THE BENEVOLENT BANDIT LABORATORY: A TESTBED FOR DISTRIBUTED ALGORITHMS USING PCS ON AN ETHERNET

Eve M. Schooler
Robert E. Felderman
Leonard Kleinrock

March 1988
CSD-880016

Table of Contents

page

Abstract 1
1 Introduction 2
2 The Environment 4
3 System Design 5
3.1 Node Manager 6
3.2 Resource Manager 8
3.3 User Interface 9
3.4 Process Manager 11
3.5 Functions Provided to the BBL Programmer 15
3.5.1 Initialization 16
3.5.2 Communication 17
3.5.3 File Operations 17
4 Performance 18
4.1 Communication Throughput 18
4.2 Applications 20
4.2.1 Merge Sort 20
4.2.2 Eight Puzzle 32
5 Restoration and Recovery 35
5.1 Token Passing 36
5.2 Music 37
6 Future Directions 39
6.1 A Graphical Interface 39
6.2 More Flexibility 39
6.3 Accounting 40
6.4 Other BBL Architectures 41
7 Conclusions 41
8 Acknowledgements 42
9 References 43

List of Figures

page

Figure 1. System Diagram. 6

Figure 2. Timing diagram for node failure. 14

Figure 3. Throughput versus message size. 19

Figure 4. Four node merge sort timing diagram (32000 data items). 21

Figure 5. Total time versus number of processors (merge sort). 22

Figure 6. Speedup versus number of processors (merge sort). 23

Figure 7. Time versus size for merging. 24

Figure 8. Time versus size for sorting. 24

Figure 9. Total time versus number of processors (32000 elements). 27

Figure 10. Speedup versus number of processors (32000 elements). 27

Figure 11. Optimum number of processors versus number of elements. 29

Figure 12. Total merge units versus number of processors. 31

Figure 13. Bounds on merge units. 32

Figure 14. Average time to solve 11 puzzle versus number of processors. 34

Figure 15. Speedup versus number of processors. 35

Figure 16. Speedup versus number of search processes. 35

The Benevolent Bandit Laboratory:
A Testbed for Distributed Algorithms using PCs on an Ethernet †

Eve M. Schooler
Robert E. Felderman
Leonard Kleinrock

Abstract

We describe the design, implementation and use of a distributed processing environment on a network of IBM PCs running DOS. Temporarily unused PCs can be accessed by other users on the network to perform distributed computations. An owner of a PC need not be aware that the machine is being used during idle times; the machine is immediately returned when the owner begins to work again. In addition, a high degree of computation resiliency is provided in this unreliable environment. If a PC is part of a distributed algorithm and is reclaimed by its owner, the system finds a replacement node (if possible), resends the affected code to the new processor, and restarts it. A distributed computation is able to proceed despite a set of transient processors. A discussion of system performance, distributed applications, and fault tolerance are included. In particular, performance improvements are demonstrated by applications like parallel merge sort and a distributed search solution to the eight puzzle.

Keywords -- Distributed processing, distributed algorithm, personal computer, DOS, local area network, message passing, Ethernet, merge sort, eight puzzle.

† This work was supported by the Defense Advanced Research Projects Agency under contract MDA 903-82-C0064, Advanced Teleprocessing Systems, and contract MDA 903-87-C0663, Parallel Systems Laboratory.

1 Introduction

During this decade we have witnessed the rapid proliferation of personal computers and workstations. Many computing environments have moved away from traditional time sharing, where one large mainframe serves an entire organization and users gain access to it through terminals, to a new environment where each user has a personal computer or workstation; and many of these are connected to a local area network. A 1984 survey by the International Data Corporation showed that the USA installed base of IBM MIPS in personal computers was ten times that installed in IBM mainframes (3030,3080,3090,4300 series). If non-IBM equipment were included, and if we made the survey today, one would expect an even greater difference between these two figures, showing that the raw computing power in these PC networks is tremendous. Yet, most of these PC MIPS are badly underutilized (what is your PC doing right now?).

Though these new PC-LANs are seemingly more practical for an environment where word processing or other small scale tasks are performed, there is precious little sharing of resources (e.g., printers and disks.) The major problem is the distribution of processing power. By moving away from time-sharing we have effectively moved from a central server system with a large capacity to many independent server systems each with a much reduced capacity and smaller arrival rate of jobs. Kleinrock has shown [KLEI84] that this leads to an inefficient use of resources. We see this when we want to execute a task such as sorting a large database. With a central processor the task is trivial, though it may be delayed if the system load is heavy. In a networked PC environment this task is usually run on a single PC. This could take many hours depending on the size of the job, especially if the job in its entirity cannot fit into the memory of the PC. Compounding our frustration with the job's slow execution is the realization that most of the processing power distributed to other PCs lies unused while the owners are off doing something else.

issue of crash recovery. Cabrera et al [CABR86] present the idea of a personal Process Manager (PPM) to relay information about node failures and to re-establish internal consistency. The PPM does not appear to manage resilient computations, though. In other words, the system will not transfer a computation to another host, at least not in this implementation. It does, however, try to combat certain failure modes by establishing a crash coordinator site to assist with temporary irregularities. Another approach to crash recovery is discussed in [GAIT87]; replicated processes or shadow copies are used to provide a high degree of failure transparency. Each process exists in multiple invocations across different workstations. Of the copies, one is considered the principal shadow. If the principal shadow fails, one of the other shadows takes over as the principal.

2 The Environment

The BBL system was developed for IBM PC-ATs running DOS 3.1 and connected together via an Ethernet. The network of approximately 100 PC-ATs was already available at UCLA. As described above, we provide an environment for distributed processing. A unique feature of this system is its ability to find idle processors and use them without the knowledge of the owner of the PC. The owner experiences no discernible delay when she begins typing at the keyboard after an idle period. The owner of the PC runs a special shell designed to emulate DOS, and to allow BBL access to the machine during the idle time. The owner, of course, can choose not to provide access to the machine. Another salient feature of the BBL system is its ability to replace processors which are part of a distributed computation, but which are reclaimed by an owner, thus allowing the computation to proceed with a set of transient processors. When this happens, the system finds another idle processor (if possible), resends the affected code to the new processor, and restarts it. The system provides the means for restoring state to this new processor, but it is partially under the control of the user/programmer. Since the BBL operating system cannot know what defines the state of a particular application, the

user writing the application code must handle some of its restoration. The alternative is to save the entire memory space and registers of the user's process and attempt to restore this state to another PC. Since each PC may be configured differently, a process may not be able to be restored in the same memory location. The PCs and DOS do not support virtual memory mapping, so this solution was ruled out.

The majority of the system was written in C with some special routines written in 8086 assembler code.

3 System Design

BBL consists of four independent modules of code. The module that resides on every PC in the network, the *Node Manager* (NM), detects when the machine goes idle and registers with the BBL system. The central resource coordinator, running on a dedicated machine, which keeps track of available PCs, is called the *Resource Manager* (RM). The code which allows a user to interact with the idle PCs is actually two modules on one machine, the *User Interface* and the *Process Manager* (UI/PM). The User Interface, as the name implies, is the interface between the user and the BBL system. The Process Manager is the lower level communication module responsible for the run-time operation of the system and for the administration of the user's algorithm processes when a user is running a distributed computation. A UI/PM machine is dedicated to a single user, although several users may be using the BBL system at once. Each of these modules will be described in more detail below.

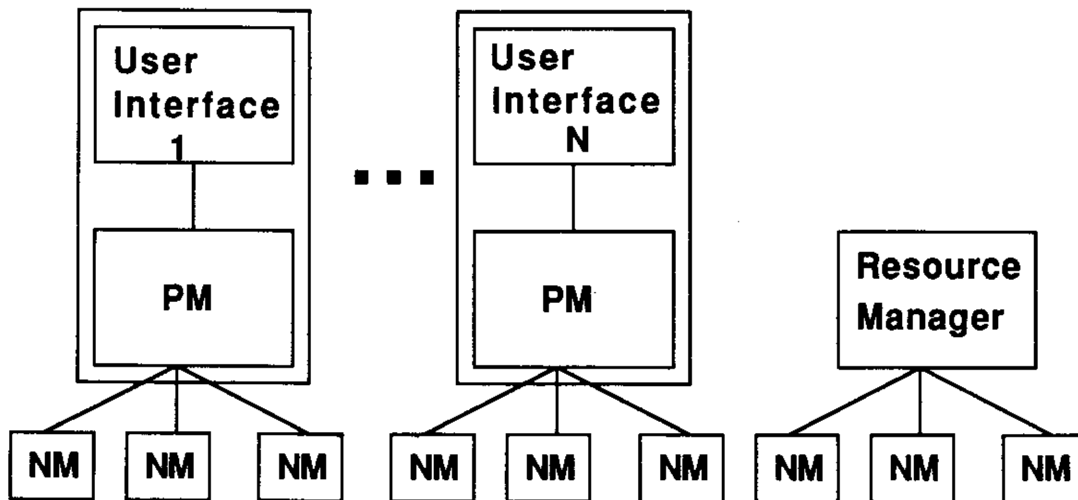


Figure 1. System Diagram.

3.1 Node Manager

The Node Manager's purpose is to benevolently steal a machine from its owner after the machine has been in the idle state for a given number of seconds. A PC is said to be in the idle state when that PC is displaying a DOS prompt, waiting for the owner to type a command. The length of time after which a machine is considered available for BBL use is simply a selected parameter which is passed to the NM program. The Node Manager is designed to run as a shell on top of DOS and emulate its operation and is normally configured to automatically execute when the system boots. The NM shell waits for the owner to type commands, decrementing a timer as it waits. When the owner completes a command by hitting the return key, the timeout counter is reset and the NM passes the command on to DOS for execution. The added overhead is imperceptible for most DOS commands, since they generally involve disk accesses which are slow compared to the time needed to parse the command and pass it to

DOS. If the owner fails to type a key within the preset time limit, the NM "takes over" the machine.

Aside from passing commands to DOS, the NM's basic operation is to send a message to the Resource Manager indicating it is free for use. The NM then waits to be assigned to a specific user's distributed computation. After being assigned to a user's UI/PM, it waits for further messages from the Process Manager which contain code to execute, possibly an input file, and addresses of other nodes involved in the distributed computation. These addresses are used by the NM to enforce topology constraints since the user code can only send packets to "neighboring" nodes, which are determined at the UI/PM. The user code sends packets by specifying a destination node's virtual id (vid). Each node in the computation is addressed by a vid, so user code need only deal with vids and never physical addresses. When one node replaces another, it receives the "dead" node's vid. In this way, replacement is transparent to the user code. The NM performs the translation from a vid into a physical address by maintaining two tables, one for input links and one for output links. The vid is used as an index into the tables which contain the physical Ethernet address of the vids linked to this node. The system does not support multi-hop routing.

If a key is hit at any time while the NM has control of the PC, the NM notifies the RM (if it is still in the RM's pool of available nodes), or the PM (if it has been assigned to a user) that it is going down and immediately returns to processing commands from its owner. The owner does not notice any delay, since the overhead for processing the context switch is imperceptible.

When the Node Manager receives code from the PM, which is its portion of a distributed computation, it places the code into a virtual disk in the memory of the PC. This downloaded code is simply an executable file. When the Process Manager notifies the NM that it is to start running the code, the NM simply executes this code by passing the file name to a copy of

DOS. When the code completes, this "subroutine call" returns and the NM notifies the PM that it has completed.

The NM is responsible for providing all the functions that the user code may need in order to perform a distributed computation; these include, but are not limited to, communication primitives and file operations. Also, since the NM is designed to run executable files, the majority of the user code can be compiled and tested outside the BBL system. Only when the bulk of the code is debugged is it necessary to run the distributed computation on the BBL system. Essentially, everything except the communication between different processors can be pre-tested.

3.2 Resource Manager

The Resource Manager is a dedicated machine responsible for keeping track of the available PCs in the network. When a PC becomes available, the RM receives a message indicating this fact from the NM running on that PC. The Resource Manager then adds this node to its pool of free nodes. Included in the "I_AM_UP" message is the amount of available space on the PC's virtual disk. Generally this is about 360K bytes. The RM uses this information to allocate nodes to users who make specific requests about the number of nodes needed and the minimum amount of memory needed per processor (to hold the downloaded code). The RM responds by sending a list of physical Ethernet addresses to the User Interface/Process Manager. The RM is able to support an arbitrary number of users of the BBL system, provided of course that there are sufficient idle PCs to fulfill all the requests.

One drawback to this design is that the RM is a dedicated machine that does nothing but handle resources for BBL. One method to eliminate the RM from the system is to allow each UI/PM to "find" its own idle PCs. This would eliminate the RM altogether, but would cause added complexity in the UI/PM, especially if the system continued to support multiple

users. In this case, when a new user wanted to run a distributed computation, the UI/PM process would need to contact any other currently running UI/PMs and ''beg'' for hoarded nodes. By centralizing the Resource Manager, each PM need only go to one location to find free nodes to replace any that go down during a computation. Since we want to replace nodes as quickly as possible, this centralized location provides the best response time when there are several UI/PMs active in the system.

To guarantee that all available processors are either in use or registered with the RM, the UI provides a function to the user which has the RM broadcast a message to all NMs. When an NM receives this message it will respond to the RM if it is not involved with a user's distributed computation. If the NM is associated with a user's PM, it sends a message to that PM asking whether it is still being used. If the PM fails to respond within a certain time, the NM re-registers with the RM. This facility was added so that if the PM fails due to a hardware or software fault, its associated NMs can eventually be returned to the RM's pool.

3.3 User Interface

The user interface serves as the link between the user and the BBL system. It is intended to aid in the management of the distributed application. A command language allows the user to logically configure the system for running the distributed algorithm, to alter the environment during run-time, to query the system about the state of the computation and system resources, to run the BBL debugger, et cetera.

The UI requires the user to select an algorithm from an already established algorithm library. The algorithm library contents are listed in the file, 'info.bbl', where each line refers to an individual algorithm configuration file (e.g. ''*.alg''). For instance, one line in the library file refers to 'BACH.alg', a distributed music program that *plays* a Bach composition (the eighth two-part invention) on several PCs. By writing output to a speaker port at different fre-

quencies, a PC can generate different pitches of sounds. A PC can be programmed to play given musical notes in this fashion. The '*.alg' file (e.g. 'BACH.alg') contains the details about the specific algorithm: a description of the program, the number of separate code segments it contains (in this case, a conductor code segment for synchronization, and two separate code segments that will actually play different notes, voice1 and voice2), the names of the code segment executable files (conduct.exe, voice1.exe, and voice2.exe), the minimum and maximum number of machines on which each code segment should run (the conductor must only run on one machine, while each voice should run on at least one machine), the names of any parameters to be passed to the executables (the conductor executable might take an input parameter to specify the tempo or speed of the music), and finally the name of any input files to be redirected to the executables (the voice parts might use input files to know which notes to play and when).

In the original version of the UI, the algorithm environment was initialized through the use of an algorithm configuration file, plus interactive questioning from the UI. After the configuration file was read, the UI would prompt the user to specify several items: how many nodes were needed from the Resource Manager, the memory requirements to be met by these nodes, the underlying logical topology for the assigned nodes, which code segments to place on which nodes, and any input parameters and/or names of input files for the executables. With the addition of debugger options, the set up of an algorithm environment became even more elaborate, if not more complicated. The format of the algorithm configuration file therefore came under scrutiny for revision. The configuration file seemed the natural place to allow the initialization of debugging options. It also seemed appropriate to eliminate, or at least minimize the use of interactive questioning and to make use of default assumptions wherever applicable. For example, by having the names of the executables files and names of any input files, the UI can approximate how much memory is needed from the nodes. Unless the user indicates a different value in the configuration file, this approximation is considered the default

value. In this version of the UI, the user is only prompted for set up information if certain key aspects about the environment have gone unspecified and no default value can be assumed. This means that in the majority of cases, the system has enough information on hand (via the configuration file) to be able to download and run user code after the user chooses an algorithm. For more information about the configuration file format, default assumptions, and the debugger see [BBL88] and [SCHO88].

Once the algorithm environment is set up and the the algorithm information is downloaded to the participating NMs, the UI runtime commands become enabled. Among other things, they let the user dynamically add nodes to or delete nodes from the algorithm, change logical link information in the topology, suspend and resume the algorithm execution, reset the environment, and ultimately exit from the BBL system. When the user makes these requests, the user interface checks their feasibility. The UI will complain if a request is contrary to the specifications of the algorithm. For instance, the UI strictly enforces code segment maxima. A code segment might have an upper limit on the number of nodes which should be running that code. In fact this is the case with the conductor code segment in the music algorithm. Only one node should be assigned the conductor code. If the user adds a node to the computation and tries to assign it the conductor code when another node has already been designated the conductor, the UI disallows the request.

3.4 Process Manager

As the name implies, the Process Manager manages the application processes owned by the user. It also provides the low-level communication needs of the UI. It handles requests between the UI and the RM, as well as between the UI and the NMs. As mentioned previously, the UI and the PM co-reside on a single node. One UI/PM node exists for each user running distributed algorithms on the system. After the first exchange of packets between a UI/PM and the RM, the Resource Manager assigns the UI/PM its own user id number. The user id number

enables the RM to differentiate between the users on the system and for NM's to respond to broadcast messages from a particular user.

After the RM allocates nodes to the UI, the PM becomes responsible for keeping track of each node's physical address. These addresses are transparent to the UI and consequently to the user and the user's algorithm. Instead, the UI identifies each node via a virtual id. The PM keeps track of the mapping between physical addresses and vids. The PM shares the mapping with the NMs since the NMs provide this same translation for the user's algorithm. During the distributed algorithm's execution, whenever one code segment of the algorithm wants to send a packet to another code segment, the user code specifies the destination node in terms of a vid. It is the NM's job to map the vid to a physical address and to ship off the packet.

This address-to-vid mapping is especially important in the event that a node is taken away from the algorithm by its owner. When this occurs, the PM tries to locate a replacement node. If a replacement is found, the PM assigns it the vid of the node being returned to its owner. The PM then notifies the necessary NMs of the new physical address for that vid. Since the user's algorithm only deals with vids, it is shielded from ever having to know about such changes. If no replacement is found and the failed node is deemed a critical node (when a node is given super node status, as described in section 3.5), the execution of the algorithm is aborted. Otherwise, the algorithm continues with one less node.

One concern with this replacement scheme is the possibility of race conditions. A send-and-wait message passing protocol, however, guards against this. For instance, consider the case where the user's algorithm is using a particular vid when the PM assigns it a new physical address (e.g. the owner has reclaimed the node). A user code segment might be trying to deliver a message to a neighboring code segment when the owner of the neighboring PC returns. Since a user code segment calls BBL library routines to send and receive packets, it actually interfaces with the NM's lower level sending and receiving routines. Unbeknownst to

the user code, when the underlying NM does not receive an acknowledgement for a packet it sent, the NM notifies the PM that the neighbor is down. The PM then goes in search of a replacement node. In this situation, the NM prolongs its normal timeout period in order to give the PM time to find a replacement. This also gives the PM time to update the failed vid's physical address information at all the NMs to which that vid is connected, including the NM which sent the original message. Meanwhile, that NM continues to try to resend the message. At some point, the NM will either receive an update message from the PM if a replacement was found or exhaust its maximum number of retries. Because the NM is interrupt driven, the PM's update packet will interrupt the NM's re-sending efforts. The failed vid's physical address is therefore never being accessed at the same time as it is being updated. Eventually the NM will resume resending the original message, only now it will send it to the vid with its updated physical address. Refer to Figure 2 for a timing diagram of PM and NM interactions during node replacements.

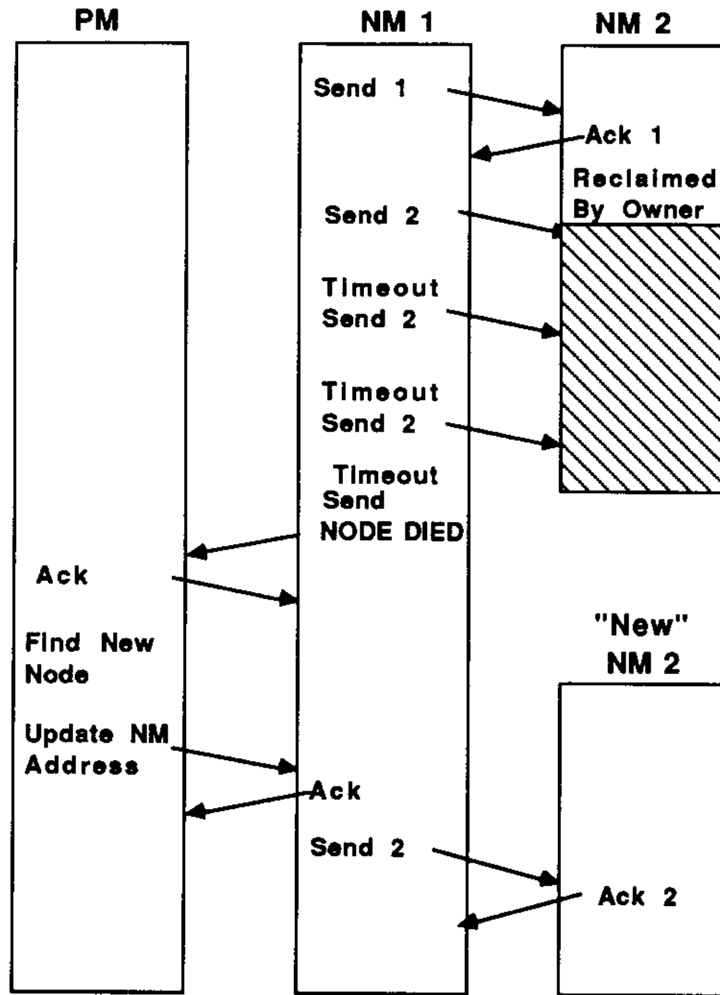


Figure 2. Timing diagram for node failure.

The PM also stores the status of each of its nodes. After a node is allocated to the PM, it can be in one of six states; **free** to be used by an algorithm, **loaded** with the algorithm code, **busy** running the code, **done** running the code, **suspended**, or **dead**. The PM relies on this information when a node is taken back by its owner. By knowing the status of the re-claimed node before it gets returned (which in turn changes its status to "dead"), the PM can try to initiate a replacement node with the same status. For example, if a node has been loaded with the algorithm code when the owner requests its return, the PM not only needs to find a replacement

node, but also needs to download code to the replacement node. Only then would the replacement node replicate the state of the old node. The more difficult scenario involving the replacement of busy nodes is discussed in section 5 under ''Restoration and Recovery''.

The fault tolerance of the PM is not particular about whether a node fails because its owner took it back or because it simply crashed. The PM is prepared to deal with both types of failures. Several forms of fault detection are embedded in the PM. The most controlled type is when an NM issues an I_AM_DOWN message to the PM, telling the PM that the owner has resumed work. Another method by which the PM learns of a node's unavailability is via one of the node's neighboring NMs. This happens during the course of the distributed algorithm's execution, when one of the algorithm's nodes tries unsuccessfully to send a message to another of its nodes. The NM of the detecting node sends the PM a NODE_DIED message. Lastly, the PM may discover a node's unresponsiveness while trying to send it a packet. This may occur at any time when the PM relays messages of any type between the UI and the NMs (during set up, downloading, run-time, debugging, whenever).

The reaction of the PM to node failures depends on how the user sets up certain system parameters. The user can set up both a replacement strategy and notification strategy; whether or not to replace a failed node with a new one, and whether or not to notify the algorithm about this activity. Naturally, the PM's job is more complicated when the user wants replacements to be found. In this situation, the PM tries to locate an idle node among its supply of allocated nodes. If all its active nodes are being used by the algorithm, then it must look to the RM's pool of resources for additional nodes.

3.5 Functions Provided to the BBL Programmer

To facilitate distributed programming for the BBL system, we have created a set of functions and placed them into a library which is linked with the user's object files. These

functions are available to the programmer constructing distributed algorithms to run on the BBL system. The functions fall into three categories: initialization, communication, and file operations. We briefly describe these functions below. More detailed descriptions of these functions and how to use the BBL environment can be found in the BBL User Manual [BBL88].

3.5.1 Initialization

Init_Vars must be called near the beginning of the user code. This is because the NM must have an address of an exit routine (passed as a parameter) to call when a key is hit. This exit routine can provide any needed fault tolerance such as saving and restoring state, and it allows the user code to terminate "cleanly" before the NM returns control of the PC to the owner. The return value from this function specifies whether or not the node is a replacement node. This means that if a node goes down and is replaced, when Init_Vars is called in the new machine, the return code will be TRUE. If the code is running for the first time, the return code is FALSE. By testing this value, a process can discover that it is replacing another node, and it may need to alert another process so as to restore certain state information. **Super_Node** indicates to the PM that it is an important node. When a node is a super node it will receive packets from the PM telling it when nodes go down and whether they get replaced. Also, if a super node goes down, the algorithm is terminated under the assumption that recovery is impossible. **Set_Parameters** tells the PM how to handle nodes going down. Parameters tell whether to replace nodes that go down and whether to notify super nodes. **Time_Synch** is used to synchronize the local clocks of all the Node Managers participating in the distributed computation. This synchronization is only approximate, but is sufficient for many applications.

3.5.2 Communication

Send_Packet is used to send a packet (message). The message may be up to 32K bytes long. This function guarantees delivery of the message into the packet buffer of the recipient. It is implemented using a stop-and-wait protocol. **Receive_Packet** waits for an incoming packet. The NM puts the packet at a user specified address. If more than one message exists in the buffer when this function is called, the oldest message is returned. Non-blocking receive is accomplished by checking flags to see if any packets have arrived as the user also has direct access to the incoming packet buffer. The user can specify a particular packet type or sender vid to receive also. **Receive_Exist** allows the user to test to see whether a packet of a particular type or from a particular vid is in the buffer. **Get_VID** returns the value of the virtual id number of the node. **Get_IN_Connections** returns the number of connections that are input links to the node along with the vid of the node on the other end of each link. **Get_OUT_Connections** returns the number of connections that are output links from the node along with the vid of the node on the other end of each link.

3.5.3 File Operations

To provide the opportunity to pass input to and to receive output from a distributed computation the system allows remote access to the file storage devices on the UI/PM machine. **BBL_Open** allows the user code to open a file on the PM machine. The return value will be a file pointer, or -1 if there was some error. The file pointer is used in subsequent calls to BBL_Read, BBL_Write, and BBL_Close. This file is opened in binary format, so only strings of bytes may be read or written. **BBL_Close** closes a remote file on the PM machine. **BBL_Write** allows the user code to write bytes to an open file on the PM machine. **BBL_Read** allows the user code to read bytes from an open file on the PM machine. Up to 32K bytes can be read or written at one time.

4 Performance

In order to test the performance and utility of the BBL system we have analyzed its operation in a variety of different circumstances. One simple measure of performance is the communication throughput between two communicating user processes in the BBL system. Another indicator of performance is the speedup achieved by applications running on the system. We examine parallel versions of merge sort and IDA*, a search algorithm used to solve the eight puzzle.

4.1 Communication Throughput

One performance measure of our system is the communication throughput between two nodes running a distributed algorithm. We chose to implement a stop-and-wait protocol for communication to make the job of programming the system much simpler. To test the throughput, we wrote two programs, one which transmitted packets, and another which received them. We experimented with different sized packets and produced the plot in Figure 3.

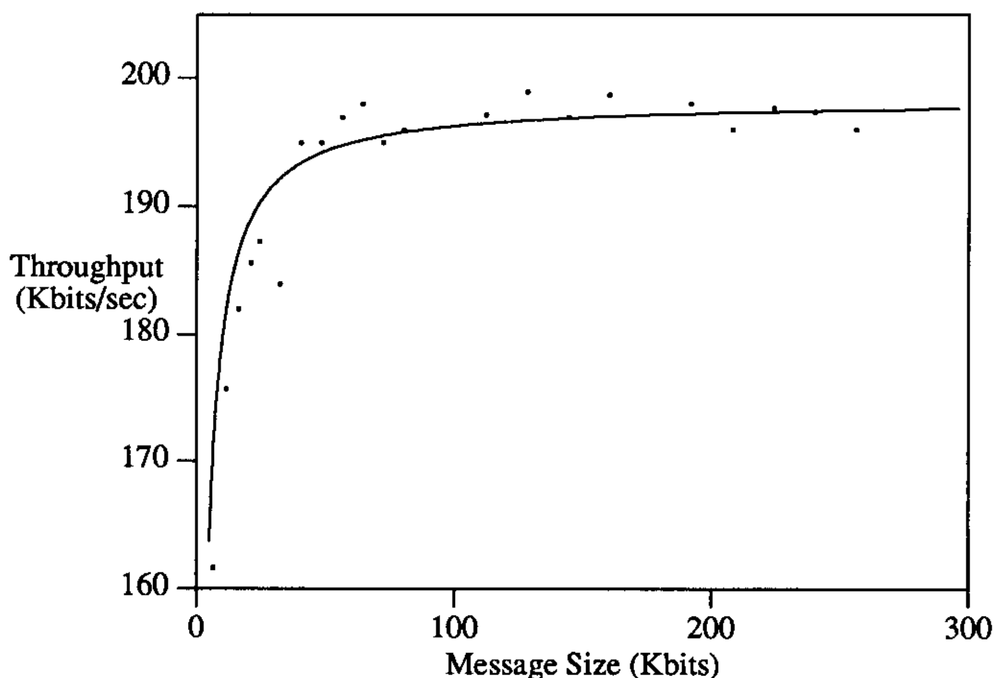


Figure 3. Throughput versus message size.

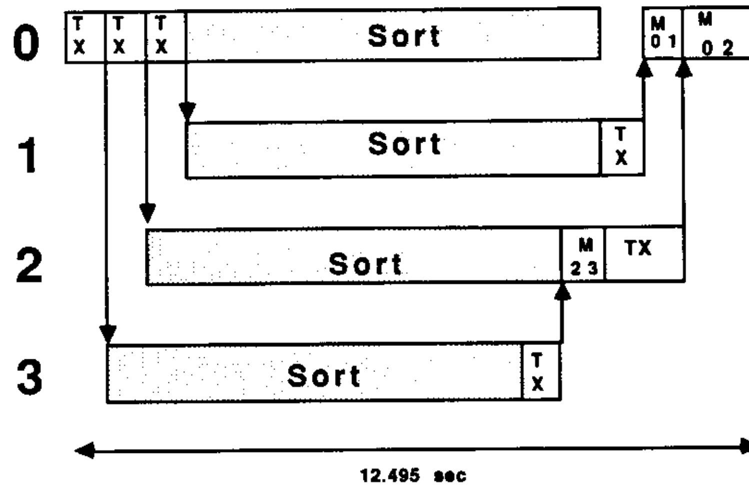
As can be seen from the plot, the maximum throughput seems to be limited to slightly under 200 Kbits/sec. The Ethernet is a 10 megabit/sec (Mbps) medium. Why then do we see only 1/50 of the maximum throughput? A major loss of throughput is due to the fact that we are using a stop-and-wait protocol with a "large" turnaround time at each processor. Since the 3Com boards installed in the PCs only have one packet buffer, every arriving packet must be read off the board, examined and then acknowledged if necessary. A turnaround time of 0.03 second will give roughly a channel utilization of 0.02, or a reduction from maximum throughput of 50 times [TANN81]. This is where most of our throughput is lost, since our overhead is fairly high. We must bear in mind though, that this is the channel throughput for **one** set of communicating processes. With several processes communicating we are able to utilize the channel more efficiently. Specifically, when running five pairs of communicating processes, the total throughput was one megabit, or five times that of a single pair. In addition, other non-

BBL processors are exchanging messages over the same network, thus utilizing a portion of the total capacity.

4.2 Applications

4.2.1 Merge Sort

One application we have coded for the BBL system is parallel merge sort. One node in the network (vid 0) generates a random list of n data elements (16 bit integers). By checking its communication links, it determines P, the number of nodes that will participate in the sorting operation; it then partitions the data into P equal size lists and transmits one list to each of the other $P-1$ processors. Each of these processors sorts its data using an $n \log n$ sort and, depending on its vid, either waits to receive data or sends its data to another processor. All waiting processors receive data and perform a merge operation before sending the data on to still another processor. Eventually, the final step involves merging two lists of size $\frac{n}{2}$ at processor 0. For example, with $P=4$, node zero sends messages of size $\frac{n}{4}$ to the other three processors. Each of these four processors sorts its part. After sorting, node one sends to node zero, and node three sends to node two. Nodes zero and two each perform a merge operation. Finally, node two sends its data to node zero which performs a merge operation to finish sorting the list. We show a timing diagram in Figure 4.



TX = Transmit data
M ij = Merge lists from processor i and j
Sort = . Sort list of data

Figure 4. Four node merge sort timing diagram (32000 data items).

Figure 5 shows the total time to sort lists of size 4000, 8000, 16000, and 32000 data items with one to nine processors.

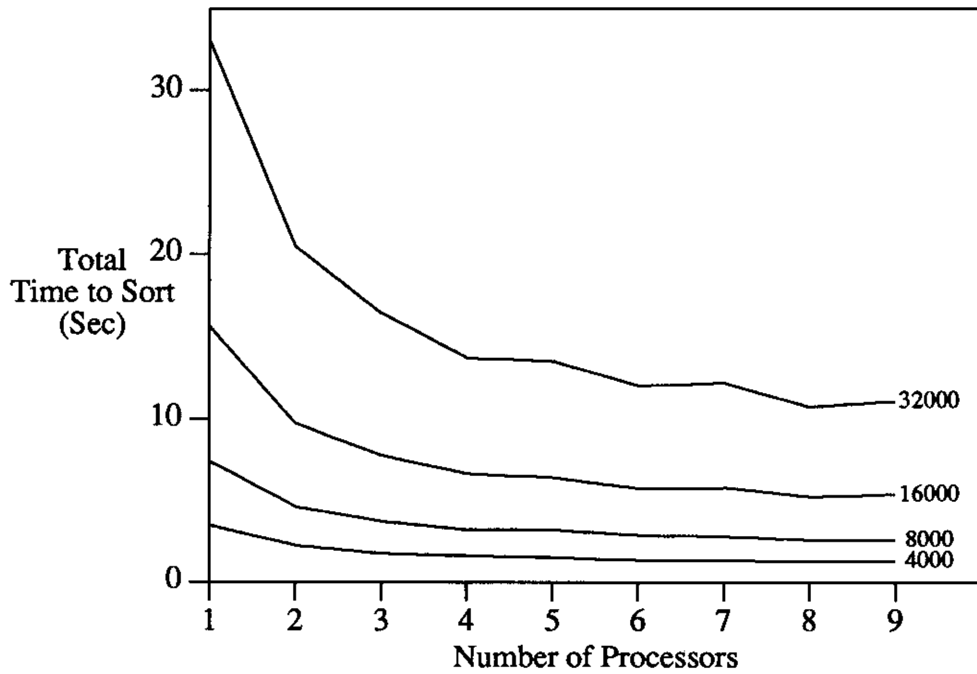


Figure 5. Total time versus number of processors (merge sort).

The corresponding speedup graph is given in Figure 6. The speedup is measured with respect to sorting the entire list on one processor with an $n \log n$ sort (no merges).

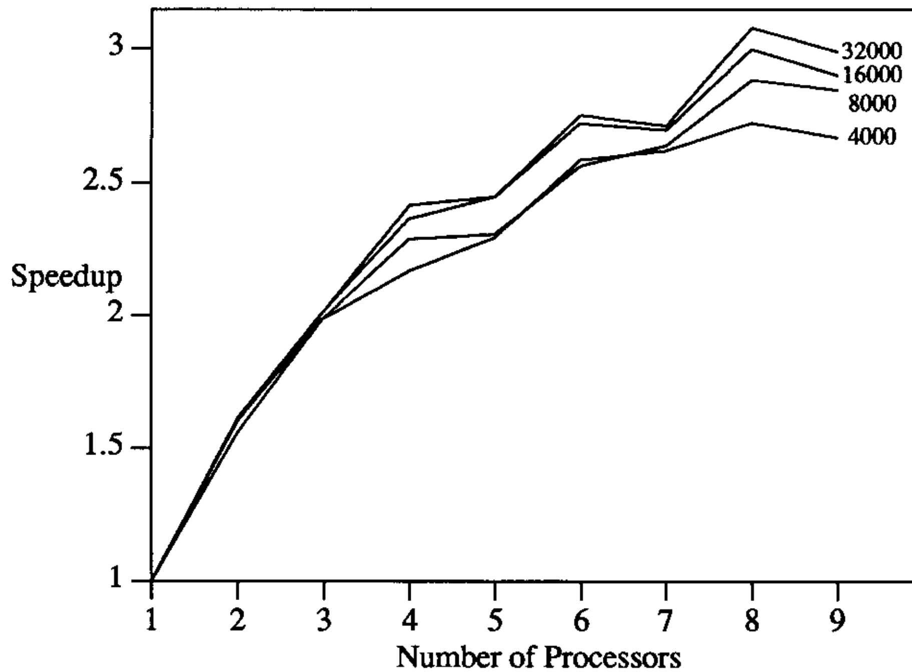


Figure 6. Speedup versus number of processors (merge sort).

In general, we would expect the speedup to increase with P and to be best where $P = 2^k$. To understand these results better, we compare them, not with linear speedup, but with the theoretical maximum speedup using this version of merge sort. We compare our results with that for a parallel merge sort where communication is free with zero delay. We can simulate these results by making approximations for the time to sort a list of data, using the fastest available sorting routine, and the time to merge two lists of a given size. We ran tests on independent (non-BBL) PCs to gather data for the time to sort a list of integers and the time to merge two sorted lists. Figures 7 and 8 show the empirical data plotted with its respective approximation curve for merging and sorting.

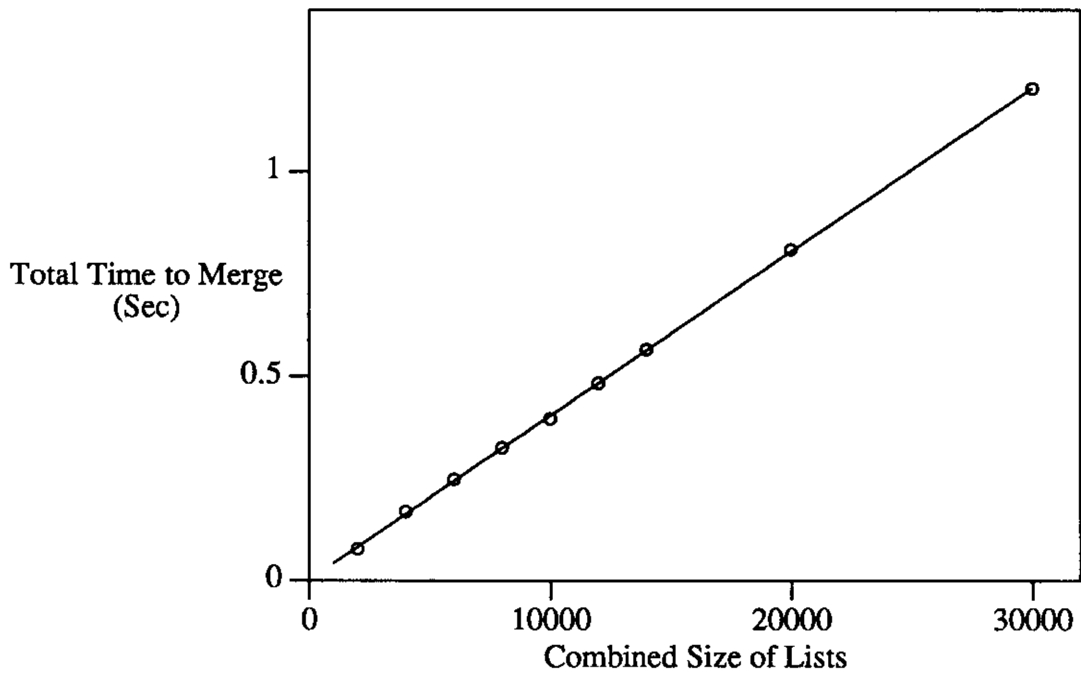


Figure 7. Time versus size for merging.

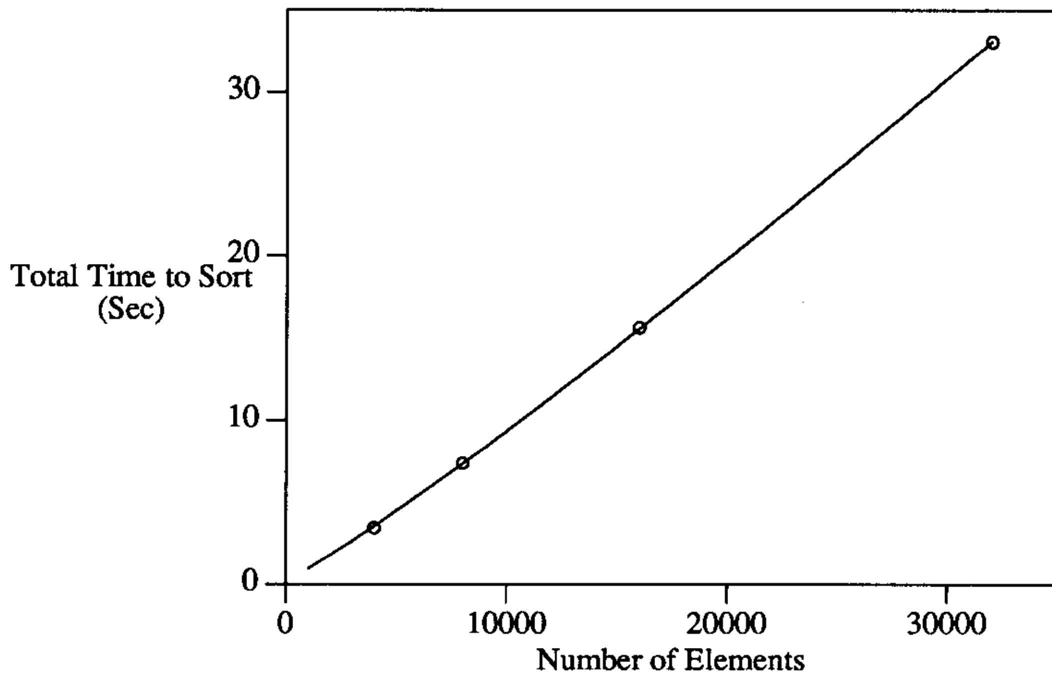


Figure 8. Time versus size for sorting.

All of our approximation formulas are of the form $Y = mX + b$ and are shown below. $T_c(n)$ is time for communicating a list of size n, $T_s(n)$ is sorting time of a list of size n. $T_m(n)$ is the merging time of two lists with combined size n, and n refers to the number of integers (data elements).

$$T_c(n) = m_c k n + b_c \tag{1}$$

$m_c = 0.00004036$ (seconds/byte)

$b_c = 0.0051134$ (seconds)

$k = 2$ (bytes/integer)

$$T_s(n) = m_s \left\lceil n \log_2 n \right\rceil + b_s \tag{2}$$

$m_s = 0.00006862$ (seconds/integer)

$b_s = 0.248614$ (seconds)

$$T_m(n) = m_m n + b_m \tag{3}$$

$m_m = 0.00004015$ (seconds/integer)

$b_m = 0.001549$ (seconds)

Using the above approximations we can compare our empirical results with results we would obtain for an ideal system with no communication overhead. For simplicity we will initially assume that $P = 2^k$ and n is the total size of the list to be sorted. The total sorting time is determined by the total time used by processor zero. It is the one that generates the random data, parcels it out, and is the place where the final merge will take place. We neglect the time to generate the list of integers. Neglecting communication, processor zero will first sort a list of size $\frac{n}{P}$, followed by a series of merge operations where the total number of items being merged is of the following format: $\frac{2n}{P}, \frac{4n}{P}, \frac{8n}{P}, \ldots, \frac{Pn}{P}$. Processor zero will perform $\log_2 P$ merges where the ith merge is of size $\frac{2^i n}{P}$. So, using our approximation formulas, the

total time spent merging data at processor zero when $P = 2^k$ is

$$M = b_m \log_2 P + m_m \left[\frac{n}{P} \sum_{i=1}^{\log_2 P} 2^i \right]$$

$$= b_m \log_2 P + m_m \frac{2n(P-1)}{P}. \tag{4}$$

When P is not a power of 2 the number of merges performed at processor zero is $\lceil \log_2 P \rceil$. The total size of merges performed is dependent on P, and we do not have a closed form expression for it. Using these formulas we can approximate the running time of the parallel merge sort in the absence of communication overhead. In Figures 9 and 10 we compare the empirical results with the approximation above for 32000 data items (which showed the largest difference). The curve labeled "BBL" is the actual measured time on the BBL system. The curve labeled "Init comm" is the approximate total time to complete when we include the cost for initially distributing the data. The curve labeled "No comm" is the approximate total time to complete the sort without adding any time for communication.

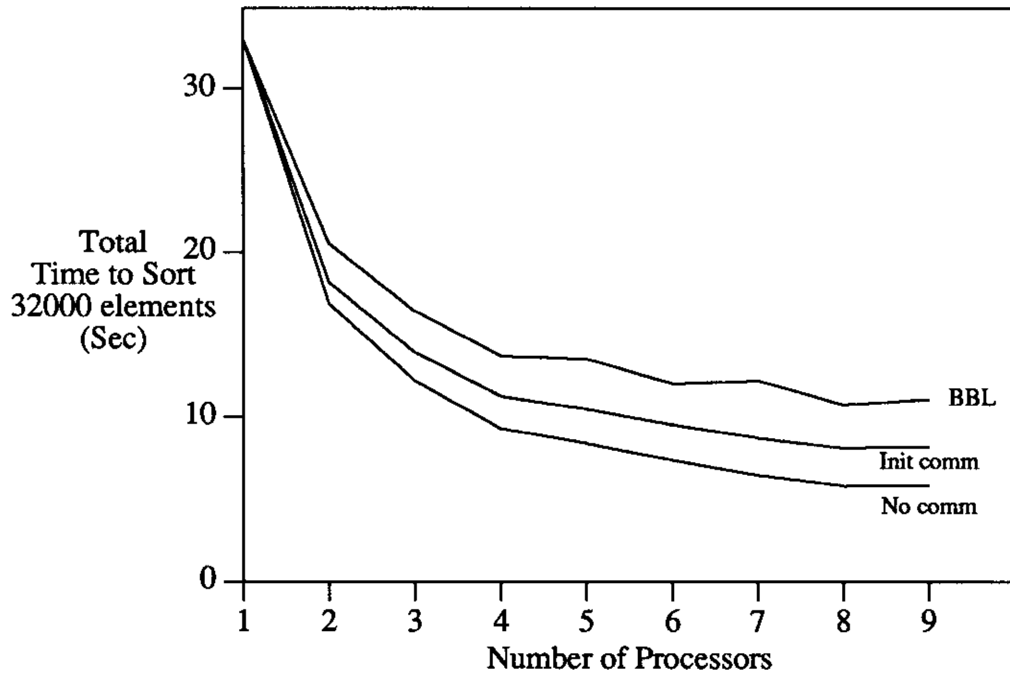


Figure 9. Total time versus number of processors (32000 elements).

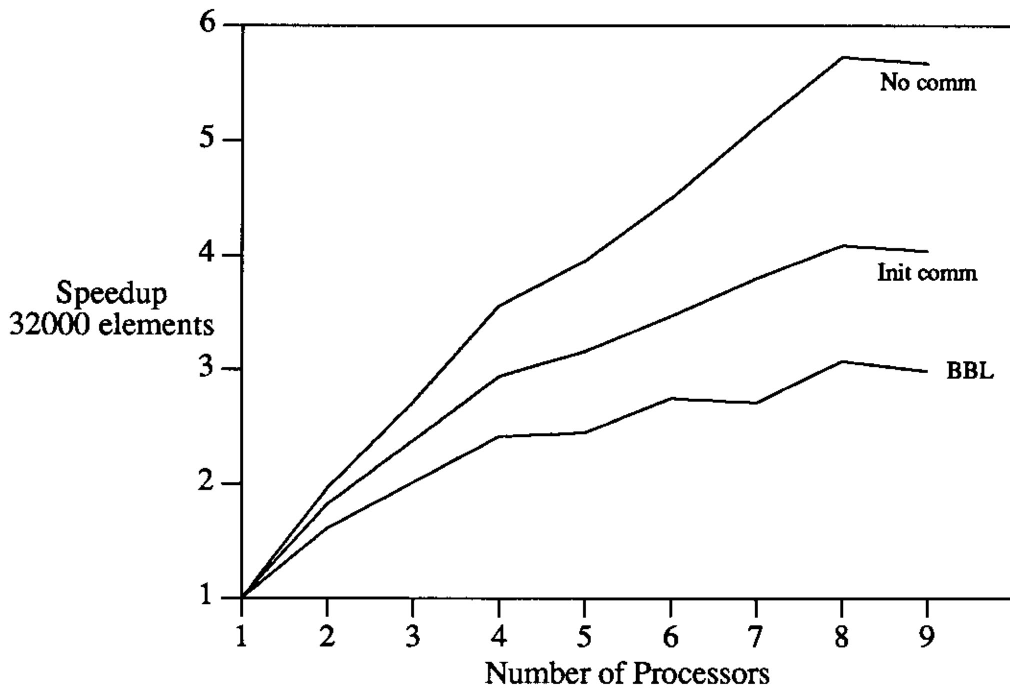


Figure 10. Speedup versus number of processors (32000 elements).

We see that communication overhead is a significant portion of the total running time of the algorithm, especially when the number of nodes and the data set is large. Due to the structure of the algorithm, we must send $P-1$ messages of size $\frac{n}{P}$ data elements to initially distribute the data. Also, when $P=2^k$ we will have to wait for one transmission of size $\frac{n}{P}$. This last cost can be seen by observing that if we send data out in reverse order (to node P, then to node $P-1$ etc.), node zero will begin sorting at the same time as node 1 (the last one to get data). In the trivial case of $P=2$, node zero and node one will finish sorting at approximately the same time. Node one then transmits its sorted list to node zero. Node zero must wait for the transmission to complete before merging its list with node one's list. It is clear that node zero will always have to wait (idle) one transmission time for the arrival of a list of size $\frac{n}{P}$ when $P=2^k$. There may also be other delays in the middle of the algorithm, but those depend on the relationship between the time to sort, merge and transmit data.

We now present a lower bound on the total time to sort a list of size n on P processors using the parallel merge sort algorithm defined above. We use the approximations discussed earlier, and the fact that we must distribute the data initially.

$$T=(P-1)\left[\frac{2m_c n}{P}+b_c\right] \qquad (\textit{initial communication})$$

$$+m_s\left[\frac{n}{P}\log_2\frac{n}{P}\right]+b_s \qquad (\textit{sorting})$$

$$+\frac{2(P-1)}{P}nm_m+b_m\log_2 P \qquad (\textit{merging}) \qquad (5)$$

We differentiate this expression with respect to P to find the optimal number of processors for this algorithm on our system. That formula is shown below.

$$\frac{dT}{dP} = \frac{n}{P^2}\left[2(m_c + m_m) - m_s\left[\log_2\frac{n}{P} + \frac{1}{\ln 2}\right]\right] + \frac{b_m}{P\ln 2} + b_c = 0 \qquad (6)$$

Unfortunately we cannot solve for P^*, the optimum number of processors to minimize T, using this equation, so we do it numerically. In Figure 11 we plot P^* versus n, the number of data elements.

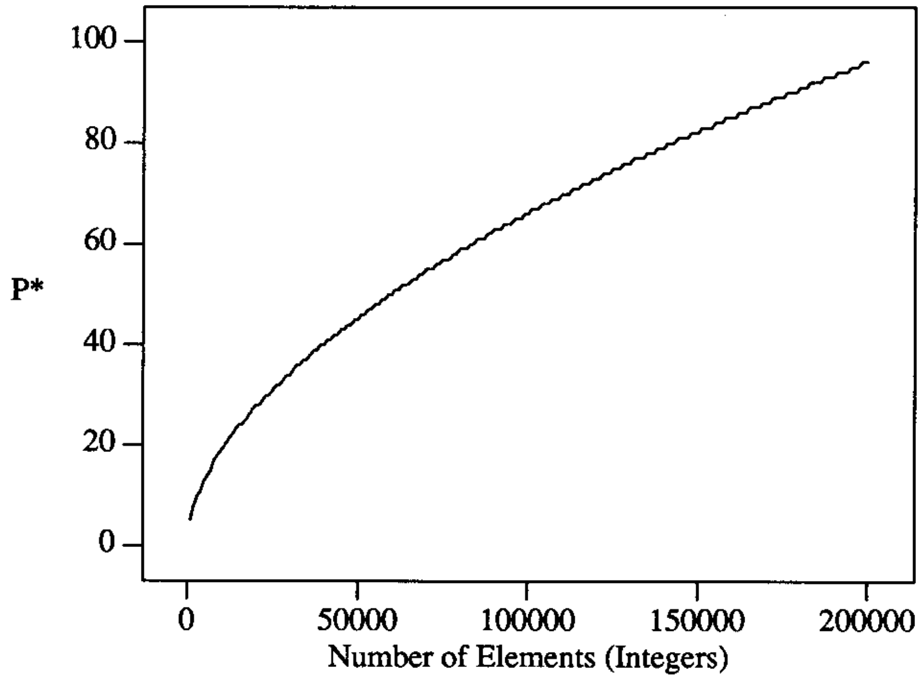


Figure 11. Optimum number of processors versus number of elements.

For example, $P^* = 36$ for 32000 data items.

We now return to examining the merging requirements of this algorithm. We have already stated that the number of times that processor zero must perform a merge operation is $\lceil \log_2 P \rceil$, but exactly how much data must be merged in each step? When $P = 2$, processor zero must perform one merge operation of size $\frac{n}{2} + \frac{n}{2} = n$. When $P = 3$ node zero must perform two merge operations, the first is of size $\frac{n}{3} + \frac{n}{3} = \frac{2n}{3}$, and the second is $\frac{2n}{3} + \frac{n}{3} = n$. Processor

zero will always perform a merge of size n (the last one) and some other merges depending on the number of processors. Since we can use a $Y = mX + b$ formula for approximating the time to merge lists, we need only look at the number of times a merge operation occurs (b term), and the total amount of data elements merged (m term). If we define a merge unit for a problem of size n as a merge operation with n elements, we see that when $P = 2$, processor zero has one merge operation with a total of one merge unit. When $P = 3$, two merges occur for a total of $\frac{5}{3}$ merge units. Figure 12 shows the exact number of merge units performed at processor zero versus the number of processors in the computation for 2 to 32 nodes.

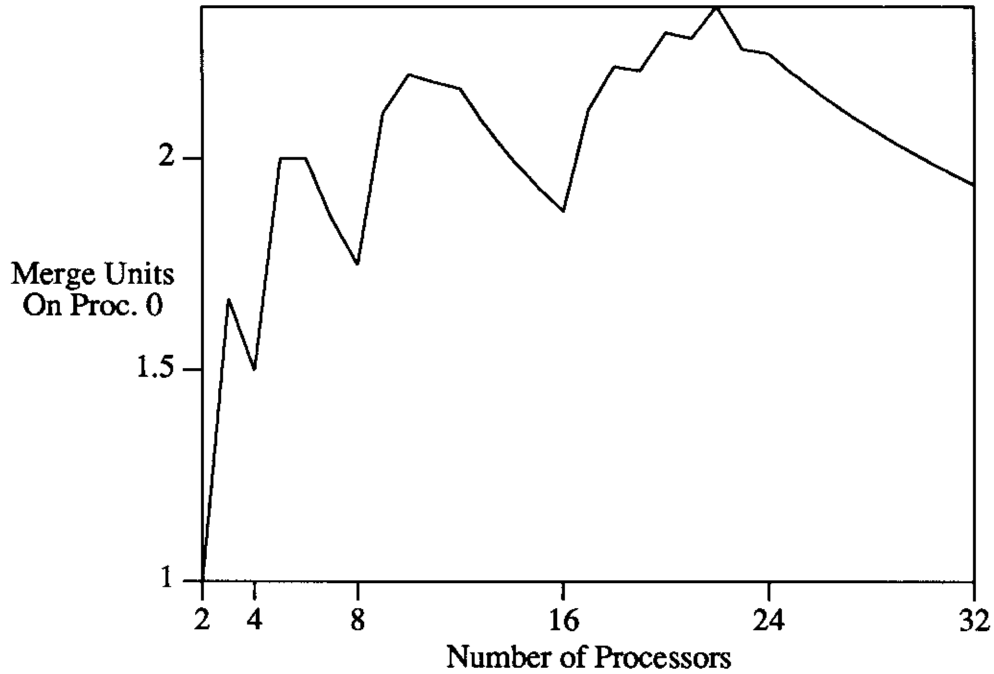


Figure 12. Total merge units versus number of processors.

We have found no closed form solution for the number of merge units versus P, but we present the following bounds. As described earlier, when $P = 2^k$ the number of merge units is exactly $\frac{2(P-1)}{P}$, which, in the limit as $P \rightarrow \infty$, equals two. This is a lower bound for all P. An upper bound is more difficult to find, and we present the following formula without proof. Let MU_u be the upper bound on the number of merge units, and MU_l be the lower bound.

$$MU_u = \frac{2P + 2^{\lceil \log_2 P \rceil - 2}}{P} \tag{7}$$

In the limit as $P \rightarrow \infty$, MU_u oscillates between two values as shown below.

$$\lim_{P \rightarrow \infty} MU_u = \lim_{P \rightarrow \infty} \frac{2P + \frac{P}{4}}{P} = \frac{9}{4} = 2.25$$

$$\lim_{P \to \infty} MU_u = \lim_{P \to \infty} \frac{2P + \frac{P}{2}}{P} = \frac{5}{2} = 2.5$$

We plot the upper and lower bounds on merge units performed at processor zero in Figure 13. The solid curve is the actual number of merge units as plotted alone in Figure 12.

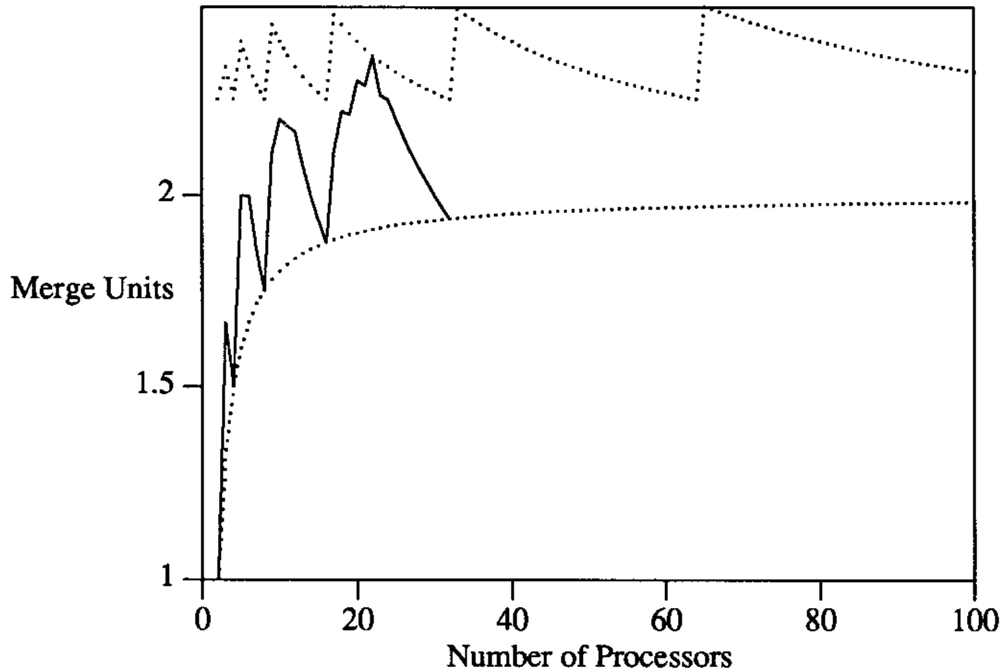


Figure 13. Bounds on merge units.

4.2.2 Eight Puzzle

Another application which was ported to the BBL system is a parallel search algorithm used to solve the eight puzzle. The algorithm is due to Curt Powley [POWL82] and is a parallel version of IDA* [KORF85]. The eight puzzle is a typical example of a search problem. As stated by Korf [KORF87], ''It consists of a 3x3 square frame containing eight numbered square tiles and an empty position called the 'blank'. The legal operators slide any tile horizontally or vertically adjacent to the blank into the blank position. The task is to rearrange the tiles from some random initial configuration into a particular desired goal configuration.'' The object of

the search is to find the optimal set of moves to solve the puzzle.

The eight puzzle runs too quickly on a single machine on the average, so parallelizing it yields no valuable information. A larger version, the Fifteen Puzzle (4x4), unfortunately can take a few hours to solve on a single PC. In order to collect data, we ran the IDA* algorithm on a 3x4 puzzle which we call the Eleven Puzzle.

Our algorithm consists of two parts, a coordinator and any number of search processes. When run on one processor, only the coordinator searches for the goal. When run on more than one processor, the coordinator runs on one node and a search process runs on each of the other nodes. When $P > 1$ the coordinator doesn't search beyond an initial threshold (a few nodes). Therefore we only begin to see speedup when $P = 3$. This is clearly inefficient, but the algorithm was ported from an application on an Intel Hypercube, where the architecture makes a coordinator desirable. Additionally, it is possible that some processors complete searching before others. Without dynamic communication between processors to share workload, we could not see linear speedup. However, this simplified version was sufficient for our needs. It is interesting to note that the conversion of the coordinator and search process code to the BBL system took less than two hours to complete. This was due to the modular structure of both the eight puzzle program and the BBL system. As a result, modification was limited to small communication modules and not to the bulk of the search code. In Figure 14 we plot the average time to solve the puzzle versus the number of processors, and in Figure 15 we show the speedup.

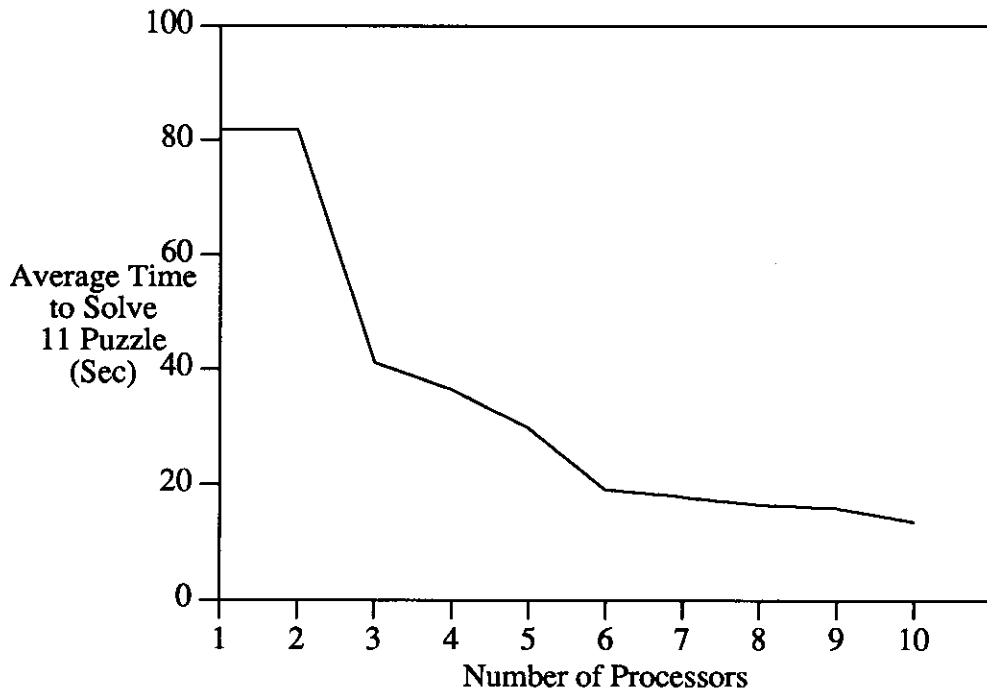


Figure 14. Average time to solve 11 puzzle versus number of processors.

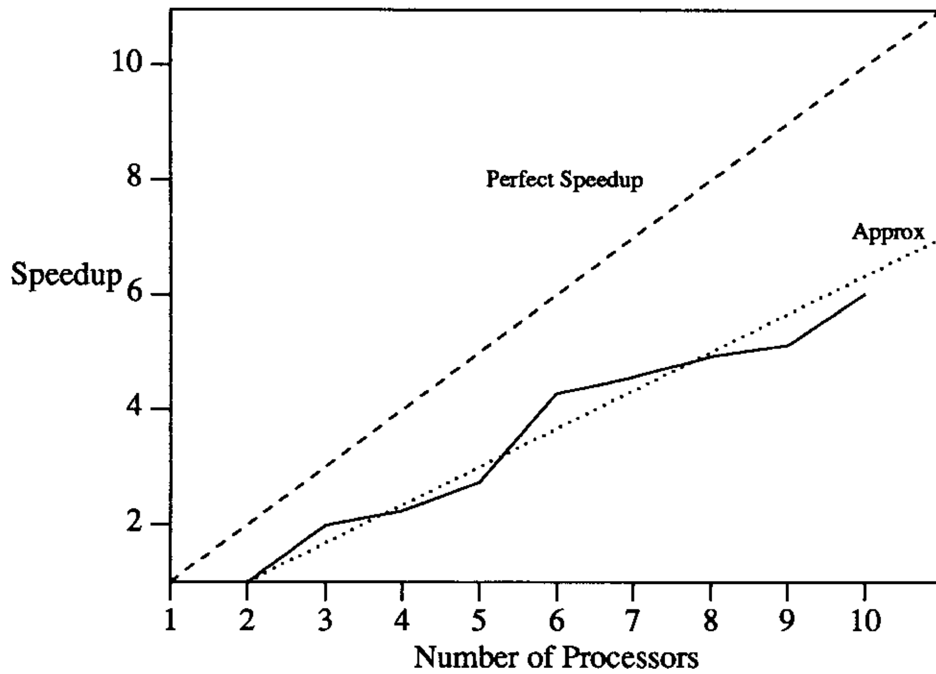


Figure 15. Speedup versus number of processors.

We ran tests on ten different puzzles and averaged the time and speedup to produce a single plot. We have included an approximation to the speedup curve as the dotted line in Figure 15. Since we have acknowledged that the coordinator performs no valuable search function, we can also plot the speedup only looking at the number of search processes available. This is simply one less than the number of processors and will produce the same speedup curve, only shifted. We plot this curve in Figure 16.

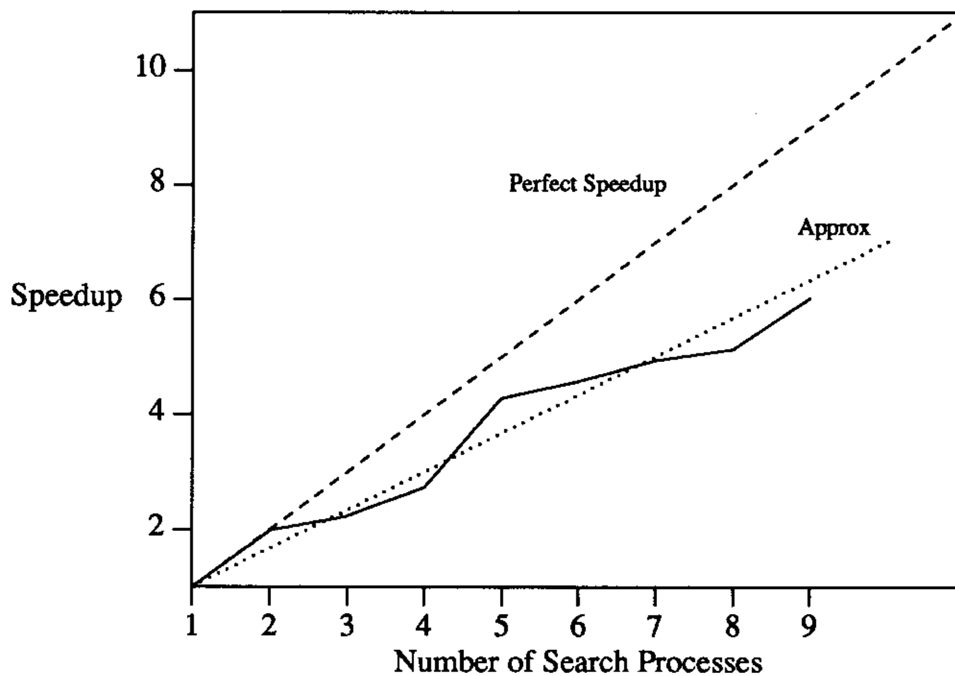


Figure 16. Speedup versus number of search processes.

5 Restoration and Recovery

We implemented two distinctly different programming approaches for handling node failures during the execution of a distributed algorithm. The first approach is exhibited in a To-

ken Passing program. Its philosophy is to allow the execution of the user program to continue until it reaches some steady state *before* returning the PC to its owner. A replacement node can therefore make assumptions about its program state when it begins running. There is no need for it to query any kind of dedicated fault-handling node or to ask for assistance from any of its neighboring nodes. In contrast to this is the second approach used by the distributed Music demo. It makes no attempt to reach some steady state in the execution before a PC is reclaimed by its owner. Instead, a newly started replacement node checks with a central, coordinator node which informs it about its current place in the computation. The tradeoff between these approaches is the delay to the PC owner versus the complexity of the recovery scheme. If a distributed algorithm has very little processing to do before it can achieve a steady state, then the first approach is acceptable, provided the delay in returning the PC to the owner remains imperceptible. Otherwise, the second approach or one similar in nature would need to be employed. A drawback, however, is that the first approach is not resilient against hard node failures, while the second approach is. If, for instance, a node crashes, the state information is unrecoverable and the algorithm will be unable to run correctly. But hard failures are rare and would kill a sequential algorithm anyway.

5.1 Token Passing

The Token Passing program was one of the original demo programs for the system. It served to test not only the user library routines for sending and receiving packets between communicating nodes, but also to exercise a program's ability to handle fault recovery. The algorithm runs with any number of nodes assigned to it and requires an underlying uni-directional token ring topology. At the algorithm's outset, the node with the lowest vid generates the ''token''. When in possession of the token, a node prints out its vid on its monitor (in a hugely oversized font), and then passes the token to the next node upstream. In this way, one can literally watch the token passing throughout the virtual ring. The interesting attribute of the

program is its behavior in the event that the PC owner takes back the PC. Before the PC is returned to its owner, the NM calls the user defined exit function as declared in the Init_Vars routine. Inside the exit function, the user code decides whether or not the computation is in a reliable state. The deciding factor is whether or not the PC has the token in its possession. If the PC does not have the token, it simply exits. If the PC does have the token, the program continues executing slightly longer. It immediately aborts displaying its vid on the monitor and shuffles the token to the next node along the ring. In this way, when a replacement node is found and is downloaded with code, it can assume it is not the owner of the token at that time. Furthermore, the token itself remains intact and the token passing continues uninterrupted.

5.2 Music

Besides the comic aspect of listening to a whole room of PCs play computerized Bach, the Music demo (Bach.alg) was instructive in that it allowed us to listen to fault tolerance at work. The algorithm is designed to run on at least three nodes. It partitions into three separate pieces of code; one plays one progression of frequencies or notes (voice1), a second (voice2) plays a different set of notes which are intended to complement the notes of voice1, and a third acts as coordinator (the conductor). The conductor code is intended to run on only one node. The other pieces of code, voice1 and voice2, should run on at least one node each. Listening to fault tolerance was best achieved though by using the minimal configuration of three nodes. If either of the nodes running voice1 or voice2 fails, it can easily be heard since only one node has been designated to play each set of specific notes. If the conductor node fails, the entire algorithm is aborted because the conductor is considered a super node. The underlying topology in this case is required to be a star network. One central node, the conductor node, is connected to all other nodes.

Before the algorithm begins, it is necessary to synchronize time at the nodes. A User Interface command option exists for this purpose or the **Time_Synch** function can be called by the conductor. Fortunately, the propagation delay of a broadcast packet is small enough so that suitable time synchronization can be achieved. The difference in node times is inaudible when the music plays. The conductor begins by sending each of its neighboring nodes a time at which to begin playing. The neighboring nodes begin by waiting for instructions from the conductor about the start time of the music. This is modeled after the protocol of a real symphony conductor who, with baton in hand, instructs the members of an orchestra when to begin playing.

If a PC's owner starts typing at the machine's keyboard, the PC exits the Music demo without storing any information about its current place in the computation. When a replacement node begins execution, it consults the conductor for state information. The conductor tells it at what time to begin playing and at what point in the music it should begin. It does not make sense for the code to repeat what it has already executed or played, since voice1 and voice2 are supposed to be synchronized and since the surviving node continues to play its series of notes even though the other voice part has disappeared temporarily. Like a well prepared symphony conductor who knows which instruments should be playing which notes at what times, the algorithm's conductor always knows how far along each node should be in the music at any given time. These calculations are time-based which is why it is imperative to synchronize the nodes before the program begins. The conductor, equipped with the starting time of the algorithm, can compute what note a node should be playing at any time, including times in the future. Because of this, the conductor can resynchronize failed nodes, even if all nodes (except for the conductor of course) fail at once. Essentially, the conductor's role is to keep the nodes, either original nodes or replacements, in synchronization with each other. The success of the conductor can be measured by the degree of polyphony or cacophony created, as the case may be.

6 Future Directions

Besides providing a robust testbed for distributed computations, the current BBL system strives to make programming in a distributed environment somewhat easier. There are however, several improvements to the system which could further simplify the programmer's tasks. Future goals include moving towards a more graphical interface, enhancing the BBL environment's flexibility, adding system accounting, assessing the portability of certain portions of the system, and perhaps making use of a central mainframe to perform the Resource Manager duties.

6.1 A Graphical Interface

The current command line orientation of the User Interface is very usable and straightforward. While it works sufficiently well, a graphical interface with a more window-plus-menus orientation might be less cumbersome. In addition, there are BBL functions that could benefit enormously from visual enhancements. For example, the BBL debugger could simultaneously display information about several nodes by using multiple windows for output.

6.2 More Flexibility

One limitation of the BBL system is its inability to handle more than one distributed algorithm per user at any given time. The idea of adding background tasks under BBL is being considered for a future implementation. This would allow several algorithms to be run simultaneously by a single user. Traditionally, multi-tasking operating systems provide this type of facility for background processes. Unfortunately, DOS is only a single-tasking operating system. In other words, DOS only provides for the execution of one process per machine. This means that the BBL system would perform the multi-tasking, but would need to simulate it at the algorithm level, as opposed to the process level. It would coordinate the usage of different sets of machines for different algorithms.

Another limitation is that it is necessary to invoke the entire BBL environment in order to run an algorithm. It has been suggested that a command line version of BBL might also be useful. A user could supply the algorithm name and any necessary parameters to a program called ''bbl'' at the command line. This command could run in the ''background'' thus allowing a user to sort a large file in parallel while editing another.

6.3 Accounting

Skeptics of the BBL system have voiced concerns about the sometimes private nature of PCs. PC owners are not always willing to make their machines readily available for public use. Some owners store confidential data on their machines, while others just make a point of not sharing their *personal* computers with anyone else. Accounting has been proposed as an inducement for owners to share PCs. The BBL system would be updated to keep track of the PCs' usage on a user basis.

In a company setting, where groups have separate budgets for their resources, BBL accounting might motivate one group to permit access to their idle machines by individuals of other groups. The loaner group could ''charge'' for the usage of their machines. The charge could translate into monetary terms or might be used in exchange for other services between the groups.

Even though the accounting facility does not currently exist under BBL, PC owners still have the choice whether or not to run under the BBL shell. PC owners can opt not to install the BBL system on their machines. They would not have to worry about their PCs being used by others. Yet, they would also no longer be able to benefit from what BBL has to offer.

Similar work at Carnegie-Mellon University indicates that users will not go out of their way to add their machines to the pool of resources (i.e. by running BBL), but if their machines automatically run the BBL-like software, users will make little effort to remove them from the

resource pool [NICH87].

6.4 Other BBL Architectures

As previously mentioned, DOS does not multi-task processes. This proved to be a surmountable obstacle, but an obstacle nonetheless. There were many occasions when a multitasking operating system would have made the implementation far easier. Because of this constraint, we have given some thought to an implementation under a multi-tasking environment, perhaps moving BBL to operate under OS/2 or UNIX. Neither the lowest level communication software, nor the expected graphic interface code would be easily portable.

Another change to the system design might entail moving the Resource Manager onto a central mainframe machine. This would eliminate the need for one of the PCs to be a dedicated RM node. Instead, the RM would simply run as a sub-process on the mainframe. The advantages here are the mainframe's fixed network address, its reliability, and its added speed.

7 Conclusions

In summary, the BBL system is an operational low cost distributed processing environment. It upholds the unique policy of borrowing idle PCs in a network and then of benevolently returning these machines to their owners when requested. It makes an effort to simplify the task of programming in a distributed system. *And, it shows that networked PCs are a viable testbed for distributed algorithms.* While the PC environment was suitable for the development of BBL, it was not without its limitations. Most notably, the lack of multi-tasking made the implementation difficult. Since BBL is a "large-grained" distributed processing system, our performance results indicate that the environment is best suited to algorithms with few communication needs. At some point, communication overhead outweighs the merits of distributing the algorithm in the first place. Therefore, computation intensive tasks are in their element under BBL. With the advent of fiber optic local area networks and associated hardware

for the PCs, BBL should work equally well for applications that are more communication intensive. This will only be possible if the hardware interfaces to the network at the PCs are able to provide short enough turn-around times to allow the PCs to fully utilize the enormous bandwidth of the fiber nets.

8 Acknowledgements

We would like to thank Naohiro Iki for his contributions to the development of the BBL system, Curt Powley for the use of his parallel search algorithm to solve the eight puzzle, and Carleton University for providing the low-level communication routines which were used as building blocks for the BBL communication primitives.

IBM, AT, DOS, and OS/2 are registered trademarks of International Business Machines Corporation. Unix is a registered trademark of AT&T. Macintosh is a trademark of McIntosh Laboratories, Inc. licensed to Apple Computer, Inc. AppleTalk is a registered trademark of Apple Computer, Inc.

9 References

[BBL88] "Benevolent Bandit Laboratory User's Manual", Technical Report 880017, Department of Computer Science, University of California, Los Angeles, CA (Mar 1988).

[BAGR85] Bagrodia, R., Chandy, K.M., Misra, J., "Distributed Computing on Microcomputer Networks", Technical Report, Department of Computer Science, University of Texas, Austin, TX (Sept 1985).

[CABR86] Cabrera, L.F., Sechrest, S., Caceres, R., "The Administration of Distributed Computations in a Networked Environment: An Interim Report", *Proceedings of the 6th International Conference on Distributed Computing Systems*, Cambridge, MA pp.389-397 (May 1986).

[COUL87] Coulas, M.F., Glenn, H.M., Marquis, G., "RNet: A Hard Real-Time Distributed Programming System", *IEEE Transactions on Computers*, vol.c-36, no.8, pp.917-932 (Aug 1987).

[GAIT87] Gait, J., "A Distributed Process Manager for an Engineering Network Computer", *Journal of Parallel and Distributed Computing*, vol.4, pp.423-437 (Aug 1987).

[GRAY86] Gray, N.A.B., Hille, R.F., "Exploiting Low Cost Computer Networks: Applications to Distributed Processing", Technical Report, Department of Computer Science, University of Wollongong, Wollongong, Australia (1986).

[JOHN87] Johnson, H.J., "Network Computing Architecture: An Overview", Internal Document, Apollo Computer, Inc., Chelmsford, MA (Jan 1987).

[KLEI84] Kleinrock, L., "On The Theory of Distributed Processing", In *Proceedings of the 22nd Annual Allerton Conference on Communication, Control and Computing*, Univ. of Illinois, Monticello, pp.60-70 (Oct 1984).

[KLEI75] Kleinrock, L., *Queueing Systems, Volume I: Theory*, Wiley Interscience, New York (1975).

[KORF87] Korf, R.E., "Real-Time Heuristic Search: First Results", *Proceedings of the 6th AAAI Conference*, Seattle Washington, pp. 133-138 (June 1987).

[KORF85] Korf, R.E., "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search", *Artificial Intelligence*, vol.27, pp.97-109 (1985).

[NICH87] Nichols, D.A., "Using Idle Workstations in a Shared Computing Environment", *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, Austin, TX, pp. 5-12 (Nov 1987).

[POWL82] Powley, C.N., "Development of a Concurrent Tree Search Program", Master's Thesis, Naval Post Graduate School, Monterey, CA (Oct 1982).

[SCHO88] Schooler, E.M., ''Distributed Debugging in a Loosely-Coupled Processing System'', Master's Thesis, Computer Science Department, University of California, Los Angeles, CA (Feb 1988).

[UNGE86] Unger, B.W., Birtwistle, G.M., Cleary, J.G., Dewar, A., ''A Distributed Software Prototyping and Simulation Environment: Jade'', *Proceedings of SCS Conference on Intelligent Simulation Environments*, San Diego, CA, pp.63-71 (Jan 1986).