University of Nevada
Reno

A Generic Queuing System and Time-Saving Region Restrictions for Calculating the Crossing Number of K_n

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
with a major in Computer Science.

by

Bei Yuan

Dr. Frederick C. Harris, Jr., Thesis advisor

December 2004

We recommend that the thesis

prepared under our supervision by

BEI YUAN

entitled

A Generic Queuing System and Time-Saving Region Restrictions for Calculating the Crossing Number of K_n

be accepted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

Dr. Frederick C. Harris, Jr., Ph.D., Advisor

Dr. Sergiu M. Dascalu, Ph.D., Committee Member

Dr. Hyunjeong Lee, Ph.D., Graduate School Representative

Marsha H. Read, Ph.D., Associate Dean, Graduate School

December 2004

Abstract

With the availability of inexpensive computer clusters it is now economically feasible to use parallel processing to attack computationally intensive problems such as calculating the crossing number of a graph. In 1996 Harris and Harris presented an algorithm for solving the crossing number problem of complete graphs, which was implemented in parallel by Tadjiev and Harris a year later. Their algorithm, though parallelized, was not load balanced and did not prune the search space with additional restrictions.

This thesis introduces an easily adaptable parallel work queue combined with a more restrictive algorithm to solve crossing number problems. Implementation of the parallel work queue offers an opportunity to get better results than previously possible on crossing number problems. Meanwhile, we also use this more restrictive algorithm to verify the efficiency of our parallel queuing system. Both the algorithm implementation and analysis are given in this thesis.

Dedication

To my grandmother, parents, and sister.

Acknowledgments

I would like to express my deepest gratitude to my thesis advisor, Dr. Frederick C. Harris, Jr., for his continuous guidance, encouragement, and for helping me through my Master's program and the entire thesis writing process.

I would also like to thank Dr. Sergiu M. Dascalu and Dr. Hyunjeong Lee, for serving on my thesis committee and spending valuable time editing and providing advice.

I am very grateful to Mrs. Judith Fredrickson, for giving all the valuable discussion and suggestions, and bearing with me all the way. Good luck with your dissertation.

I would like to add a very special thanks to my family for their unconditional love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The year was 1944, the location Hungary. Paul Turán documented a personal experience with the following description [5]:

> We worked near Budapest, in a brick factory. There were some kilns where the bricks were made and some open storage yards where the bricks were stored. All the kilns were connected by rail with all the storage yards. The bricks were carried on small wheeled trucks to the storage yards. All we had to do was to put the bricks on the trucks at the kilns, push the trucks to the storage yards, and unload them there. We had a reasonable piece rate for the trucks, and the work itself was not difficult; the trouble was only at the crossings. The trucks generally jumped the rails there, and the bricks fell out of them; in short this caused a lot of trouble and loss of time which was precious to all of us. We were all sweating and cursing at such occasions, I too; but *nolens volens* the idea occurred to me that this loss of time could have been minimized if the number of crossings of the rails had been minimized. But what is the minimum number of crossings?

Turán had conceptualized and documented the crossing number problem. Known as Turán's Brick Factory Problem, Turán's query is considered the birth of the crossing

number problem. Over the years, many have studied this easily stated problem, but little is known about general solutions. Garey and Johnson [9] proved that finding the minimum crossing number of a graph was NP-complete. Because of the difficulty of this problem, work turned away from finding the crossing number of a graph to sub-problems and other related works. Pach and Tóth [15] is a good resource for an overview of references and current open problems on crossing numbers for various graph families. An interesting discussion of the many different interpretations of the term 'graph drawing' can be found in [16].

Almost twenty years before the proof of the crossing number problem as NP-complete, Guy initiated the pursuit of the crossing number of complete graphs [10]. A graph with n vertices is complete, denoted K_n, if every pair of vertices are directly connected. Figure 1.1 shows the complete graph with 3, 4, and 5 vertices. Our current work joins in Guy's pursuit. We pursue complete graphs because they are a well studied graph family. Some known answers exist upon which to test our theories.

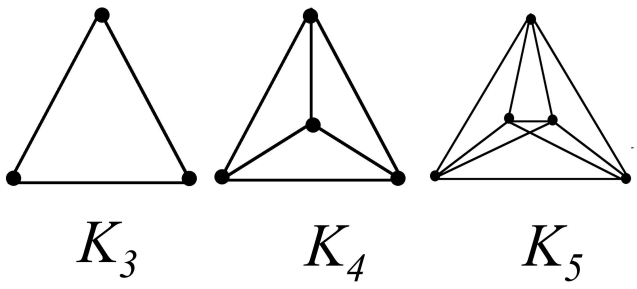

Figure 1.1: Graphs of K_3, K_4, and K_5

We employ the algorithm proposed by Harris and Harris in [13]. The algorithm uses an exhaustive search to find the crossing number of a graph. As graph size increases, exhaustive searches can become computationally intensive, taking months or

even years to complete on a single computer. Parallel computers offer the hope for providing the computational power to meet the demands of running these complex algorithms. However, the difficulty in parallelizing most algorithms lies in properly and efficiently dividing the work to ensure maximum concurrency and simultaneous termination. Work queues have proven to be an efficient means of ensuring load balancing, but they often require large amounts of message passing as slave processes request work from the queue. These messages make proper synchronization of the utmost importance, but they also make it difficult to achieve. We attempt to simplify this problem by developing a generic work queuing system for medium to large Beowulf clusters, which are made of network computers. Using a medium to large Beowulf cluster and a parallel queuing system that can be optimized for problem granularity, along with an updated definition of a good drawing, we have efficiently solved the crossing number problem for a complete graph with less than nine vertices. Parallel implementation of the Harris and Harris algorithm has gone through several iterations and improvements in recent times [8, 14, 19, 20, 26]. By harnessing the power of computer clusters and tightening the definition of a good graph drawing from a region perspective, the crossing number problem may be solved for larger graphs.

The remainder of the thesis is laid out as follows: Chapter 2 gives a brief overview of crossing number problems and the queuing system in general. Chapter 3 introduces our improvements based on the Harris and Harris algorithm and implementation details of a parallel work queue. Chapter 4 presents the results, including analysis and comparisons, of running the algorithm in various situations. Conclusions and directions for future work are provided in Chapter 5.

Chapter 2

Literature Review

This chapter presents the crossing number problems and the queuing system in general. Section 2.1 introduces the terminology, definition, and equations associated with the algorithm of calculating the crossing number of complete graphs. Section 2.2 provides a detail description on the previously used Harris and Harris algorithm. A parallel queuing system and its unique problems are addressed in section 2.3.

2.1 Crossing Number Problem

2.1.1 Terminology and Definitions

We now informally define some terms from graph theory which will be used in the rest of the paper. These definitions are based on [5].

For our purposes, a *compact-orientable 2-manifold*, or simply a *surface*, may be thought of as a sphere or a sphere with some number of handles placed on it.

Definition 2.1.1 *The genus of a surface S is the number of handles on the surface.*

Definition 2.1.2 *The genus of a graph G, gen(G), is the smallest genus of all surfaces on which G can be embedded.*

Definition 2.1.3 *A graph G is embeddable on a surface S if it can be drawn on S*

such that any two edges that intersect do so only at a vertex of G mutually incident with them.

Definition 2.1.4 *A graph is a planar graph if it can be embedded on the plane.*

Definition 2.1.5 *A planar graph that is embedded on the plane is called a plane graph.*

Definition 2.1.6 *A region of the plane graph G is a connected portion of the plane remaining after all curves and points corresponding to edges and vertices of G have been deleted.*

Definition 2.1.7 *A region is called a 2-cell region if any simple closed curve in that region can be continuously deformed or contracted in that region to a single point.*

Definition 2.1.8 *An embedding is a 2-cell embedding if all the regions in an embedding are 2-cell.*

Definition 2.1.9 *A good drawing of a graph G is one with the following properties:*

- *adjacent edges never cross,*
- *two nonadjacent edges cross at most once,*
- *no edge crosses itself,*
- *no more than two edges cross at a point of a plane, and*
- *the (open) arc in the plane corresponding to an edge of the graph contains no vertex of the graph.*

Definition 2.1.10 *The crossing number of a graph G, denoted $\nu(G)$, is the minimum number of edge crossings among all the good drawings of G in the plane.*

Finally, the relationship between the number of regions of a graph and the surface on which it is embedded is described by the well-known generalized Euler's Formula[5]:

Let G be a connected graph with n vertices and m edges with a 2-cell embedding on the surface of genus g and having r regions, then

$$n - m + r = 2 - 2g$$

The definitions related to embeddings and Euler's Formula are vital to the Harris and Harris algorithm as it employs Edmonds' Rotational Embedding Scheme. We discuss both in Section 2.2.

2.1.2 Minimum Crossing Number

As stated, we are concentrating on the crossing number of K_n. First, we differentiate between two general categories of crossing numbers for graphs. The crossing number, denoted $\nu(G)$, allows for edges that are nonlinear. Another category of crossing numbers is the rectilinear crossing number, denoted $\bar{\nu}(G)$, which is defined similarly but with the added constraint that each edge is a straight line segment.

For complete graphs of size 10 and less, a simple formula provides the crossing number [11]:

$$\nu(K_n) = \frac{\lfloor \frac{n}{2} \rfloor \lfloor \frac{n-1}{2} \rfloor \lfloor \frac{n-2}{2} \rfloor \lfloor \frac{n-3}{2} \rfloor}{4} \tag{2.1}$$

The rectilinear crossing number also holds for this formula, with the notable exception of graphs with 8 vertices. For K_8, the crossing number is 18; while the rectilinear crossing number is 19. As the crossing problem increases in size (11 vertices or more), a different formula appears to hold true for the rectilinear case:

$$\bar{\nu}(K_n) \leq \lfloor \frac{(7n^4 - 56n^3 + 128n^2 + 48n\lfloor \frac{(n-7)}{3} \rfloor + 108)}{432} \rfloor \tag{2.2}$$

Guy proposed this formula in 1972 [11] and conjectured that equality held. The equality conjecture was shown to be false when Thorpe and Harris [21] generated a

rectilinear drawings of K_{12} with 155 crossings and K_{13} with 229 crossings instead of 156 and 231 as conjectured by Guy. Recent work by Brodsky *et al.* [4] and Aichholzer *et al.* [1] proves that $\bar{\nu}(K_{10}) = 62$, and [2] proves $\bar{\nu}(K_{11}) = 102$ and $\bar{\nu}(K_{12}) = 153$.

Also in 1972, Guy [11] conjectured that equation 2.1 holds true for all n. To date this conjecture still stands. In private communication Guy anticipates that divergence around n of size 12 or 13 is possible as was seen in the rectilinear case [12].

2.2 An Overview of the Algorithms Used

As noted, we employ an exhaustive-search, branch-and-bound algorithm proposed by [13]. An overview of it is presented to allow for understanding of the region restriction presented in this section.

2.2.1 Edmonds' Rotational Embedding Scheme

We start with a look at the Rotational Embedding Scheme, first formally introduced by Edmonds [6] in 1960 and then discussed in detail by Youngs [25] a few years later. The following is the formal statement of the Rotational Embedding Scheme as presented in [5].

> Let G be a nontrivial connected graph with $V(G) = \{v_1, v_2, \ldots, v_n\}$. For each 2-cell embedding of G on a surface there exists a unique n-tuple $(\pi_1, \pi_2, \ldots, \pi_n)$, where for $i = 1, 2, \ldots, n$, $\pi_i : V(i) \rightarrow V(i)$ is a cyclic permutation that describes the subscripts of the vertices adjacent to v_i. Conversely, for each such n-tuple $(\pi_1, \pi_2, \ldots, \pi_n)$, there exists a 2-cell embedding of G on some surface such that for $i = 1, 2, \ldots, n$ the subscripts adjacent to v_i and in the counterclockwise order about v_i, are given by π_i.

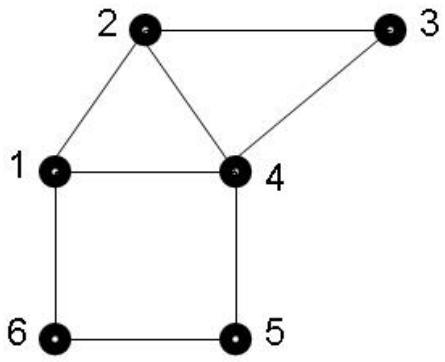

Figure 2.1: A Planar Embedding of a Graph

For example, consider Figure 2.1 which gives a planar embedding of a graph. From this graph we obtain the following counterclockwise permutations associated with each vertex:

$$\begin{array}{ll} \pi_1 = (6,4,2) & \pi_2 = (1,4,3) \\ \pi_3 = (2,4) & \pi_4 = (3,2,1,5) \\ \pi_5 = (4,6) & \pi_6 = (5,1) \end{array}$$

From these permutations we can obtain the edges of the graph and the number of regions of the graph. For instance, this graph has 4 regions. The edges for one of these regions can be traced as follows:

1) Start with edge (1,2)
2) Go to permutation π_2 and find out who follows 1, and it is 4. Therefore the second edge is (2,4).
3) Go to permutation π_4 and find out who follows 2, and it is 1. Therefore the third edge is (4,1).
4) Go to permutation π_1 and find out who follows 4 and it is 2. This yields the original edge (1,2), so we are finished.

The region traced was bounded by the edges (1,2), (2,4), and (4,1). The other regions and edges can be found in a similar manner.

It is important to note the converse portion of the Rotational Embedding Scheme: every collection of vertex permutations corresponds to an embedding on some surface. Given a set of permutations, we can trace the edges and determine the genus of the surface.

2.2.2 The Harris and Harris Algorithm

In this algorithm [13] the solution space of the crossing number problem is mapped onto a tree. The tree is then searched for the crossing number with a branch-and-bound depth-first search (DFS). A DFS searches more deeply into the tree for a solution whenever possible. Once a path is found from the root to a leaf representing a solution, the search backtracks to explore the nearest unsearched portion of the tree. This continues until the entire tree has been traversed. Branch and bound allows us to change one small part of the DFS algorithm. When the cost to get to a vertex v exceeds the current optimal solution, we tell the DFS algorithm not to traverse the subtree having vertex v as its root. This saves us from having to cover a section of the search space that is guaranteed to cost more than the current optimal solution.

First, we begin with the vertex set for the graph in question and begin to add edges. After each edge is added, we determine whether the partial graph is still planar. This is done using the Rotational Embedding Scheme. Once we cannot add any edges and keep the partial graph planar we are ready to begin our mapping to the tree.

At this stage, the partial graph is the root of the tree. The root of the tree has a branch for each possible planar embedding. We select the first embedding and begin to build the rest of its tree. Regions compose the levels of the tree and edge segments through the regions are the branches to the next level. Tree building begins by considering laying down the next edge (which will go from vertex i to vertex j).

The first decision we need to make is: through which one of the m regions, to which vertex i is adjacent, should this edge should leave? These m regions represent the next layer of our tree. Once the region is selected, the next decision is: which of the l edges of that region the edge will cross? When we have made this decision, we will create a cross vertex (degree 4) and place an edge from vertex i to the cross vertex and then try to lay the edge from the cross vertex to vertex j. This may be possible directly, or it may require more cross vertices.

For all the rest of the edges we lay them down in a similar manner, and when we have finally laid them all down we have reached a leaf in the tree. At this point we have a cost for the current solution which is the number of cross vertices we have added. This number of crossings becomes the new bound. The DFS continues by backing up and trying other paths through the tree using the bound as a stopping criteria. We proceed until the entire solution space is traversed. In order to understand this, we will walk through the algorithm with K_5 as our example. In Figure 2.2 we have the vertex set for the graph with all the edges that can be added to the graph while it remains planar.

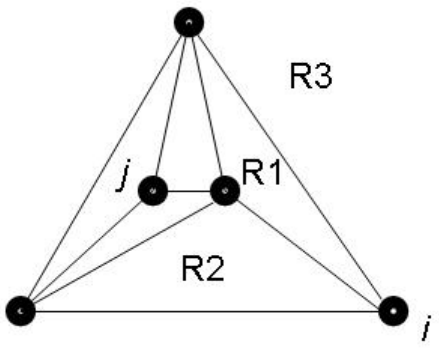

Figure 2.2: Planar Portion of K_5

At this point, the algorithm states that we are to take all remaining edges and try

to lay them down one at a time. This is fairly simple in this case because there is only one edge left to be added to construct a graph of K_5, and that is from vertex i to vertex j. Next we choose a region through which to lay the edge. We have three choices, and these are the three regions to which vertex i is adjacent (R1, R2, and R3). We select R1 which has 3 edges. We cannot legally cross two of the edges (since they are incident with i) so there is only one choice. We then place a cross vertex on this edge and connect an edge from i to the cross vertex as shown in Figure 2.3. We then find out if we can draw the edge from the cross vertex k to vertex j and take the cross vertex k as on real vertex instead of a crossing, we can still keep the graph planar. In this case we can and we are finished with this edge and have one crossing after we connect vertex k with j. This solution is shown in Figure 2.4. The algorithm then backtracks and tries the other regions to which i is adjacent and finds out that there are multiple ways to draw K_5 with one crossing, but none with zero crossings.

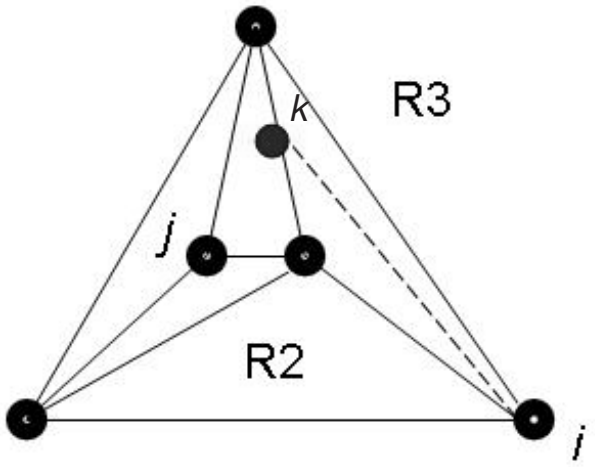

Figure 2.3: Beginning to Lay Down an Edge for K_5

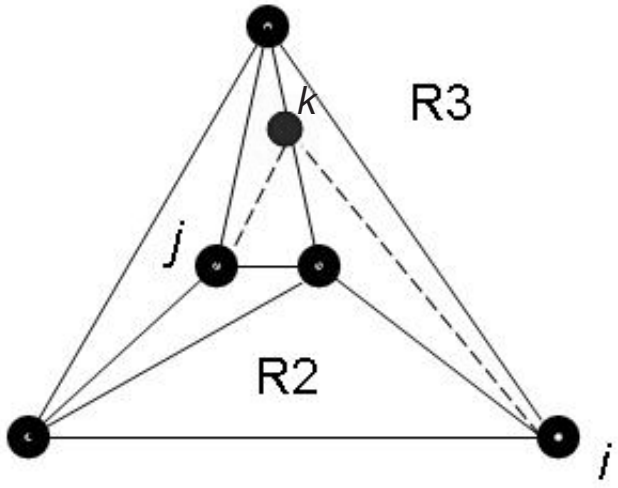

Figure 2.4: One Drawing of K_5 with One Crossing

2.3 Queuing System Overview

2.3.1 Parallel Work Queues

Problems such as solving the crossing number of a complete graph are computationally intensive and virtually impossible to solve using a conventional single processor machine. Fortunately, such problems can often be easily broken down into smaller problems. The size, or granularity, of these smaller problems can vary greatly, but the concept of separate processors tackling portions of the overall problem applies regardless of size.

It is often advantageous to use a divide and conquer strategy when dealing with difficult and/or lengthy problems. These problems may have data sets that are completely defined prior to run-time and are often divided among the processors, each computing its own results from its data set. This method is referred to as static partitioning. If one processor finishes working it must wait idly while the other processors catch up. When a processor is idle, the benefits of concurrent execution are not

realized in their entirety. In addition, heterogeneous processor clusters may make effective partitioning difficult. Static partitioning is effective when data sets are known prior to run-time and the execution time of the data can be easily determined. This is seldom the case, however. One solution is dynamic load balancing.

A work queue is one method of implementing dynamic load balancing and thereby ensuring a work load that is evenly distributed across many processors or machines [18]. This load balancing can be either centralized, residing with a master process, or decentralized, controlled by each slave. A combination of these two systems may also be used. The work queue is especially useful in load balancing with irregular data structures such as an unbalanced search tree [24].

In centralized load balancing, the tasks to be performed are held by the master and distributed to the slaves as they finish other tasks and become idle. This process minimizes the time each slave is idle, thereby maximizing efficiency. One disadvantage of centralized load balancing is the possibility of a bottleneck while the master distributes tasks. A bottleneck occurs when many slaves request tasks simultaneously, but the master can issue only one task at a time. Another disadvantage of this centralized balancing is the heavy network load. It needs more network bandwidth to send commands and data to maintain the balance. In decentralized load balancing, local processors keep their own work pools. This strategy has the benefit of avoiding the bottleneck mentioned above. Decentralized load balancing is similar to static partitioning and has the same apparent problems. In more complex systems, the slaves may request work from each other or from a centralized master queue.

2.3.2 Cluster Information

Conventional supercomputers or high performance computing systems are expensive and beyond the budgets of many researchers. A lower cost option available to such research groups is a Beowulf cluster, a cluster of high performance workstations designed to work in parallel. Beowulf cluster technology uses clusters of commodity machines, relying on standard Ethernet networks as a system interconnection, which allows the construction of distributed memory parallel computers to serve as a "supercomputer". The normally used network connection is 100Mb/s Ethernet, although more expensive technology such as Myrinet [3] and QsNet [17] may be used as well. These more expensive technologies provide better performance because of their low latency, high bandwidth and ability to scale well among larger numbers of nodes.

The typical steps employed in building a Beowulf cluster are as follows [22]:

- Install an operating system on the front end.
- Install each compute node and link them together to form a cluster.
- Set up a single system view.
- Install parallel computing systems.

To use a cluster for parallel processing, a software layer must be installed to make a virtual parallel machine from a group of nodes. In general, this software layer is in charge of parallel task creation/deletion, passing data among tasks, and synchronizing task execution. Current public tools available are PVM, MPI, and BSP, which make cluster computing handle a wider range of applications for low cost.

Our cluster currently has 128 compute processors. 60 of these are Pentium III 1 GHz and 68 are Pentium IV XEON 2.2 GHz Processors. Each of these CPUs have 2GB of RAM and in total the cluster has more than a half terabyte of disk storage.

These compute nodes are connected with 1Gbps Ethernet (for NFS) and Myrinet 2000 (for communication), where both MPI and PVM techniques are supported. We just received funding to add 80 more processors. These processors will be 64 bit processors with 2 GB of memory per CPU.

2.3.3 Previous Solutions and Problems

We began with an existing implementation of the sequential algorithm to find the crossing number of a complete graph given by Harris and Harris in [13]. As we presented, this algorithm is a depth-first-search branch-and-bound algorithm which exhaustively covers the entire search space. This implementation was parallelized using static partitioning by Tadjiev in [19] and [20]. Due to the fact that static partitioning does little to ensure load balancing, Tadjiev found that it was common in a long run for one processor to be working while the others sat idle. This idleness can be attributed to an unbalanced tree that was statically partitioned across the processors. An example of such a tree is shown in Figure 2.5. In this example, processors 2 and 3 sit idle while processor 1 works toward a solution.

Without the minimal guarantee of load balancing, Tadjiev also showed that very little speedup was noticed in solving larger vertex sets when static partitioning was run on many processors. By utilizing a generic work queuing system, the crossing number problem and other problems can be solved faster and more efficiently with minimal adaptation.

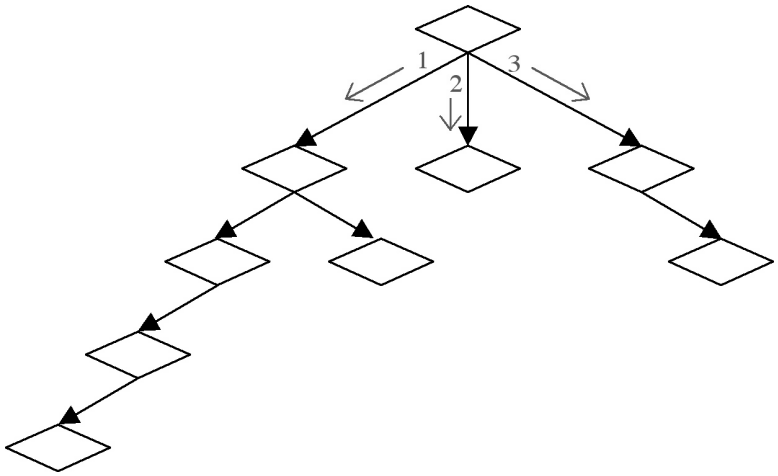

Figure 2.5: Unbalanced Search Tree

Chapter 3

Algorithm Implementation and Improvement

In this chapter, we first describe our region restricted algorithm for calculating the crossing number of K_n. Then we provide a detailled implementation of the parallel queuing system. The application of combining the above two scheme to solve crossing number problems is given at the end.

3.1 Crossing Number Algorithm Improvement

We show four possibilities of how we can lay down an edge from u to v starting through region R1 in Figure 3.1 . Based on Definition 2.1.9 we can see all of the graphs in Figure 3.1 are good drawings. R1 has five edges, three of which can be legally crossed. Recall that each edge adjacent to R1 not incident to vertex u will be used in generating all valid uv edges, so we are assured that all edge placements shown in Figure 3.1 will be generated. Figures 3.1(a) and (b) both illustrate placement of edge uv with one crossing, while Figures 3.1(c) and (d) have two and four crossings, respectively. Note that the new edge in both (c) and (d) exits R1 on the same edge and enters R3 on the same edge, but the edge in Figure 3.1 (d) travels less efficiently, creating more crossings than in (c). Thus the drawing in Figure 3.1 (d), though a

good drawing, will not aid in producing a complete graph with a minimum number of crossings because the drawing in (c) is more efficient.

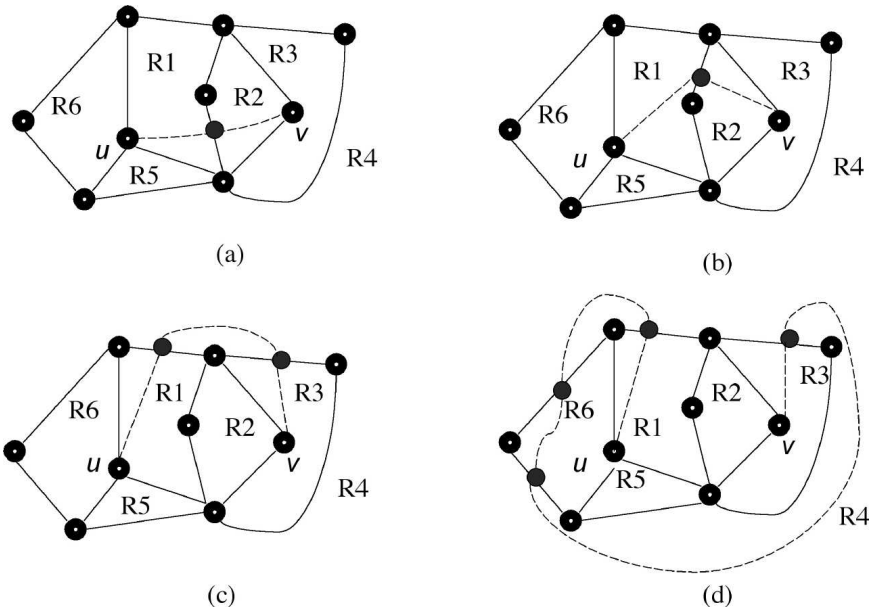

Figure 3.1: Good Drawings for Placement of Edge uv

With this observation we add a further restriction to the definition of a good drawing to include a region restriction.

Let $f : u = p_0, p_1, ...p_n, p_{n+1} = v$ denote the edge laid down from two non-adjacent vertices u, v of G. Let $H = G + f$.

The graph H is defined to be a *region restricted good drawing* if every region of H is adjacent to at most two vertices on f. This new restriction eliminates the graph in Figure 3.1 (d) from being generated.

Figure 3.2 shows the laying down of the six initial edge segments emanating from vertex u in laying of all valid edges from u to all other nonadjacent vertices. For any of the edges from u to v of G, the edges will be restricted from entering regions R1, R6 and R5 after laying down the initial edge segment. With each additional edge segment placed, the relevant regions will join the list of restricted regions for any

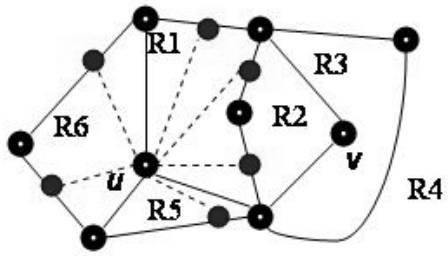

Figure 3.2: Initial Edge Segments From Vertex u to Other Nonadjacent Vertices

given edge.

Modification to the graph generation algorithm was minimal. Whenever an edge segment is laid down, its initiating vertex is used to add to, or create, a list of restricted regions for that edge.

3.2 The Generic Queuing System

Similarly to crossing number problems, many computationally expensive problems have had parallel algorithms either proposed or implemented that make it easy to break problems into jobs (or sub-jobs) [18]. Therefore, we decided to build a queue of jobs (work) that can be distributed across a cluster to harness the parallel computation power available, allowing researchers to use it with little or even no knowledge needed of the parallel programming details.

A queue is a well-known data structure used to produce first-in, first-out behavior for lists of data. A powerful tool can be built by parallelizing the queue data type, allowing the first-in, first-out properties to be exhibited across computers. A parallel job queue is a specialized form of parallel queue that is designed around the processing and distribution of jobs.

The implementation of the parallel job queue discussed in this thesis has several

key features. One primary goal of the parallel job queue is to have a queue that is as generic as possible, allowing many types of problems to use the data structure without having to reconfigure (or even be knowledgeable about) its inner workings. Another major goal is to hide as much of the parallel programming as possible. In fact, nearly all parallel aspects can be hidden. A secondary goal is that all message passing can be performed through the parallel queue interface.

To keep the queue generic enough to work with many types of problems, an abstract data type called package is employed. A package is a generic data storage object that holds three pieces of information:

1. an identifier telling what type of message is being processed,
2. the number of bytes in the data portion, and
3. the data portion (represented by raw byte data).

A package is then expected to be sub-classed so that additional functionality (such as specific encoding and decoding) can occur. For example, a job package is a subclass of package that handles the decoding of the raw data into the job format, whatever that may be. For the parallel job queue, a job is a description of the data or processing that needs to take place. Job packages can be enqueued on the parallel queue so that other processes can retrieve the job package at a later time. Jobs can be dequeued simply by asking for a package from the queue.

The second goal of hiding the parallel programming was successful because we were able to encapsulate all of the general message passing into the queue itself and use the message types to differentiate job packages from control packages, described in the next subsection, virtually all of the parallel programming is hidden. By using a single abstract data type, a package, to encode all data transfer, a great amount of flexibility and portability is ensured.

3.2.1 Queuing System Implementation

The queuing system was designed to be as generic as possible, allowing it to be adapted to a variety of problem sizes and types. The system works around jobs. A job represents whatever amount of work can be executed independently. Thus, the system is easily tunable by the user to ensure parallel efficiency. The size of each job, or the amount of work the job contains, is referred to as the granularity of a task. Coarse granularity describes a task with a large number of sequential instructions that takes a significant amount of time to execute [23].

In our proposed implementation, which is illustrated in Figure 3.3, first the master will create the first m jobs and place them on the central queue. In general, m is determined by the problem under investigation and should be greater than n, where n is the number of slave processors. The master controls the job distribution, and is the only processor that can have the access to the central queue. Then it sends a job to each processor. Each processor will create more jobs while processing. These jobs are kept in a local work queue by each slave. When a slave's local work queue reaches some user-defined limit, it will send the extra jobs to the master for placement in the central queue. When a slave is idle, it sends a request for a job to the master and, upon receipt, begins computation and the filling of its local queue. After initial startup, we have configured the master to hold a certain amount of jobs in the central queue so that jobs can be delivered reasonably fast whenever any idle slave requests. If a slave is idle and the central queue is empty, the idle slave takes a job from another working slave *via* the master. Process termination occurs when all slaves are idle and the central queue is empty.

We also named signals that the queue will deliver to specified nodes right away

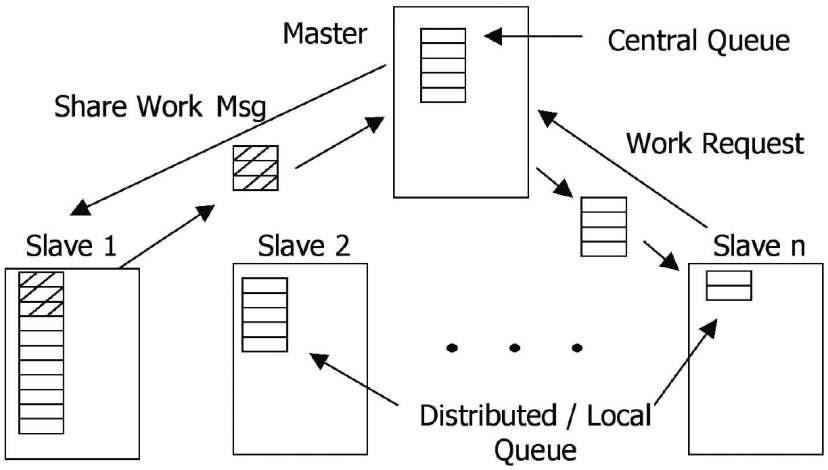

Figure 3.3: Parallel Work Queue

instead of considering them as a package to be queued. For example, the `STOP` signal informs the slave to stop working. `SLAVE_NEED_JOB` keeps the master updated on the slave's status, allocating jobs to the slave if necessary. `MASTER_FULL` helps each slave to know when not to send jobs to the master queue.

The user is responsible for coding two functions and describing the job package. The first function that must be written is the `master_create_jobs()` function which creates all of the initial jobs and places them in the central queue. The second function is the `slave_process_jobs()` function that is passed a job from the queue and is responsible for processing that job and possibly creating more jobs to be placed on the local queue. A job package is normally a C/C++ data structure that needs to be defined and subclassed by the user if it is not represented using a built-in data type.

3.3 Application with the Crossing Number Problem

For the crossing number problem, the job is simply an integer array filled in with its graph description (such as its current crossings, region list, edges needs to be added to make it complete, etc.). The layout of our job is illustrated in Figure 3.4. The `Array Size` is the number of integers in the array. The `Current Crossings` the number of crossings that this job currently has. The `Graph Size` is the number of vertices other than crossings. The `Edge List` is a list of edges to be added to the rest of the graph represented by this array. The `Region List` is a list of regions that represent the embedding of the current graph. Whenever we lay down an edge segement, a job is generated.

Figure 3.4: Job Data for the Crossing Number Problem

The `master_create_jobs()` function is quite simple. It reads data in from a file and places those jobs into the initial queue. We designed it this way so that we could easily modify the process to solve different sub-problems that will arise in our future work. The `slave_process_jobs()` function will receive a job, decode the integer array, modify the data using the algorithm presented in [13], and encode it to a new integer array as a new job.

The last major thing we had to define were special signals. In solving the crossing number problem we only needed one new signal, `MCN_UPDATE`. This signal broadcasts a better minimum crossing number that a slave has found to all of the other slaves. This signal allows every slave to throw away those jobs that were in the queue that have more crossings then the current best solution.

Chapter 4

Results

4.1 Performance of Restricted Good Drawing Algorithm and Queuing System

Table 4.1 shows a comparison of the number of partial edges laid down when building K_n for $5 \leq n \leq 8$ with and without the restricted good graph. Tests were run on 6 parallel processors. For all n in the table, the restricted good drawing tests generated graphs with the true minimum crossing number.

A substantial search space reduction resulted as can been seen by comparing results for $n = 7$. The search space is reduced by a factor of nearly 75. Using the good drawing definition alone, we were unable to generate any complete graphs for $n = 8$ after over a week of runtime. With the restricted good drawing, we fully processed all jobs in under four hours. This time reduced to just over 2 hours when running on

Vertices (n)	$\nu(K_n)$	**Good Drawing**	**Restricted Good Drawing**
5	1	3	3
6	3	203	71
7	9	1,498,775	19,979
8	18	*	46,697,854

Table 4.1: Good Drawing *versus* Restricted Edges Placed

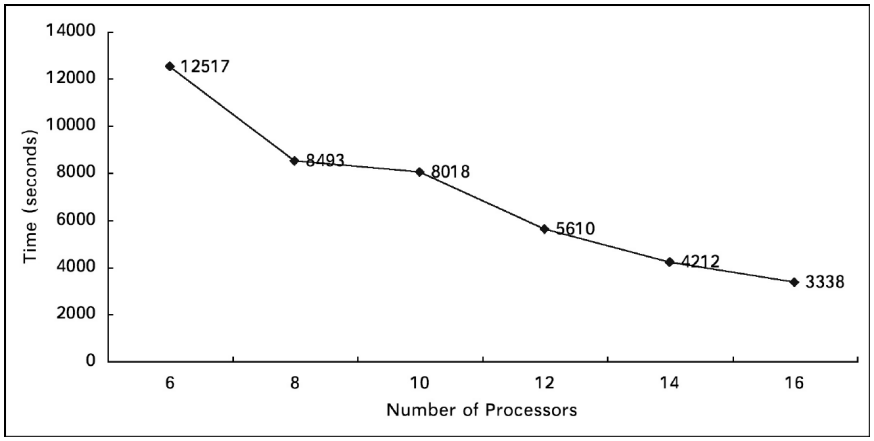

Figure 4.1: Execution Time for K_8 Run with a Number of Processors

eight processors.

Tests were run on K_9, and results for the crossing number were not achieved. These results indicate that tighter restrictions on the search space will be needed. Figure 4.1 shows execution time of our algorithm run on K_8 with a variety of processors.

The time needed to process one job varies depending on the definition of the job. The algorithm we use to calculate minimum crossings of complete graphs breaks the problem down into very small parts. Hence, a job is composed of only a couple hundred integers, which requires a trivial amount of time to process.

We pull out two test cases from Figure 4.1: K8 runs with 6 and 8 processors, and get the following two tables to briefly show the activity of the queuing system. In Tables 4.2 and 4.3, the second column is the number of jobs that each slave handled. The next three columns are the number of jobs sent from the master to the slave or the slave to the master. The master only sends jobs to the slaves when they request jobs. The slaves send jobs to the master when the slave's queue is full or when the master requests jobs so an idle slave can continue working.

	# jobs processed	# jobs master sent to slave	# jobs to master	
			slave full	master requests
Slave1	8,342,722	1,514,960	500,512	394,885
Slave 2	9,798,531	1,276,270	1,200,044	122,004
Slave 3	5,663,196	890,280	1,195,091	316,733
Slave 4	5,280,137	517,435	1,043,396	116,819
Slave 5	5,997,357	942,616	375,144	114,378
Slave 6	11,615,911	1,942,186	1,601,353	103,384
Total	46,697,854	7,083,747		

Total time taken: ≈ 3.5 hours

Table 4.2: K_8 Run with 6 Slaves

	# jobs processed	# jobs master sent to slave	# jobs to master	
			slave full	master requests
Slave 1	5,860,119	1,066,788	124,171	168,214
Slave 2	5,852,334	672,011	106,191	205,251
Slave 3	6,252,721	500,014	352,193	1,066,193
Slave 4	5,918,188	618,149	509,321	452,096
Slave 5	5,437,214	925,381	157,449	168,598
Slave 6	5,207,427	897,845	856,689	164,729
Slave 7	5,652,174	1,096,038	837,283	152,531
Slave 8	6,219,062	353,452	391,920	335,845
Total	46,399,239	6,129,678		

Total time taken: ≈ 2.4 hours

Table 4.3: K_8 Run with 8 Slaves

4.2 Discussion on Experiments and Results

Table 4.2 shows the data for K_8 run with 6 slaves, and Table 4.3 shows the same problem with 8 slaves. Figure 4.2 shows the same problem run with a variety of processing slaves. This figure shows that the more nodes used, the more evenly the jobs are distributed among the slaves. The queue can manage a large numbers of jobs

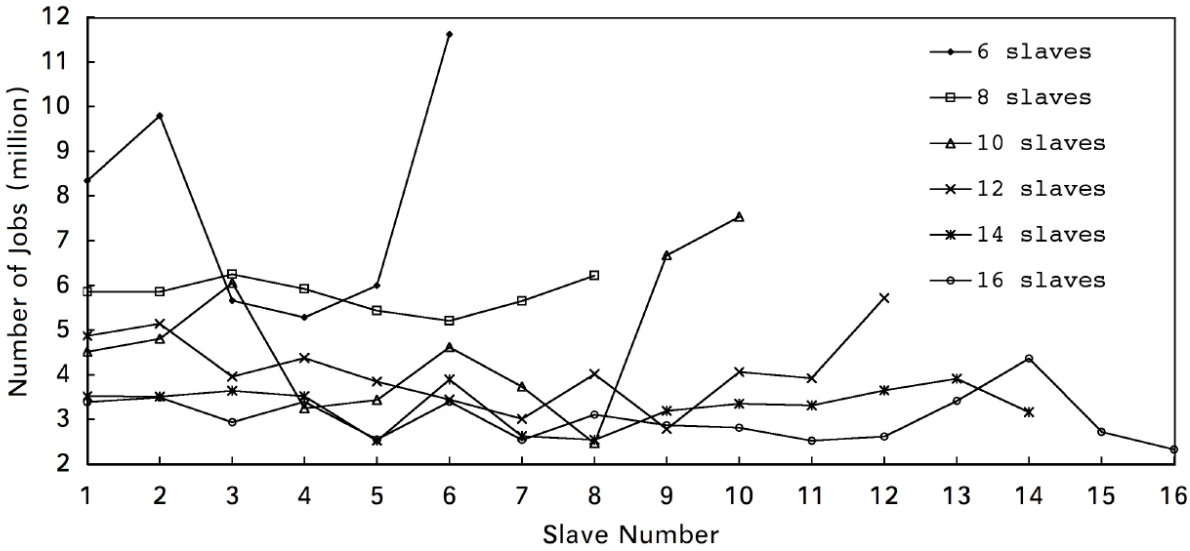

Figure 4.2: Jobs Processed by Each Slave for K_8

(up to several million) and signals while maintaining a balanced load. We also found several features that we did not expect.

1. There is a trade-off between the number of slaves to be used and the number of jobs to be processed. The total number of jobs handled by all the slaves is different even if they deal with the same vertex set. Since the processes run in parallel, the time required to generate the first best solution is roughly the same. Therefore, the more nodes we use, the bigger the total amount of jobs created.

2. A good maximum queue size selection makes a difference. Generally speaking, the machine will complete jobs faster when there are not many jobs piled in the queue taking up system resources. Each slave will send extra jobs from its local queue whenever the maximum size is reached. Most of the time, we tend to increase the number of nodes in order to expedite the processing speed. As a consequence, there will be a huge number of jobs passed back to the master

from slaves. We need to be cautious when choosing the number of nodes to be used so that the number of jobs that slaves send back will not be beyond the master machines capacity. When we ran K_8, we set the maximum size of slave queues to 1 million jobs, and the master queue size to 2 million. This works well for 5 to 16 processors. We also tested smaller and larger queue sizes and believe that this range is a good general configuration.

3. Currently the master sends one job at a time. We are unable to show whether it is more practical to send multiple jobs because the speed of dealing with one job varies as time goes by. For the crossing number problem, the time to complete a job decreases as better solutions are generated. If the master sends multiple jobs to slaves, the slaves will be kept busy and less communication time will be spent as slaves ask for fewer jobs. However, sending multiple jobs may result in a slightly unbalanced load prompting requests for more work from some slaves.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

We first presented a tightening of the definition of a good graph drawing for use with a parallel, branch-and-bound, exhaustive-search algorithm for finding the crossing number of complete graphs. The new definition examines graphs not only as sets of vertices and edges but also in terms of a set of regions. Implementation of the region restriction on the laying down of an edge during graph building has greatly reduced the search space of this problem for $n < 9$. This initial step in the direction of region examination for search space reduction is promising as it has prompted new questions related to even tighter region restrictions to be explored.

We have documented our creation of a generic queuing system for computationally intensive problems including crossing number problems. By developing a standard queuing system for use in Beowulf clusters, we have opened the door for easy adaptation of existing algorithms to solve a wide variety of problems. Using our system, costly development can be avoided. The system allows for easy tuning for optimum performance based on the computational intensity or job granularity of the chosen algorithm.

We have explained how this system works and have demonstrated its usefulness

by presenting results from the crossing number problem. These results have laid the foundation for what we envision as a fruitful path of research.

We plan to use this queuing system to solve the crossing number problem for larger graphs. Harris and Thorpe showed in [21] that the actual rectilinear crossing number diverges from the currently accepted value given by Guy in [7]. This result leads us to believe that the non-rectilinear conjectured formula is also not a tight bound. Using our queuing system as a framework to achieve load balancing, the non-rectilinear problem may be solved for K_{11} or greater. We also plan to use the queuing system to find the crossing number of a bipartite graph. Very little adaptation should be necessary to use our queuing system for this and other similar graph theory problems.

5.2 Future Work

The success of the restricted good graph for $n < 9$, using queuing system has prompted a further restriction that is currently an open question. A radical region restriction is a greedy edge placement based on the analysis of the number of regions between vertices. The question we pose: If when laying down all edges from vertex u to vertex v of graph G, can we generate only those edges that are of the minimum region separation from u to v? For example, Figure 5.1 shows four of the ten possible uv edge placements for the given graph. Figure 5.1 (a) and (b) are the only two possible uv edge placements that have the minimum number of regions (in this case two) separating u and v. The minimum number of separating regions, being two, indicates that only one crossing will be generated in each of these cases. Figure 5.1 (c) and (d) show two other uv edge placements of the remaining eight, each of which separates u and v by three regions. A three region separation will introduce two crossings. Can we generate the two shortest path edges and ignore the rest? In other

words: Will the greedy solution get the optimal result?

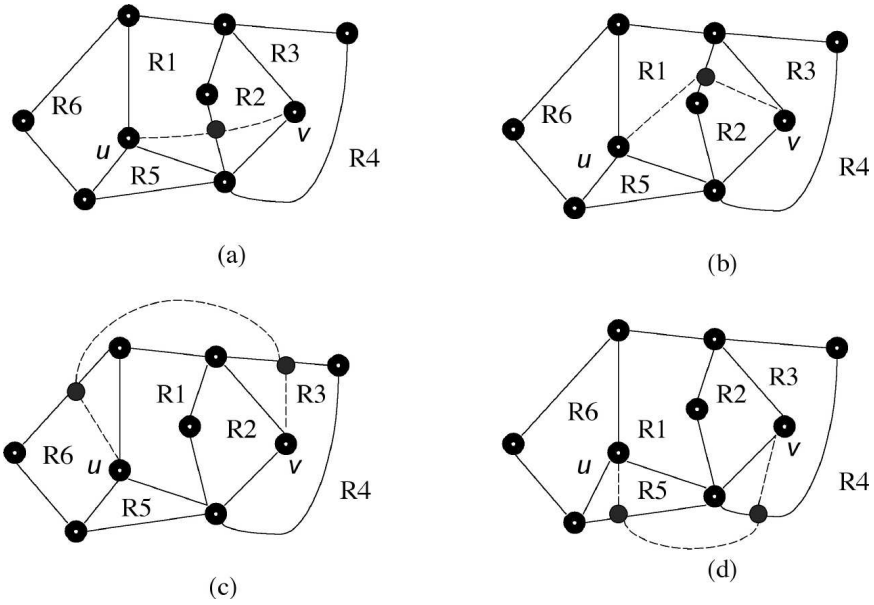

Figure 5.1: Edge Placement for Edge uv

Another issue is that of order. Does it matter in which order the edges are laid down? If order doesn't matter, we will be able to greatly prune the search tree of possible edge placements. We look to generate complete sets of crossing number graphs for small n and separate the results into non-isomorphic families. At this point, region restrictions and order questions can be tested to see if any of the families are lost. Once we know that non-isomorphic families are not missing, we can fine tune a tree pruning method that will not overlook the minimum crossing number for a graph.

Once we know that we are locating the crossing number for K_n with our pruning methods intact, we hope we can generate a counterexample to the conjecture, equation 2.1, posed by Guy [11] many years ago as the exact solution of this problem. It is anticipated that divergence may take place around $n = 12$ as it did in the rectilinear case [21].

Another avenue of exploration is trying to generate the crossing number of K_n using the non-isomorphic crossing number graph families of K_{n-1} as feeder graphs. A construction-based method for larger n may be attainable in the future.

At the same time, we hope to make the queuing system even more general by looking at the possibility of slaves being able to query other slaves and being allowed to transfer jobs directly from each other, without the use of the master as an intermediary. We plan to make queue size variables user definable at run-time rather than only at compile time.

Bibliography

[1] O. Aichholzer, F. Aurenhammer, and H. Krasser. Enumerating order types for small point sets with applications. In *Proc.* 17^{th} *Annual ACM symposium on Computational Geometry*, pages 11–18, 2001.

[2] O. Aichholzer, F. Aurenhammer, and H. Krasser. On the crossing number of complete graphs. In *Proc.* 18^{th} *Annual ACM symposium on Computational Geometry*, Barcelona, Spain, June 2002.

[3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.K. Si. Myrinet – a gigabit-per-second local area network. *IEEE Micro*, 15(1):29–38, 1995.

[4] A. Brodsky, S. Durocher, and E. Gether. The rectilinear crossing number of k10 is 62. *The Electronic J. of Combinatorics*, 8, 2001. Research Paper 23.

[5] G. Chartrand and L. Lesniak. *Graphs and Digraphs*. Chapman & Hall/CRC, Boca Raton, FL, 3rd. edition, 1996.

[6] J. Edmonds. A combinatorial representation for polyhedral surfaces. *Notices Amer. Math. Soc.*, **7**:646, 1960.

[7] P. Erdös and R. Guy. Crossing number problems. *The American Mathematical Monthly*, 80:52–58, January 1973.

[8] J. R. Fredrickson, B. Yuan, and F. C. Harris, Jr. A time-saving region restriction for calculating the crossing number of k_n. *Submitted*, May 2004.

[9] M.R. Garey and D.S. Johnson. Crossing number is NP-Complete. *SIAM J. of Alg. Disc. Meth.*, **4**:312–316, 1983.

[10] R.K. Guy. A combinatorial problem. *Nabla (Bulletin of the Malayan Mathematical Society)*, 7:68–72, 1960.

[11] R.K. Guy. Crossing numbers of graphs. In Y. Alavi, D.R. Lick, and A.T. White, editors, *Graph Theory and Applications*, pages 111–124, Berlin, 1972. Springer-Verlag.

[12] R.K. Guy, November 24, 1992. Private Communication with R.P. Pargas.

[13] F. C. Harris, Jr. and C. R. Harris. A proposed algorithm for calculating the minimum crossing number of a graph. In Yousef Alavi, Allen J. Schwenk,

and Ronald L. Graham, editors, *Proceedings of the Eighth Quadrennial International Conference on Graph Theory, Combinatorics, Algorithms, and Applications*, Kalamazoo, Michigan, June 1996. Western Michigan University.

[14] S. C. Martin. A parallel queuing system for computationally intensive problems on medium to large beowulf clusters. Master's thesis, University of Nevada, Reno, NV 89557, December 2003.

[15] J. Pach and G. Toth. Thirteen problems on crossing numbers. *Geombinatorics*, 9:225–246, 2000.

[16] J. Pach and G. Toth. Which crossing number is it, anyway? *J. Combinatorial Theory B*, 80:225–246, 2000.

[17] F. Petrini, S. C. Coll, E. Frachtenberg, and A. Hoisie. Performance evaluation of the quadrics interconnection network. *Journal of Cluster Computing*, 2002.

[18] M.J. Quinn and N. Deo. An upper bound for the speedup of parallel best-bound branch-and-bound algorithms. *BIT*, **26**(1):35–43, 1986.

[19] U. Tadjiev. Parallel computation and graphical visualization of the minmum crossing number of a graph. Master's thesis, University of Nevada, Reno, NV 89557, August 1998.

[20] U. Tadjiev and F. C. Harris, Jr. Parallel computation of the minimum crossing number of a graph. In Michael Heath, Virginia Torczon, Greg Astfalk, Petter E. Bjorstad, Alan H. Karp, Charles H. Koelbel, Vipin Kumar, Robert F. Lucas, Layne T. Watson, and David E. Womble, editors, *Proc. of the* 8^{th} *SIAM Conf. on Parallel Process. for Sci. Comput.*, Minneapolis, Minnisota, March 1997. SIAM.

[21] J. T. Thorpe and F. C. Harris, Jr. A parallel stochastic optimization algorithm for finding mappings of the rectilinear minimal crossing problem. *Ars Comb.*, **43**:135–148, 1996.

[22] P. Uthayopas, T. Aungsakul, and J. Maneesilp. Building a parallel computer from cheap pcs: Smile cluster experience. In *Proceedings of the Second Annual National Symposium on Computational Science and Engineering*, Bangkok, Thailand, 1998. National Science and Technology Development Agency.

[23] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, Upper Saddle River, NJ, 1999.

[24] K. Yelick. Programming models for irregular applications. *ACM SIGPLAN Notices*, 28(1):10, January 1993.

[25] J. Youngs. Minimal imbeddings and the genus of a graph. *J. Math. Mech.*, 12:303–315, 1963.

[26] B. Yuan, S. C. Martin, J. R. Fredrickson, and F. C. Harris, Jr. A generic queuing system for computationally intensive problems. *Submitted*, May 2004.