIMD lead us to represent graphs as a bit string of edges which is different from the conventional way of storing graphs with pointers or as a matrix. Using bits allowed us to use bitwise AND and XOR to more quickly determine whether or not a graph has a red or blue clique of a specific size.

5.1 Comparing a Slow Example

A slow example with our algorithm is R(2, 8). We're benchmarking against this because R(3, 4) would take about 10 days to calculate on our i3-6100U computer without SIMD, and all other examples take a small amount of time. R(2, *x*) is always *x*, so it would be trivial to optimize, but we're using the example to show how big problems like R(3, 4) would scale with SIMD.

Running without SIMD on a single thread, calculating R(2, 8)=8 took 45 minutes and 42.461 seconds. With SIMD, calculating R(2, 8)=8 took 47 minutes and 33.573s seconds.

5.2 Future Work

Our algorithm could possibly be improved by packing our smaller graph colorings into an 256-bit register instead of having each graph coloring taking up a whole 8 256-bit values. For graph sizes with less than or equal to 128 edges, we can pack 2 graph colorings into one register. For graph sizes with less than or equal to 64 edges, we can pack 4, less than or equal to 32 packs 8, and less than or equal to 16 packs 16. For this last one, we could do 16 tests in the same amount of time it currently takes us to do one test.

Additionally, we could examine the assembly code output for our SIMD algorithm and possibly find something that the Rust compiler put in to link the code together which is not optimal. In this case, we could write and link in assembly rather than calling the intrinsics from Rust. This could allow us to do more optimizations.

Another thing we could try is to see if loading and storing into SIMD registers takes a while, then we could put the AND and XOR operations in the same function called right after each other. The idea is that this would result in less assembly instructions total in the inner loop.

We might also try optimizing generating the possible cliques with SIMD which uses a greater number assembly instructions that take more clock cycles. Our uses of AND and XOR are faster instructions than addition, which is used a lot for generating the possible cliques.

5.3 Conclusion

Even though our use of SIMD didn't show us an improvement in speed, there are other algorithms that are faster with SIMD. Finding that this algorithm doesn't optimize well with SIMD has helped us understand for which algorithms it might. Additionally, there may still be parts of the code for generating Ramsey numbers where SIMD usage would be beneficial.

In

References

[1] The OEIS Foundation Inc. *A212954 - OEIS*, https://oeis.org/A212954, March 2019.
[2] Helsinki University of Technology Laboratory for Theoretical Computer Science. *COMPUTATIONAL METHODS FOR RAMSEY NUMBERS*, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.3.5547&rep=rep1&type=pdf, 2000.
[3] Lisandro Dalcin. *MPI for Python*, https://mpi4py.readthedocs.io/en/stable/, Feb 15, 2019.
[4] Anaconda, Inc. *Dask.distributed*, http://distributed.dask.org/en/latest/, 2016.
[5] Pascal Schweitzer. *Problems of Unknown Complexity*, https://people.mpi-inf.mpg.de/~pascal/docs/thesis_pascal_schweitzer.pdf, April 2012.
[6] Brendan D. McKay. *R(4,5) = 25*, http://users.cecs.anu.edu.au/~bdm/papers/r45.pdf, 1995.