Optimization of computation and storage of Ramsey graphs using High Performance Computing Clusters

---

A thesis

presented to

the faculty of the Department of Mathematics

East Tennessee State University

In partial fulfillment

of the requirements for the degree

Master of Science in Applied Computer Science

---

by

Jwalanta D. Shrestha

May 2014

Christopher Wallace, Ph.D., Chair

Martin Barrett, Ph.D.

Selim Kalayci

Keywords: graph theory, Ramsey numbers, parallel, HPC, MPI

ABSTRACT

Optimization of computation and storage of Ramsey graphs using High Performance Computing Clusters

by

Jwalanta D. Shrestha

Insert text of abstract here.

ACKNOWLEDGMENTS

Insert text of acknowledgements here.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 TITLE OF FIRST CHAPTER

Test of Chapter 1.

1.1 Subsection Title

Text of subsection.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Typi non habent claritatem insitam; est usus legentis in iis qui facit eorum claritatem. Investigationes demonstraverunt lectores legere me lius quod ii legunt saepius. Claritas est etiam processus dynamicus, qui sequitur mutationem consuetudium lectorum. Mirum est notare quam littera gothica, quam nunc putamus parum claram, anteposuerit litterarum formas humanitatis per seacula quarta decima et quinta decima. Eodem modo typi, qui nunc nobis videntur parum clari, fiant sollemnes in futurum.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <bitset>
#include <algorithm>
```

```
#include <string.h>
#include "solver.h"
#include "utils.h"

/*
int constraint_sort(Constraint a, Constraint b){
    if (a.sign=='>' && b.sign=='<') return true;
    return (ffsll(a.lhs)>ffsll(b.lhs));
}
*/

Solver::Solver(){

    new_graphs_ptr = &new_graphs;
    old_graphs_ptr = &old_graphs;

    // get MPI information

    // Get the number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &mpi_num_processes);

    // Get my rank among all the processes
    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_this_process);

    // parallel partition not started
    mpi_parallel_start = 0;

}

Solver::~Solver(){

}

void Solver::add_constraint(Constraint c){

    // compute a and b

    // first bit set to 1
    // http://linux.die.net/man/3/ffs
    /*
```

```
    int n = ffsll(c.lhs);

    if (n>0){
        n--;

        // set a to 0-bit for upto the first set bit in lhs
        c.a = 0;     // set all to 0
        c.a = ~c.a;      // set all to 1
        c.a <<= n;   // shift n bits left

        // set b to 1-bit at first set bit in lhs
        c.b = 1ULL<<n;
    }
    */
    constraint.push_back(c);

}

void Solver::clear_constraints(){
    constraint.clear();
}

void Solver::print_constraint(){

    int l_count = 0, g_count = 0;
    std::cout << std::endl;

    for (int i=0;i<constraint.size();i++){
        //std::bitset<64> x(constraint[i].lhs);
        std::cout << "[" << binary_str(constraint[i].lhs,64) << "]" <<
            constraint[i].sign << constraint[i].rhs << std::endl;

        if (constraint[i].sign=='<') l_count++; else g_count++;
    }

    std::cout << "<: " << l_count << " >: " << g_count << std::endl;

}

/*
```

```
int Solver::solve(int vertices, string filename=""){

    int i, count=0;
    int satisfy;

    BIGINT t,max,n,on,jumps=0;

    //sort(constraint.begin(), constraint.end(), constraint_sort);
    //print_constraint();

    // size of graph
    int m=vertices*(vertices-1)/2;

    max = 1ULL<<m;

    //cout << "m: " << m << "max: " << max << endl;

    ofstream ofile;
    if (filename!=""){
        ofile.open(filename.c_str());
    }

    //#pragma omp parallel for shared(count)
    for (n=0;n<max;n++){

        if (check(n)) {

            //#pragma omp atomic
            count++;

            if (filename != ""){
                ofile << get_g6(n, vertices) << endl;
            }

        }
        else{
            // jump


        }
```

```
    }

    if (filename!=""){
        ofile.close();
    }

    //cout << "Jumps: " << jumps << endl;

    return count++;

}
*/

void Solver::solve(int vertices){

    BIGINT one=1,n,i;
    int shift;

    int max;

    max = 1<<(vertices-1);

    //std::cout << "Max: " << max << std::endl;

    //std::cout << constraint.size() << " constraints, ";

    shift = (vertices-1)*(vertices-2)/2;

    // loop through all old constraints
    int count=0;
    for (std::set<BIGINT>::iterator it=old_graphs_ptr->begin(); it!=
        old_graphs_ptr->end(); ++it){
        for (i=0;i<max;++i){

            // create n using old graph and new edge
            n=i;
            n<<=shift;
            n|=*it;
```

```
            /*
            if (vertices == 9 || vertices == 10){
                std::cout << "Old graph: " << binary_str(*it, 64)
                            << ", Edge: " << binary_str(i, 16)
                            << ", New graph: " << binary_str(n, 64)
                            << std::endl;
            }
            */

            if (check(n)==0){
                new_graphs_ptr->insert(canon_label(n,vertices));
                //new_graphs_ptr->insert(n);
            }

        }
    }
}

void Solver::add_edge(BIGINT y, BIGINT edge, int vertices, int
    edge_start, int shift){

    if (edge_start>=vertices) return;

    BIGINT n, e;

    // prepare new graph using this edge
    n=edge;
    n<<=shift;
    n|=y;

    int c = check(n);
    if (c==0){
        // ramsey graph
        new_graphs_ptr->insert(canon_label(n,vertices));
    }
    else{
        // since this is not a ramsey graph
        // no point adding new edges

        // c==1 means the check failed for cliques
```

```
        // c==2 means it failed for independent set

        if (c==1) return;
    }

    for (int i=edge_start; i<vertices-1; ++i){

        // prepare edge
        e=1;
        e<<=i;

        add_edge(y, edge|e, vertices, i+1, shift);

    }

}


void Solver::solve_using_edges(int vertices){

    // shift count for new edges
    int shift = (vertices-1)*(vertices-2)/2;

    // partition the graphs for parallel execution
    // start parallel after hitting 100 thousand graphs
    if (!mpi_parallel_start && old_graphs_ptr->size() > 100000){

        float partition_size = old_graphs_ptr->size() /
            mpi_num_processes;

        int st = (int)(mpi_this_process * partition_size);
        int en = (int)(mpi_this_process * partition_size +
            partition_size);

        int count = -1;

        // add edges to each Ramsey graph from previous graph order
        for (std::set<BIGINT>::iterator it=old_graphs_ptr->begin(); it
            != old_graphs_ptr->end(); ++it){
```

```
            count++;

            // skip anything not in range
            if (count < st || count >= en) continue;

            add_edge(*it, 0, vertices, 0, shift);
            old_graphs_ptr->erase(it);
        }

        // parallel started, dont partition anymore
        mpi_parallel_start = 1;

    }
    else{
        // add edges to each Ramsey graph from previous graph order
        for (std::set<BIGINT>::iterator it=old_graphs_ptr->begin(); it
            != old_graphs_ptr->end(); ++it){
            add_edge(*it, 0, vertices, 0, shift);
            old_graphs_ptr->erase(it);
        }
    }

}


int Solver::check(BIGINT n){

    BIGINT t;

    // check n against all constraints
    for (int i=0;i<constraint.size();++i){

        t = constraint[i].lhs & n;

        if (constraint[i].sign == '<'){
            if (popcount(t) >= constraint[i].rhs){
                // doesnt satisfy
                return 1;
            }
        }
```

```
        if (constraint[i].sign == '>'){
            if (popcount(t) <= constraint[i].rhs){
                // doesnt satisfy
                return 2;
            }
        }

    }

    return 0;


}

void Solver::solve_ramsey(int s, int t){

    // start clock
    begin = clock();

    // ramsey graphs upto n<=2 are same

    // create graphs for n=2

    old_graphs_ptr->clear();
    new_graphs_ptr->clear();

    // two graphs for n=2
    old_graphs_ptr->insert(0);
    old_graphs_ptr->insert(1);

    // graph order
    int n=2, total;

    // vector to store edges of complete graph
    std::vector<BIGINT> v;

    // output sugar
    std::cout << "# " << mpi_this_process << ": R(" << s << "," << t <<
        "," << "1) = 1 [0s]" << std::endl;
```

```
std::cout << "# " << mpi_this_process << ": R(" << s << "," << t <<
    "," << "2) = 2 [0s]" << std::endl;

while (n++){

    //std::cout << "# " << mpi_this_process << ": R(" << s << "," <<
        t << "," << n << ") = " << std::flush;

    // calculate edges
    int e=n*(n-1)/2, i;

    Constraint c;
    std::vector<int> tmp;

    // clear constraints
    clear_constraints();

    // computing minimum size of constraint
    int shift = (n-1)*(n-2)/2;

    // remove all unnecessary constraints
    BIGINT min_constraint = 1;
    min_constraint<<=shift;

    // get combinations for K_s
    v.clear();
    tmp.clear();
    get_combinations(v,n,s,tmp);
    c.sign = '<';
    c.rhs = s*(s-1)/2;
    for (i=0;i<v.size();++i){
        c.lhs = v[i];
        if (c.lhs >= min_constraint) add_constraint(c);
        //add_constraint(c);
    }


    // get combinations for K_t
    v.clear();
    tmp.clear();
```

```
    get_combinations(v,n,t,tmp);
    c.sign = '>';
    c.rhs = 0;
    for (i=0;i<v.size();++i){
        c.lhs = v[i];
        if (c.lhs >= min_constraint) add_constraint(c);
        //add_constraint(c);
    }

    //print_constraint();

    //solve(n);
    solve_using_edges(n);

    std::cout << "# " << mpi_this_process << ": R(" << s << "," << t
        << "," << n << ") = "
              << new_graphs_ptr->size() << " [" << (double(clock() -
                   begin) / CLOCKS_PER_SEC) << "s]" << std::endl;

    if (new_graphs_ptr->size()==0) {
        // no ramsey graphs found
        // current n is the Ramsey Number
        break;
    }

    // point old graphs to new graphs for next iteration
    std::set<BIGINT> *tmp_graphs;
    tmp_graphs = old_graphs_ptr;
    old_graphs_ptr = new_graphs_ptr;
    new_graphs_ptr = tmp_graphs;

    // clear new graphs (previously old graph)
    // EDIT: no need to clear, this set is already emptied
    // new_graphs_ptr->clear();

}

// wait till all processes are done
mpi_wait();
```

```
}

void Solver::mpi_wait(){
    int a = 0;

    // block till all processes are done
    if (mpi_this_process==0){
        MPI_Bcast(&a, 1, MPI_INT, 0, MPI_COMM_WORLD);
    }
    else{
        MPI_Bcast(&a, 1, MPI_INT, 0, MPI_COMM_WORLD);
    }
}

/*
void Solver::solve_ramsey(int s, int t, int n){

    int e=n*(n-1)/2, i;

    old_graphs_ptr->clear();
    new_graphs_ptr->clear();

    old_graphs_ptr->insert(0);

    // create constraints
    Constraint c;
    std::vector<int> tmp;

    // clear constraints
    clear_constraints();

    // computing minimum size of constraint
    int shift = (n-1)*(n-2)/2;

    // vector to store edges of complete graph
    std::vector<BIGINT> v;

    // get combinations for K_t
    v.clear();
```

```
tmp.clear();
get_combinations(v,n,t,tmp);
for (i=0;i<v.size();i++){
    c.lhs = v[i];
    c.sign = '>';
    c.rhs = 0;
    add_constraint(c);
}

// get combinations for K_s
v.clear();
tmp.clear();
get_combinations(v,n,s,tmp);
for (i=0;i<v.size();i++){
    c.lhs = v[i];
    c.sign = '<';
    c.rhs = s*(s-1)/2;
    add_constraint(c);
}


BIGINT y;
for (int edge_count = 1; edge_count<=e; edge_count++){

    for (std::set<BIGINT>::iterator it=old_graphs_ptr->begin(); it!=
        old_graphs_ptr->end(); ++it){

        // for each old graph, create graphs with one more edge and
            check
        for (shift=0;shift<e;shift++){
            // create n using old graph and new edge
            y=1;
            y<<=shift;
            y|=*it;

            std::cout << binary_str(y,e) << std::endl;

            if (y==*it) continue;

            if (check(y)){
```

```
                new_graphs_ptr->insert(canon_label(y,n));
            }
        }


    }

    std::cout << "R(" << s << "," << t << "," << edge_count << ","
       << n << ") = " << new_graphs_ptr->size() << std::endl;

    // point old graphs to new graphs for next iteration
    std::set<BIGINT> *tmp_graphs;
    tmp_graphs = old_graphs_ptr;
    old_graphs_ptr = new_graphs_ptr;
    new_graphs_ptr = tmp_graphs;

    // clear new graphs (previously old graph)
    new_graphs_ptr->clear();

    if (old_graphs_ptr->size()==0){
        // no graphs created
        // manually create graphs
        v.clear();
        tmp.clear();
        get_combinations(v,e,edge_count,tmp);
        for (i=0;i<v.size();i++){
            old_graphs_ptr->insert(v[i]);
        }
    }

  }


}
*/


std::string Solver::get_g6(BIGINT n, int vertices){

  int i,t;
```

```
    int m = vertices*(vertices-1)/2;
    std::string s="";

    if (vertices<=62){
        s += (char)(vertices + 63);
    }

    BIGINT one = 1;
    BIGINT tmp = 0;

    while (m>0){

        t=0;
        // get six bits from back
        for (i=0;i<6;i++){

            t<<=1;

            // check if last bit is set
            tmp = n&one;
            if (tmp==one) t|=1;

            n>>=1;
        }

        t+=63;

        s+=(char)t;

        m-=6;
    }

    return s;

}

/*
int Solver::popcount(BIGINT n){
```

```
    bitset<64> b(n);
    return b.count();

}
*/

#ifndef MPZ_BIGINT

inline int Solver::popcount(BIGINT x){

    int pop_count;
    for (pop_count=0; x; pop_count++)
        x &= x-1;
    return pop_count;

}

#else

inline int Solver::popcount(BIGINT x){
    return (int)mpz_popcount(x.get_mpz_t());
}

#endif


/*
int __popcount(BIGINT x){
    x =  x       - ((x >> 1)  & k1); // put count of each 2 bits into
        those 2 bits
    x = (x & k2) + ((x >> 2)  & k2); // put count of each 4 bits into
        those 4 bits
    x = (x       +  (x >> 4)) & k4 ; // put count of each 8 bits into
        those 8 bits
    x = (x * kf) >> 56; // returns 8 most significant bits of x + (x<<8)
         + (x<<16) + (x<<24) + ...
    return (int) x;
}
*/
```

Theorem 1.1 [1] *A bounded function f is Riemann integrable on $[a, b]$ if and only if the set of discontinuities of f has measure* 0.

Table 1: Caption for the first table.

Column 1	Column 2
1	2
3	4

2 TITLE OF SECOND CHAPTER

Text of Chapter 2.

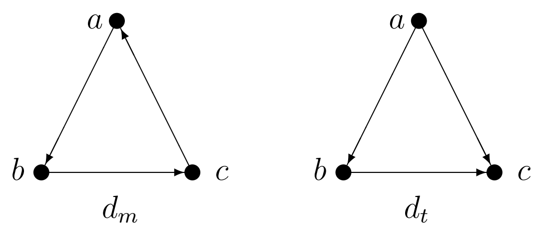

Figure 1: Caption for the first figure.

BIBLIOGRAPHY

[1] *An Introduction to Analysis*, 2nd edition, by J. R. Kirkwood, Published by PWS Publishing Company and Waveland Press, Inc. 1995.

VITA

BOB GARDNER

Education:	B.S. Mathematics, Auburn University in Montgomery, Montgomery, Alabama 1984
	M.S. Mathematics (Combinatorics), Auburn University Auburn, Alabama 1987
	M.S. Zoology and Wildlife (Population Genetics), Auburn University, Auburn, Alabama 1992
	Ph.D. Mathematics (Complex Analysis), Auburn University Auburn, Alabama 1991
Professional Experience:	Assistant Professor, LSU in Shreveport, Shreveport, Louisiana, 1991–1993
	Assistant, Associate, Full Professor, East Tennessee State University, Johnson City, Tennessee, 1993–present
Publications:	R. Gardner and N. K. Govil, "Some Inequalities for Entire Functions of Exponential Type," *Proceedings of the American Mathematical Society*, **123**(9) (1995) 2757–2761.