University of Nevada
Reno

A Parallel Queuing System for Computationally Intensive Problems on Medium to Large Beowulf Clusters

A professional paper submitted in partial fulfillment of the
requirements for the degree of Master of Science
with a major in Computer Engineering

by

Sean Christopher Martin

Dr. Frederick C. Harris, Jr., Advisor

December 2003

Dedication

To my wife Julie, for all her patience and love.

Acknowledgements

I would like to thank my advisor, Dr. Harris for the huge amount of help he has given me. Fred, I couldn't have done it without you!

I would also like to thank Umid Tadjiev for his version of Minimum Crossing Number code and Bei Yuan and Judy Fredrickson for their hurried, 11th hour work on the Minimum Crossing Number problem.

My appreciation goes out to Dr. Egbert and Dr. Leone for their role as committee members.

Finally, I want to give a big capital THANK YOU to the Computer Science department at the University of Nevada, Reno for getting me through my "first" graduate degree.

Abstract

Large groups of networked workstations, commonly referred to as Beowulf clusters, require a systematic approach to load balancing. Many applications require extensive message passing and synchronization to take full advantage of the available processing power. We have endeavored to simplify this task by developing a generic queuing system that can be adapted to different applications. This system is particularly suited to graph theory problems, many of which require a high ratio of computation to message passing. We have used the queuing system to solve a common graph theory problem, finding the Minimum Crossing Number of a complete graph.

Table of Contents

List of Figures

Chapter 1 Introduction

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. Work queues 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. By developing a generic work queuing system for medium to large Beowulf clusters, we attempt to simplify this problem.

The algorithm Harris and Harris presented in [1] uses an exhaustive search to find the Minimum Crossing Number (MCN) of a graph. As problem size grows, 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 MCN of a complete graph for a number of vertices less than nine. The MCN of complete graphs with vertices numbering less than 10 was conjectured by Guy in [2] and has been proven in [10], allowing us to verify our results. The MCN for larger vertex sets has not been proven, but by harnessing the power of large computer clusters, this problem may finally be solved. 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: Chapter 2 describes background information concerning parallel load balancing, work queues, and introductory graph theory concepts. It describes the Minimum Crossing Number of a complete graph and a generic work queuing system. Chapter 2 also includes descriptions of branch and bound algorithms, breadth first and depth first searches and a brief introduction to Beowulf clusters. Chapter 3 relates the proposed implementation of our queuing system. Chapter 4 describes the “job” used to solve the Minimum Crossing Number Problem and the Traveling Salesman Problem in our queuing system and lists the results obtained, and Chapter 5 finishes the paper with our conclusions and future work.

Chapter 2 Background and Literature Review

Problems such as the Minimum Crossing Number of a complete graph are computationally intensive and virtually impossible to solve using a conventional single processor machine. Fortunately, these 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 each processor tackling only a portion of the overall problem applies regardless of size. In this chapter we will begin with a look at work queues in Section 2.1, parallel queues in Section 2.2, branch and bound algorithms, breadth-first and depth-first searches in Section 2.3, graphs and the Minimum Crossing Number of a graph in Section 2.4 and Beowulf clusters in Section 2.5.

2.1 Work Queues

A work queue is one method of ensuring a balanced work load that is evenly distributed across many processors or machines. 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 [3].

In centralized load balancing, the tasks to be performed are held by the master and meted out to the slaves as they finish other tasks and become idle. This process minimizes the time each slave is idle, thereby maximizing efficiency. A variation of this strategy involves the distribution of non-homogenous tasks. These tasks are usually arranged in descending order of size and/or complexity. This approach allows some slaves to work on smaller tasks (deeper in the queue) while waiting for the early larger tasks to complete. The wisdom in this ordering is readily apparent: if a large or time consuming task is left for last, processes could sit idle while the last task is computing. One disadvantage of centralized load balancing is the possibility of a bottleneck while the master distributes tasks—many slaves may request tasks, but the master can only issue one at a time.

In decentralized load balancing, local processes 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 attendant problems. In more complex systems, the slaves may request work from each other or from a centralized master queue.

The difficulty in parallelizing most algorithms lies in dividing the work to ensure maximum concurrency and synchronizing the processes to achieve simultaneous termination. There exist several ways to implement dynamic load balancing in a multi-processor system. One of the most common methods is the work pool or processor farm. These constructs are referred to as a queue and have the First-In, First-Out characteristic common to all types of queues. These queues often require large amounts of message passing as slave processes request jobs from the queue, making the difficult task of proper synchronization very important. Our generic work queuing system is an attempt to simplify the problem stated above.

2.2 Parallel Work Queue

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 that data set. This is referred to as static partitioning. If one processor finishes working it must wait idle while the other processors "catch up." When a processor is idle, the benefits of concurrent execution are not realized to their entirety. In addition, heterogeneous processor clusters can make effective partitioning very difficult. Static partitioning is effective when data sets are known prior to runtime and the execution time of the data can be easily determined. This is seldom the case, however. One solution is dynamic load balancing.

2.2.1 Eager Task Creation

A central work queue is an effective tool in load balancing during distributed, concurrent execution. When each process becomes idle, it can query a master process and remove the next job from the queue. This approach is referred to in [4] as a form of eager task creation (ETC). But, by using a central work queue there exists the likelihood of a bottleneck when removing jobs from the queue. Additionally, if two or more processes attempt to access the queue simultaneously, data may be corrupted and the possibility of a resultant race condition or deadlock may require additional overhead to ensure mutual exclusion (*e.g.*, a semaphore). By using local work queues in addition to a central one, these problems are largely avoided.

2.2.2 Lazy Task Creation

A task with too fine a grain can create a suboptimum work distribution when utilizing eager task creation. One solution presented by Mohr, *et al.* in [5] is referred to as lazy task creation (LTC). Initial work in [5] focuses on a mechanism they call breadth-first until saturation, then depth-first (BUSD). BUSD uses an initial breadth-first task creation until all processors are busy and then a depth-first execution and further task creation. This method can also utilize a run-time construct called task stealing. When one task is idle it may "steal" a task from another processor, thus helping to balance the workload.

2.3 Depth-first Search, Breadth-first Search and Branch and Bound

In many cases, a search space may be mapped to a tree structure. In such a tree, nodes further from the root represent a more complete possible solution. Each leaf node signifies a complete solution. Two methods used to search trees for solutions are depth-first search (DFS) and breadth-first search (BFS).

In a depth-first search, the search traverses the tree until a complete solution, a leaf node, or a dead-end (*i.e.*, no solution) is reached. The search then returns to the node closest to the root that has an alternate path and again moves down the tree toward a solution.

In a breadth-first search, the search will visit every node adjacent to the root and then move to the next level and visit every node adjacent to every node at that level. All possible solutions are visited when the search completes searching all leaf nodes.

Branch and bound refers to the process of using a known solution to prune branches from the search tree. Once one solution is found, that value becomes the bound, the yardstick by which all other solutions are measured. It is also possible to use a predetermined good solution. If, during a search, the partial solution has exceeded the bound, that branch of the search tree is abandoned. This technique is particularly effective in an exhaustive search when the initial search space is very large. By eliminating branches as close to the root as possible, this method can significantly reduce the actual number nodes searched.

2.4 Graphs

A graph is an ordered pair $G = (V, E)$, where V is a finite, non-empty set of objects called vertices, and E is a (possibly empty) set of unordered pairs of distinct vertices (*i.e.*, 2-subsets of V) called edges. The set V [or $V(G)$ to emphasize that it belongs to the graph G] is called the vertex set of G, and E [or $E(G)$ to emphasize as above] is called the edge set of G. These sets must conform to the following rules as given in [6]:

- no edge contains a vertex other than its endpoints
- no two adjacent edges share a point other than their common endpoint
- two nonadjacent edges share at most one point at which they cross transversally
- and no three edges cross at the same point

If $e = \{u, v\} \in E(G)$, we say that vertices u and v are adjacent in G and that e joins u and v. We also say that u and v are the ends of e. The edge e is said to be incident with u (and v), and vice versa. We write uv (or vu) to denote the edge $\{u,v\}$ with the understanding that no order is implied [6]. Note that in Figure 2.1 the graph G has sets V and E as follows:

$V = \{a, b, c, d, e\}$

$E = \{ab, bc, cd, de, ec, bd, ac\}$

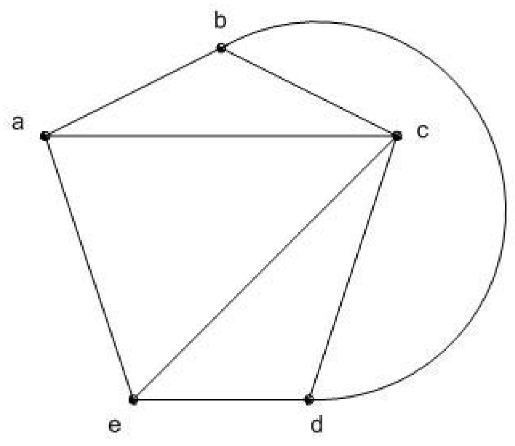


Figure 2.1: Graph G

2.4.1 Complete graphs

A complete graph (denoted K_n) is a graph with n vertices in which each vertex is connected to each of the others (with one edge between each pair of vertices). Thus in a complete graph all vertices have an edge incident to every other vertex. The graph in Figure 2.1 is not a complete graph because it is missing the edges (*ad*) and (*eb*). Figure 2.2 is a complete graph on five vertices (K_5).

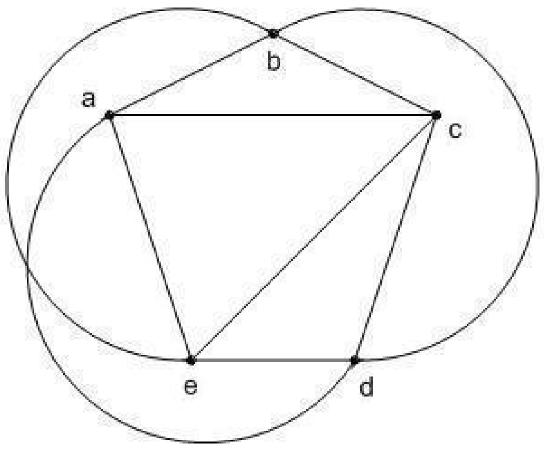


Figure 2.2: Complete Graph (K_5)

2.4.2 Planar graphs

A graph is considered planar if it can be drawn on a plane and the edges intersect only at their ends. A drawing of a graph on a plane is called a planar embedding of that graph. A planar embedding divides a graph into connected regions. It is important to note that one of the regions lies outside all edges and is infinite [6]. Figure 2.2 represents a non-planar embedding of K_5. It is worth noting that K_5 is not a planar graph.

2.4.3 Bipartite graphs

A graph is bipartite if its vertices can be partitioned into two disjoint subsets U and V such that each edge connects a vertex from U to one from V. A bipartite graph is a complete bipartite graph if every vertex in U is connected to every vertex in V.

Figure 2.3 shows a complete bipartite graph with 3 vertices in set *U* and 3 in set *V*. This graph is denoted $K_{3,3}$.

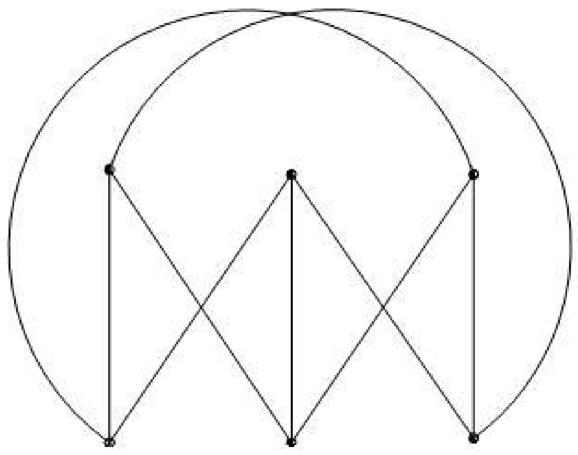


Figure 2.3: Complete Bipartite graph ($K_{3,3}$)

2.4.4 Crossing Number

The crossing number of a graph refers to the number of times its edges cross. It has been conjectured by Guy in [7] that the following formula holds true for the minimum crossing number of a complete bipartite graph.

$$Cr(n,m) = \left(\frac{n}{2}\right)\left(\frac{n-1}{2}\right)\left(\frac{m}{2}\right)\left(\frac{m-1}{2}\right)$$

where $[r]$ denotes the integer part of r

Figure 2.4 shows a complete bipartite graph ($K_{3,3}$) with the minimum number of crossings.

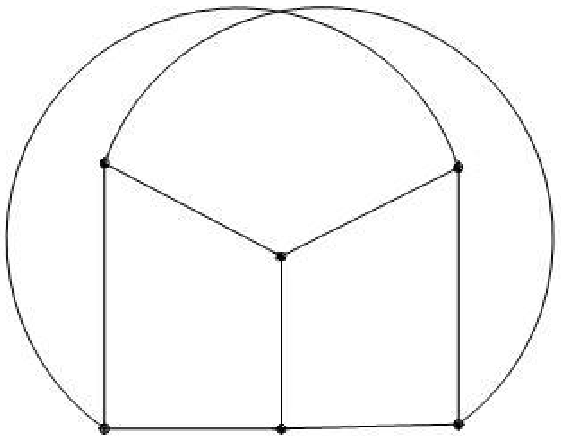


Figure 2.4: ($K_{3,3}$) with Minimum Number of Crossings

2.4.5 Minimum Crossing Number Problem

Paul Turan was the first to recognize the minimum crossing number problem for what it is today. The following anecdote from [8] relates his discovery:

> In July 1944 the danger of deportation was real in Budapest, and a reality outside Budapest. 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 rather precious to all of us (for reasons not to be discussed here). 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?

As a result, this problem has become informally known as "Turan's Brickyard Problem."

Calculating the MCN of a complete graph is a problem with the following characteristics:

- every vertex must have an edge incident to every other vertex
- the graph must be planar
- no edge crosses itself
- no pair of adjacent edges cross
- two edges cross at most once
- no more than two edges cross at one point

This problem was proved to be NP-complete by Garey and Johnson in [10]. Guy and Erdös presented a survey of the minimum crossing number of several families of graph including K$_n$ in [9]. The accepted value (v) for the minimum crossing number of a member of the family of complete graphs (K$_n$) was conjectured by Richard Guy in [2] and is given by the formula:

$$v(K_n) = \frac{\left(\frac{n}{2}\right)\left(\frac{n-1}{2}\right)\left(\frac{n-2}{2}\right)\left(\frac{n-3}{2}\right)}{4}$$

Where n is the number of vertices of the complete graph. This has been proven for all $n \leq 10$ [10].

2.5 Beowulf Clusters

Thomas Lawerence Sterling begins the introduction of his computer cluster building and maintenance guide with the following:

> Beowulf was the legendary sixth-century hero from a distant realm who freed the Danes of Heorot by destroying the oppressive monster Grendel. As a metaphor, "Beowulf" has been applied to a new strategy in high performance computing that exploits mass-market technologies to overcome the oppressive costs in time and money of supercomputing, thus freeing scientists, engineers, and others to devote themselves to their respective disciplines [11].

Beowulf clusters are groups of linked workstations that can be used to solve computationally intensive problems. By using single, relatively cheap workstations, these clusters are both economical and scalable.

The Beowulf Project was developed at NASA's Goddard Space Flight Center by Thomas Sterling and Donald Becker in the summer of 1994. Because of the availability of low cost PC workstations a 16 node cluster was built for less than the target of $50,000 [12]. The project was driven by the need for high performance workstations in the Earth and space sciences community [13]. The original benchmark goal for the project was one Gigaflops, or one thousand million (10^9) floating point operations per second. This was achieved in 1996 using 16 100 MHz Intel processors.

A Beowulf cluster uses a single computer as a node; these nodes are interconnected using a variety of networking configurations, utilizing Ethernet for their interconnectedness. Messaging is managed through the TCP/IP protocol stack. The nodes usually run a UNIX-based operating system such as Linux [11]. Linux is an open source operating system under the GNU standard, allowing users to change components as necessary for different applications. This adaptability, as well as the inclusion of the TCP/IP protocol, made Linux particularly suited to use as an operating system in the Beowulf Project [12].

The Beowulf architecture emphasizes several key hardware and networking attributes. The system must be built from mass market commodity components; it must have dedicated processors, rather than borrowing processing from idle workstations and should have its own private system area network [12]. By using consumer products, these clusters were developed to benefit from competition in the PC markets. The designers intended for the cluster nodes to use free operating systems and distributed computing tools whenever possible, thereby significantly cutting costs. Their main goal was high performance at an affordable price.

In effect, a multi-node cluster acts as a supercomputer, giving large amounts of processing power for a fraction of the cost of traditional multi-processor systems. By utilizing current high performance processors and networking standards, new Beowulf clusters have taken multi-processor computation to new heights of performance. The Beowulf architecture gives anyone with a need for high processing power a relatively inexpensive, easy to construct computing option.

Chapter 3 Proposed Implementation

We have devised a parallel queuing system utilizing some aspects of both ETC and LTC. This system is being used and will be used to solve computationally intensive problems, particularly those involving graph theory. The system has been used to solve the MCN problem for a complete graph. Message synchronization and load balancing is achieved using a combination local and centralized queuing system. In this case, these jobs can also spawn other jobs. It is this property that the system of central and distributed work queues is designed to exploit.

We began with an existing implementation of an algorithm to find the MCN of a complete planar graph given by Harris in [1]. This implementation had also been parallelized using static partitioning by Tadjiev in [14, 15]. As mentioned in the previous chapter, static partitioning generally does nothing to ensure load balancing. Very little speedup was noticed in solving larger vertex sets, even when the method was run on many processors. Frequently, only several hours into a many-day run, one processor would be working while the others sat idle. By utilizing a generic work queuing system, the MCN and other problems could be solved with a minimum of adaptation.

In the rest of this chapter we will explain our implementation, describe where it fits in this taxonomy, and give a detailed overview of our implementation.

3.1 The Generic Queuing System

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 a job. This job represents whatever amount of work can be executed independently. Thus, the system is easily tunable by the user to ensure good parallel efficiency. This amount of work is referred to as the granularity of a task. Coarse granularity refers to a task with a large amount of sequential instructions that take a significant amount of time to execute [16]. In this system, tasks with a finer granularity take better advantage of the central and distributed queues. An example job is given in Chapter 4 for the MCN problem.

Many system variables are user definable, including: the maximum queue size of both the local and central queues, the amount of work reserved by the slaves when sending a packed queue, and the number of jobs created and enqueued by the master prior to concurrent execution. These variables allow the queuing system to be adapted to a variety of situations and granularities.

The normally high communication overhead of fine-grained tasks is minimized by the use of the local queue. In addition, the queue sizes may be tuned to allow coarse grained tasks to benefit from the central and distributed queue system.

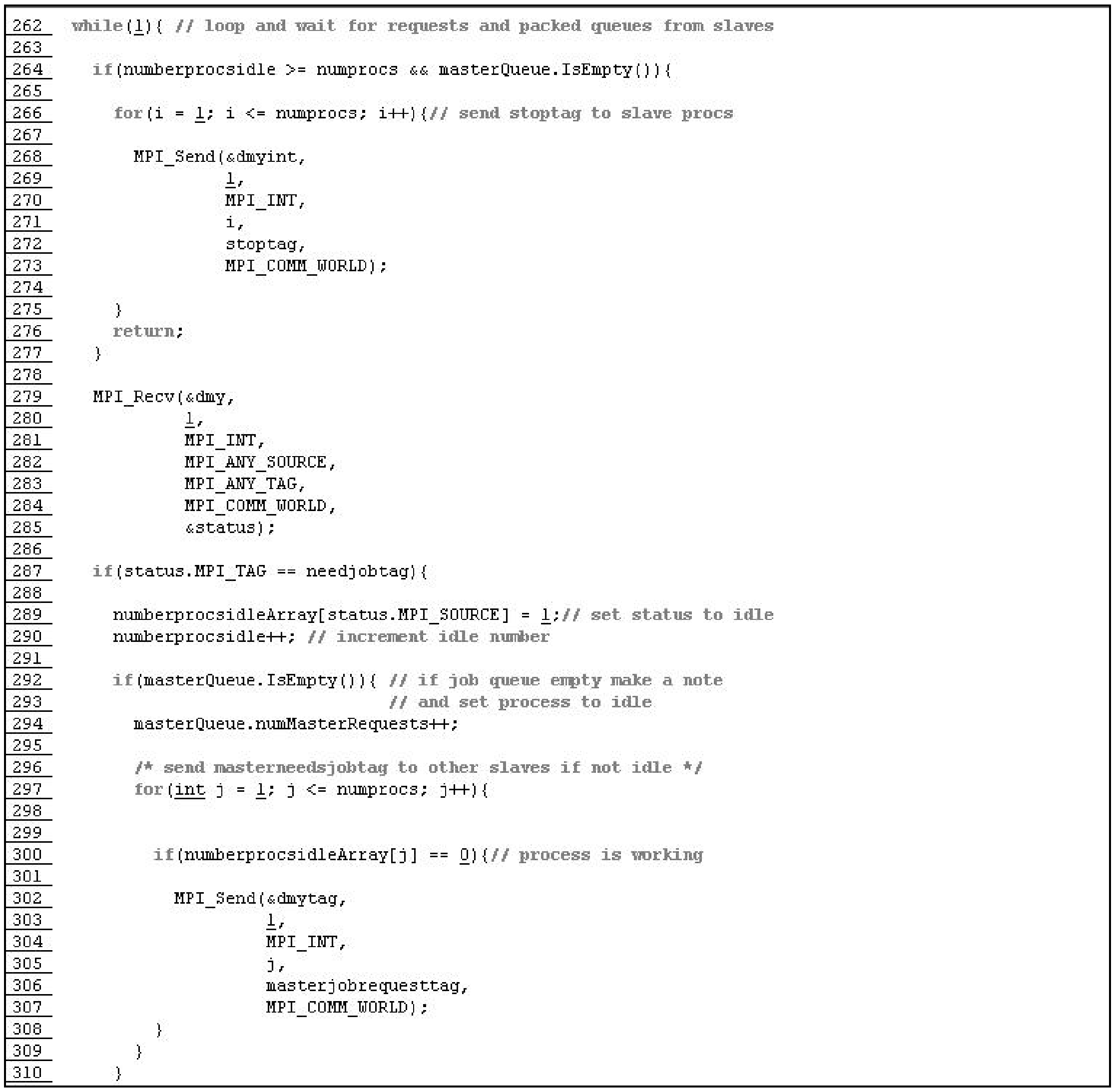
3.2 Implementation Overview

In our proposed implementation, the master will create the first *m* jobs. In general, *m* is determined by the problem under investigation and should be greater than *n*, where *n* is the number of slave processors. These jobs are put in the master's queue. The master then sends a job to each processor. From there, each processor will create more nodes for the tree while working. These jobs are kept in a local work queue by each slave. When a slave's local work queue reaches some user defined size, the queue is packed (reserving a predetermined number of jobs to continue working) and sent back to the master for placement in the master 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. If a slave is idle and the master queue is empty, the idle slave "steals" a job from another slave *via* the master. Process termination occurs when all slaves are idle and the master's work queue is empty.

3.2.1 The Master Process

The master creates enough jobs to fill its queue and provide a job for each slave. It then sends a job to each slave and waits in an idle loop for messages. These messages can be one of several types: a request for a job, a packed-up job queue from a slave whose local queue was overflowing, or an update of a new best solution. Upon receiving a request for a job, the master will dequeue from its local queue, provided it is not empty, and send the job to the requesting slave. If the master's queue is empty, it will send a request to a portion of the working processes to pack and send a portion of their queues to the master. The master only requests jobs from a portion of the working slaves to avoid communication bottleneck. By using a round-robin approach each portion of the slaves will be polled in turn for available jobs. This portion is user definable based on the number of slaves and the task granularity. By gathering available jobs in this fashion, the master's queue is adequately stocked, removing the need for frequent task stealing. Figure 3.1 shows the loop that receives messages from the slaves (lines 262+).

Depending on the value of the received message, the master either sends a new job to the slave or prepares to receive a packed job queue to enqueue locally.

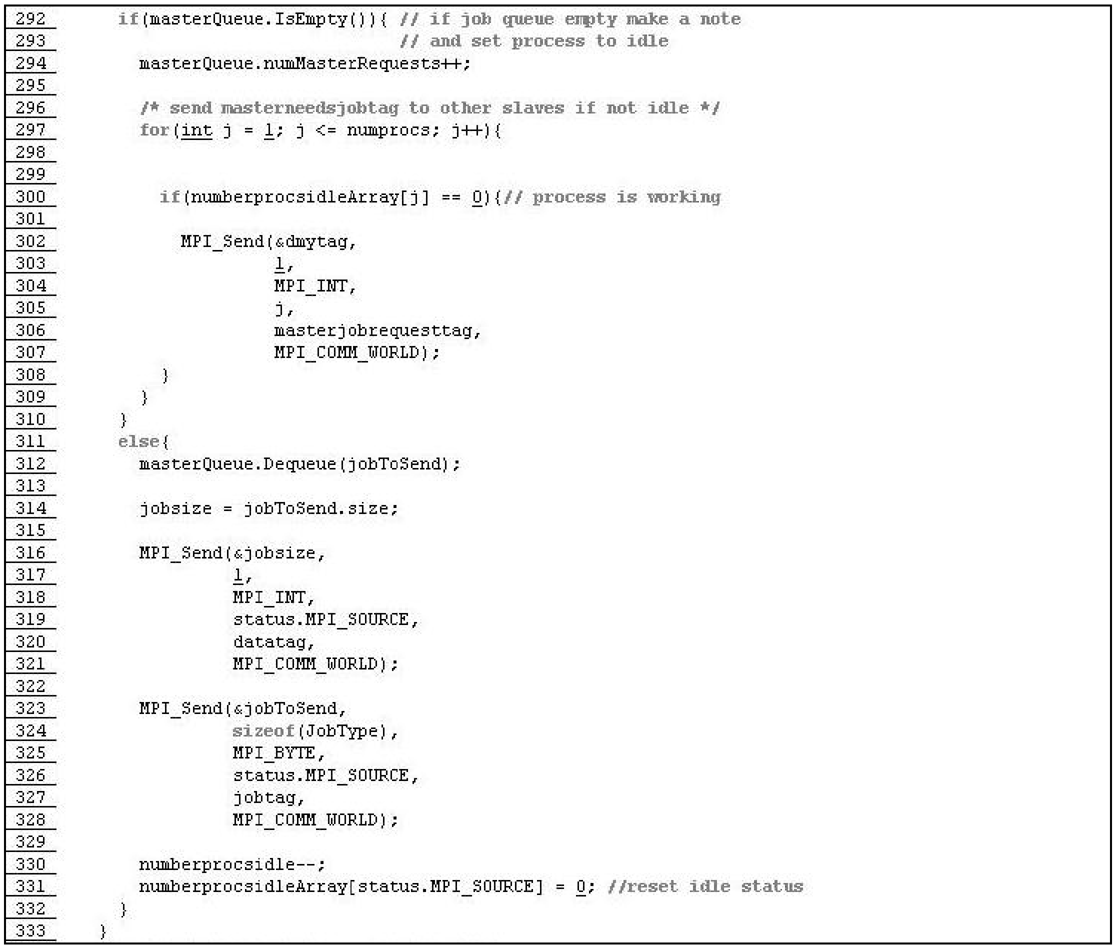


```
262   while(1){ // loop and wait for requests and packed queues from slaves
263
264     if(numberprocsidle >= numprocs && masterQueue.IsEmpty()){
265
266       for(i = 1; i <= numprocs; i++){// send stoptag to slave procs
267
268         MPI_Send(&dmyint,
269                  1,
270                  MPI_INT,
271                  i,
272                  stoptag,
273                  MPI_COMM_WORLD);
274
275       }
276       return;
277     }
278
279     MPI_Recv(&dmy,
280              1,
281              MPI_INT,
282              MPI_ANY_SOURCE,
283              MPI_ANY_TAG,
284              MPI_COMM_WORLD,
285              &status);
286
287     if(status.MPI_TAG == needjobtag){
288
289       numberprocsidleArray[status.MPI_SOURCE] = 1;// set status to idle
290       numberprocsidle++; // increment idle number
291
292       if(masterQueue.IsEmpty()){ // if job queue empty make a note
293                                  // and set process to idle
294         masterQueue.numMasterRequests++;
295
296         /* send masterneedsjobtag to other slaves if not idle */
297         for(int j = 1; j <= numprocs; j++){
298
299
300           if(numberprocsidleArray[j] == 0){// process is working
301
302             MPI_Send(&dmytag,
303                      1,
304                      MPI_INT,
305                      j,
306                      masterjobrequesttag,
307                      MPI_COMM_WORLD);
308           }
309         }
310       }
```

Figure 3.1 Master Code

In Figure 3.2, line 292, the master checks to see if it has jobs available in its queue to send to a slave. If no jobs are available, the master will make a note of which slaves are idle and broadcast a “needjob” message to the slaves who are still working. The frequency of task stealing can be tuned by altering the min_queue_size variable. The slaves will not respond to a needjob query if their queue is smaller than this size. If a slave requests a job and the master’s queue is not empty, the master simply dequeues a job and sends it to that slave.

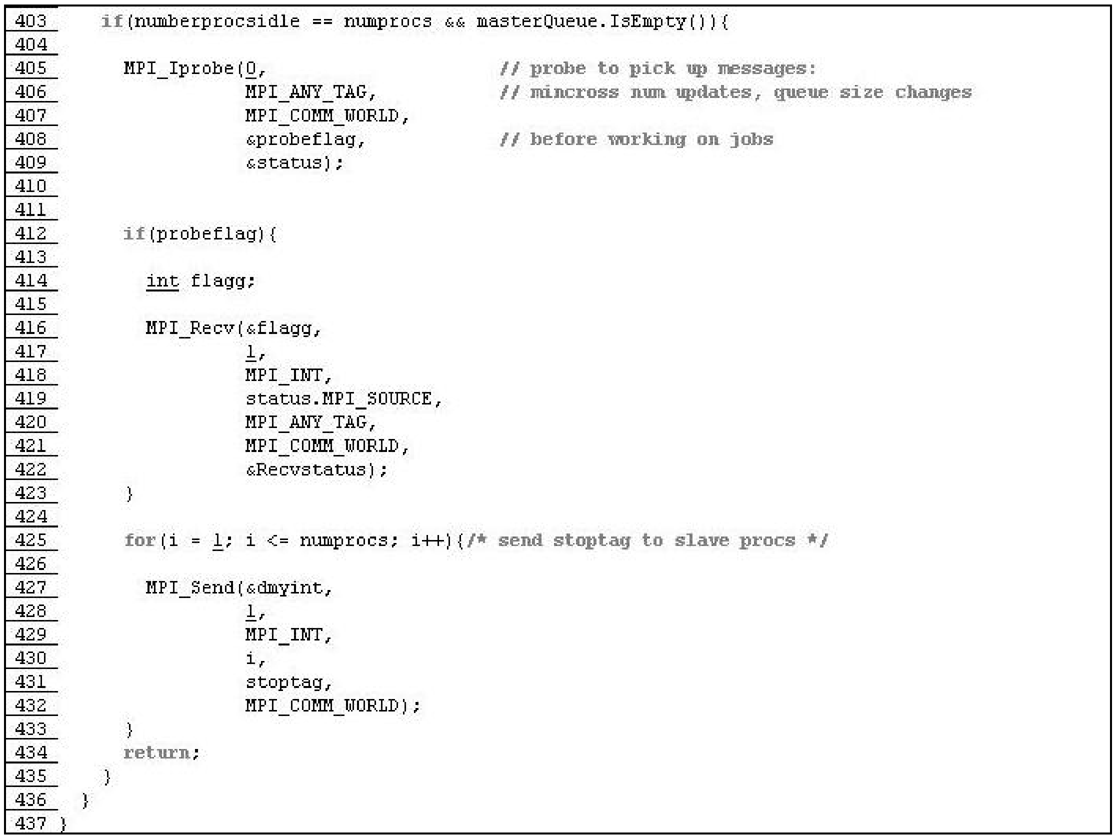
If the master receives a best solution update, it changes its local best solution variable and then sends the new value to the slaves. This approach allows the system to take full advantage of a branch-and-bound algorithm.



```
  if(masterQueue.IsEmpty()){ // if job queue empty make a note
                             // and set process to idle
    masterQueue.numMasterRequests++;

    /* send masterneedsjobtag to other slaves if not idle */
    for(int j = 1; j <= numprocs; j++){


      if(numberprocsidleArray[j] == 0){// process is working

        MPI_Send(&dmytag,
                 1,
                 MPI_INT,
                 j,
                 masterjobrequesttag,
                 MPI_COMM_WORLD);
      }
    }
  }
  else{
    masterQueue.Dequeue(jobToSend);

    jobsize = jobToSend.size;

    MPI_Send(&jobsize,
             1,
             MPI_INT,
             status.MPI_SOURCE,
             datatag,
             MPI_COMM_WORLD);

    MPI_Send(&jobToSend,
             sizeof(JobType),
             MPI_BYTE,
             status.MPI_SOURCE,
             jobtag,
             MPI_COMM_WORLD);

    numberprocsidle--;
    numberprocsidleArray[status.MPI_SOURCE] = 0; //reset idle status
  }
}
```

Figure 3.2: Master Code Part 2

Figure 3.3, lines 403-434, show the termination routine. If all processes are idle and the master queue is empty, the master sends a stop message to all slaves and then returns. All processes will return upon receipt of this message and results, if any, will be output prior to program termination.

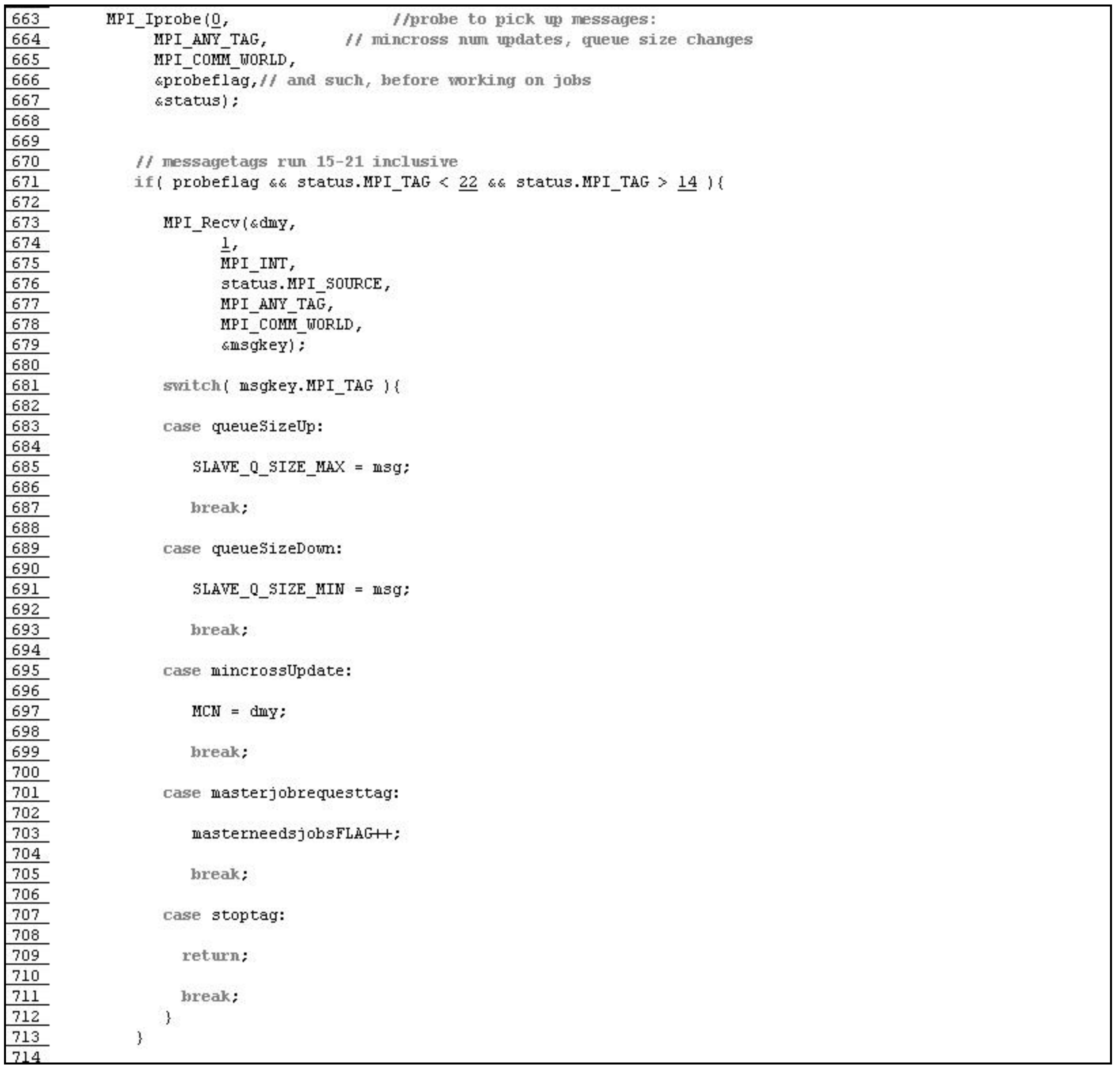


```
403     if(numberprocsidle == numprocs && masterQueue.IsEmpty()){
404
405       MPI_Iprobe(0,                     // probe to pick up messages:
406                  MPI_ANY_TAG,           // mincross num updates, queue size changes
407                  MPI_COMM_WORLD,
408                  &probeflag,            // before working on jobs
409                  &status);
410
411
412       if(probeflag){
413
414         int flagg;
415
416         MPI_Recv(&flagg,
417                  1,
418                  MPI_INT,
419                  status.MPI_SOURCE,
420                  MPI_ANY_TAG,
421                  MPI_COMM_WORLD,
422                  &Recvstatus);
423       }
424
425       for(i = 1; i <= numprocs; i++){/* send stoptag to slave procs */
426
427         MPI_Send(&dmyint,
428                  1,
429                  MPI_INT,
430                  i,
431                  stoptag,
432                  MPI_COMM_WORLD);
433       }
434       return;
435     }
436   }
437 }
```

Figure 3.3 Master Code Part 3

3.2.2 The Slave Process

Each slave process requests an initial job from the master, receives it, and begins processing. If new jobs are created, they are put into the slave's local queue. When the current job is finished, the slave dequeues a job from the local queue and continues to process. If the slave's queue reaches a maximum size or the master requests jobs from the slave, the queue is packed and sent to the master. A small portion of the jobs are kept in the local queue for the slave to continue working. This portion is user definable. The slave terminates upon receiving a stop message from the master. Figure 3.4 shows the variety of messages the slave may receive from the master. This switch construct (line 681) is easily altered because additional messages for expanded functionality can be added by the user.



```
663      MPI_Iprobe(0,                //probe to pick up messages:
664          MPI_ANY_TAG,         // mincross num updates, queue size changes
665          MPI_COMM_WORLD,
666          &probeflag,// and such, before working on jobs
667          &status);
668
669
670        // messagetags run 15-21 inclusive
671        if( probeflag && status.MPI_TAG < 22 && status.MPI_TAG > 14 ){
672
673          MPI_Recv(&dmy,
674               1,
675               MPI_INT,
676               status.MPI_SOURCE,
677               MPI_ANY_TAG,
678               MPI_COMM_WORLD,
679               &msgkey);
680
681          switch( msgkey.MPI_TAG ){
682
683          case queueSizeUp:
684
685             SLAVE_Q_SIZE_MAX = msg;
686
687             break;
688
689          case queueSizeDown:
690
691             SLAVE_Q_SIZE_MIN = msg;
692
693             break;
694
695          case mincrossUpdate:
696
697             MCN = dmy;
698
699             break;
700
701          case masterjobrequesttag:
702
703             masterneedsjobsFLAG++;
704
705             break;
706
707          case stoptag:
708
709            return;
710
711            break;
712          }
713        }
714
```

Figure 3.4 Slave Probe Construct

By fine-tuning the queue sizes and other variables, this queuing system can be easily adapted to a wide variety of problem types. Additional message types may be added to increase or change functionality as need be. Optimal granularity varies greatly from problem to problem. By leaving some control in the hands of the user, we have increased the usefulness of this system.

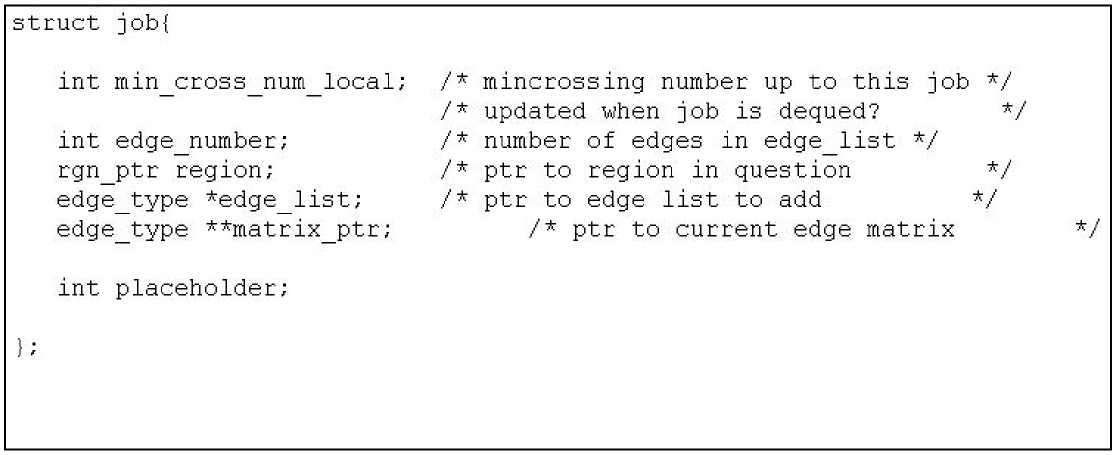
Chapter 4 Results

In order to test the effectiveness of our queuing system, we used several problems. The first was a basic simulation, the second a graph theory problem described in detail in Chapter 2 called the Minimum Crossing Number of a complete graph and the third a classic problem referred to as the Traveling Salesman Problem (TSP). The "job" structure used in these problems consists of a C/C++ data structure as shown in Figure 4.1 and Figure 4.2. These jobs are representative of those that could be used to solve other graph theory problems.

4.1 Testing Setup

The region lists for the MCN problem were developed using the rotational embedding scheme found in [17]. We have found the MCN by mapping the search space to a tree using the algorithm found in Harris [1] and performing a breadth-first search on that tree. We used a branch-and-bound algorithm to limit search space. In these types of algorithms, current solutions are compared to best solutions allowing that branch of the tree to be discarded if a better solution is already known. Thus, by limiting the search space we theoretically reduce execution time.

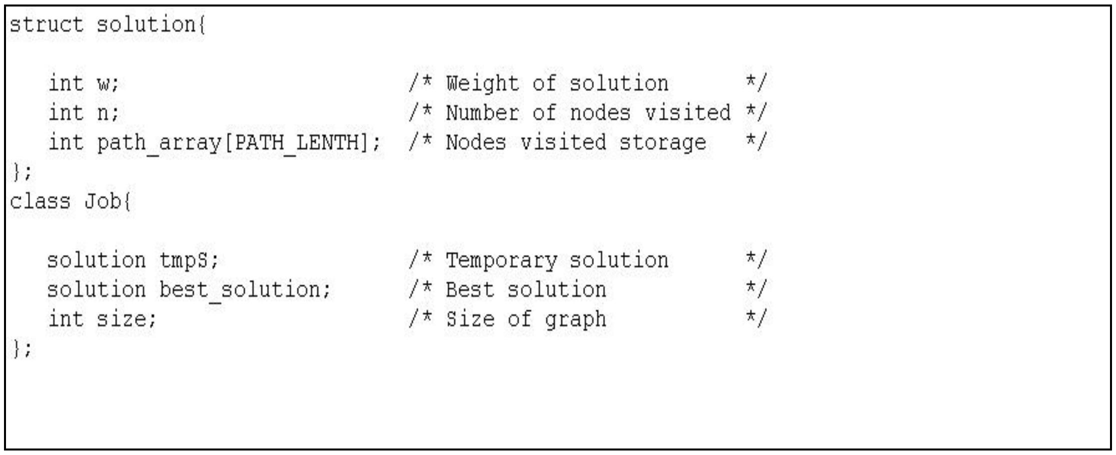
Figure 4.1 shows the job used in the queuing system to solve the MCN problem. This C language structure can be adapted to the problem as needed. The generic nature of the queuing system and its reusability for a variety of problems is achieved by using a templated queue and a user defined job type.



```
struct job{

   int min_cross_num_local;  /* mincrossing number up to this job */
                             /* updated when job is dequed?        */
   int edge_number;          /* number of edges in edge_list */
   rgn_ptr region;           /* ptr to region in question         */
   edge_type *edge_list;     /* ptr to edge list to add          */
   edge_type **matrix_ptr;         /* ptr to current edge matrix        */

   int placeholder;

};
```

Figure 4.1 MCN Job Structure

The Traveling Salesman Problem (TSP) is a common graph theory problem described thus, "…given a finite number of "cities" along with the cost of travel between each pair of them, find the cheapest way of visiting all the cities and returning to your starting point." [18] This problem is commonly represented as a complete graph. Figure 4.2 shows the data structure used to hold data for the TSP.



```
struct solution{

   int w;                       /* Weight of solution      */
   int n;                       /* Number of nodes visited */
   int path_array[PATH_LENTH];  /* Nodes visited storage   */
};
class Job{

   solution tmpS;               /* Temporary solution      */
   solution best_solution;      /* Best solution           */
   int size;                    /* Size of graph           */
};
```

Figure 4.2 TSP Job Structure

4.2 Details

Because of the problems in passing dynamically allocated memory using MPI, the user may define two "packing" functions, unpackjobs() and packjobs(). By putting all data members into a structure and giving a size in bytes, the queuing system can easily pass work and information, of any data type from slave to master and *vice versa*. This also adds to the reusability of the queuing system—only the size, not the content, of the data in bytes is relevant.

The master unpacks the jobs and places them in its queue and waits for a slave to request more work. Upon receiving the packed job from the master, a slave unpacks and enqueues it. This allows the queuing system to be as generic as possible.

4.3 Testing

This structure was tested using a simulated job creation system. The job structure was made up of one integer. The amount of work done by each slave was determined randomly. Some slaves created jobs, sending back packed queues to the master. Others requested jobs from the master. Upon receiving a certain number of requests for work from the slaves, the master purged its queue and requested jobs from working slaves. This simulation was designed to test all functionality of the generic queuing system.

The minimum crossing code was developed from work by Tadjeiv in [15] and by Yuan in [19] and was used as an example of the types of problems that the queuing system can be used to solve. Figure 4.2 shows the number of jobs created by the slaves for the MCN problem using the queuing system. Results for two different

K_6 starting graphs are included. More results and a full description of these starting graphs will be found in [19].

K6-1	**Number of Jobs Enqueued**	**Number of Requests For work**	**Number of Jobs Sent to Master**
Master	36	16	
Slave 1	476		8
Slave 2	297		6
Slave 3	305		2
Slave 4	432		14
K6-2			
Master	7	10	
Slave 1	498		1
Slave 2	455		0
Slave 3	455		0
Slave 4	497		2

Figure 4.3 Job Statistics for K_6

The first column of Figure 4.3 lists the number of jobs queued by each slave and the master. This includes jobs the master created to start the search process. The second column lists the number of requests for work received by the master, the third the number of jobs each slave sent to the master process.

The TSP had a much finer granularity. As shown in Figure 4.4, many thousands of jobs were created. The number of nodes refers to the number of “cities” the salesman must visit. Each job was one step in a breadth-first search for the optimal solution.

8 nodes	**Number of Jobs Enqueued**	**Number of Requests for Work**	**Number of Jobs Sent to Master**
Master	37	16	
Slave 1	11,508		10
Slave 2	6,052		7
Slave 3	2,513		7
Slave 4	1,281		6
Slave 5	1,078		0
11 nodes			
Master	30, 549	258	
Slave 1	530,933		29534
Slave 2	505,204		745
Slave 3	374,969		260
Slave 4	69,821		0

Figure 4.4 Job Statistics for the TSP

Figure 4.4 is broken up into two test graphs, an eight node graph and an eleven node graph. The first column of Figure 4.4 lists the number of jobs queued by each slave and the master. Note the much larger numbers than the MCN. The second column lists the number of requests for work received by the master, the third the number of jobs returned to the master by each slave process.

Figure 4.5 and Figure 4.6 show the slowdown as a result of the queuing system in the TSP.

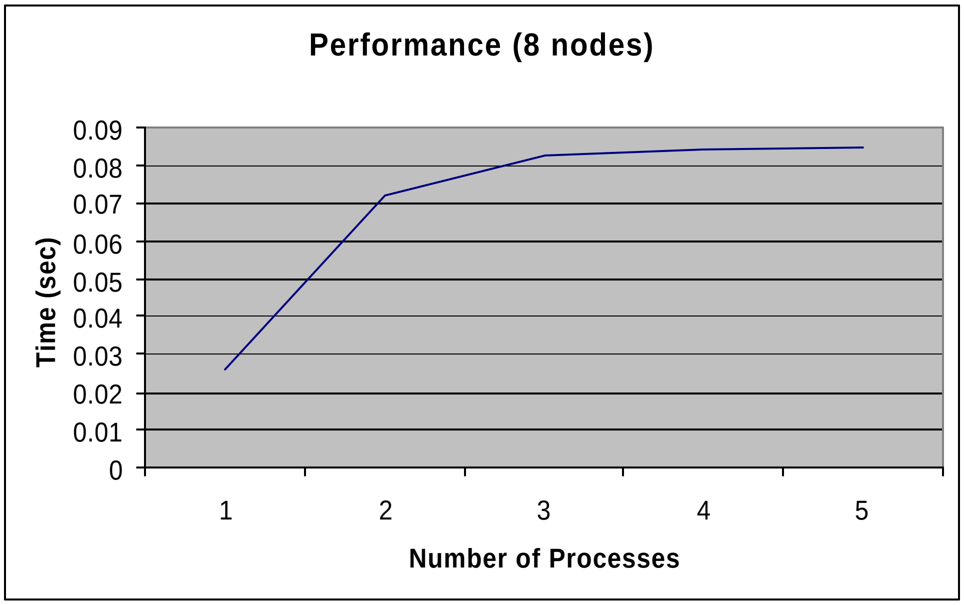


Figure 4.5 TSP Performance for Eight Nodes

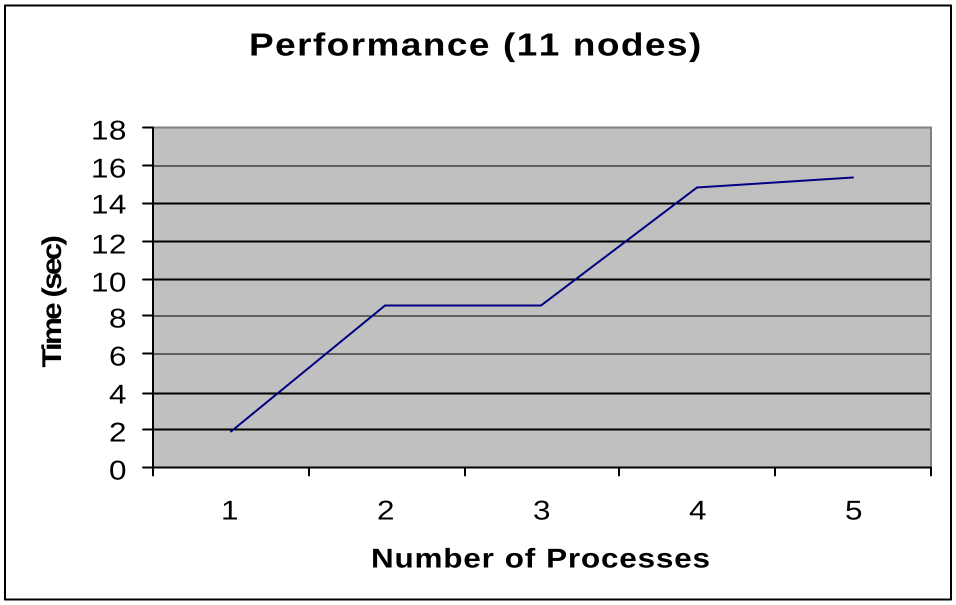


Figure 4.6 TSP Performance for Eleven Nodes

Chapter 5 Conclusions and Future work

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, large amounts of 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 plan to use this queuing system to solve the MCN problem for larger vertex sets. Harris and Thorpe showed in [20] that the actual rectilinear minimum crossing number diverges from the currently accepted value given by Guy in [2]. 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 develop a “job” with which the minimum crossing number of a bipartite graph may be determined. Very little adaptation should be necessary to use our queuing system for this and other similar graph theory problems.

The construct that allows messages such as slave queue size changes or updates to a “best solution” could be encapsulated in function “hooks”. The implementation of function parameters passed as void pointers is also in the works. These additions would make the system even more general.

In future versions of this system, we plan on looking at the possibility of slaves being able to query other slaves and being allowed to steal jobs directly from each other—without the use of the master as an intermediary.

We also plan to make queue size variables user definable at run-time rather than just at compile time. This will possibly be implemented to include a graphical user interface developed using an open source, readily available video library such as GTK.

The performance slowdowns caused by the extensive message passing of the queuing system are less important than the ability to create, store and manage large numbers of jobs. We are working toward a system to write the queue to a hard disk or some other form of non-volatile storage to prevent excessive memory consumption. Saving the queue contents would also allow the suspension of long running jobs. Easy stopping and restarting of large jobs will allow a cluster to be used for shorter processes without requiring starting computation from the beginning.

References

[1] F. Harris and C. Harris. A proposed algorithm for calculating the MCN of a graph. In Yousef Alavi, Allen J. Schwenk and Ronald L. Graham, editors. *Proceedings of the Eighth Quadrenial International Conference on Graph Theory, Combinatorics, Algorithms and Applications,* Kalamazoo, Michigan, June, 1996. Western Michigan University.

[2] R. Guy. A minimal problem concerning comple plane graphs. In Y. Alavi, D.R. Lick, and A.T. White, editors, *Graph Theory and* Applications, pages 111-124, Berlin, 1972. Springer-Verlag.

[3] K. Yelick. Programming Models for Irregular Applications. In *Proceedings of the Workshop on Languages, Compilers and Run-Time Environments for Distributed Memory Multiprocessors*, volume 28(1). ACM SIGPLAN Notices, January 1993. 10

[4] M. Feeley. An Efficient and General Implementation of Futures on Large Scale Shared Memory Multiprocessors. PhD thesis, Brandeis University, Waltham, MA, 1993.

[5] E. Mohr, D. Kranz, and R. Halstead. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 185-197, June 1990.

[6] G. MacGillivray. *UVic Discrete Mathematics Study Guide*. Retrieved on June 14, 2003 from http://www.math.uvic.ca/faculty/gmacgill/guide/M222Graphs.pdf.

[7] R. Guy. The decline and fall of Zarankiewicz's theorem. In F. Harary, editor, *Proof Techniques in Graph Theory,* Academic Press, New York (1969) 63-69.

[8] P. Turan. A note of welcome. *J. Graph Theory*, **1**:7-9, 1997.

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

[10] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[11] T. Sterling. *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters Scientific and Engineering Computation*. MIT Press, Cambridge, Mass, 1999.

[12] D. Ridge, D. Becker, P. Merkey, and T. Sterling, Beowulf: Harnessing the power of parallelism in a pile-of-pcs. *Proceedings, IEEE Aerospace*, 1997.

[13] T. Sterling, D. Becker, D. Savarese, et al. BEOWULF: A Parallel Workstation for Scientific Computation. *Proceedings of the 1995 International Conference on Parallel Processing*. Vol. 1, pp. 11-14. August 1995

[14] U. 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.

[15] U. Tadjiev and F. Harris, Jr., Parallel Computation of the Minimum Crossing Number of a Graph. *Proc. 8th SIAM Conf. on Parallel Process. for Sci. Comput*. Minneapolis, Minnesota, March 14-17, 1997.

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

[17] C. Lovegrove and R. Ringeisen. Crossing numbers of permutation graphs. *Congress Numer.*, **67**:125-135, 1988.

[18] *Solving Traveling Salesman Problems*. Retrieved November 20, 2003 from http://www.math.princeton.edu/tsp/index.html.

[19] B. Yuan. Master's Thesis, University of Nevada, Reno, Reno, NV 89557, due for completion May 2004.

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