A Generic Queuing System for Computationally Intensive Problems

Bei Yuan, Sean C. Martin, Judith R. Fredrickson, Frederick C. Harris, Jr.
Department of Computer Science and Engineering
University of Nevada
Reno, Nevada 89557
{bei, scmartin, fredrick, fredh}@cs.unr.edu

Abstract

With the availability of inexpensive computer clusters it is now economically feasible to attack computationally intensive problems using parallel processing. This paper presents an easily adaptable parallel work queue for solving these types of problems, many of which lie in the field of graph theory. This system allows the user to focus on the problem to be solved without having to become an expert in parallel processing details (message synchronization, work load balancing, etc.). Initial results are presented for calculating the crossing number of complete graphs using this system.

Keywords: parallel, crossing number, complete graph, Beowulf cluster

1 Introduction

Many combinatorial optimization problems are NP-complete. Consequently, computational requirements for algorithms are unpredictable and sometimes intense. 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.

The algorithm Harris and Harris presented in [13] 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 processor. Using a medium to large Beowulf cluster and a parallel queuing system that can be optimized for

problem granularity, we have solved the crossing number problem for a complete graph of size less than nine. However, by harnessing the power of large computer clusters and optimizing algorithm granularity, this problem may finally be solved for larger graphs. Our queuing system has been designed to aid in solving this, and other, large computationally intensive problems.

The remainder of this paper is laid out in the following manner: Section 2 presents background information concerning the crossing number of complete graphs and parallel work queues, gives a brief introduction to Beowulf clusters, and addresses some of the work that has been done previously. Section 3 describes the proposed implementation of our queuing system. Section 4 describes the solution of the crossing number problem using our queuing system and lists the results obtained, and Section 5 finishes the paper with our conclusions and future work.

2 Background and Problem Description

2.1 The History of the Crossing Number Problem

Determining the crossing number of a graph is an important problem with applications in areas such as circuit design and network configuration [16]. It is this importance that has driven our work in finding the minimum crossing number of a graph.

Informally, the *crossing number* of a graph G, denoted $\nu(G)$, is the minimum number of crossings among all good drawings of G in the plane, where a good drawing has the following properties:

(a) No edge crosses itself,
(b) No pair of adjacent edges cross,
(c) Two edges cross at most once, and
(d) No more than two edges cross at one point.

Although the problem is easily stated and has been well studied, not much is known about the general solution. Erdös and Guy [7] compiled a survey of what was known in 1973, and Turán discussed it *via* his brick factory problem [29]. However, it was not until 1983 that Garey and Johnson [8] proved that finding the minimum crossing number of a graph is an NP-complete problem. Because of the difficulty of this problem, the focus of research turned away from finding the minimum crossing number of a graph to sub-problems and other related problems.

With regard to the sub-problems, there has been work on product graphs ranging from the product of C_n and graphs of order four [2] to $C_3 \times C_n$ [24], $C_4 \times C_4$ [6], $C_5 \times C_5$ [23], and $C_5 \times C_n$ [15]. Results have also been published for the bipartite graphs $K_{3,n}$ [22], and $K_{5,n}$ [14], and the torroidal crossing

number of $K_{m,n}$[11] has been found. In 1991 Beinstock [3] related the crossing number of a graph to the arrangement of pseudolines, a topic well studied by combinatorialists.

There have also been several related problems that have been studied over the years. These range from the rectilinear crossing problem [1, 10, 27, 28] to the maximum crossing number of a graph or subgraph [9, 12] to the thrackle conjecture [5]. Pach and Tóth [18] 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 [19].

2.2 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 [21]. 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 [32].

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. 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 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 [4] and QsNet [20] 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 [30]:

- 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. Cluster computing can address a wider range of applications than previously thought for low cost [17].

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 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.

2.4 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]. 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 [25, 26]. As we mentioned earlier, 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 1. In this example, processors 2 and 3 sit idle while processor 1 works toward a solution.

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.

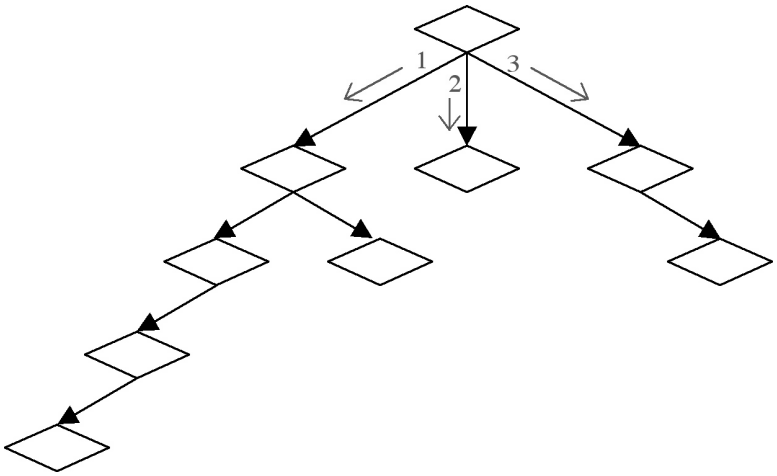

Figure 1: Unbalanced Search Tree

3 The Generic Queuing System

Many computationally expensive problems have had parallel algorithms either proposed or implemented, which makes it easy to break problems into jobs (or sub-jobs). Therefore, we decided to build a queue of jobs (work) that can be distributed across a cluster to harness the parallel com-

putation 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 paper 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.

3.1 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

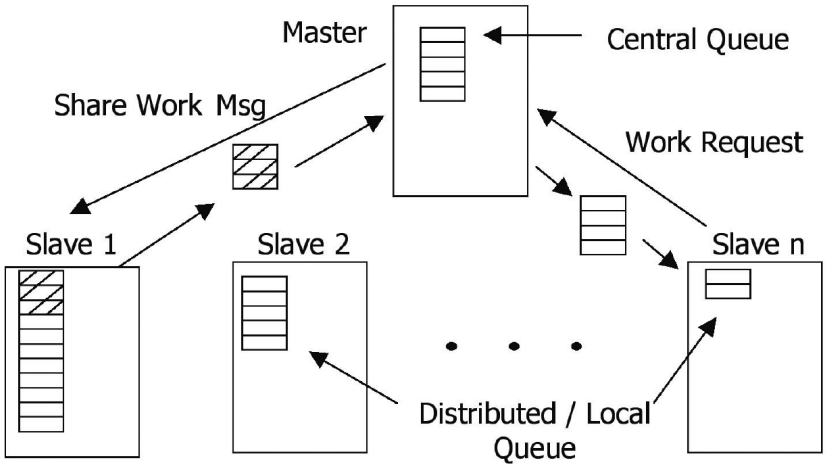

Figure 2: Parallel Work Queue

parallel efficiency. The size of each job, or the amount of work they contain, 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 [31].

In our proposed implementation, which is illustrated in Figure 2, 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 then 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 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 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()` 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 overloaded by the user if it is not represented using a built-in data type.

3.2 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. 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.

Figure 3: 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 subproblems that will arise in our future work. The `slave()` 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 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.

4 Results

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 thousand integers, which requires a trivial amount of time to process.

In Tables 1 and 2, 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 1: 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 2: K_8 Run with 8 Slaves

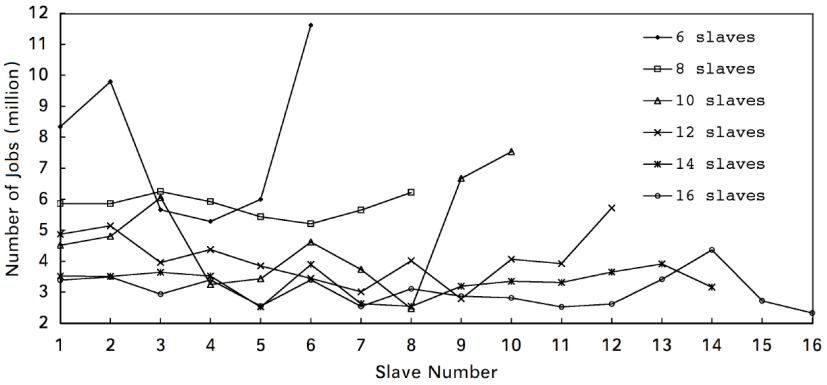

Figure 4: Jobs Processed by Each Slave for K_8

Table 1 shows the data for K_8 run with 6 slaves, and Table 2 shows the same problem with 8 slaves. Figure 4 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 (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.

5 Conclusions and Future Work

In this paper we have documented our creation of a generic queuing system for computationally intensive 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 [28] 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.

In future versions, we want to make the 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. This will possibly be implemented to include a graphical user interface developed using an open source, readily available library such as GTK.

References

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

[2] L. Beineke and R.D. Ringeisen. On the crossing number of products of cycles and graphs of order four. *J. Graph Theory*, **4**:145–155, 1980.

[3] D. Bienstock. Some provably hard crossing number problems. *Disc. Comput. Geom.*, **6**:443–459, 1991.

[4] 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.

[5] J.E. Cottingham. *Thrackles, Surfaces, and Maximum Drawing of Graphs.* PhD thesis, Clemson, University, Clemson, SC 29634, August 1993.

[6] A.M. Dean and R.B. Richter. The crossing number of $C_4 \times C_4$. *J. Graph Theory*, **19**(1):125–129, 1995.

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

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

[9] R. Guy, H. Harborth, B. Piazza, R. Ringeisen, and S. Stueckle. Crossings in the complete graph. preprint.

[10] 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.

[11] R.K. Guy and T.A. Jenkyns. The torroidal crossing number of $K_{m,n}$. *J. Combin. Theory*, **6**:235–250, 1969.

[12] F. Harary, P.C. Kainen, and A.J. Schwenk. Torroidal graphs with arbitrarily high crossing numbers. *Nanta Math.*, **6**:58–67, 1973.

[13] Frederick C. Harris, Jr. and Cynthia 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] D.J. Kleitman. The crossing number of $K_{5,n}$. *J. Combin. Theory*, **9**:315–323, 1971.

[15] M. Klesc, R.B. Richter, and I. Stobert. The crossing number of $C_5 \times C_n$. *J. Graph Theory*, **22**(3):239–243, 1996.

[16] F.T. Leighton. *Complexity Issues in VLSI.* MIT Press, Cambridge, MA, 1983.

[17] W. B. Ligon and R. B. Ross. Implementation and performance of a parallel file system for high performance distributed applications. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 471–480. IEEE Computer Society Press, August 1996.

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

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

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

[21] 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.

[22] R.B. Richter and J. Siran. The crossing number of $K_{3,n}$ in a surface. *J. Graph Theory*, **21**(1):51–54, 1996.

[23] R.B. Richter and C. Thomassen. Intersection of curve systems and the crossing number of $C_5 \times C_5$. *Discrete Comput. Geom.*, **13**:149–159, 1995.

[24] R.D. Ringeisen and L.W. Beineke. The crossing number of $C_3 \times C_n$. *J. Combin. Theory, Ser. B*, **24**:134–136, 1978.

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

[26] Umid Tadjiev and Frederick 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.

[27] C. Thomassen. Rectilinear drawings of graphs. *J. Graph Theory*, **12**:335–341, 1988.

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

[29] P. Turan. A note of welcome. *J. Graph Theory*, **1**:7–9, 1977.

[30] 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.

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

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