How Many People Does it Take to…: A Parallel Approach to the Party Problem

David Toth
Merrimack College

1 The Party Problem

The party problem is a problem in an area of mathematics known as Ramsey Theory. The R(m, n) instance of the party problem asks what is the fewest number of people that must attend a party to guarantee that at the party, there is a group of m people who all know each other or a group of n people who are all complete strangers or both (so this is the logical or rather than the exclusive or). Thus, the solution to the R(m, m) instance of the party problem indicates the fewest number of people required to be invited to a party to **guarantee** that at the party, there will be a group of m people who all know each other or a group of m people who are all complete strangers. While the party problem has been solved for some small values of m and n, it has yet to be solved for values of m and n that are equal and at least 5 [1]. Bounds on the answers to the problem have been established for a number of values of m and n [1]. For example, it is known that $43 \leq R(5, 5) \leq 49$ [1]. Figure 1 shows the currently known solutions and bounds on solutions to the Party Problem.

n m	3	4	5	6	7	8	9	10	11	12	13	14	15
3	6	9	14	18	23	28	36	40 43	46 51	52 59	59 69	66 78	73 88
4		18	25	35 41	49 61	56 84	73 115	92 149	97 191	128 238	133 291	141 349	153 417
5			43 49	58 87	80 143	101 216	125 316	143 442	159 633	185 848	209 1139	235 1461	265 1878
6				102 165	113 298	130 495	169 780	179 1171	253 1804	262 2566	317 3705	5033	401 6911
7					205 540	216 1031	237 1713	289 2826	405 4553	416 6954	511 10581	15263	22116
8						282 1870	317 3583	6090	10630	16944	817 27490	41525	861 63620
9							565 6588	580 12677	22325	39025	64871	89203	
10								798 23556		81200			1265

Figure 1 - Known Solutions and Bounds on Solutions to the Party Problem [1]

In order to study the Party Problem, people model the situation with complete graphs where vertices represent people. Each edge of the graph is one of two colors, indicating whether the people represented by the vertices the edge connects know each other or do not know each other. For the rest of this lesson, we assume that a blue edge between vertices indicates that the people represented by the vertices know each other and a red edge between vertices indicates that the people represented by the vertices do not know each other. An example of a party with five people is shown in Figure 2.

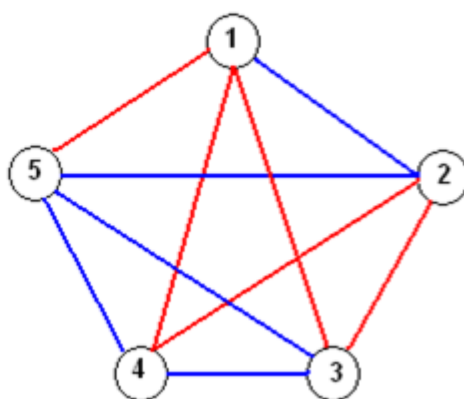

Figure 2 - A Party with Five People

In Figure 2, person 1 knows person 2, but nobody else. Person 2 knows person 1 and person 5, but not person 3 and person 4. Person 3 knows person 4 and person 5, but not person 1 and person 2. Person 4 knows person 3 and person 5, but not person 1 and person 2. Person 5 knows person 3 and person 4, but not person 1 and person 2.

A complete graph on v vertices, denoted K_v, contains $v2-v2$ edges. We encourage the instructor to consult a reference on Graph Theory for a formal proof of this if they are uncomfortable with the following informal explanation of why this is true. In a complete graph on v vertices, each of the v vertices is connected to the other v-1 vertices by an edge, giving us a total of $v * (v-1)$ or $v^2 - v$ edges. However, this counts each edge connecting a pair of vertices twice, and thus we must divide the $v^2 - v$ edges by 2 to get the number of edges in K_v.

Because each one of these edges can be either red or blue, there are two ways to color each edge in K_v. That means there are 2 ^ $(v2-v2)$ different ways to color the edges of a

complete graph. A group of x people who all know each other can be visualized as a graph with x vertices where each edge of that graph is blue and group of x people who do not all know each other can be visualized as a graph with x vertices where each edge of that graph is red. In order to show that $R(x, x) = v$, one must show that both conditions below hold true.

Condition 1: Each of the 2 ^ $(v2-v2)$ different colorings of K_v contains a red K_x subgraph, a blue K_x subgraph, or both.
Condition 2: Each of the different colorings of complete graphs with more than v vertices contains a red K_x subgraph, a blue K_x subgraph, or both.
Condition 3: There is no smaller v such that every colorings of K_v contains a red K_x subgraph, a blue K_x subgraph, or both.

We note that every complete graph with x+1 vertices can be formed by adding a vertex to a graph with x vertices and adding the edges from the new vertex to the original x vertices of the graph. Thus, if every complete graph with x vertices has a red or blue K_5 subgraph, then every complete graph with x+1 vertices must have either a red or blue K_5 subgraph. Therefore, it is sufficient to show that conditions 1 and 3 hold.

We note that some of the different colorings of K_v are isomorphic, and thus there are shortcuts that can be taken to show that every K_v contains a red K_5 subgraph, a blue K_5 subgraph, or both. We encourage the instructor to consult a reference on Graph Theory for an explanation of isomorphism if they are curious, as a student may ask about this concept. However, for this module, we are ignoring the fact that some graphs are isomorphic and just employ a brute force strategy for our algorithm, so the instructor need not worry about isomorphic graphs. It should be clear that the minimum value of R(n, n) is n and if one wanted to prove that R(n, n) = n, one must show that all 2 ^ $(n2-n2)$ different colorings of K_n contains a red or blue K_n. Therefore, viewing n as the input size of the problem, at a minimum, the complexity of our brute force algorithm is 2 ^ $(n2-n2)$ which is clearly exponential. We observe that this is the minimum complexity and for n > 2, R(n, n) > n. In reality, to improve the known bounds for R(5, 5), the brute force algorithm might have to test 2^{903} graphs rather than 2 ^ $(52-52)$ or 2^{10} graphs. As the n in R(n, n) increases, this gets significantly bigger as shown by the known bounds in Figure 1.

The bounds, in turn, cause the number of graphs that need to be tested to increase at an extremely high rate because of the number of vertices in the graphs that must be tested.

In this module, we will write our code to try to tighten the bounds on R(5, 5). Specifically, we will assume that R(5, 5) = 46 and then test every graph with 45 vertices. If any graph with 45 vertices has no red K_5 subgraph and no blue K_5 subgraph, then we will have shown that the lower bound on R(5, 5) is 46, rather than the current lower bound of 43. If every graph with 45 vertices has a red K_5 subgraph or a blue K_5 subgraph, then we will have shown that the upper bound on R(5, 5) is 46, rather than the current upper bound of 49.

2 Assignment #1 – Developing a Sequential Program to Solve the Party Problem

We begin by having students write a sequential program to determine if 46 is an upper or lower bound for R(5, 5), as discussed in the previous section. The program should test every possible graph to see if the graph contains a red K_5 or blue K_5 until it has tested every graph or finds a graph that contains neither a red nor blue K_5. If a graph does not contain a red K_5 and does not contain a blue K_5, the program terminates and says that 46 is a new lower bound for R(5, 5). If the program determines that all of the graphs contains a red or blue K_5 subgraph, then the program should output that 46 is the new upper bound for R(5, 5). We note that students should not expect that the program to terminate in their lifetime because of the number of graphs that need to be tested. To determine if a graph contains a red or blue K_5, the program should test every set of 5 vertices in the 45-vertex graph until it has found a red or blue K_5 subgraph or has tested every set of five vertices without finding a red or blue K_5 subgraph. To ensure the program completes in a reasonable amount of time, only have it test a small number of graphs at first. Later, when you do performance comparisons between the other versions of the program, you can select a number of graphs that makes sense based on your hardware. Our sequential version of the program tested 335,544,320,000 graphs in 220 minutes using one core on our 2.2 GHz quad-core AMD CPU. We chose to test 335,544,320,000 graphs based on a research project we did [2]. In our research project, 335,544,320,000 "was a multiple of the number of cores in every one of the GPUs we used and the algorithm took about an hour to test that many graphs using all 4 CPU cores. By picking a value that took the CPU a long time, we expected to see what, if any, performance gains we could reasonably expect under normal conditions of the

cores having a huge number of graphs to test, which is what we would encounter if we tried to solve the problem with GPUs or CPUs" [2].

2.1 Using a Two-Dimensional Array

One can use a two-dimensional array to represent a graph that is being tested and 5 loops nested one inside another to write a sequential program to test if 46 is an upper or lower bound for R(5, 5). The outermost loop corresponds to the first vertex in the set of vertices, the loop immediately inside the outermost loop corresponds to the second vertex in the set of vertices, and so on. Example code to do this looks like this:

```
const int NUMBER_OF_VERTICES = 45;
const int RED = 0;
const int BLUE = 1;
…
int graph[NUMBER_OF_VERTICES][NUMBER_OF_VERTICES];
bool foundK5 = false;
int firstVertex = 0;
int secondVertex = 0;
int thirdVertex = 0;
int fourthVertex = 0;
int fifthVertex = 0;

// Initialize graph array to the adjacency matrix of the graph to examine here.

while ((foundK5 == false) && (firstVertex < NUMBER_OF_VERTICES - 4))
{
   secondVertex = firstVertex + 1;
   while ((foundK5 == false) && (secondVertex < NUMBER_OF_VERTICES - 3))
   {
      thirdVertex = secondVertex + 1;
      while ((foundK5 == false) && (thirdVertex < NUMBER_OF_VERTICES - 2))
      {
         fourthVertex = thirdVertex + 1;
         while ((foundK5 == false) && (fourthVertex < NUMBER_OF_VERTICES - 1))
         {
            fifthVertex = fourthVertex + 1;
            while ((foundK5 == false) && (fifthVertex < NUMBER_OF_VERTICES))
            {
                if ((graph[firstVertex][secondVertex] == RED) &&
                          (graph [firstVertex][thirdVertex] == RED) &&
                          (graph [firstVertex][fourthVertex] == RED) &&
                          (graph [firstVertex][fifthVertex] == RED) &&
                          (graph [secondVertex][thirdVertex] == RED) &&
                          (graph [secondVertex][fourthVertex] == RED) &&
                          (graph [secondVertex][fifthVertex] == RED) &&
                          (graph [thirdVertex][fourthVertex] == RED) &&
                          (graph [thirdVertex][fifthVertex] == RED) &&
                          (graph [fourthVertex][fifthVertex] == RED))
                {
                   foundK5 = true;
                }
                else
```

```
                        {
                           if ((graph [firstVertex][secondVertex] == BLUE) &&
                                  (graph [firstVertex][thirdVertex] == BLUE) &&
                                  (graph [firstVertex][fourthVertex] == BLUE) &&
                                  (graph [firstVertex][fifthVertex] == BLUE) &&
                                  (graph [secondVertex][thirdVertex] == BLUE) &&
                                  (graph [secondVertex][fourthVertex] == BLUE) &&
                                  (graph [secondVertex][fifthVertex] == BLUE) &&
                                  (graph [thirdVertex][fourthVertex] == BLUE) &&
                                  (graph [thirdVertex][fifthVertex] == BLUE) &&
                                  (graph [fourthVertex][fifthVertex] == BLUE))
                           {
                                  foundK5 = true;
                           }
                        }
                      fifthVertex ++;
                   }
                   fourthVertex ++;
               }
               thirdVertex ++;
           }
           secondVertex ++;
       }
       firstVertex ++;
}
```

We point out that the graph array could have been an array of Boolean values instead of integers. However, we used the integers for ease of understanding as we developed a function to generate graphs.

2.2 Using a One-Dimensional Array

To make it easier to generate the entire set of graphs, we chose to remove the redundant and unnecessary information from our two-dimensional array and flatten it into a one-dimensional array as shown in Figure 3. We point out that a complete graph, by definition does not contain loops, or edges from one vertex to itself. Therefore, we have placed -9 in the long diagonal of the array that runs from the top left to the bottom right. We also point out that the information below that diagonal is a duplicate of the information above the diagonal. Therefore, we can eliminate the information on and below the diagonal, as shown in Figure 3. The locations in the one-dimensional array that make up a subgraph on 5 vertices are not as intuitive to determine, as the locations in the two-dimensional array are. While this makes a great problem for the students to solve if they have lots of spare time, we strongly suggest that in the interest of time, the instructor give the students the solution in class.

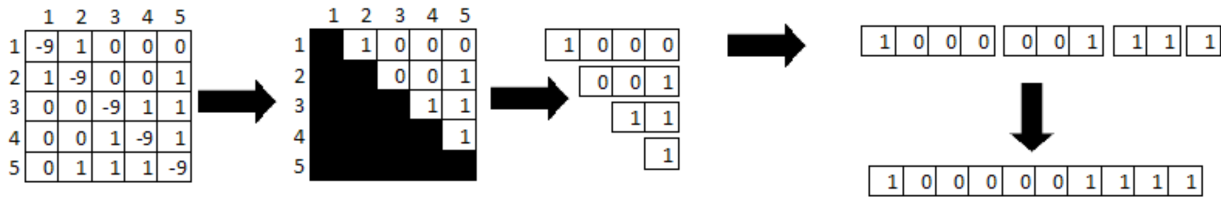

Figure 3 - Converting the Two-Dimensional Array to a One-Dimensional Array

To modify the if/else statement in Section 2.2 to use a one-dimensional array instead of a two-dimensional array, one can replace the individual parts of the if statement as shown:

```
graph[firstVertex][secondVertex] == RED
```

becomes

```
graph[(NUMBER_OF_VERTICES * firstVertex) + secondVertex - (((firstVertex + 1)*(
firstVertex + 2))/ 2))] == RED
```

Students will need to write a function that generates each subsequent graph after the first graph. To generate the next graph, consider each value in the array as a digit in a binary number and add one to the rightmost value in the array, propagating carry values to the left as appropriate. File **Sequential_Ramsey_final.cpp** is an example sequential program.

3 Assignment #2 – Developing a Parallel Program to Solve the Party Problem with OpenMP

To create a parallel version of the sequential program using OpenMP, we can just make some minor modifications to the sequential program. The modifications divide the number of graphs to test by the number of threads that the user instructs the program to use which is specified as a command line argument, and parcel out these sets to a number of threads. We specified the number of threads as a command line argument because we wanted to be able to test our program using different numbers of threads in an automated manner. We note that the student could also determine the number of threads to use when running the program in the code without the argument by using the **omp_get_num_threads()** function call. When we ran our program, we used a number of threads equal to the number of CPU cores in our computer and had each core test $1/n^{th}$ of the graphs, assuming there are n CPU cores. To do this efficiently,

you want the program to create n ranges of consecutive graphs and have each core test the graphs in a range. File **OpenMP_Ramsey_final.cpp** is an example OpenMP program. The OpenMP version of the program launches a number of threads (in the example program, the number of threads is specified as a command line argument). To convert the program from the sequential version to the parallel version using OpenMP, we wrap the bulk of the main program from the sequential version in the **#pragma omp parallel** directive. Within that directive, we also add code to generate the first graph from each of the sets of graphs that the whole set was partitioned into. We note that the **void generateNextGraph(int graph[])** and **bool hasK5(int graph[])** functions from the sequential version of the program are unchanged from the sequential version of the program.

4 Assignment #3 – Developing a Parallel Program to Solve the Party Problem with CUDA

To create a CUDA version of the program, we use a similar strategy to the OpenMP version of the program. Instead of CPU cores doing the work, the GPU cores will do the work. We need to have a file that holds the code for the CUDA kernel. The kernel will have each thread create the first graph it is supposed to test. Then the kernel tests the graph. In the event that a graph with no red or blue K_5 was found, the kernel records this fact and the fact that the nth graph tested was the graph so the graph can be recreated easily after the program finishes. The contents of the kernel file, which just contains one function, is essentially the code from the **main()** function of the OpenMP version of the program. However, we also take the code from the OpenMP version that tests a graph for a K_5 and the code to generate the next graph and move them into the kernel function in the appropriate places so that the kernel can just run one function. This allows the program to just call one kernel function, reducing the complexity of the CUDA portion of the program.

The other source code file contains the code for the CPU to run. This code should be minimal - just enough to set up the initial graph, allocate memory on the GPU, transfer the initial graph to the GPU, invoke the kernel, and transfer the status of whether the program found a graph with no red or blue K_5 in it back to the CPU after the GPU finishes checking the graphs. Files **template_kernel.cu** and **template.cu** form an example CUDA solution.

5 Performance Testing Instructions

To test the performance of the code on your hardware, recompile the code and then run it as is on your hardware. That will provide you with one set of performance data. The OpenMP code can be run using a number of CPU cores specified on the command line so if you have a computer with n cores, you can specify how many to use and see the performance differences based on the number of cores you use. The hardware we used for testing was an upgraded Gateway GT5674 computer with an AMD Phenom 9500 2.2 GHz quad-core CPU, 4 GB of RAM, and a 650 watt power supply. The computer ran the Windows Vista operating system. The OpenMP version of the program took about 220, 114, and 54 minutes using 1, 2, and 4 cores of the processor, respectively. The CUDA version took about 38 minutes on a GeForce 9500 GT, about 9 minutes on a GeForce GT 240, about 8 minutes on a GeForce GTS 450, about 5 minutes on a Quadro FX 5800, and about 2.5 minutes on a GeForce GTX 480.

To get a second set of performance data for your systems, we suggest running the programs with a different set of graphs by modifying the starting values in the

```
int baseGraph[NUM_EDGES];
```

array. By choosing a base graph that does not have a red or blue K_5 subgraph that can be detected almost immediately, you will lengthen the runtime of the program. We note that this is not an easy task (otherwise R(5, 5) likely would be known by now) and suggest filling the array with a random set of ones and zeros.

6 Questions for Students to Answer

These questions are provided for students to answer to demonstrate some things they have learned from this module.

1. In the OpenMP version of the program, if instead of creating n consecutive ranges of graphs and having each CPU core test a range, you have each core test every nth graph, what is the performance impact on the program? Why? **We would expect the program to slow down because each core would have to perform the function that generates the next graph n times instead of one time to find the next graph to test.**
2. One could rewrite the OpenMP code to have only a single shared variable that keeps track of the current graph being tested and have each core simply get the next graph

based on that variable. That would allow each core to run until all the graphs were tested, rather than forcing each core to test the same number of graphs. Are there advantages/disadvantages to that modification? **An advantage is that if one core gets behind the other cores in terms of the graphs to test because of some graphs taking longer to test than others, the other cores could share the graphs the core that's behind hasn't gotten to, decreasing the wall clock time to complete the program. A disadvantage is that we will need to protect the variable that keeps track of the next graph so that we don't hit any race conditions with it. Protecting it may slow down the program a fair amount.**

3. Question about what performance advantage does the OpenMP & sequential code have over the CUDA code? **The sequential and OpenMP code will indicate to the user if graph that contains neither a red K_5 nor a blue K_5 is found faster than the CUDA code because the program outputs such a result immediately if it finds such a graph, while the CUDA version continues until it tests all the graphs in the range and then checks for a graph with no red K_5 and no blue K_5.**
4. In the CUDA version of the code, we purposely allow a race condition to occur. We allow multiple threads to write to the Boolean variable that tracks whether a graph with neither a red K_5 nor a blue K_5 was found. Although this is poor style and dangerous, we can get away with doing that in this situation. Why? **The variable is initialized to false and never written to except to change its value to true. Thus, every thread would be overwriting the location with the same value, changing it from the initial state of false to the new state of true. Therefore, as long as the variable is set to true when the program ends, we can tell that a graph containing neither a red K_5 nor a blue K_5 was found.**

7 Possible Extensions

The instructor could have the students apply what they have learned to other problems that require brute force solutions to find the optimal solution, such as NP-complete problems the traveling salesman problem, graph coloring problems, the knapsack problem, and the bin packing problem. All of these problems have the same basic method to solve them. The key to solving each one with parallelism is to determine how to efficiently partition the search space to

allow multiple cores (CPU or GPU) to work on the problem without having to wait for each other or duplicate effort unnecessarily. Another extension of this problem would be to have students write additional versions of the program that used MPI, a hybrid of MPI and OpenMP, or a hybrid of MPI and CUDA.

8 References

1 S. P. Radziszowski, Small Ramsey Numbers, The Electronic Journal of Combinatorics. DS1.10. (originally published July 3, 1994, last updated August 4, 2009), http://www.combinatorics.org/Surveys/ds1/sur.pdf.

2 Michael V. Bryant and David Toth, A Performance Comparison of a Naive Algorithm to Solve the Party Problem using GPUs, submitted to the Journal of Computational Science Education on March 2, 2012.