The Benevolent Bandit Laboratory: A Testbed for Distributed Algorithms

ROBERT E. FELDERMAN, EVE M. SCHOOLER, AND LEONARD KLEINROCK, FELLOW, IEEE

Abstract—**We describe the design, implementation, and use of a distributed processing environment on a network of IBM PC's running DOS. Temporarily unused PC's 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, some 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. Thus, a distributed computation is able to proceed despite a set of transient processors. A discussion of system performance, distributed applications, and fault tolerance is included. In particular, performance improvements are demonstrated by applications like parallel merge sort and a distributed search solution to the eight puzzle.**

I. Introduction

During this decade, we have witnessed the rapid proliferation of personal computers and workstations. 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, toward environments where each user has a personal computer or workstation connected to a local area network. A 1984 survey by the International Data Corporation showed that the U.S. 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 a much greater difference between these two figures, showing that the raw PC computing power is tremendous. Yet most of these PC MIPS are badly underutilized (what is your PC doing right now?).

This situation was understandable as long as the PC was, indeed, a *personal* computer; however, many of these PC's are now connected together via local area networks (LAN's) and, as such, now comprise a system. Thus, we now have the opportunity to recapture those unused MIPS. Although these new PC-LAN's are seemingly more practical for an environment where small scale tasks such as word processing are performed, there is precious little sharing of resources. 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 systems each with a much reduced capacity and smaller arrival rate of jobs; Kleinrock has shown [8] that this leads to an inefficient use of resources. For example, consider the task of sorting a large database. With a mainframe, the task is trivial, although it may be delayed if the system load is heavy. By contrast, in a networked PC environment the task usually runs on a single PC. The job would take many hours depending on its size, especially if it cannot fit into the memory of the PC in its entirety. Compounding our frustration with the job's slow execution is the realization that additional idle processing power of other PC's lies unused while the owners are off doing something else.

The aforementioned problems led us to create a system in which temporarily unused PC's can be accessed by other users on the network to perform distributed computations. The owner of the 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. It is this notion of transparency to the owner of a PC that led us to call our system the Benevolent Bandit Laboratory (BBL), since PC CPU cycles are "stolen" in a benevolent fashion and then the PC's are returned to their owners upon request. Another motivation for the construction of the system was the need for a testbed for distributed algorithms. Much current research is directed toward designing distributed algorithms, but many of these algorithms go untested because distributed systems are unavailable. The BBL system is a low cost solution to this problem. The network, developed originally for file sharing, was already in place, we simply created software to allow the network to support distributed processing.

Although other loosely coupled distributed programming environments exist [2]–[7], [16], none incorporates the notion of benevolent CPU stealing. The only system we know of which is designed specifically to run on a low cost network of PC's is from the University of Wollongong [6]. It uses a number of Macintosh computers linked together through the AppleTalk network. The main limitation of that system is the network's low rate of data transmission. At best, the data transfer capac-

Manuscript received November 1, 1987; revised August 1, 1988. 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.

R. E. Felderman and L. Kleinrock are with the Department of Computer Science, University of California, Los Angeles, CA 90024-1596.

E. M. Schooler was with the Department of Computer Science, University of California, Los Angeles, CA. She is now with Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292-6695.

IEEE Log Number 8825036.

ity of the network is 200K bytes/s. This proved to impair the usefulness of distributing computations, since communication overhead was so high. The other distributed environments are composed of more powerful machines (workstations, servers, mainframes) running multitasking operating systems (primarily UNIX). Naturally, this leads to more complicated process management. Of these systems, only a few address the issue of crash recovery. Cabrera *et al.* [3] present the idea of a personal Process Manager (PPM) to relay information about node failures and to reestablish internal consistency. The PPM does not appear to manage resilient computations. 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 [5]; 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.

II. The Environment

The BBL system was developed to provide an environment for distributed processing on IBM PC-AT's running DOS 3.1 and connected together via an Ethernet. The network of approximately 100 PC-AT's was already available at UCLA. A unique feature of BBL 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 using the machine 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. 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, the computation may 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 PC's and DOS do not support virtual memory mapping, so this solution was ruled out.

III. 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 it with the BBL system. The central resource coordinator, running on a dedicated machine, which keeps track of available PC's, is called the *Resource Manager* (RM). The code which allows a user to interact with the idle PC's is actually two modules on one machine, the *User Interface* and the *Process Manager* (UI/PM). The User Interface provides the link 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 below. The logical interconnection between the modules is shown in Fig. 1. A more detailed description of the system can be found in [14].

A. Node Manager

The Node Manager is designed to run as a shell on top of DOS and emulate its operation. Normally, a PC is configured to automatically execute the NM when the system boots. 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 the PC is displaying a DOS prompt, waiting for the owner to type a command. The length of time after which a machine is considered idle, or available for BBL use, is simply a selected parameter which is passed to the NM program. While the NM shell waits for the owner to type commands, it decrements a timer. If 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. If the timer expires, the NM registers itself with 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 messages from the Process Manager which contain code to execute, possibly an input file, and addresses of other nodes involved in the distributed computation.

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.

Packets are used to exchange information. 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 is responsible for the translation of a vid into a physical address. The NM

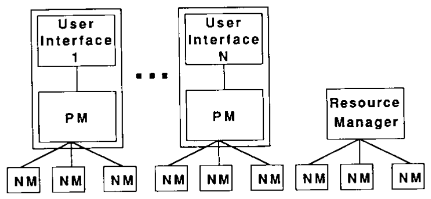

Fig. 1. System diagram.

is also 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, initialization routines, communication primitives, and file operations which allow the NM's to read from and write to files on the PM's disk. Since the NM is designed to run executable files, the majority of the user code can be compiled and tested outside the BBL environment. Only when the bulk of the code is debugged is it necessary to run the distributed computation on the BBL system. Essentially, everything except communication between different processors can be pretested.

B. Resource Manager

The Resource Manager is a dedicated machine responsible for keeping track of available PC's 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. The RM uses this pool 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 PC's to fulfill all the requests.

One drawback to this design it that the RM is a dedicated machine that does nothing but handle resources for BBL. A method to eliminate the RM from the system would be to allow each UI/PM to "find" its own idle PC's. This would eliminate the RM altogether, but would add complexity to 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 other currently running UI/PM's and "beg" for hoarded nodes. By centralizing the Resource Manager, each PM need only go to one location to find free nodes, and the RM can provide equitable sharing of resources between multiple users.

C. 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 runtime, to query the system about the state of the computation and system resources, to observe output, to run the BBL debugger, etc.

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 on several PC's. 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 voice 2), the names of the code segment executable files, 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, and finally the name of any input files to be redirected to the executables.

In the original version of the UI, the algorithm environment was initialized through the use of an algorithm configuration file plus lengthy interactive questioning from the UI. With the addition of debugger options, the setup of an algorithm environment became even more elaborate, if not more complicated. Accordingly, we expanded the configuration file to allow the user to provide all the necessary information, thus eliminating the lengthy interactive setup. For more information about the configuration file format, default assumptions, and the debugger see [1] and [13].

Once the algorithm environment is set up and the algorithm information is downloaded to the participating NM's, 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.

D. Process Manager

The Process Manager coordinates 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 NM's. 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 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 NM's since the NM's provide this same translation for the user's algorithm. 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 NM's 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, the execution of the algorithm is aborted. Otherwise, the algorithm continues with one fewer node.

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.

IV. 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.

A. Communication Throughput

One performance measure of our system is the communication throughput between two nodes running a distributed algorithm. 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 Fig. 2.

As can be seen from the plot, the maximum throughput is limited to slightly under 200 kbits/s. The Ethernet is a 10 Mbit/s medium. Why then do we see only 1/50 of the maximum throughput? A major loss of throughput is due to the use of a stop-and-wait protocol with a "large" turnaround time at each processor. Since the boards which connect the PC's to the Ethernet only have one packet buffer, every arriving packet must be read off the board, examined, and then acknowledged if necessary. Although the actual turnaround time is difficult to measure, a time of 0.03 s will give, roughly, a channel utilization of 0.02, or a factor of 50 reduction from maximum throughput [15]; since we observe a factor of 50, we suspect that 0.03 s is a good estimate. This is where most of our throughput is lost, since our overhead is fairly high. We must bear in mind, however, 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 is 1 Mbit 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.

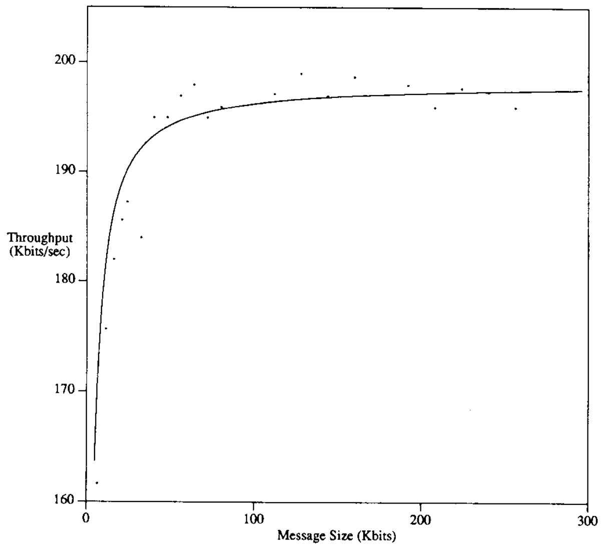

Fig. 2. Measured throughput versus message size.

B. Merge Sort

One application we have coded for the BBL system is the 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 $n/2$ at processor 0. For example, with $P = 4$, node zero sends messages of size $n/4$ to the other three processors. Each of the 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 sorted data ($n/2$ data elements) to node zero which performs a merge operation to finish sorting the list. We show a timing diagram in Fig. 3.

Fig. 4 shows the total time to sort lists of size 4000, 8000, 16 000, and 32 000 data items with one to nine processors.

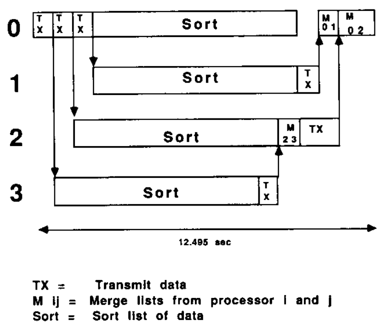

Fig. 3. Four node merge sort timing diagram (32 000 data items).

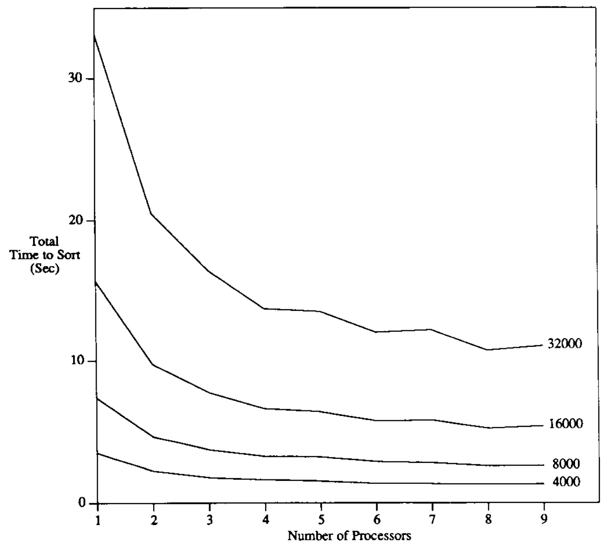

Fig. 4. Total time versus number of processors (merge sort).

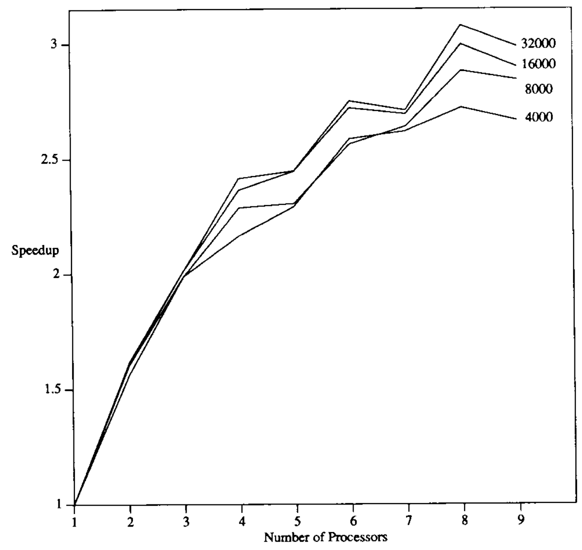

Fig. 5. Speedup versus number of processors (merge sort).

The corresponding speedup graph is given in Fig. 5. The speedup is measured with respect to sorting the entire list on one processor with an $n \log n$ sort (no merges). In general, we would expect the speedup to increase with P and to be best where $P = 2^k$.

To understand these results better, let us compare them, not to linear speedup, but to the theoretical maximum speedup using this version of merge sort. Since there are two sources of imperfect speedup, namely, the algorithm itself and the architecture it runs on, we attempt to isolate the contribution of each. Specifically, we compare our results to that for a parallel merge sort where communication is free with zero delay; this eliminates any loss due to the architecture. We 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) PC's to gather data for the time to sort a list of intergers and the time to merge two sorted lists. From the measured data, we created the following linear approximations to the actual time to perform the various parts of the algorithm. All of our approximation formulas are of the form $Y = mX + b$. $T_c(n)$ is the 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).

$$\begin{aligned} T_c(n) &= m_c kn + b_c \\ m_c &= 0.00004036 \quad &&\text{(seconds/byte)} \\ b_c &= 0.0051134 &&\text{(seconds)} \\ k &= 2 &&\text{(bytes/integer)} \qquad (1) \end{aligned}$$

$$\begin{aligned} T_s(n) &= m_s(n \log_2 n) + b_s \\ m_s &= 0.00006862 \quad &&\text{(seconds/integer)} \\ b_s &= 0.248614 &&\text{(seconds)} \qquad (2) \end{aligned}$$

$$\begin{aligned} T_m(n) &= m_m n + b_m \\ m_m &= 0.00004015 \quad &&\text{(seconds/integer)} \\ b_m &= 0.001549 &&\text{(seconds).} \qquad (3) \end{aligned}$$

Using the above approximations, we compare our empirical results to results we would obtain for an ideal system with no communication overhead. For simplicity, initially assume that $P = 2^k$ and n is the total size of the list to be sorted. Further assume that n is evenly divisible by P. 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 takes place. We ignore the time to generate the list of integers. Neglecting communication, processor zero will first sort a list of size n/P, followed by a series of merge operations where the total number of items being merged is of the following format: $2n/P$, $4n/P$, $8n/P$, $\cdots$, Pn/P. That is, processor zero will perform $\log_2 P$

merges where the ith merge is of size $2^i n/P$. 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 a parallel merge sort in the absence of communication overhead. In Figs. 6 and 7 we compare the empirical results to the approximation above for 32 000 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 only 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. 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. As the number of processors increases, the time to sort and merge the data decreases, but the amount of time spent on communication increases. The relative cost of communication versus processing determines the optimum number of processors to use for a particular data set.

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.

$$T = (P-1)\left(\frac{2m_c n}{P} + b_c\right) \quad \text{(initial communication)}$$
$$+ m_s\left(\frac{n}{P}\log_2\frac{n}{P}\right) + b_s \quad \text{(sorting)}$$
$$+ \frac{2(P-1)}{P} n m_m + b_m \log_2 P \quad \text{(merging)}. \tag{5}$$

We differentiate this expression with respect to P, and obtain the expression which must be satisfied by P^* to minimize T for this algorithm on our system. That expression is shown below.

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

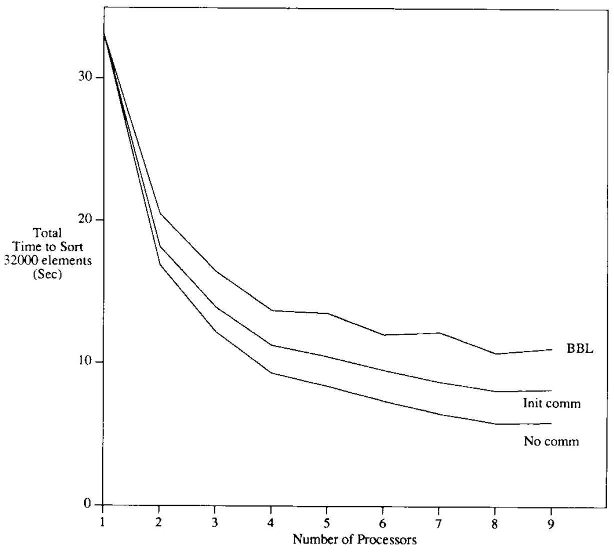

Fig. 6. Total time versus number of processors (32 000 elements).

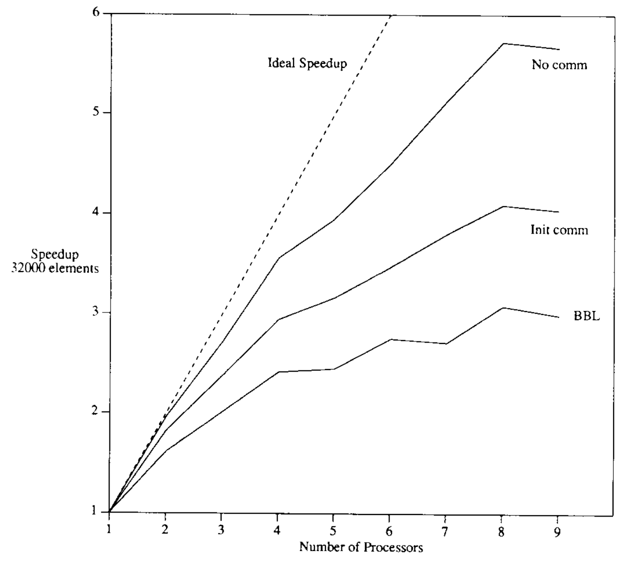

Fig. 7. Speedup versus number of processors (32 000 elements).

Unfortunately we cannot solve for P^*, so we do it numerically. In Fig. 8, we plot P^* versus n, the number of data elements. For example, $P^* = 36$ for 32 000 data items.

C. Search: The Eleven 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 Powley [12] and is a parallel version of IDA* [9]. The eight puzzle is a typical example of a search problem and, as stated by Korf [10], "consists of a 3 × 3 square frame containing eight numbered square tiles and an empty position called the "blank." The legal operators slide any tile horizontally

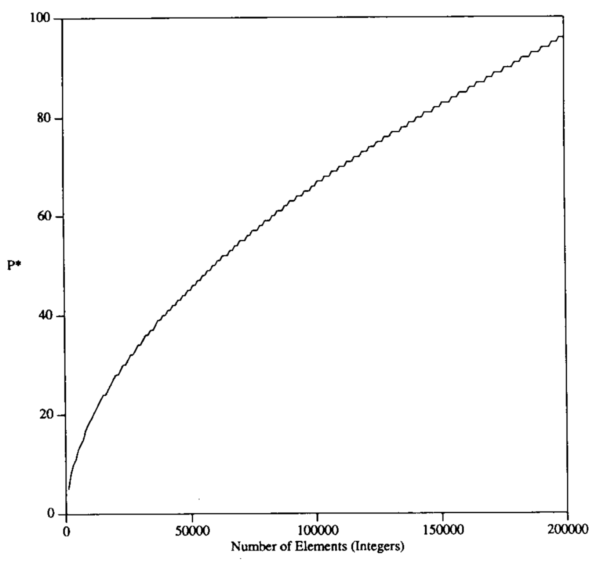

Fig. 8. Optimum number of processors versus number of elements.

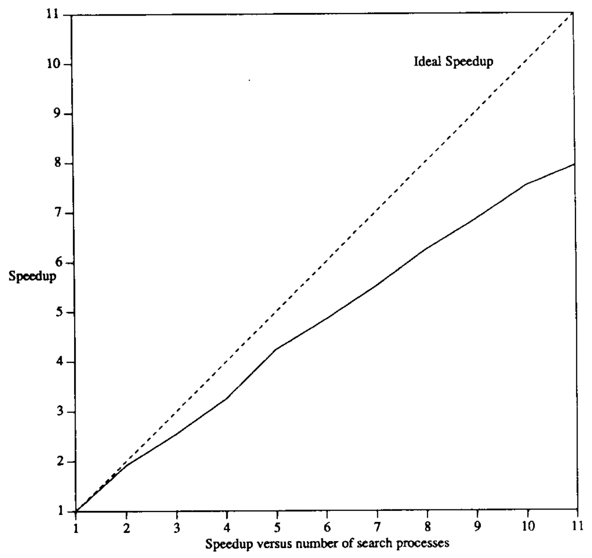

Fig. 9. Speedup versus number of search processes for the 11 puzzle.

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 (i.e., minimal number) 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 (4 × 4), however, can take up to 100 h to solve on a single PC. Therefore, to collect data, we ran the IDA* algorithm on a 3 × 4 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 does not search beyond an initial threshold (a few nodes). Therefore, we only begin to see speedup when $P = 3$. This is clearly inefficient, but, for simplicity, we ported the algorithm 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. Interestingly enough, the conversion of the coordinator and search process code to the BBL system took less than 2 h 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.

We ran tests on ten different puzzles and averaged the speedup to produce a single plot. Since we have acknowledged that the coordinator performs no valuable search function, we 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 a shifted speedup curve. We plot this curve in Fig. 9. We see a relatively good speedup as compared to perfect speedup.

Other applications coded for the BBL system include matrix multiplication, linear programming, traveling salesman, and two-player game tree search.

V. Restoration and Recovery

We implemented two distinctly different programming approaches for handling node failures during the execution of a distributed algorithm. A node failure occurs when the owner of a machine reclaims it during the operation of a distributed algorithm. The first approach is exhibited in a Token Passing program. Its philosophy is to try to allow the execution of the user program to complete execution *before* returning the PC to its owner. When running this distributed program, several nodes are logically connected in a unidirectional ring. The first processor begins execution by printing its vid on the PC screen in a very large font. After clearing the screen, it sends a message (token) to its neighbor. Each processor waits for the token, prints its vid once it receives the token, then passes the token on. If a key is hit on some processor, that processor simply tests whether or not it has the token; if so, it simply forwards it to the next node before returning control of the PC to the owner. If the node does not have the token, it simply terminates. 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 complete execution of the task at hand before the 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. In this algorithm, nodes reclaimed by their owners simply terminate by notifying the PM. Each replacement node, when it starts execution, checks with a "conductor" node. The conductor responds with a time and location in the music where the replacement node should start *playing*.

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 complete, 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 protected 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 node failures are rare and would kill a sequential algorithm anyway.

VI. Security Issues

Unfortunately, using DOS and IBM PC's limits the amount of security we can provide the owner of a PC. Because DOS is single-tasking, there are no restrictions on the operation of a process. The entire memory space is accessible to a process as are all peripherals (i.e., disks). Without herculean effort, we cannot discover that the user code is operating maliciously. One option, not yet explored, is to modify the user code compiler to prevent operations like disk reads and writes. Unfortunately, a clever programmer could circumvent these precautions. With the current version of BBL, owners must trust the honesty of the users of the system. Work is continuing in this area.

Skeptics of the BBL system have voiced concerns about the sometimes private nature of PC's. 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. The question arises whether anyone will allow their machines to be accessed by a BBL-like system. Fortunately, some similar work at Carnegie-Mellon University indicates that users do not go out of their way to add their machines to the pool of resources (by running BBL-like software), but if their machines automatically run BBL-like software, they make little effort to remove them [11].

VII. Conclusions

In summary, the BBL system is an operational low cost distributed processing environment. It upholds the unique policy of borrowing idle PC's 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 PC's are a viable computing resource.* While the PC environment was suitable for the development of BBL, it was not without its limitations. Most notably, the lack of multitasking made the implementation difficult. Our performance results indicate that the environment is best suited to algorithms with few communication needs, since BBL is a "large-grained" distributed processing system; at some point, communication overhead outweighs the benefits of distributing the algorithm in the first place. Therefore, computation intensive tasks are in their element running under BBL.

Note: IBM, AT, and DOS 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.

Acknowledgment

Special thanks to the reviewers for their insightful comments and suggestions. We would like to thank Carleton University for providing the low-level communication routines which were used as building blocks for the BBL communication primitives.

References

[1] "Benevolent Bandit Laboratory user manual," Tech. Rep. CSD-880017, Dep. Comput. Sci., Univ. Calif., Los Angeles, CA, Mar. 1988.

[2] R. Bagrodia, K. M. Chandy, and J. Misra, "Distributed computing on microcomputer networks," Tech. Rep., Dep. Comput. Sci., Univ. Texas, Austin, TX, Sept. 1985.

[3] L. F. Cabrera, S. Sechrest, and R. Caceres, "The administration of distributed computations in a networked environment: An interim report," in *Proc. 6th Int. Conf. Distributed Comput. Syst.*, Cambridge, MA, pp. 389-397, May 1986.

[4] M. F. Coulas, H. M. Glenn, and G. Marquis, "RNet: A hard real-time distributed programming system," *IEEE Trans. Comput.*, vol. C-36, pp. 917-932, Aug. 1987.

[5] J. Gait, "A distributed process manager for an engineering network computer," *J. Parallel Distributed Comput.*, vol. 4, pp. 423-437, Aug. 1987.

[6] N. A. B. Gray and R. F. Hille, "Exploiting low cost computer networks: Applications to distributed processing," Tech. Rep. Dep. Comput. Sci., Univ. Wollongong, Wollongong, Australia, 1986.

[7] H. J. Johnson, "Network computing architecture: An overview," Internal Document, Apollo Computer, Inc., Chelmsford, MA, Jan. 1987.

[8] L. Kleinrock, "On the theory of distributed processing," in *Proc. 22nd Annu. Allerton Conf. Commun. Contr. Comput.*, Univ. Illinois, Monticello, pp. 60-70, Oct. 1984.

[9] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial Intell.*, vol. 27, pp. 97-109, 1985.

[10] ——, "Real-time heuristic search: First results," in *Proc. 6th AAAI Conf.*, Seattle, WA, pp. 133-138, June 1987.

[11] D. A. Nichols, "Using idle workstations in a shared computing environment," *Proc. ACM Symp. Operating Syst. Principles*, Austin, TX, pp. 5-12, Nov. 1987.

[12] C. N. Powley, "Development of a concurrent tree search program," Master's thesis, Naval Post Graduate School, Monterey, CA, Oct. 1982.

[13] E. M. Schooler, "Distributed debugging in a loosely-coupled processing system," Master's thesis, Comput. Sci. Dep., Univ. Calif., Los Angeles, CA, Feb. 1988.

[14] E. M. Schooler, R. E. Felderman, and L. Kleinrock, "The Benevolent Bandit Laboratory: A testbed for distributed computing using PC's on an Ethernet," Tech. Rep. CSD-880016, Comput. Sci. Dep., Univ. Calif., Los Angeles, CA, Mar. 1988.

[15] A. S. Tanenbaum, *Computer Networks.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[16] B. W. Unger, G. M. Birtwistle, J. G. Cleary, and A. Dewar, "A distributed software prototyping and simulation environment: Jade," in *Proc. SCS Conf. Intell. Simulation Environ.*, San Diego, CA, pp. 63-71, Jan. 1986.

Robert E. Felderman was born in Evanston, IL, in 1962. He received the B.S.E. degree (magna cum laude) from Princeton University in 1984 with a double major in EECS and engineering and management systems, and he received the Master's degree in computer science from UCLA in 1986.

He spent a year at Hughes Aircraft Company, working on guidance systems in the Underwater Systems Lab. He is currently pursuing the Ph.D. degree, specializing in distributed systems.

Mr. Felderman is a member of Tau Beta Pi and Sigma Xi.

Eve M. Schooler was born in New York, NY, in 1961. She received the B.S. degree in computer science from Yale University in 1983 and the Master's degree in computer science in 1987 from UCLA.

She worked in R&D at Apollo Computer, Inc. for two years. She is currently a Research Scientist at Information Sciences Institute in the Systems Division and is involved in a multimedia conferencing project.

Leonard Kleinrock (S'55-M'64-SM'71-F'73) received the B.S. degree in electrical engineering from the City College of New York in 1957 (evening session) and the M.S.E.E. and Ph.D.E.E. degrees from the Massachusetts Institute of Technology in 1959 and 1963, respectively.

While at M.I.T. he worked at the Research Laboratory for Electronics, as well as with the Computer Research Group of Lincoln Laboratory in advanced technology. He joined the Faculty at the University of California, Los Angeles, in 1963. He is a Professor of Computer Science at U.C.L.A. His research interests focus on local area networks and computer networks, performance evaluation of distributed systems. He has had over 150 papers published and is the author of five books—*Communication Nets: Stochastic Message Flow and Delay* (1964); *Queueing Systems, Volume I: Theory* (1975); *Queueing Systems, Volume II: Computer Applications* (1976); *Solutions Manual for Queueing Systems, Volume I* (1982), and, most recently, *Solutions Manual for Queueing Systems, Volume II* (1986). He is Co-director of the U.C.L.A. Computer Science Department Center for Experimental Computer Science and is a well-known lecturer in the computer industry. He is the principal investigator for the DARPA Advanced Teleprocessing Systems contract at U.C.L.A. He is also founder and CEO of Technology Transfer Institute, a computer/communications seminar and consulting organization located in Santa Monica, CA.

Dr. Kleinrock is a member of the National Academy of Engineering, a Guggenheim Fellow, a member of the IBM Science Advisory Committee, and in 1986, he became a member of the Computer Science and Technology Board of the National Research Council. He has received numerous best paper and teaching awards, including the ICC 1978 Prize Winning Paper Award, the 1976 Lanchester Prize for outstanding work in Operations Research, and the Communications Society 1975 Leonard G. Abraham Prize Paper Award. In 1982, as well as having been selected to receive the C.C.N.Y. Townsend Harris Medal, he was co-winner of the L. M. Ericsson Prize, presented by His Majesty King Carl Gustaf of Sweden, for his outstanding contribution in packet switching technology. In July of 1986, he received the 12th Marconi International Fellowship Award, presented by His Royal Highness Prince Albert, brother of King Baudoin of Belgium, for his pioneering work in the field of computer networks.