THE BENEVOLENT BANDIT LABORATORY
USER MANUAL FOR SOFTWARE VERSION 3.0

Eve Schooler
Robert Felderman

March 1988
CSD-880017

Table of Contents

The Benevolent Bandit Laboratory †

User Manual
For Software Version 3.0

Abstract

This paper is the User Manual for the Benevolent Bandit Laboratory (BBL). BBL is 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.

1 Introduction

This manual is designed to facilitate the installation and use of the Benevolent Bandit Laboratory (BBL) software. The BBL system is a software package that runs on IBM PC's running DOS. The system allows the user to access idle CPU cycles on other machines to perform a distributed computation.

2 Installation

This software requires the following hardware:

1) At least 3 IBM PC-AT or compatible systems
2) 3Com EtherLink Board #3C501 for each PC
3) Ethernet interconnection network

and software:

1) DOS version 3.1 including virtual disk installation (e.g. D: disk).

A BBL software release consists of three executable software modules:

1) Resource Manager (RM)
2) Node Manager (NM)
3) User Interface / Process Manager (UI/PM)

and two files (BBLUSER.H, BBLUSER.LIB) for creating code to run on the system.

† 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.

The Resource Manager must be installed and running on only ONE (1) machine on the network. The other two modules can be installed and running on any number of machines on the network. Each piece of software should carry the same version number, otherwise compatibility problems may result. To find the version number of a module, consult the **Operation** subsection of the appropriate module's section.

3 Resource Manager

3.1 Description

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 an "I_AM_UP" message from the NM running on that PC. The Resource Manager then adds this node to its pool of free nodes. Included in this 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 regarding 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 UI/PM. 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.

3.2 Operation

The RM is invoked by running the executable file "RM.EXE". Execute the RM by typing:

```
RM
```

When the RM begins execution, the following message appears on the PC screen:

```
BBL Resource Manager Version X.XX
```

where "X.XX" will be the version number of the RM. This version number should match those on both the UI/PM and all the NMs. This manual is intended to describe the operation of software version 3.0.

After printing its version number, the RM prints additional status information. Currently it broadcasts a RESET_CHANNEL message, delays, then broadcasts an I_AM_THE_RM message. If any NMs are on the net and up and running, the RM indicates that it has acquired NMs or registered users by printing the number of users on the system, the number of available NMs and the current time on the screen. As these numbers change, the current values will be updated on the screen. A typical example will look like this:

```
BBL Resource Manager Version  3.00

Sending reset channel message
Delaying
Sending I_AM_THE_RM message
Delaying
Users =  0    Nodes =  1     16:33:11.74
WHERES_THE_RM
Users =  0    Nodes =  1     16:33:35.69
Users =  0    Nodes =  1     16:33:42.28
Users =  1    Nodes =  0     16:33:46.95
Users =  1    Nodes =  0     16:33:56.12
```

3.3 Termination

The RM can be terminated by simply typing any key on the keyboard. The system monitors the keyboard continuously, and terminates the execution of the RM whenever a key is hit. Once a key is hit the following messages should appear:

```
GOT KEY HIT.... TERMINATING
Pointer is NULL
First Reset got (80)
Second Reset got (89)
RESTORING INTERRUPT VECTOR
```

followed by the DOS prompt. These are simply diagnostic messages. If they do not appear exactly as listed above, there is no cause for alarm. The RM can be restarted by simply typing "RM".

4 Node Manager

4.1 Description

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 idle when it 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. The system is normally configured to automatically execute the shell 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. 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.

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 (D:) 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.

4.2 Operation

The NM runs as a process on top of DOS. It is invoked by running the executable file called "SHELL.EXE". The Node Manager will begin execution by simply typing

SHELL n ("n" replaced by an integer)

The parameter passed to the shell is the number of seconds that the machine must be idle before the Node Manager takes over. The only indication that the NM is running is by the prompt printed on the screen. It is generally of the form "BBL [C:TMP]". The prompt begins with "BBL" and then contains the name of the current directory inside brackets. The above example shows that the default directory is "TMP" on the C disk (hard disk). Operation at this point will appear to the user to be the same as if DOS were running. Any DOS commands simply pass through the program and are

executed by DOS. There are some exceptions to this. The current version of the NM (3.0) will not work in conjunction with PC Interface (PCI), the program to allow file access to a mainframe running a PCI server. This is because both the NM and PCI use the 3Com EtherLink board and we have yet to resolve the contention. Therefore, **do not run the NM and PCI simultaneously!** They are actively hostile to each other and the machine. You will be forced to reboot the machine if these two programs are running concurrently.

There are several commands that are not passed on to DOS for execution. These are special commands reserved to allow the user to get some information from the NM. These commands are listed below:

BBLADDRESS
BBLDEBUG
BBLHELP
BBLVER
BBLEXIT
BBLQUIT

"BBLADDRESS" will type the address of the 3Com EtherLink board on the screen. The address is listed in octal and hex formats. This function is normally only needed by the system programmers for diagnostic purposes. "BBLDEBUG" toggles the NM debugging flag. When the flag is set to TRUE, debugging information is printed out on the screen. "BBLHELP" will list the BBL commands and their actions. "BBLVER" will type the following message on the screen:

BBL Node Manager Version X.XX

where "X.XX" will be the version number of the software (e.g. 3.0). The last two commands (BBLEXIT,BBLQUIT) cause the NM to terminate and return direct control to DOS. If the owner opts not to include her PC in the pool of BBL NMs, the NM shell can be terminated by typing "BBLEXIT" or "BBLQUIT" to it.

4.3 Termination

As mentioned above, to terminate the NM shell, simply type "BBLQUIT" or "BBLEXIT" at the BBL prompt.

5 User Interface/Process Manager

5.1 Description

The User Interface is invoked by running the executable file "UIPM.EXE". Execute the UI/PM by typing:

UIPM

When the UI/PM begins execution, the following message appears on the PC screen:

> The Resource Manager has *x* nodes available
>

The prompt for the UI/PM environment is a ">". The value of *x* is the number of nodes the RM has available at the time the UI/PM is initialized. If there is no message from the RM, it implies no connection has been established yet between the UI/PM and RM. In this case, type the **rm** command and follow it by typing **free** to find out how many nodes exist at the RM.

The User Interface serves as the interface between the user and the BBL environment. It is intended to aid in the management of the distributed application. Although there are hopes for a graphical interface, the UI is currently table-driven. 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 turn on debugging facilities, et cetera.

The Process Manager manages the application processes owned by the user. It also provides the low-level communication needs of the UI. It handles the 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 RM allocates nodes to the UI/PM, the PM becomes responsible for keeping track of the mapping between physical addresses and vids. The PM shares the mapping with the NMs for use by the user code. This address-to-vid mapping is especially important in the event that a node is taken away from the algorithm by its owner. If a replacement node is found, the PM assigns it the vid of the node being returned to its owner. Since the user's algorithm only deals with vids, it is shielded from ever having to know that the underlying physical address was changed.

The PM also stores the status of each of its nodes. A node allocated to the PM 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. The PM's strategies on fault tolerance and on avoidance of race conditions are discussed further in [SCHO88].

5.2 Operation

The UI is command-line oriented. The UI command shell prompts the user for commands and takes actions accordingly. There are two classes of commands; those considered legal during the initialization of a distributed algorithm (the set up phase) and those intended for use during and after the algorithm's execution (the runtime phase). Thus, the UI presents a different set of commands depending on whether the system is in the set up or the runtime phase. A user selects a command by either typing out the name in full or typing however many characters are needed for the UI to recognize the name as unique. A description of the UI command language follows;

User Interface Command Language		
Command	**Phase†**	**Description**
algorithm	S	Choose an algorithm from the "info.bbl" algorithm library file
allocate_nodes	S	Indicate the number of nodes to allocate to the UI/PM
bye	B	Exit from the BBL environment
change_links	R	Change the logical topology contained in the connection matrix; either alter the underlying physical address of an endpoint of a link, or enable or disable a link
code_class	S	Map virtual ids to code classes (or code segments)
command_line	R	Modify the command line for running the distributed algorithm (this is useful when one wants to experiment with input parameters to a program without having to re-download code)
data	S	Set up input file or input parameter data
debug_control	B	Establish environment variable defaults for the collection of traces and performance measurements; provide debug control functions for observing and modifying ipc message queues; analyze either trace files or performance feedback data
dos	B	Invoke a DOS environment
download	B	Download code to NMs
fault_tolerance	B	Establish environment variable defaults for fault tolerance
flow_control_diagram	B	Diagram of the flow of control of the UI command language
free_nodes	B	Ask the RM how many NMs are available
heart_beat	B	Ask the RM to broadcast a heartbeat message to pick up any stray NMs
help	B	Present the current UI command language options
quit	B	Exit the BBL environment (same as the bye command)

† The UI phase during which the command is enabled; S=Setup, R=Runtime, B=Both.

User Interface Command Language (continued)		
Command	**Phase†**	**Description**
reset	B	Reset the algorithm to its initial state
rm_addr	B	Broadcast to find the address of the RM (when the RM is started after the UI/PM)
runtime	S	Enable the runtime phase routines
setup	S	Enable the set up phase routines
start	R	Start the execution of the user's distributed algorithm
suspend_nodes	R	Suspend program execution at the NMs
time	B	Synchronize time at the NMs
topology	S	Select an underlying logical topology for the algorithm
view_data	B	Allow the user to view system state information (e.g. which algorithm is running, the state of various vids, the mapping between nodes and code classes, et cetera)
"?"	B	Present the current UI command language options (same as the help command)

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. Algorithm configuration files, at very least, contain information about the number of executable files (or code classes) involved in the algorithm, the names of the executable files, whether or not the executables need input parameters and/or input files, and the minimum and maximum number of nodes needed by each executable. In the original version of the UI, an algorithm environment was initialized in part by the algorithm configuration file, in part by 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 RM, 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. In this version, the format of the configuration file has been extended to allow the entire set up of the algorithm environment from within it. The user is only prompted for set up information if certain key aspects about the environment have gone unspecified, are stated incorrectly, or no default values can be assumed. Algorithm configuration file details are explained more fully in the section **Writing Distributed Software for BBL** under the subsection **Algorithm Configuration Files**. In addition, see **Appendix B** for a demonstration of the UI/PM environment.

† The UI phase during which the command is enabled; S=Setup, R=Runtime, B=Both.

5.3 Termination

The commands **bye** or **quit** are used to exit the UI/PM.

6 The Debugger

6.1 Description

An integrated debugger exists as part of the BBL environment. It focuses on debugging message-passed interprocess communication (ipc). Control of either one, several, or all processes takes place from a master debugger process, which in the BBL environment is one and the same with the UI/PM. The BBL debugger combines three approaches to ipc debugging; monitoring, dynamic control, and performance measuring.

During monitoring, program state information is extracted at significant points in a program's execution. No control over program state is provided. The *audit trail* or *trace* collected by the monitor is used later to shed light on unexpected errors. The analysis of trace histories can be performed either at a breakpoint during the course of the program or retrospectively after program execution. It is a reliable means of discovering synchronization errors as well as deadlocks between competing processes.

In contrast to monitoring, dynamic debugging provides control over the execution of a program. It allows a user to interactively set breakpoints, observe intermediate values of variables, control program flow by single-stepping, et cetera. The advantage of using dynamic debugging is twofold. First, dynamic debugging supplements the nonintrusive nature of monitoring by providing an interactive form of debugging. With this approach, a program can be suspended at significant points in time, allowing program state to be analyzed or changed. When done iteratively over time, this technique helps localize faulty program behavior. Second, dynamic debugging allows the programmer to dynamically modify the scope of what is being debugged. This is crucial to a large distributed system where an overwhelming volume of information may be produced.

In order to implement the aforementioned mechanisms for ipc monitoring and control, probes (or hooks, if you will) were placed in most BBL system code involved in ipc. Having done this, it was easy to include code for performance feedback. Because the debugger gathers ipc measurements and statistics, it in effect provides an additional level of debugging. It not only tracks the performance of distributed algorithms, but also analyzes the BBL system itself. Such analysis might result in rethinking BBL's design, the redistribution of algorithm process tasks to achieve load balancing, or the reorganization of algorithms to reduce ipc overhead.

6.2 Operation

The UI command specific to the debugger is **debug_control**. During the set up phase, it allows the user to adjust environment variables for ipc traces, performance measurements, and debug printout messages.

```
> debug
Which debugger operation to set ? (default is none)
        (1) set trace event parameters
        (2) set performance parameters
        (3) print out debug messages
```

After the user selects the runtime option from the command language, the runtime

routines are enabled. This enables a modified command language. All algorithm-related set up information is presumed to be in place by this point, so commands like those for selecting an algorithm or for establishing an underlying topology are no longer available during this phase. The **debug_control** command, however, is still available. A user continues to be able to change most debug environment options, although performance measurement parameters can only be set while the algorithm is not running. At this stage, the command also provides tools to dynamically control ipc.

```
> debug
Which debugger operation to set ?
(default is none)
        (1) set trace event parameters
        (2) control or view ipc queue
        (3) print out debug messages
        (4) set ipc breakpoints
        (5) single step
```

Finally, once the program has completed, it allows the user to analyze and filter any performance and/or trace information collected during the program's execution.

```
> debug
Which debugger operation to set ?
(default is none)
        (1) set trace event parameters
        (2) control or view ipc queue
        (3) print out debug messages
        (4) set ipc breakpoints
        (5) single step
        (6) analyze trace files
        (7) set performance parameters
        (8) view performance feedback
```

6.2.1 Distributed Monitoring

In order to trace a distributed algorithm, tools to set up, collect, and examine traces must be made available. A collection of system events were defined (see the next table for a complete list) that were deemed important enought to warrant monitoring. Each event represents an activity outside the full control of a single process or is, in other words, ipc-related. The user decides at set up time which ipc events to trace and if event data should be traced as well. For example, if one process sends data to another process, the trace file can simply note the occurrence of the transmission of the message, or can also include the contents of the message sent. IPC trace files may be examined after program completion or during a breakpoint in the program's execution. Dynamic adjustments to the set of events being traced, to either reduce or expand the set, can occur at any point during runtime. In this implementation, traces are performed on a per process basis. Each process writes to a separate trace file as opposed to a single file with the merged history of all events. When trace files are generated, they reside at the UI/PM. They are named "log.x", where x is the vid of the NM from which the trace events are coming. They are created in the same directory in which the UI/PM is run. To set up trace events interactive-

ly (versus from within the configuration file) a user might type the following:

```
> debug
Which debug operation to perform ?
(default is none)
        (1) set trace event parameters
        (2) set performance parameters
        (3) print out debug messages
1

How would you like to change debugger traces ? (default is none)
        (1) change debugger trace event settings
        (2) change the style in which traces are performed
1

Which debugger trace event setting to change ? (default is none)
        (1)  do NOT trace DATA associated with ipc events
        (2)  do NOT trace send packets
        (3)  do NOT trace waiting to receive packets
        (4)  do NOT trace receive packets
        (5)  do NOT trace suspend packets
        (6)  do NOT trace resume packets
        (7)  do NOT trace when NM goes down
        (8)  do NOT trace when algorithm execution begins
        (9)  do NOT trace when algorithm execution completes
        (10) do NOT trace link_change requests
        (11) do NOT trace when NMs open files
        (12) do NOT trace when NMs close files
        (13) do NOT trace when NMs read files
        (14) do NOT trace when NMs write to files
        (15) do NOT trace when NMs are reset
        (16) do NOT trace when the user exits BBL
        (17) all of the above
2

Which debugger trace event setting to change ? (default is none)
        (1)  do NOT trace DATA associated with ipc events
        (2)  trace send packets
        (3)  do NOT trace waiting to receive packets
        (4)  do NOT trace receive packets
        (5)  do NOT trace suspend packets
        (6)  do NOT trace resume packets
        (7)  do NOT trace when NM goes down
        (8)  do NOT trace when algorithm execution begins
        (9)  do NOT trace when algorithm execution completes
        (10) do NOT trace link_change requests
        (11) do NOT trace when NMs open files
        (12) do NOT trace when NMs close files
        (13) do NOT trace when NMs read files
        (14) do NOT trace when NMs write to files
        (15) do NOT trace when NMs are reset
        (16) do NOT trace when the user exits BBL
```

(17) all of the above

How should the parameters be set?
(default is all)
(1) set individual nodes
(2) set all nodes

Which debug operation to perform ? (default is none)
(1) set trace event parameters
(2) set performance parameters
(3) print out debug messages

.
.
.

It is apparent that distributed debugging places heavy demands on the amount of space needed to store trace data. One way to limit the size of trace files is to make them cyclic. Trace files would only get so big before they begin to overwrite outdated information. Although the BBL debugger allows the user to restrict the size of trace files, by default these files are unbounded. These aspects about trace logs are what is meant by "the style in which traces are performed".

The system also provides the use of a debug flag to turn on or off system print-out statements at various processors. These statements can appear on the console of the PM, the RM, or any NM involved in the running the algorithm. The user visually follows the course of the algorithm by seeing which BBL code is invoked and under what conditions. These messages are generally only of use to BBL system programmers. The following table summarizes the BBL debugger trace options.

Debugger Trace Options† *(for Ipc Monitoring)*	
Option	**Description**
Event	Record an entry in the trace file, if the flags for the following events are set;
send	user algorithm code sends a message
rcv_wait	an NM waits to receive a message
receive	a message is received
run	the program begins running
open	user code opens a file
close	user code closes a file
read	user code reads from a file
write	user code writes to a file
suspend	the algorithm has been suspended
resume	the algorithm has been resumed
link change	an NM's logical link information changes
complete	the algorithm completes
i_am_down	the node is re-claimed by its owner or has crashed
reset	the algorithm is reset
quit	the BBL environment has been exited
Data	Include message data in the event trace
Limit	Limit the size of trace files
Printout*	Print out extensive debug messages to the display monitor
Style	Trace at the message or packet level

6.2.2 Dynamic Debugging

The dynamic tools of the BBL debugger concentrate on interactive manipulation of process ipc queues: a user has the ability to inquire about the state of a process' queue (how many incoming messages are queued to be processed, from which processes were they sent, what user-defined type messages were they, what are the contents of the messages in the ipc queue, and so on), to alter ipc queues (packets can be re-ordered, added, deleted, saved, or altered), to insert ipc-related breakpoints so that execution will halt before or after the delivery of certain messages, and to force the single-stepping of program execution (at the granularity of ipc steps). Basically, this functionality results in the close control of both the contents and the scheduling of messages passed between processes. As with tracing, one can fine tune the handling of any or all processes. Dynamic debugger instructions can apply exclusively to one process, have an affect on clusters of processes, or can be destined for all processes at once. These functions are displayed below. The following table summarizes the BBL debugger queue functions.

† These options can be set on a per process basis. * Printout statements can be turned on at any NM, as well as the PM and RM.

Debugger Queue Functions† *(for Ipc Control)*	
Function	**Description**
View	View the message queue of a particular NM
Add	Add a message to the message queue
Delete	Delete a message from the message queue
Change	Modify the contents of a particular message
Re-order	Re-order the messages in the message queue
Save	Save the contents of a message to a file
Breakpoint	Set a breakpoint either before or after the delivery of a particular message
Single-step	Single-step program execution, pausing before or after the delivery of a particular message

6.2.3 Performance Feedback

A user has the ability to instruct the debugger to gather feedback on the frequency and type of message traffic generated. Messages are first classified into BBL and non-BBL packets. This proves to be useful in determining the impact of network load on an algorithm's performance. BBL traffic, in turn, is broken down into user, system, or debugger initiated. The culpability for high ipc overhead can therefore be assessed further. If user initiated communication is sufficiently time consuming, a user might opt to change the algorithm's communication model or to change the algorithm altogether.

Timings are also calculated upon request. The debugger will time how long an algorithm and each of its components took to complete. Furthermore, it will break down algorithm execution time into time spent on communication versus time spent actually processing the algorithm. Communication time can be further scrutinized by breaking it down into time spent sending messages and time spent blocked waiting for the receipt of specific messages.

Besides ipc measurements, the system keeps track of fault frequency, tallying how often a node outright crashes versus how often it is re-claimed by its owner. Additionally, it stores NM state changes. The NM's PC can be in one of three states: in use by its owner, available for BBL usage, or being used by a BBL user application. This is referred to as the NM's history. When the user requests history statistics (e.g. the percentage of time an NM has spent in a particular state), the user can obtain information about NMs in the possession of the UI/PM or about those resident at the

† Most of these functions apply to one vid queue at a time. Breakpointing and Single-stepping can be performed on all processors or clusters of processors.

RM. These functions help with the analysis of the overall behavior and utility of the BBL system itself. Developed one step further, they might also help track the distribution of the interarrival time between keystrokes at a PC's keyboard or help dynamically determine the optimal amount of time the NM should wait before considering a machine idle.

Unlike other debugger options, performance parameters must be set up and remain unchanged throughout program execution. Otherwise, the measurements have no real context for making comparisons. The following table summarizes the BBL debugger performance measurement options.

Debugger Measurement Options† *(for Performance Feedback)*	
Option	**Comments**
Completion Time	For the algorithm overall, as well as on a per process basis
NM Completion Time	Minus overhead time of communication
Execution Time	Versus the time spent on communication
Time Spent Downloading	Code segment information, plus an input file and/or input parameters
Total Messages Passed	Broken down into overall and per process, and into those sent and those received
Distribution of Message Types	Classified into system, user, and debugger generated, and into those sent and those received
Non-BBL Net Activity	During algorithm execution
Number Node Failures	Nodes being used by the algorithm which either crashed or were returned to their owners during program execution
Completion status	Per NM
State changes	Per NM event histories

6.3 Termination

Access to the debugger is terminated when the UI/PM is terminated. Any trace log files that were generated by the debugger, however, are readily accessible outside the BBL environment. For further details about debugger functionality and usage see **Appendix B.**

† These options can be set on a global basis.

7 Writing Distributed Software for BBL

The BBL system does not provide an environment for the development or writing of software. It is merely the execution environment. Because we found the TURBO C environment so well suited to program development, all code development is done outside the confines of BBL. However, a library of BBL routines were written to provide the basic services one expects in a distributed system, such as packet sending and receiving, et cetera. These functions supplement standard C routines and will be described in the sections that follow.

The BBL system uses executable files as the modules to send to and execute on each processor. Therefore, each separate piece of code to run on different processors must be written and compiled into a ".EXE" file. Remember that DOS is not a multitasking operating system, so each processor has only one task running on it. Also, any attempt to write complex assembly language programs which use interrupts will probably not work as expected, and will possibly cause the BBL system to crash. Additionally, each executable module must call the BBL provided function **Init_Vars** as the first line in the program. This allows the NM to set up various data structures and links into the user's code, specifically an area for packet buffers and associated control structures and the address of an exit routine to be called when the NM is forced to terminate the user's code. See the next section for further details.

In summary, the BBL restrictions are that:

1) Code must be written in C (to link with the BBL provided routines)
2) Each piece that will run on a separate processor
   must be compiled into a ".EXE" file
3) No interrupt driven routines
4) First executable statement MUST be a call to the Init_Vars routine
5) Must be compiled with the large model

7.1 BBL Provided Functions

This section describes the functions which are available to the user of the BBL system. These functions may be included in the user's code which is compiled and then run under the BBL environment. These functions are actually embedded in the NM. A set of library routines written in assembler are placed into a library called "BBLUSER.LIB". The user must link her code with this library in order to access these functions. For version 3.00 of the Node Manager, the following functions are defined:

```
int Init_Vars(struct user_packet *buffer,int *size, int* free, int Q_size, void (*user_exit)());
Super_Node(void);
Set_Parameters(int replace, int notify);
int Time_Synch(void);
Send_Packet(int vid, int type, char *paddr, int length, int *ret_code);
int Receive_Exist(int vid, int type);
Receive_Packet(int *vid, int *type, struct user_packet *paddr, int *size);
Get_VID(int *vid);
Get_IN_Connections(int *num, int list[]);
Get_OUT_Connections(int *num, int list[]);
```

BBL_Open(char *path, int flags, int exclusive, int *retval, int *errno);
BBL_Close(int file_handle, int *retval, int *errno);
BBL_Write(int file_handle, char *addr, int size, int *retval, int *errno);
BBL_Read(int file_handle, char *addr, int size, int *retval, int *errno);

The functions provided by the NM are described in the following paragraphs. Each is introduced by its format (in bold letters) and functional description.

int Init_Vars(struct user_packet *buffer, int *size, int* free, int Q_size, void (*user_exit)());

This function must be called at the beginning of the user code. This is because the NM must have an address of an exit routine to call when a key gets hit, or when instructed to kill the node. The address of the exit routine is passed as a parameter to Init_Vars as "user_exit". This exit routine can provide any needed fault tolerance, must free any allocated space, or simply be the routine "exit()". This routine **must** end with a call to the function "exit()". It must not return. The other parameters passed are the address of the user packet buffer, the address of an array to indicate the size of the packet placed in the buffer by the NM, the address of an array of flags indicating the availability of the buffers, and an integer which indicates the number of buffers allocated.

The return value from this function call is TRUE if this is a restarted 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. These values are useful for user designed fault tolerance.

Generally the call will look like this:

Init_Vars(input_buffer, input_size, input_free, U_QUEUE_SIZE, exit);

Super_Node(void);

This is called by the user to indicate 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. Also see Set_Parameters which allows the node to turn off notification and/or replacement.

Set_Parameters(int replace, int notify);

This function is called by any node to tell the PM how to handle nodes going down. The two parameters are integers. In the absence of any instructions from the usercode, the default is to replace nodes without notification.

replace:	0= no replacement (let the node just die) 1= replace the node and restart code, if running
notify	0= no notification 1= notify each "super node"

The down_packet structure is defined in BBLUSER.H and is the format of the packet which is sent from the PM to the user code when a node dies if notify = 1.

```
struct down_packet {
        int To;         /* sent to VID */
        int From;       /* sent from VID  = PM_VID = 0x8000 */
        int Type;       /* packet type = 0x8001 */
        long Count;     /* "Timestamp" incremented for each pkt rcvd */
        int eom;        /* end of message used by system */
        int vid;        /* vid that went down */
        int ack;        /* if a replacement was found */
};
```

int Time_Synch(void);

This function is used to synchronize the clocks of each of the nodes participating in a computation. All nodes associated with a given user will be synchronized. Only one node involved in the computation needs to call this function. It operates by telling the PM to broadcast a message that sets the clock of the appropriate NM's. Unfortunately, this is not a guaranteed function. To increase the probability of successful completion, do not execute any function that requires DOS operations (printing, reading disks et cetera) immediately after the call to Time_Synch. The return value from this function indicates whether this node received the TIME_SET packet from the PM. It gives an idea as to whether the nodes are now synchronized. It is possible that some node didn't receive the message.

Send_Packet(int vid, int type, char *paddr, int length, int *ret_code);

This function is used to send a packet (message). The message may be up to 32767 bytes long. Ret_code will be non-zero in the case of an error ("-1" indicates that the requested VID is not connected to this node). Any other non-zero return code indicates an error at the packet sending level. The Type parameter is a packet type which allows a receiver process to wait for packets of a particular type. Paddr is a pointer to the start of the user data.

Receive_Packet(int *vid, int *type, struct user_packet *paddr, unsigned *size);

This function waits for an incoming packet. This is a blocking receive. The NM puts the packet at paddr and returns the packet length in size. By setting the vid and type variables to appropriate values, this function can receive from a particular vid or a particular type only. If vid points to -1 then any vid is acceptable. If type points to a value of -1 then any type is acceptable. These variables will be updated with the vid and type of the packet that is actually received. This function will handle receiving a multi-packet message. If more than one message exists in the buffer when this function is called, the oldest message is returned.

Non-blocking receive can be accomplished by checking the input_free flags to see if any packets have come in. If the flag is FALSE, the NM has placed a packet into this buffer. When the user is done with the packet, the input_free flag should be set to TRUE. **Note**: if the user wants to use non-blocking receive on messages larger than one packet, it is the user's responsibility to put the packet back together. This is

non-trivial, but is accomplished by using the From, Type, Count and eom fields of the incoming packets.

The following packet-related information is provided for packet reception in BBLUSER.H.

```
#define U_QUEUE_SIZE    42      /* number of packet buffers (any size) up to 42 */
#define MAX_LINKS       25      /* max num of input or output links (any size)*/

struct user_packet {
        int To;         /* sent to VID */
        int From;       /* sent from VID */
        int Type;       /* packet type */
        long Count;     /* "Timestamp" incremented for each pkt rcvd */
        int eom;        /* end of message flag (used by system) */
        unsigned char Data[1480];
};

struct user_packet input_buffer[U_QUEUE_SIZE];
int input_size[U_QUEUE_SIZE];
int input_free[U_QUEUE_SIZE];
```

int Receive_Exist(int vid, int type);

This function returns TRUE if a packet of the particular type from the particular vid is in the packet queue. Vid or type may be set to -1 to allow any vid or any type respectively.

Get_VID(int *vid);

This function will return the value of the virtual id number of the node in the variable vid.

Get_IN_Connections(int *num, int list[]);

This function will return the number of connections that are input links to the node along with the vid of the node on the other end. The total number of input links is returned in num. The format of list is simply an array of integers one for each of the connections.

Get_OUT_Connections(int *num, int list[]);

This function will return the number of connections that are output links from the node along with the vid of the node on the other end. This has the same format as the Get_IN_Connections.

BBL_Open(char *path, int flags, int exclusive, int *retval, int *errno);

This function allows the user code to open a file on the PM machine. The call has nearly the same format as the standard open statement in C. Path is a pointer to a null-terminated string containing the path name of the file. Flags are the same as the normal open command. Exclusive is TRUE if this node should have exclusive read/write for this file. Retval will contain the file pointer, or -1 if there was some error. The file pointer is to be 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. Errno should point to the global variable errno so that any error conditions can be returned to the user.

BBL_Close(int file_handle, int *retval, int *errno);

This function closes a remote file on the PM machine. The file handle is the value returned in the BBL_Open call. Errno should point to the global variable errno so that any error conditions can be returned to the user.

BBL_Write(int file_handle, char *addr, int size, int *retval, int *errno);

This function allows the user code to write to an open file on the PM machine. Up to 32767 bytes can be written at one time. The file handle is the value returned in the BBL_Open call. Retval is the number of bytes written. If size is not equal to retval, there probably is an error. Errno should point to the global variable errno so that any error conditions can be returned to the user.

BBL_Read(int file_handle, char *addr, int size, int *retval, int *errno);

This function allows the user code to read from an open file on the PM machine. Up to 32767 bytes can be read at one time. The file handle is the value returned in the BBL_Open call. Retval is the number of bytes read, or -1 if there is an error. Errno should point the global variable errno so that any error conditions can be returned to the user.

The file BBLUSER.H must be included in the user code C files. The current version of it is listed in **Appendix C.**

7.2 Task Partitioning

While the BBL provided C functions allow separate processes of a distributed algorithm to communicate with each other, they do not instruct a user on how to actually distribute an algorithm. Toward this end, we step through the construction of a sample algorithm to illustrate the use of the BBL system.

The example distributed algorithm is a parallel version of merge sort. One of the processors in the network generates a list of data (integers) to be sorted. This processor then divides the list into as many pieces as there will be processors participating in the computation. It sends one piece of the list to each processor and keeps one for itself. Every processor then sorts its piece of the data. After sorting, half of the processors send their list to another waiting processor. The waiting processors take the newly received list and merge it with their own list. This process continues until the original node merges two lists to create the final sorted list.

The first job is to split up the problem into its constituent parts. It seems as if there are two different types of nodes. The first type of node creates the original list and sorts the final list at the end. This processor is referred to as the "MERGER". The other processors in the system all perform essentially the same task, sorting and merging, and they are called "SORTERS". Our parallel algorithm will consist of one MERGER and *N-1* SORTERS.

7.3 The Algorithm Configuration File (MERGSORT.ALG)

After writing and compiling the algorithm code into two files called "MERGER.EXE" and "SORTER.EXE", we are ready to integrate it into the BBL system. We do so by creating an algorithm configuration file called "MERGSORT.ALG" (It is advantageous to name the ".ALG" file with some significant name, since when the number of available algorithms increases, it is difficult to remember that "ALGO1" is actually a mergesort routine). The configuration file name "MERGSORT.ALG" must be added to the list of other configuration file names in the "info.bbl" library file, if it is to be accessible from the BBL environment.

The ".ALG" file offers a way for the user to specify the requirements of the distributed algorithm's ".EXE" files. In addition, the revised configuration file format allows the user to specify all set up information from within this file. This approach bypasses questioning from the UI, unless a discrepancy is found in the file or some request cannot be fulfilled (e.g. the number of nodes needed by the algorithm). In this fashion, the BBL user need only select an algorithm, at which point the system is ready to invoke runtime commands.

7.3.1 Required Data

The beginning entries in a configuration file are required, while the latter are optional. The first required lines of the file are comments meant to describe the algorithm. Therefore, "MERGSORT.ALG" might begin something like:

```
This is the mergesort algorithm written by John Doe.
It sorts a list of integers on an arbitrary number of processors.
There is only one "MERGER" and any number of "SORTERS".

2

MERGER.EXE                    /* code class 0 */
1
1
1
0

SORTER.EXE                    /* code class 1 */
0
-1
0
0
```

A blank line terminates the comment section. This is followed by an integer which indicates the total number of executable files (or code classes) associated with the algorithm. In our case, there are 2 executable files. Information pertaining to each code class is expected next.

Each code class description has five components; the name of the ".EXE" file, the minimum number of processors on which this code should run, the maximum number of processors on which this code should run, whether or not any input parameters are to be passed to the executable, and whether or not any input file is to be redirected to the executable.

Only one processor should act as the MERGER, so both the upper and lower bound for "MERGER.EXE" are set to 1. Because the code has been written to expect an input parameter (to indicate the size of the list to sort), the parameter field is set to 1 in the description. In this implementation of the algorithm, no input file is needed which is indicated by the use of a 0 in the description.

In the case of "SORTER.EXE", the minimum number of nodes needed is zero since the MERGER code was written to be able to sort the list by itself. The maximum is set to -1, indicating an unlimited upper bound. The SORTER gets all of its information from the MERGER, so no parameters or input data are needed.

7.3.2 Optional Data

The optional entries need not be arranged in any particular order. For clarity purposes, they have been organized here first into options directly related to the initialization of the algorithm, and then into those specific to the debugger. The option names in the configuration file have the same property as command names within the UI. As long as a uniquely recognizable prefix is used, the name is considered legitimate. Comments are entered as in the C programming language (between '/*' and '*/') and legal delimiters (between multiple arguments on a line) are spaces, commas, or colons. The optional data for the sample algorithm might look as follows;

```
topology: connected
nodes: 4, 3
memory: 14
map: 0 0
map: 1 1 2
params_in: 0 4000

recovery
notify
supernode:0
trace: send, receive, rcv_wait
perform: all
debug: pm
```

These configuration file options include requests that the underlying logical topology by fully connected, that 4 nodes be set aside for the user, while 3 are to be actually used by the algorithm, that any node allocated have at least 14K of memory available for use by the algorithm, that vid 0 be mapped into code class 0, whereas vids 1 and 2 are to be mapped into code class 1, and that vid 0 has an input parameter "4000".

Additionally, nodes are to be replaced when they fail, any supernodes should be notified about failed nodes and their replacements, and vid 0 is to be a supernode (if it fails the algorithm should be abandoned). Finally, debugger tracing should note when nodes send, receive, and are waiting to receive messages, all performance measurements should be collected, and PM debug messages should occur while the system runs.

7.3.3 Backus-Naur Form

The configuration file format and options can be described in more detail in Backus-Naur Form (BNF). For the purpose of describing configuration file formats, the following symbols will be considered meta-symbols of the BNF formalism. They are not part of the configuration file format.

= < > | [ ] "

An equal sign denotes a production rule. An argument enclosed in angle brackets (e.g. "<" and ">") is at some point expandable as the left hand side of a production rule. The symbol "|" is a logical OR function. An expression enclosed in square brackets (e.g. "[" and "]") is considered legally repeatable zero or more times. Followed by a "+", a specific number, or a range of numbers, an expression in square brackets is to be repeated at least 1 or more times, a specific number of times, or within some range of times respectively. Any item enclosed in double quotes (e.g. ") is meant as a string of characters. The highest level abstraction of the format looks as follows;

<description of algorithm>
<blank line>
<n, the number of code classes in the algorithm>
[<executable description>]^{n}
[<optional info>]

If there are n code classes, there must be as many executable descriptions. Each argument expands into statements shown below. Any description in italics is difficult to place in BNF, but should be intuitive. First, the required data is expanded.

<description of algorithm> = *descriptive text about the algorithm*
<n, the number of code classes in the algorithm> = <vid range>
<executable description> =
    <name of code class executable file>
    <minimum nodes in code class>
    <maximum nodes in code class>
    <input parameters needed?> <parameters>

    <input file needed?> [<input file name>]$^{0\text{-}1}$]

<name of code class executable file> = <legal name>.exe
<minimum nodes in code class> = <vid range> | -1
<maximum nodes in code class> = <vid range> | -1
<input parameters needed?> = 0 | 1
<input file needed?> = 0 | 1

The value -1 is valid for specifying the minimum or maximum number of nodes in a code class; in either case it specifies an unbounded limit. Next, the optional data is expanded.

```
<optional info> = <set up info> | <debug info>

<set up info> =
        [<nodes> <number to get> [<number to use>]] |
        [<memory> <memory requirement>] |
        [<topology> <topology choice> [<start> | <width> <length>]] |

        [<map> <code class> [<vid>]+ ] |
        [<input parameters> <vid> <parameters>] |
        [<input files> [<vid> <input file name>]]
<nodes> = "nodes"
<number to get> = <vid range>
<number to use> = <vid range>
<memory> = "memory"
<memory requirement> = 0..32767
<topology> = "topology"
<topology choice> = "bblsort" | "tree" | "nonstandard" | "torus" |
        "connected" | "clear" | "star" | "grid" | "biring" | "uniring"
<start> = <vid range>
<width> = <vid range>
<length> = <vid range>
<map> = "map"
<code class> = <vid range>
<input parameters> = "params_in"
<input files> = "files_in"

<debug info> =
        [<recovery> <true or false>] |
        [<notify> <true or false>] |

        [<supernode> [<vid>]+ ] |

        [<trace> [<trace choice>]+ [vid]] |

        [<perform> [<perform choice>]+ [vid]] |

        [<debug> [<debug choice>]+ [vid]]

<recovery> = "recovery"
<notify> = "notify"
<supernode> = "supernode"
<trace> = "trace"
<trace choice> = "send" | "receive" | "rcv_wait" | "run" | "open" |
        "close" | "read" | "write" | "suspend" | "resume" |
        "link_change" | "complete" | "i_am_down" | "reset" | " quit" |
        "data" | "all"
<perform> = "perform"
<perform choice> = "completion_t" | "execution_t" | count_types" |
```

"count_pkts" | "failures" | "all"
<debug> = "debug"
<debug choice> = "pm_print" | "rm_print" | "nm_print" | "all"

Lastly, miscellaneous arguments are expanded.

<blank line> = <space> CR LF
<vid range> = 0...100
<parameters> = [<parameter> [<space>]] <parameter>
<parameter> = *any legal parameter*
<space> = *any white space characters*
<input file name> = <legal name>
<legal name> = <char> [<char> | <digit>]$^{1-7}$
<char> = a...zA...Z | *other legal characters*
<digit> = 0...9
<vid> = <vid range>
<true or false> = "true" | "false"

The required portions of the configuration file have a fixed ordering, while the optional portions do not. In the general style of other names within the BBL environment, the whole string of an option under <topology choice>, <trace choice>, <perform choice>, and <debug choice> need not be specified, so long as its uniquely identifiable. Note that input parameters/files can be specified on a code class basis (i.e. ALL vids in that code class use the same input parameters/files) or on a per node basis. Both approaches can be used in the same configuration file; any conflicts are resolved in favor of using the node specific data. Finally, notice that <memory requirement> is in kilobytes per node.

7.3.4 Default Assumptions

Default values for optional entries follow:

Default Assumptions	
Option	**Default**
topology	fully connected unless one of the following sub-options follows; *connected, clear, star, grid, biring, uniring, bblsort, tree, nonstandard, torus*
nodes	sum of the code segment minimums
memory	maximum code segment size
map	starting with the lowest vid, first meet code class maximums; if any left over, continue to allocate (cycling through the code classes) until either code class maximums are reached or all nodes are allocated
params_in	no individual input parameters are needed
files_in	no individual input files are needed
recovery	replace nodes that fail
notify	do not notify supernodes of node failures
supernode	no nodes are set
trace	no tracing is done unless one of the following sub-options follows; *send, receive, rcv_wait, run, open, close, read, write, suspend, resume, link_change, complete, i_am_down, reset, quit, data, all*
log_style	change the style in which tracing is performed if one of the sub-options follows; *limit, circularity, overwrite, append, messages, packets*
perform	no performance measurements are performed unless one of the sub-options follows; *completion_t, execution_t, download_t, count_pkts, count_types, failures, nm_history, all*
debug	no printout messages are printed unless one of the sub-options follows; *pm_print, rm_print, nm_print, all*

If no vid or collection of vids is specified, then all vids are considered. Many topologies require a starting vid; the default start vid is 0. Other topologies require length and width variables; by default these are 0. The *nodes* option allows a user to specify both the number of nodes to allocate and the number of nodes to actually use during the algorithm. If the latter is omitted, it is set equal to the number of nodes allocated.

8 References

[SCHO88] Schooler, E.M., Felderman, R.E., Kleinrock, L., "The Benevolent Bandit Laboratory: A Testbed for Distributed Algorithms Using PCs on an Ethernet", Technical Report CSD-880016, Computer Science Department, University of California, Los Angeles, (Mar 1988).

9 Appendix A

9.1 The Mechanics of User Function Calls

Since the NM code and the user code is compiled separately there must be a method to link the function calls in the user's code with the actual routines in the NM at runtime. The library BBLUSER.LIB contains short assembler routines for each of the user functions. These assembler functions only provide a link to the actual routines in the Node Manager. These routines read the address of a jump table from a fixed location in memory (0:0188,0:018A). Then, depending on which function is called, the routine indexes into the jump table to find the start address of the called function. Important: This jump table is defined in "loadaddr.asm" a file which is part of the Node Manager. The jump table must always be in synch with the assembler routines in the user's library. Since each NM routine's address is placed in a specific location in the jump table, the assembler routines in the user's library must also access this specific location. Once the proper address has been located, the routine performs a JUMP to the NM routine. The NM routine will terminate with a Subroutine Return, thus returning directly to the user's code. When the NM takes over a node it performs several initialization steps. One of these steps is to load the address of the jump table into the fixed location mentioned above. Once this is accomplished, the user code can call any of the NM provided functions.

Here is the jump table as defined in NM version 3.00.

```
/* the jump table */

/* the order is EXTREMELY important, the functions linked into
   the user code depend on this order. */

Jump_Table      NM Function Name    User Function Name

          DD    _NM_Send_Packet     (Send_Packet)
          DD    _NM_Super_Node      (Super_Node)
          DD    _NM_Rcv_Packet      (Receive_Packet)
          DD    _NM_Get_VID         (Get_VID)
          DD    _NM_Get_IN          (Get_IN_Connections)
          DD    _NM_Get_OUT         (Get_OUT_Connections)
          DD    _NM_Terminate       (Completion)
          DD    _NM_Write           (BBL_Write)
          DD    _NM_Read            (BBL_Read)
          DD    _NM_Set_Parameters  (Set_Parameters)
          DD    _NM_Init_Vars       (Init_Vars)
          DD    _NM_Open            (BBL_Open)
          DD    _NM_Close           (BBL_Close)
          DD    _NM_Rcv_Exist       (Receive_Exist)
          DD    _NM_Time_Synch      (Time_Synch)

/* jump table address is stored at 0:0188, 0:018A
   which is interrupt 62, an unused interrupt */
```

9.2 Adding New User Functions

Adding a new user function is a complicated task and should only be performed by a competent BBL system programmer. The files which will need to be modified are listed below:

1) USERLIB.ASM (BBLUSER.LIB)
2) BBLUSER.H
3) LOADADDR.ASM (JUMP TABLE)
4) NEWUFUNC.C

The first file to change is the library file linked into the user's code. The code for this library is in the file USERLIB.ASM. By examining this file one can see the basic structure of a user function in the library. Let's pretend we are going to add a function called "nothing()" which does nothing when called. We need to add a function "_nothing" (**Note the initial underscore**) to the file USERLIB.ASM. The easiest way to do this is to copy the text of a complete function, the last one in the file, to the bottom of the file, then change it appropriately. Four changes need to be made: three to the names and one for the location in the jump table. Addresses in the jump table are four bytes long, so this new function will be at whatever location the previous function uses plus four. In our example, the previous function "_Time_Synch" is at location 56, so we'll put our function at location 60. The assembler code needed for our new function "nothing" is shown below.

```
PUBLIC      _nothing
_nothing    PROC FAR
            PUSH AX
            PUSH BX
            PUSH CX
            PUSH ES
            PUSH DS
            PUSH SI
            PUSH DI
            MOV  AX,0
            MOV  ES,AX
            MOV  AX,ES:0188h; offset
            MOV  BX,ES:018Ah; segment
            ADD  AX,60;  <=====    *** Offset in jump table ***
            MOV  DS,BX
            MOV  SI,AX
            MOV  AX,CS
            MOV  ES,AX
            MOV  DI,OFFSET CS:Jump_Addr
            MOV  CX,2
            REP  MOVSW
            POP  DI
            POP  SI
            POP  DS
            POP  ES
            POP  CX
            POP  BX
            POP  AX
            JMP  [CS:Jump_Addr]
_nothing    ENDP
```

This file is then assembled with the Microsoft assembler (MASM) using the "m" and "x" switches as follows:

masm /mx userlib.asm;

Assuming there are no assembler errors, the file userlib.obj will have been created. This is the object file used to make the library "BBLUSER.LIB". First, delete the old version of BBLUSER.LIB and create a new one by using the system function "LIB". Details of how to use this function are contained in any DOS manual and will not be described here.

The next file to change is BBLUSER.H. This is the user include file for BBL. All we need to do is add an external function definition to the list of other BBL functions at the bottom of this file. For our function we'll add the line:

extern nothing(void);

This completes the changes on the user's side of the function, now we need to change things on the NM's side.

The first NM change to make is to write the actual code for the function "nothing" and place it into the file "NEWUFUNC.C". This file contains all of the functions provided to the user by the NM. Since our function doesn't do anything, it will be very simple. The code is shown below.

```
NM_nothing(void)
{
        set_DS();       /* sets registers to proper values */

        /* code of function */

        fix_DS();       /* restore registers */
}
```

The two functions "set_DS()" and "fix_DS()" are necessary to set the SS,ES and DS registers to the values used by the NM and to restore them after execution. They must be the first and last statement of the routine except for a return. These functions must always "return", they cannot exit. Also, returning the value of a variable is not possible. For example a set of statements like:

```
int i;

set_DS();
i = 256;
fix_DS();
return(i);
```

will not work as expected since by calling "fix_DS()" we lose the ability to access the variable "i" correctly. Therefore only calls like

```
return(256);    /* a constant */
```

are possible. To return some arbitrary value, you must pass the address of the variable as a parameter to the function, using this address the code can modify the value stored at the address. Take a look at some of the other NM provided functions to see how this address passing works.

Finally, the jump table contained in the file "LOADADDR.ASM" must be updated. We will add the address of the function we have just created "NM_nothing" to the end of the table. We'll add two lines so the file will look something like this:

```
EXTRN _NM_Send_Packet  :FAR
EXTRN _NM_Super_Node   :FAR
EXTRN _NM_Rcv_Packet   :FAR
EXTRN _NM_Get_VID      :FAR
EXTRN _NM_Get_IN       :FAR
EXTRN _NM_Get_OUT      :FAR
EXTRN _NM_Terminate    :FAR
EXTRN _NM_Write        :FAR
EXTRN _NM_Read         :FAR
EXTRN _NM_Set_Parameters:FAR
EXTRN _NM_Init_Vars    :FAR
```

```
EXTRN _NM_Open          :FAR
EXTRN _NM_Close         :FAR
EXTRN _NM_Rcv_Exist     :FAR
EXTRN _NM_Time_Synch    :FAR
EXTRN _NM_nothing       :FAR        /* NEW LINE */

        .
        .
        .

Jump_Table  DD  _NM_Send_Packet     ; 0
            DD  _NM_Super_Node      ; 4
            DD  _NM_Rcv_Packet      ; 8
            DD  _NM_Get_VID         ; 12
            DD  _NM_Get_IN          ; 16
            DD  _NM_Get_OUT         ; 20
            DD  _NM_Terminate       ; 24
            DD  _NM_Write           ; 28
            DD  _NM_Read            ; 32
            DD  _NM_Set_Parameters  ; 36
            DD  _NM_Init_Vars       ; 40
            DD  _NM_Open            ; 44
            DD  _NM_Close           ; 48
            DD  _NM_Rcv_Exist       ; 52
            DD  _NM_Time_Synch      ; 56
            DD  _NM_nothing         ; 60    /* NEW LINE */
```

Note: the location in the jump table MUST correspond to the location (number) placed in the code in the file USERLIB.ASM. We add our new function to the end of the table (16th function) so the number placed in the code for _nothing in USERLIB.ASM is (16-1)*4 = 60

10 Appendix B: Demonstration of the BBL System

The following demo of the BBL debugger environment makes use of the example algorithm discussed in section 7.3, **The Algorithm Configuration File (MERGSORT.ALG)**. The configuration file for the mergesort algorithm looks as follows;

```
This is the mergesort algorithm written by John Doe.
It sorts a list of integers on an arbitrary number of processors.
There is only one MERGER and any number of SORTERs.

2

MERGER.exe
1
1
1
0

SORTER.exe
0
-1
0
0

topology: connected
nodes: 5, 4
memory: 14
map: 0 0
map: 1 1 2 3
params_in: 0 4000

recovery
notify
supernode: 0
perform: all
debug: pm
```

The required entries in the configuration file are identical to those discussed in the earlier example. The optional entries are slightly different in that 5 nodes are set aside for the user, while 4 are to be actually used by the algorithm, that vid 0 be mapped into code class 0, whereas vids 1, 2, and 3 are to be mapped into code class 1, and that initially no tracing is turned on.

We assume that at least one node in the network is running the RM module and that all other nodes are running the NM module.

To begin the demonstration of the BBL environment, the user types "UIPM" to the DOS prompt.

BBL User Interface Version 3.00

> The Resource Manager has 8 free nodes
> help

Command Language :
==========
<debug_control>
<dos>
<fault_tolerance>
<flow_control_diagram>
<free_nodes>
<help | ?>
<quit | bye>
<reset>
<setup>
<time>
<view_data>

> set
SETUP routines have been enabled

Choose one of the following algorithm choices: (default is 'BACH.alg')
(0) BACH.alg
(1) BBLSORT.alg
(2) BOBSTEST.alg
(3) EIGHT.alg
(4) FILETEST.alg
(5) IBM.alg
(6) INFINITE.alg
(7) LOGO.alg
(8) MERGSORT.alg
(9) MESSAGE.alg
(10) OFF.alg
(11) SHELL.alg
(12) TOKEN.alg
(13) TEST.alg
(14) TIME.alg
8
Optional configuration file information...
Nodes requested = 5, nodes to use = 4
Error: algorithm needs at least 26K

Apparently, the estimation for the amount of memory needed to run the program is low. The user updates the "MERGSORT.alg" file by entering DOS, editing the file, then re-selecting the algorithm.

> dos
DOS shell started, type EXIT to return to BBL

<DOS> tc mergsort.alg
(edit the file)
<DOS> exit
Returning to BBL

> help

```
        Command Language :
        ==================
        <algorithm>                 <help | ?>
        <debug_control>         <quit | bye>
        <dos>                       <reset>
        <fault_tolerance>       <time>
        <flow_control_diagram>  <view_data'>
        <free_nodes>

> alg
Choose one of the following algorithm choices: (default is 'BACH.alg' )
        (0) BACH.alg
        (1) BBLSORT.alg
        (2) BOBSTEST.alg
        (3) EIGHT.alg
        (4) FILETEST.alg
        (5) IBM.alg
        (6) INFINITE.alg
        (7) LOGO.alg
        (8) MERGSORT.alg
        (9) MESSAGE.alg
        (10) OFF.alg
        (11) SHELL.alg
        (12) TOKEN.alg
        (13) TEST.alg
        (14) TIME.alg
8
Optional configuration file information...
Nodes requested = 5, nodes to use = 4
Allocated 5 node(s) with 28K of memory
Topology is completely connected
Node class 0: 0
Node class 1: 1 2 3
Input parameters for vid 0: 4000
Supernodes: 0
Collect performance info at ALL nodes:
        count packets sent and received
        count the types of packets sent and received
        break down execution time
        trace node failures during execution
        trace NM histories
        automatically report measurements on algorithm completion
UI/PM debug msgs on
free_config
```

The message "free_config" is a result of having turned on debug messages at the UI/PM. Once the algorithm environment is set up correctly, we turn off these messages, download the code, view the state of the system, enable the RUNTIME phase routines, and then start the algorithm.

```
> debug
Which debugger operation to set ? (default is none)
        (1) set trace event parameters
        (2) set performance parameters
        (3) print out debug messages
3

Change which node's debug messages ? (default is none)
        (1) do NOT print out debug messages at the NM(s)
        (2) print out debug messages at the PM
        (3) do NOT print out debug messages at the RM
        (4) all of the above
2
```

Change which node's debug messages ? (default is none)
(1) do NOT print out debug messages at the NM(s)
(2) do NOT print out debug messages at the PM
(3) do NOT print out debug messages at the RM
(4) all of the above

Which debugger operation to set ? (default is none)
(1) set trace event parameters
(2) set performance parameters
(3) print out debug messages

> down
How should the download be performed ?
(default is all)
(1) download individual nodes
(2) download all nodes

Once the download has completed, we look at the **view** *command.*

> v

Data available for viewing : (a blank line exits the routine)
(1) Connection matrix
(2) Node status information
(3) Node address information
(4) Vid information
(5) Node tasks
(6) User Id
(7) Fault tolerance strategies
(8) Algorithm choice
(9) Topology choice
(10) Channels
(11) Supernodes
(12) Debugger info
1

Connection Matrix:
==========
0 1 1 1 0
1 0 1 1 0
1 1 0 1 0
1 1 1 0 0
0 0 0 0 0

Data available for viewing : (a blank line exits the routine)
(1) Connection matrix
(2) Node status information
(3) Node address information
(4) Vid information
(5) Node tasks
(6) User Id
(7) Fault tolerance strategies
(8) Algorithm choice
(9) Topology choice
(10) Channels
(11) Supernodes
(12) Debugger info
2

```
Node Status:
=========
Idle Vids:  4
Loaded Vids:  0 1 2 3
Busy Vids:
Suspended Vids:
Done Vids:
Dead Vids:

Data available for viewing : (a blank line exits the routine)
        (1) Connection matrix
        (2) Node status information
        (3) Node address information
        (4) Vid information
        (5) Node tasks
        (6) User Id
        (7) Fault tolerance strategies
        (8) Algorithm choice
        (9) Topology choice
        (10) Channels
        (11) Supernodes
        (12) Debugger info
3

Node Address Info:
==============
UI address 002 140 214 010 105 030
vid 0, addr 002 140 214 007 226 001
vid 1, addr 002 140 214 010 103 001
vid 2, addr 002 140 214 010 062 107
vid 3, addr 002 140 214 010 062 102
vid 4, addr 002 140 214 007 225 163

Data available for viewing : (a blank line exits the routine)
        (1) Connection matrix
        (2) Node status information
        (3) Node address information
        (4) Vid information
        (5) Node tasks
        (6) User Id
        (7) Fault tolerance strategies
        (8) Algorithm choice
        (9) Topology choice
        (10) Channels
        (11) Supernodes
        (12) Debugger info
4

Vids in Use:
=========
0 1 2 3 4

Data available for viewing : (a blank line exits the routine)
        (1) Connection matrix
        (2) Node status information
        (3) Node address information
        (4) Vid information
        (5) Node tasks
        (6) User Id
        (7) Fault tolerance strategies
        (8) Algorithm choice
        (9) Topology choice
```

```
        (10) Channels
        (11) Supernodes
        (12) Debugger info
5

Node Tasks:
==========
Total nodes allocated = 5
Nodes in use = 4
Nodes done = 0
Dead nodes = 0

Data available for viewing : (a blank line exits the routine)
        (1) Connection matrix
        (2) Node status information
        (3) Node address information
        (4) Vid information
        (5) Node tasks
        (6) User Id
        (7) Fault tolerance strategies
        (8) Algorithm choice
        (9) Topology choice
        (10) Channels
        (11) Supernodes
        (12) Debugger info
6

User Id = 0

Data available for viewing : (a blank line exits the routine)
        (1) Connection matrix
        (2) Node status information
        (3) Node address information
        (4) Vid information
        (5) Node tasks
        (6) User Id
        (7) Fault tolerance strategies
        (8) Algorithm choice
        (9) Topology choice
        (10) Channels
        (11) Supernodes
        (12) Debugger info
12

Which debug information to view ?
(default is none)
        (1) trace event parameters
        (2) performance parameters
        (3) debug printout message option
        (4) ipc control information
2
        count packets sent and received
        count the types of packets sent and received
        break down execution time
        trace node failures during execution
        trace NM histories
        automatically report measurements on algorithm completion

Which debug information to view ?
(default is none)
        (1) trace event parameters
        (2) performance parameters
```

(3) debug printout message option
(4) ipc control information

Data available for viewing : (a blank line exits the routine)
(1) Connection matrix
(2) Node status information
(3) Node address information
(4) Vid information
(5) Node tasks
(6) User Id
(7) Fault tolerance strategies
(8) Algorithm choice
(9) Topology choice
(10) Channels
(11) Supernodes
(12) Debugger info

> run
RUNTIME routines have been enabled

Command Language :
==========

<add_nodes>	<help | ?>
<change_links>	<monitor>
<command_line>	<quit | bye>
<debug_control>	<reset>
<delete_nodes>	<resume_nodes>
<dos>	<help | ?>
<download>	<suspend_nodes>
<fault_tolerance>	<time>
<flow_control_diagram>	<view_data>
<free_nodes>	

> start
How should the code be run ?
(default is all)
(1) run individual nodes
(2) run all nodes

> debug
Which debug operation to perform ?
(default is none)
(1) set trace event parameters
(2) control or view ipc queue
(3) print out debug messages
(4) set ipc breakpoints
(5) single step

Once the RUNTIME routines are enabled, the **debug_control** *command displays different options. Earlier, in the SETUP phase, it was only possible to set trace event parameters, to set performance parameters, and to set print out debug messages. Although there is a wider assortment of functionality during the RUNTIME phase, the performance parameters can only be set while the algorithm is not executing. Therefore, that option is currently unavailable. Once the algorithm completes, the* **debug_control** *command displays even more options.*

> v

Data available for viewing : (a blank line exits the routine)
(1) Connection matrix
(2) Node status information
(3) Node address information

(4) Vid information
(5) Node tasks
(6) User Id
(7) Fault tolerance strategies
(8) Algorithm choice
(9) Topology choice
(10) Channels
(11) Supernodes
(12) Debugger info

2

Node Status:
========
Idle Vids: 4
Loaded Vids:
Busy Vids:
Suspended Vids:
Done Vids: 0 1 2 3
Dead Vids:

Data available for viewing : (a blank line exits the routine)
(1) Connection matrix
(2) Node status information
(3) Node address information
(4) Vid information
(5) Node tasks
(6) User Id
(7) Fault tolerance strategies
(8) Algorithm choice
(9) Topology choice
(10) Channels
(11) Supernodes
(12) Debugger info

> debug
Which debug operation to perform ?
(default is none)
(1) set trace event parameters
(2) control or view ipc queue
(3) print out debug messages
(4) set ipc breakpoints
(5) single step
(6) analyze trace files
(7) set performance parameters
(8) view performance feedback

8

Which measurement are you interested in seeing ? (default is none)
Timings:
(1) completion time for the entire algorithm
(2) completion time per processor
(3) nm version of completion time per processor
(4) break down of algorithm execution time computation
(5) time algorithm spent downloading
Messages:
(6) total number of messages passed
(7) total number of messages passed per processor
(8) distribution of message types
(9) distribution of message types per processor
(10) non-bbl network activity during algorithm execution
Status:
(11) number of failed nodes owned by user

(12) completion status per NM
(13) state changes per NM (event history)

1

Completion time for the entire algorithm: 00:00:01.58

Which measurement are you interested in seeing ? (default is none)
Timings:
(1) completion time for the entire algorithm
(2) completion time per processor
(3) nm version of completion time per processor
(4) break down of algorithm execution time computation
(5) time algorithm spent downloading
Messages:
(6) total number of messages passed
(7) total number of messages passed per processor
(8) distribution of message types
(9) distribution of message types per processor
(10) non-bbl network activity during algorithm execution
Status:
(11) number of failed nodes owned by user
(12) completion status per NM
(13) state changes per NM (event history)

2

How should the process completion time be calculated ?
(default is all)
(1) calculate individual nodes
(2) calculate all nodes

Completion Time:
==========
Vid 0, 00:00:01.58
Vid 1, 00:00:01.46
Vid 2, 00:00:01.48
Vid 3, 00:00:01.52

Which measurement are you interested in seeing ? (default is none)
Timings:
(1) completion time for the entire algorithm
(2) completion time per processor
(3) nm version of completion time per processor
(4) break down of algorithm execution time computation
(5) time algorithm spent downloading
Messages:
(6) total number of messages passed
(7) total number of messages passed per processor
(8) distribution of message types
(9) distribution of message types per processor
(10) non-bbl network activity during algorithm execution
Status:
(11) number of failed nodes owned by user
(12) completion status per NM
(13) state changes per NM (event history)

4

How should the break down of execution time be calculated ?
(default is all)
(1) calculate individual nodes
(2) calculate all nodes

```
Break down of execution time:
=============================
Vid: 0, total time 00:00:01.58
        communication time 00:00:00.42 (22 %)
        send time 00:00:00.24 (15 %)
        blocked receive time 00:00:00.11 (7 %)
Vid: 1, total time 00:00:01.46
        communication time 00:00:00.78 (51 %)
        send time 00:00:00.07 (5 %)
        blocked receive time 00:00:00.67 (46 %)
Vid: 2, total time 00:00:01.48
        communication time 00:00:00.80 (52 %)
        send time 00:00:00.09 (6 %)
        blocked receive time 00:00:00.68 (46 %)
Vid: 3, total time 00:00:01.52
        communication time 00:00:00.76 (48 %)
        send time 00:00:00.14 (9 %)
        blocked receive time 00:00:00.59 (39 %)

Which measurement are you interested in seeing ? (default is none)
        Timings:
        (1) completion time for the entire algorithm
        (2) completion time per processor
        (3) nm version of completion time per processor
        (4) break down of algorithm execution time computation
        (5) time algorithm spent downloading
        Messages:
        (6) total number of messages passed
        (7) total number of messages passed per processor
        (8) distribution of message types
        (9) distribution of message types per processor
        (10) non-bbl network activity during algorithm execution
        Status:
        (11) number of failed nodes owned by user
        (12) completion status per NM
        (13) state changes per NM (event history)
5

Total time for most recent download: 00:00:06.39

Which measurement are you interested in seeing ? (default is none)
        Timings:
        (1) completion time for the entire algorithm
        (2) completion time per processor
        (3) nm version of completion time per processor
        (4) break down of algorithm execution time computation
        (5) time algorithm spent downloading
        Messages:
        (6) total number of messages passed
        (7) total number of messages passed per processor
        (8) distribution of message types
        (9) distribution of message types per processor
        (10) non-bbl network activity during algorithm execution
        Status:
        (11) number of failed nodes owned by user
        (12) completion status per NM
        (13) state changes per NM (event history)
6

Total # of messages: sent 209, received 125

Which measurement are you interested in seeing ? (default is none)
```

```
        Timings:
        (1) completion time for the entire algorithm
        (2) completion time per processor
        (3) nm version of completion time per processor
        (4) break down of algorithm execution time computation
        (5) time algorithm spent downloading
        Messages:
        (6) total number of messages passed
        (7) total number of messages passed per processor
        (8) distribution of message types
        (9) distribution of message types per processor
        (10) non-bbl network activity during algorithm execution
        Status:
        (11) number of failed nodes owned by user
        (12) completion status per NM
        (13) state changes per NM (event history)
7

How should the total number of messages passed be displayed ?
(default is all)
        (1) display totals for individual nodes
        (2) display totals for all nodes

Total # of messages:
==========
Vid 0,   sent 94, received 52
Vid 1,   sent 31, received 21
Vid 2,   sent 33, received 21
Vid 3,   sent 51, received 31

Which measurement are you interested in seeing ? (default is none)
        Timings:
        (1) completion time for the entire algorithm
        (2) completion time per processor
        (3) nm version of completion time per processor
        (4) break down of algorithm execution time computation
        (5) time algorithm spent downloading
        Messages:
        (6) total number of messages passed
        (7) total number of messages passed per processor
        (8) distribution of message types
        (9) distribution of message types per processor
        (10) non-bbl network activity during algorithm execution
        Status:
        (11) number of failed nodes owned by user
        (12) completion status per NM
        (13) state changes per NM (event history)
8

Which distribution are you interested in seeing ?
(default is none)
        (1) types of messages sent
        (2) types of messages received
1

Distribution of packet types sent:

ACK: 144
SET_FT_PARAMS: 1
SUPER_NODES: 4
USER_DATA: 60
```

```
User: 65 (31 %), System: 144 (69 %), Debugger: 0 (0 %)

Which distribution are you interested in seeing ?
(default is none)
        (1) types of messages sent
        (2) types of messages received
2

Distribution of packet types received:

ACK: 65
USER_DATA: 60

User: 60 (48 %), System: 65 (52 %), Debugger: 0 (0 %)

Which distribution are you interested in seeing ?
(default is none)
        (1) types of messages sent
        (2) types of messages received

Which measurement are you interested in seeing ? (default is none)
        Timings:
        (1) completion time for the entire algorithm
        (2) completion time per processor
        (3) nm version of completion time per processor
        (4) break down of algorithm execution time computation
        (5) time algorithm spent downloading
        Messages:
        (6) total number of messages passed
        (7) total number of messages passed per processor
        (8) distribution of message types
        (9) distribution of message types per processor
        (10) non-bbl network activity during algorithm execution
        Status:
        (11) number of failed nodes owned by user
        (12) completion status per NM
        (13) state changes per NM (event history)
9

Vids with status DONE: 0 1 2 3
================

Which vid's message distribution to display ? (default is none)
0

Which distribution are you interested in seeing ?
(default is none)
        (1) types of messages sent
        (2) types of messages received
1

Distribution of packet types sent for vid 0:

ACK: 62
SET_FT_PARAMS: 1
SUPER_NODE: 1
USER_DATA: 30

User: 32 (34 %), System: 62 (66 %), Debugger: 0 (0 %)

Which distribution are you interested in seeing ?
(default is none)
```

(1) types of messages sent
(2) types of messages received

Which vid's message distribution to display ? (default is none)

Which measurement are you interested in seeing ? (default is none)
Timings:
(1) completion time for the entire algorithm
(2) completion time per processor
(3) nm version of completion time per processor
(4) break down of algorithm execution time computation
(5) time algorithm spent downloading
Messages:
(6) total number of messages passed
(7) total number of messages passed per processor
(8) distribution of message types
(9) distribution of message types per processor
(10) non-bbl network activity during algorithm execution
Status:
(11) number of failed nodes owned by user
(12) completion status per NM
(13) state changes per NM (event history)
10

The non-bbl net activity during algorithm execution:
18 packets OR 9 percent of all packets sent

Which measurement are you interested in seeing ? (default is none)
Timings:
(1) completion time for the entire algorithm
(2) completion time per processor
(3) nm version of completion time per processor
(4) break down of algorithm execution time computation
(5) time algorithm spent downloading
Messages:
(6) total number of messages passed
(7) total number of messages passed per processor
(8) distribution of message types
(9) distribution of message types per processor
(10) non-bbl network activity during algorithm execution
Status:
(11) number of failed nodes owned by user
(12) completion status per NM
(13) state changes per NM (event history)
11

The number of failed nodes owned by the user: 0
(key hits: 0, node crashes: 0)

Which measurement are you interested in seeing ? (default is none)
Timings:
(1) completion time for the entire algorithm
(2) completion time per processor
(3) nm version of completion time per processor
(4) break down of algorithm execution time computation
(5) time algorithm spent downloading
Messages:
(6) total number of messages passed
(7) total number of messages passed per processor
(8) distribution of message types
(9) distribution of message types per processor
(10) non-bbl network activity during algorithm execution

Status:
(11) number of failed nodes owned by user
(12) completion status per NM
(13) state changes per NM (event history)

For comparison purposes, we can re-run the algorithm with a larger data set simply by using the **command** *option.*

> command
Which vid's command line would you like to change ?
(default is to leave command line as is)
0

Please enter the new parameters for node 0 (separated by spaces):
(default parameters are '4000')
8000

>start
How should the code be run ?
(default is all)
(1) run individual nodes
(2) run all nodes

Again, after the algorithm has completed, performance feedback can be obtained about the computation. Once performance numbers have been gleaned from the algorithm execution, we turn off performance measuring and turn on ipc tracing. The events of interest include when packets are sent, when processes wait for them to be delivered, and when they are actually received. This can be established from within the configuration file (by removing the line referring to the **perform** *option and adding one which begins with the* **trace** *option and contains the settings send, receive, recv_wait). This can be also be established interactively, as follows;*

> debug
Which debug operation to perform ?
(default is none)
(1) set trace event parameters
(2) control or view ipc queue
(3) print out debug messages
(4) set ipc breakpoints
(5) single step
(6) analyze trace files
(7) set performance parameters
(8) view performance feedback
7

Which algorithm performance option to change ? (default is none)
(1) count packets sent and received
(2) count the types of packets sent and received
(3) break down execution time
(4) trace node failures during execution
(5) trace NM histories
(6) automatically report measurements on algorithm completion
(7) all of the above
7

Which algorithm performance option to change ? (default is none)
(1) do NOT count packets sent and received
(2) do NOT count the types of packets sent and received
(3) do NOT break down execution time
(4) do NOT trace node failures during execution
(5) do NOT trace NM histories
(6) do NOT automatically report measurements on algorithm completion
(7) all of the above

```
Which debug operation to perform ?
(default is none)
        (1) set trace event parameters
        (2) control or view ipc queue
        (3) print out debug messages
        (4) set ipc breakpoints
        (5) single step
        (6) analyze trace files
        (7) set performance parameters
        (8) view performance feedback
1

How would you like to change debugger traces ? (default is none)
        (1) change debugger trace event settings
        (2) change the style in which traces are performed
1

Which debugger trace event setting to change ? (default is none)
        (1) do NOT trace DATA associated with ipc events
        (2) do NOT trace send packets
        (3) do NOT trace waiting to receive packets
        (4) do NOT trace receive packets
        (5) do NOT trace suspend packets
        (6) do NOT trace resume packets
        (7) do NOT trace when NM goes down
        (8) do NOT trace when algorithm execution begins
        (9) do NOT trace when algorithm execution completes
        (10) do NOT trace link_change requests
        (11) do NOT trace when NMs open files
        (12) do NOT trace when NMs close files
        (13) do NOT trace when NMs read files
        (14) do NOT trace when NMs write to files
        (15) do NOT trace when NMs are reset
        (16) do NOT trace when the user exits BBL
        (17) all of the above
2

Which debugger trace event setting to change ? (default is none)
        (1) do NOT trace DATA associated with ipc events
        (2) trace send packets
        (3) do NOT trace waiting to receive packets
        (4) do NOT trace receive packets
        (5) do NOT trace suspend packets
        (6) do NOT trace resume packets
        (7) do NOT trace when NM goes down
        (8) do NOT trace when algorithm execution begins
        (9) do NOT trace when algorithm execution completes
        (10) do NOT trace link_change requests
        (11) do NOT trace when NMs open files
        (12) do NOT trace when NMs close files
        (13) do NOT trace when NMs read files
        (14) do NOT trace when NMs write to files
        (15) do NOT trace when NMs are reset
        (16) do NOT trace when the user exits BBL
        (17) all of the above
3

Which debugger trace event setting to change ? (default is none)
        (1) do NOT trace DATA associated with ipc events
        (2) trace send packets
        (3) trace waiting to receive packets
        (4) do NOT trace receive packets
```

(5) do NOT trace suspend packets
(6) do NOT trace resume packets
(7) do NOT trace when NM goes down
(8) do NOT trace when algorithm execution begins
(9) do NOT trace when algorithm execution completes
(10) do NOT trace link_change requests
(11) do NOT trace when NMs open files
(12) do NOT trace when NMs close files
(13) do NOT trace when NMs read files
(14) do NOT trace when NMs write to files
(15) do NOT trace when NMs are reset
(16) do NOT trace when the user exits BBL
(17) all of the above

4

Which debugger trace event setting to change ? (default is none)
(1) do NOT trace DATA associated with ipc events
(2) trace send packets
(3) trace waiting to receive packets
(4) trace receive packets
(5) do NOT trace suspend packets
(6) do NOT trace resume packets
(7) do NOT trace when NM goes down
(8) do NOT trace when algorithm execution begins
(9) do NOT trace when algorithm execution completes
(10) do NOT trace link_change requests
(11) do NOT trace when NMs open files
(12) do NOT trace when NMs close files
(13) do NOT trace when NMs read files
(14) do NOT trace when NMs write to files
(15) do NOT trace when NMs are reset
(16) do NOT trace when the user exits BBL
(17) all of the above

How should the parameters be set ?
(default is all)
(1) set individual nodes
(2) set all nodes

Which debug operation to perform ?
(default is none)
(1) set trace event parameters
(2) control or view ipc queue
(3) print out debug messages
(4) set ipc breakpoints
(5) single step
(6) analyze trace files
(7) set performance parameters
(8) view performance feedback

> start
How should the code be run ?
(default is all)
(1) run individual nodes
(2) run all nodes

The execution of the algorithm produces four log files, one for each NM. The log file for vid 0, "log.0", looks as follows;

14:47:14.99 SEND PACKET: to vid 1, size 2002
14:47:15.21 SEND PACKET: to vid 2, size 2002
14:47:15.71 SEND PACKET: to vid 3, size 2002

14:47:16.53 RECEIVE WAIT:
14:47:16.58 RECEIVE PACKET: from vid 1, size 2002, eom
14:47:16.64 RECEIVE WAIT:
14:47:16.97 RECEIVE PACKET: from vid 3, size 4002, eom

The log file "log.1" for vid 1;

14:47:14.39 RECEIVE WAIT:
14:47:15.05 RECEIVE PACKET: from vid 0, size 2002, eom
14:47:15.93 SEND PACKET: to vid 0, size 2002

While the log for vid 2, "log.2", looks as follows;

14:47:14.44 RECEIVE WAIT:
14:47:15.27 RECEIVE PACKET: from vid 0, size 2002, eom
14:47:16.09 SEND PACKET: to vid 3, size 2002

And that of vid 3, "log.3";

14:47:14.06 RECEIVE WAIT:
14:47:15.71 RECEIVE PACKET: from vid 0, size 2002, eom
14:47:16.53 RECEIVE WAIT:
14:47:16.58 RECEIVE PACKET: from vid 2, size 2002, eom
14:47:16.97 SEND PACKET: to vid 0, size 4002

Each event traced is locally timestamped and contains information about the type of event, the vid involved, and the size of the message. The "eom" marking specifies whether or not the packet received is the end of the message. It is possible to change the style of debug trace files; events can be logged on a per packet or per message basis (messages are broken down into packets if they exceed the network packet size). By default, debug trace files log events on a per message basis, so only end-of-message events appear in this example.

Next, assume the debugger settings for traces have been turned off. IPC breakpoints are now established.

> debug
Which debug operation to perform ?
(default is none)
(1) set trace event parameters
(2) control or view ipc queue
(3) print out debug messages
(4) set ipc breakpoints
(5) single step
5

Single stepping is now turned ON
Which debug operation to perform ?
(default is none)
(1) set trace event parameters
(2) control or view ipc queue
(3) print out debug messages
(4) set ipc breakpoints
(5) single step

> start
How should the code be run ?
(default is all)
(1) run individual nodes
(2) run all nodes

> (BREAKPOINT: vid 0 sent message to vid 1, message type -1)

Instead of single stepping (or halting after each message is sent), just break execution when vid 0 receives messages of any type (specified by the use of -1) from any other vid (also specified by -1).

```
> debug
Which debug operation to perform ?
(default is none)
        (1) set trace event parameters
        (2) control or view ipc queue
        (3) print out debug messages
        (4) set ipc breakpoints
        (5) single step
4

Which vid/type combination should cause breakpoints ? (enter on the same line)
(default is none)
-1 -1
Breakpoint at which nodes ?
(default is all)
        (1) individual nodes
        (2) all nodes
1
Vids of nodes: (one per line, a blank line signals the end of the list)
0

Which vid/type combination should cause breakpoints ? (enter on the same line)
(default is none)

Which debug operation to perform ?
(default is none)
        (1) set trace event parameters
        (2) control or view ipc queue
        (3) print out debug messages
        (4) set ipc breakpoints
        (5) single step
5

Single stepping is now turned OFF
Which debug operation to perform ?
(default is none)
        (1) set trace event parameters
        (2) control or view ipc queue
        (3) print out debug messages
        (4) set ipc breakpoints
        (5) single step

> resume
> (BREAKPOINT: vid 1 sent message to vid 0, message type -1)
```

Now that the desired breakpoint has occurred, let us examine the message queue of vid 0. We find that the queue has a user-established length of 20 buffers, with the first buffer, buffer 0, containing the message from vid 1. Buffers are numbered beginning at 0. Although all other buffers in the queue are empty, buffer 0 has 2002 bytes of data. We save the entire message buffer to a file "msg.0" so we can restore it later on (after we play with vid 0's queue). Note that any time one wishes to perform functions on ipc queues, the system asks for a particular vid. By default, it assumes the user wants to handle the queue of the most recently used vid, unless no previous vid has been established yet.

When we add a message to the ipc queue, we can select to add the message in its entirety from a file (like "msg.0") or to input the message information (type, sender, and data) interactively. In turn, the message data can be entered from a file or interactively. In this case, we opt to use a data file called "msgdata.0", essentially a file similar to "msg.0", minus type and sender information.

```
> debug
```

```
Which debug operation to perform ?
(default is none)
        (1) set trace event parameters
        (2) control or view ipc queue
        (3) print out debug messages
        (4) set ipc breakpoints
        (5) single step
2

Which ipc queue operation to perform ?
(default is none)
        (1) view the queue contents
        (2) add a message
        (3) delete a message
        (4) change the contents of a message
        (5) save message to a file
        (6) re-order messages
1

In which vid are you interested ? (default is none)
0

What information are you interesting in viewing ? (default is none)
        (1) Queue length
        (2) Buffer Status
        (3) Size of messages
        (4) Message data
1

Queue Length for vid 0: 20
What information are you interesting in viewing ? (default is none)
        (1) Queue length
        (2) Buffer Status
        (3) Size of messages
        (4) Message data
2

Vid 0's Buffers In use: 0
What information are you interesting in viewing ? (default is none)
        (1) Queue length
        (2) Buffer Status
        (3) Size of messages
        (4) Message data
3

Message Buffer Sizes for vid 0 :
Message 0: 2002 bytes
What information are you interesting in viewing ? (default is none)
        (1) Queue length
        (2) Buffer Status
        (3) Size of messages
        (4) Message data
4

In which message are you interested ? (0-19)
(default is none)
1

No data: message buffer is free
In which message are you interested ? (0-19)
(default is none)
0
```

The packet is from vid 1 and is of type -1
How many bytes of user data to display ? (0-2002)
(default is none)

In which message are you interested ? (0-19)
(default is none)

What information are you interesting in viewing ? (default is none)
(1) Queue length
(2) Buffer Status
(3) Size of messages
(4) Message data

Which ipc queue operation to perform ?
(default is none)
(1) view the queue contents
(2) add a message
(3) delete a message
(4) change the contents of a message
(5) save message to a file
(6) re-order messages
5

In which vid are you interested ? (default is 0)

Which message buffer would you like to save ?
(default is none)
0

The name of an output file?
msg.0

Which message buffer would you like to save ?
(default is none)

Which ipc queue operation to perform ?
(default is none)
(1) view the queue contents
(2) add a message
(3) delete a message
(4) change the contents of a message
(5) save message to a file
(6) re-order messages
2

In which vid are you interested ? (default is 0)

Before which buffer would you like to add an ipc message ?
(default is none)
0

How will the message be entered?
(1) interactively
(2) via an input file
1

From which vid is this packet ?
2

What is the packet type?
-1

How many bytes of data in the packet ? (0-2002)
2002

How will the data be entered ?
(1) interactively
(2) via an input file
2

The name of the input file?
msgdata.0

Before which buffer would you like to add an ipc message ?
(default is none)

Which ipc queue operation to perform ?
(default is none)
(1) view the queue contents
(2) add a message
(3) delete a message
(4) change the contents of a message
(5) save message to a file
(6) re-order messages
3

In which vid are you interested ? (default is 0)

Which message buffer would you like to delete from vid 0's queue ?
(default is none)
0

Which message buffer would you like to delete from vid 0's queue ?
(default is none)

Which ipc queue operation to perform ?
(default is none)
(1) view the queue contents
(2) add a message
(3) delete a message
(4) change the contents of a message
(5) save message to a file
(6) re-order messages
4

In which vid are you interested ? (default is 0)

Which message buffer would you like to change ?
(default is none)
0

In what manner would you like to change the message ?
(1) the message 'sender' vid
(2) the message 'type'
(3) message data
(4) all of the above (the message in full)
4

How will the message be entered ?
(1) interactively
(2) via an input file
2

The name of the input file?

```
msg.0

Which message buffer would you like to change ?
(default is none)

Which ipc queue operation to perform ?
(default is none)
        (1) view the queue contents
        (2) add a message
        (3) delete a message
        (4) change the contents of a message
        (5) save message to a file
        (6) re-order messages

Which debug operation to perform ?
(default is none)
        (1) set trace event parameters
        (2) control or view ipc queue
        (3) print out debug messages
        (4) set ipc breakpoints
        (5) single step
```

Finally, assume we have reset the system and turned on debug printout messages at the UI/PM (either interactively or from the configuration file). During the download phase, vid 1 is re-claimed by its owner after the RM has allocated it to the UI/PM. Printout messages show how the PM probes the NM, finds it not responding, replaces it with the fifth and extra NM already assigned to the UI/PM, updates connection information (so all neighboring nodes know about the new physical address of the replacement node) and then re-downloads the code to the new NM.

```
> down
How should the download be performed ?
(default is all)
        (1) download individual nodes
        (2) download all nodes

vdload: downloading vid  0
send_pac: allocated buffer space 15612, ptrp 56EB:000E
code size: 15504 bytes
code checksum is BF73
vdload: vid 0 DOWNLOAD successful
vdload: downloading vid  1
send_pac: allocated buffer space 25758, ptrp 56EB:000E
code size: 25650 bytes
code checksum is B622
e_send: Never got ack  ptype = 2  seqno = -1  time_out = 9400
        size = 372  send_size = 372  rem_size = 372  pkt_cnt = 18
vdload: vid 1 DOWNLOAD retval = 85
dload: DOWNLOAD for vid 1 failed
maximum number of resends exceeded
Error: node 1 at address 002 140 214 010 105 046  NOT responding
i_am_down: finding a replacement node for vid 1 ...
i_am_down: asking the PM for a replacement node
i_am_down_r: PM finds a replacement node for vid 1
i_am_down_r: updating connection info
vdload: downloading vid  1
send_pac: allocated buffer space 25758, ptrp 56EB:000E
code size: 25650 bytes
code checksum is B622
vdload: vid 1 DOWNLOAD successful
vdload: downloading vid  2
send_pac: allocated buffer space 25758, ptrp 56EB:000E
code size: 25650 bytes
```

```
code checksum is B622
vdload: vid 2 DOWNLOAD successful
vdload: downloading vid  3
send_pac: allocated buffer space 25758, ptrp 56EB:000E
code size: 25650 bytes
code checksum is B622
vdload: vid 3 DOWNLOAD successful
> reset
reset_sys: send FREE_NM
reset_sys: free up allocated space
free algorithm related space
reset_sys: send SUICIDE
reset_sys: reset channels
reset_sys: free up debug info
reset_sys: re-initialize debug info
reset_sys: re-initialize structures
> quit
pm_quit: send SUICIDE
pm_quit: send FREE_NM
pm_quit: free up allocated space
free algorithm related space
pm_quit: free up debug info
First reset got (89)
Second reset got (89)
RESTORING INTERRUPT VECTOR
```

11 Appendix C: BBLUSER.H

```
/*  BBLUSER.H
 *
 * Include file for user code
 */

#ifndef FALSE
#define FALSE       0
#define TRUE        -1
#endif
#define CR_LF "\015\012"


#define U_QUEUE_SIZE        42      /* (42 max) number of packet buffers */
#define MAX_LINKS           25      /* max num of input or output links */

/* set parameters defines */
#define REPLACE             TRUE
#define NO_REPLACE          FALSE
#define NOTIFY              TRUE
#define NO_NOTIFY FALSE


/* Packet Types used by PM */
#define DEAD_NODE           0x8001
#define AVAIL_NODE          0x8002

/* VID of PM */
#define PM_VID              0x8000

#define PACKET_HEADER_SIZE          12

struct user_packet {
        int To;     /* sent to VID */
        int From;   /* sent from VID */
        int Type;     /* packet type */
        long Count;             /* "Timestamp" incremented for each pkt rcvd */
        int eom;    /* end of message flag (used by system) */
        unsigned char Data[1480];
};

struct down_packet {
        int To;     /* sent to VID */
        int From;   /* sent from VID */
        int Type;     /* packet type = 0x8001 */
        long Count;             /* "Timestamp" incremented for each pkt rcvd */
        int eom;    /* end of message used by system */
        int vid;    /* vid that went down */
        int ack;    /* if a replacement was found */
};

struct user_packet input_buffer[U_QUEUE_SIZE];
int input_size[U_QUEUE_SIZE] = {0};
int input_free[U_QUEUE_SIZE] = {TRUE} ;

/* must call Init_Vars(input_buffer,input_size,input_free,U_QUEUE_SIZE,exit_routine)
 * to initialize these variables Init_Vars returns TRUE if this node is restart
 * of a node that went down
 */

extern int Init_Vars(struct user_packet *buffer,
                        int *size, int* free, int Q_size, void (*user_exit)());
extern Super_Node(void);
extern Set_Parameters(int replace, int notify);
extern Send_Packet(int vid, int type, char *paddr, int length, int *ret_code);
extern int Receive_Exist(int vid, int type);
extern Receive_Packet(int *vid, int *type, struct user_packet *paddr, unsigned int *size);
extern Get_VID(int *vid);
extern Get_IN_Connections(int *num, int list[]);
extern Get_OUT_Connections(int *num, int list[]);
extern BBL_Open(char *path, int flags, int exclusive, int *retval, int *errno);
extern BBL_Close(int file_handle, int *retval, int *errno);
extern BBL_Write(int file_handle, char *addr, int size, int *retval, int *errno);
extern BBL_Read(int file_handle, char *addr, int size, int *retval, int *errno);
extern int Time_Synch(void);
```

12 Appendix D: Sample Code

The following three "C" files make up the code for the mergesort distributed algorithm. The file "merger.c" is the code that runs on one node (vid 0). "Sorter.c" runs on all the other nodes. The third file "mergsort.c" contains routines used by both the merger and sorter, such as the actual merging routine.

12.1 Merger.c

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>
#include <time.h>
#include <process.h>
#include <conio.h>
#include <dos.h>
#include <math.h>
#include "language.h"
#include "bbluser.h"

extern int compare(int *i1, int *i2);                    /* comparison function for qsort */
extern dump_buffer(int buf);
extern long convert_time(struct time *tim);
extern send_data(int vid, int *final_array);
extern int *wait_and_merge(int *final_array, int *data_count);

#define NUM_LOOPS       1

#define SORTED          100     /* packet type */
#define NOT_SORTED      200             /* packet type */

int location;

#define DEFAULT_DATA 100
int *all_data;                          /* all of the data */
int *final_array;                       /* final sorted list */
int *temp_array;                        /* temp sorted list */
int *split_data[MAX_LINKS];             /* each node's data */
int out_links[MAX_LINKS];               /* output links */
int sorted[MAX_LINKS];                  /* tells whether its sorted yet */
int first_time;

int *data_list;                         /* used to cast packet data */
int buffer_size = 0;

int copy_spot,each_total,num_out,remain,total_data,total_nodes,my_vid;
int retval,up_to,data_count,done,got_one;
int read_spot = 0;
int merge_count;
int debug = FALSE;
void user_exit(void);
int *temp_ptr;
long start_mem,end_mem;


long convert_time(struct time *buf);
struct time start_time,end_time,current_time;
long    total_time,start_hund,diff_hund,curr_hund;  /* time in hundredths */


main(argc, argv)
int argc;
char *argv[];
{
        int i,j;
        unsigned int rand_init;
        int loop, hsecs;

        Init_Vars(input_buffer,input_size,input_free,U_QUEUE_SIZE,user_exit);

        start_mem = farcoreleft();

        total_time = 0;
```

```
for (i=0; i<23; i++) WRITELN;
WRITE("MERGER\n\n");

if (argc>1) sscanf(argv[1], "%d",&total_data);
else total_data = DEFAULT_DATA;

if (argc>2) debug = TRUE;

gettime(&start_time);
rand_init = (unsigned)(start_time.ti_hund * start_time.ti_sec);
srand(rand_init);
WRITE("random generator starting at %u \n",rand_init);

Set_Parameters(NO_REPLACE, NO_NOTIFY);
Super_Node();

Get_VID(&my_vid);
WRITE("My vid = %d \n",my_vid);

Get_OUT_Connections(&num_out,out_links);
WRITE("I have %d output links\n",num_out);

total_nodes = num_out + 1;                              /* include me */

WRITE("Total_nodes = %d \n",total_nodes);


for (loop=0; loop<NUM_LOOPS; loop++) {

        all_data = (int *)farmalloc(total_data*sizeof(int));        /* get space for data */
        if (!all_data) {
                WRITE("Couldn't allocate space for all_data\n");
                exit(0);
        }

        for (i=0; i< total_data; i++) all_data[i] = rand();        /* generate random data */


        if (total_nodes == 1) {
                WRITE("Only me doing the sorting\n");

                gettime(&start_time);
                start_hund = convert_time(&start_time);


                qsort(all_data,total_data,sizeof(int),compare);

                /* write time end */

                WRITE("All done \n");
                gettime(&end_time);
                WRITE("total_data = %d total_nodes = %d  ",
                        total_data,total_nodes);


                diff_hund = convert_time(&end_time) - start_hund;

                WRITE(" time  %6ld.%02ld \n",(diff_hund/100),(diff_hund MOD 100));
                total_time += diff_hund;

                farfree(all_data);
                all_data = NULL;
        }


        /*****************  MORE THAN ONE NODE  ********************/
        else {
                /* more than 1 node */

                gettime(&start_time);
                start_hund = convert_time(&start_time);


                remain = total_data MOD total_nodes;
                each_total = total_data/total_nodes;
                copy_spot = 0;
```

```
if (remain) {
        up_to = total_nodes-1;
        remain = remain + each_total;
}
else up_to = total_nodes;


/* allocate space and copy data to separate arrays */
for (i=0; i<(up_to); i++) {
        split_data[i] = (int *) farmalloc((each_total+1)*sizeof(int));
        if (!split_data[i]) {
                WRITE("Couldn't allocate space for split_data[%d]\n",i);
                for (j=0;j<i;j++) {
                        farfree(split_data[j]);
                        split_data[j] = NULL;
                }
                farfree(all_data);
                all_data = NULL;
                exit(0);
        }
        for (j=1; j<=each_total; j++) split_data[i][j] = all_data[copy_spot++];
        split_data[i][0] = each_total;
}

if (remain) {
        i = total_nodes -1;
        split_data[i] = (int *) farmalloc((remain+1)*sizeof(int));
        if (!split_data[i]) {
                WRITE("Couldn't allocate space for split_data[%d]\n",i);
                for (i=0;i<up_to; i++) {
                        farfree(split_data[i]);
                        split_data[i] = NULL;
                }
                farfree(all_data);
                all_data = NULL;
                exit(0);
        }
        for (j=1; j<=remain; j++) split_data[i][j] = all_data[copy_spot++];
        split_data[i][0] = remain;
} /* endif */

/* download data to each processor */

for (i=0; i<(total_nodes-1); i++) {

        if (debug) WRITE("Sending to node %d \n",out_links[i]);

        retval = 0;
        Send_Packet(out_links[i],0,(char *)split_data[out_links[i]],
                ((split_data[out_links[i]][0]+1)*sizeof(int)),
                        &retval);
        sorted[out_links[i]]=FALSE;

        if (retval) {
                WRITE("Error in sending to vid %d:  %X\n",out_links[i],retval);
                WRITE("Exiting\n");
                for (j=0;j<total_nodes;j++) {
                        farfree(split_data[j]);
                        split_data[j] = NULL;
                }
                farfree(all_data);
                all_data = NULL;
                exit(0);
        }
} /* endfor */


if (debug) WRITE("Sorting my piece\n");
qsort(&split_data[my_vid][1],split_data[my_vid][0],sizeof(int),compare);
sorted[my_vid] = TRUE;

final_array = split_data[my_vid];
data_count = split_data[my_vid][0];

/* free up other arrays */
for (i=0; i<total_nodes; i++) {
        if (i != my_vid) {
                if (split_data[i]) {
```

```
                                        farfree(split_data[i]);
                                        split_data[i] = NULL;
                                }
                        }
                }

                first_time = TRUE;
                temp_ptr = NULL;
                done = FALSE;
                while (!done) {

                        /* wait for packets and process them */
                        if (debug) WRITE("Waiting for packet  %d/%d \n",final_array[0],total_data);

                        temp_ptr = wait_and_merge(final_array, &data_count);

                        if (first_time) {
                                farfree(split_data[my_vid]);
                                split_data[my_vid] = NULL;
                                first_time = FALSE;
                        }
                        else {
                                if (final_array) {
                                        farfree(final_array);
                                        final_array = NULL;
                                }
                        }
                        final_array = temp_ptr;

                        if (data_count>=total_data) done = TRUE;

                } /* endwhile !done*/

                gettime(&end_time);
                WRITE("%2d.  total_data = %d total_nodes = %d  ",(loop+1),total_data,total_nodes);

                diff_hund = convert_time(&end_time) - start_hund;

                WRITE(" time   %6ld.%02ld \n",(diff_hund/100),(diff_hund MOD 100));
                total_time += diff_hund;


                if (argc >3) {
                        for (i=0; i<total_data; i++)
                        WRITE("%2d   %6d    %6d \n",i,all_data[i],final_array[i+1]);
                }


        if (temp_ptr) farfree(temp_ptr);
        temp_ptr = NULL;
        if (all_data) farfree(all_data);
        all_data = NULL;
        for (i=0; i<total_nodes; i++) {
                if (split_data[i]) farfree(split_data[i]);
                split_data[i] = NULL;
        }

        } /* else more than one node */

} /* for loop */

hsecs = 100 * NUM_LOOPS;

WRITE("\nAVG TOTAL_TIME = %6ld.%02ld \n",(total_time/hsecs),(total_time MOD hsecs));

end_mem = farcoreleft();
WRITE("\n start_mem = %ld    end_mem = %ld    diff = %ld \n",
                start_mem,end_mem,(start_mem-end_mem));
}


void user_exit(void)
{
        int i;

        WRITE("USER_EXIT: exit function called by NM\n");

        if (temp_ptr) farfree(temp_ptr);
```

```
        if (all_data) farfree(all_data);
        for (i=0; i<total_nodes; i++) if (split_data[i]) farfree(split_data[i]);

        end_mem = farcoreleft();
        WRITE("\n start_mem = %ld   end_mem = %ld   diff = %ld \n",
                                start_mem,end_mem,(start_mem-end_mem));

        exit(0);
}
```

12.2 Sorter.c

```
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#include <dos.h>
#include <math.h>
#include <alloc.h>
#include "language.h"
#include "bbluser.h"

/* function prototypes */
extern int compare(int *i1, int *i2);                    /* comparison function for qsort */
extern dump_buffer(int buf);
extern long convert_time(struct time *tim);
extern send_data(int vid, int *final_array, int type);
extern int *wait_and_merge(int *final_array, int *data_count);
extern dump_location(void);

#define NUM_LOOPS          1
#define MAXSIZE            0x8100          /* buffer for packet */
#define SORTED             100       /* packet type */
#define NOT_SORTED         200             /* packet type */

#define NOTHING            0               /* location */
#define WAITING            1               /* location */
#define MERGING            2               /* location */
#define SORTING            3               /* location */
#define SENDING            4               /* location */
int location = NOTHING;

long start_mem,end_mem;

#define W -1        /* wait */
#define X  0        /* nothing */
#define S0 0
#define S1 1
#define S2 2
#define S3 3
#define S4 4
#define S5 5
#define S6 6
#define S7 7
#define S8 8
#define S9 9
#define S10         10


/* "where" tells sorters when to wait and when to send
        indexing:  where[total_nodes][cycle-1][my_vid]
*/


int where[11] [3] [10] = {
                    {{X,X,X,X,X,X,X,X,X,X},{X,X,X,X,X,X,X,X,X,X},{X,X,X,X,X,X,X,X,X,X}},
                    {{X,X,X,X,X,X,X,X,X,X},{X,X,X,X,X,X,X,X,X,X},{X,X,X,X,X,X,X,X,X,X}},
                    {{W,S0,X,X,X,X,X,X,X,X},{X,X,X,X,X,X,X,X,X,X},{X,X,X,X,X,X,X,X,X,X}},
                    {{W,S0,S0,X,X,X,X,X,X,X},{W,X,X,X,X,X,X,X,X,X},{X,X,X,X,X,X,X,X,X,X}},
                    {{W,S0,S3,W,X,X,X,X,X,X},{W,X,X,S0,X,X,X,X,X,X},{X,X,X,X,X,X,X,X,X,X}},
                    {{W,S0,S3,W,S0,X,X,X,X,X},{W,X,W,S0,X,X,X,X,X,X},{X,X,X,X,X,X,X,X,X,X}},
                    {{W,S0,S3,W,S5,W,X,X,X,X},{W,X,X,S0,X,S0,X,X,X,X},{W,X,X,X,X,X,X,X,X,X}},
                    {{W,S0,S3,W,S5,W,S5,X,X,X},{W,X,X,S0,X,W,X,X,X,X},{W,X,X,X,X,S0,X,X,X,X}},
                    {{W,S0,S3,W,S5,W,S7,W,X,X},{W,X,X,S0,X,S7,X,W,X,X},{W,X,X,X,X,X,X,S0,X,X}},
                    {{W,S0,S3,W,S5,W,S7,W,S0,X},{W,X,X,S5,X,W,X,S0,X,X},{W,X,X,X,X,S0,X,X,X,X}},
                    {{W,S0,S3,W,S5,W,S7,W,S9,W},{W,X,X,S5,X,W,X,S9,X,W},{W,X,X,X,X,S0,X,X,X,S0}}
                        };


int *data_list;                                  /* used to cast packet data */

int retval;
int my_vid;
struct user_packet *buffer;
int buffer_size;
int *final_array;
int data_count;
```

```
int send_to;
int total_nodes;
int num_out[MAX_LINKS];
int done,merge;
int cycle = 1;
int max_cycles;
int debug = FALSE;
void user_exit(void);
main(argc)
{
        int i;
        int *temp_ptr;
        int first_time;
        int loop;
        int from,type;

        Init_Vars(input_buffer,input_size,input_free,U_QUEUE_SIZE,user_exit);

        start_mem = farcoreleft();

        if (argc > 1) debug = TRUE;
        Super_Node();

        Get_VID(&my_vid);
        for (i=0; i<23; i++) WRITELN;
        WRITE("SORTER   Running   vid = %d\n",my_vid);

        Get_OUT_Connections(&total_nodes, num_out);
        total_nodes++;          /* include me */

        max_cycles = (int) (0.0 + log10((double)total_nodes)/log10(2));
        WRITE("Max cycles = %d\n",max_cycles);


        for (loop=0; loop<NUM_LOOPS; loop++) {

                location = NOTHING;
                buffer = (struct user_packet *)farmalloc(MAXSIZE);
                if (!buffer) {
                        WRITE("Couldn't allocate space for buffer\n");
                        exit(-1);
                }
                WRITE("Waiting\n");
                /* wait for packet */
                from = -1;
                type = -1;
                Receive_Packet(&from, &type, buffer, &buffer_size);
                location = SORTING;

                data_list = (int *)buffer->Data;
                if (debug) WRITE("Got data total = %d\n",data_list[0]);

                final_array = data_list;
                data_count = data_list[0];
                /* sort my piece */
                qsort(&data_list[1],data_list[0],sizeof(int),compare);
                location = SORTED;


                /* decide whether to wait or send */
                cycle = 1;
                done = FALSE;
                first_time = TRUE;
                while (!done) {

                        if (where[total_nodes][cycle-1][my_vid]==-1) {
                                temp_ptr = final_array;
                                final_array = wait_and_merge(final_array,&data_count);
                                if (first_time) {
                                        if (buffer) farfree(buffer);
                                        buffer = NULL;
                                        first_time = FALSE;
                                }
                                else if (temp_ptr) farfree(temp_ptr);
                        }
                        else {
                                send_to = where[total_nodes][cycle-1][my_vid] ;
                                send_data(send_to, final_array, SORTED);
```

```
                                        done = TRUE;
                                        location = NOTHING;
                                }

                                cycle++;

                        } /* while !done */

                        if (buffer) farfree(buffer);
                        else if (final_array) farfree(final_array);

                } /* for loop */

                end_mem = farcoreleft();
                WRITE("\n start_mem = %ld   end_mem = %ld   diff = %ld \n",
                                start_mem,end_mem,(start_mem-end_mem));

}

void user_exit(void)
{

                WRITE("USER_EXIT: exit function called by NM\n");

                dump_location();

                WRITE("Current location: ");

                switch (location){

                        case NOTHING:
                                WRITE("NOTHING \n");
                                break;

                        case SORTED:
                        case WAITING:
                                WRITE("WAITING, so send data to vid = 0\n");
                                send_data(0, final_array, SORTED);
                                break;

                        case SORTING:
                                WRITE("SORTING, sending to 0 \n");
                                send_data(0, final_array, NOT_SORTED);
                                break;

                        case MERGING:
                                WRITE("MERGING, this is the tough one\n");
/*                              send_data(0, final_array, NOT_SORTED); */
                                break;

                        default: WRITE("DEFAULT\n");

                } /* switch */


                if (buffer) farfree(buffer);
                if (final_array) farfree(final_array);

                exit(0);
}
```

12.3 Mergsort.c

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>
#include <time.h>
#include <dos.h>
#include "language.h"
#include "bbldefs.h"

#define MAXSIZE         0x7FFF+PACKET_HEADER_SIZE /* max packet buffer */
#define SORTED          100     /* packet type */
#define NOT_SORTED      200             /* packet type */
#define NOTHING         0               /* location */
#define WAITING         1               /* location */
#define MERGING         2               /* location */
#define SORTING         3               /* location */
#define SENDING         4               /* location */

extern int debug;
extern int location;


int *merge_sort(int *array1,int *array2,int *final_size);
extern user_exit(void);


/*
 * merge_sort
 * takes ptr1, size1, ptr2, size2, ptr(size3)
 * returns pointer to merged list
 *  NOTE: array[0] is the number of elements in the array
 */

int *merge_sort(array1,array2,size3)
int *array1;
int *array2;
int *size3;
{
        int *array3;
        int pt1,pt2,pt3;
        int end1,end2;

        location = MERGING;
        if (debug) WRITE("Merge_Sort: two lists of size  %d  &  %d \n",
                                        array1[0],array2[0]);

        array3 = (int *)farmalloc((array1[0]+array2[0]+1)*sizeof(int));
        if (!array3) {
                WRITE("Merge_Sort couldn't allocate space \n");
                WRITE("Free space = %ld \n",farcoreleft());
                user_exit();
        }
        end1 = array1[0];
        end2 = array2[0];
        pt1 = pt2 = pt3 = 1;

        while ((pt1 <= end1) AND (pt2 <= end2)) {
                if (array1[pt1] < array2[pt2]) array3[pt3++] = array1[pt1++];
                else array3[pt3++] = array2[pt2++];
        }

        /* copy remaining elements */
        while (pt1 <= end1) array3[pt3++] = array1[pt1++];

        while (pt2 <= end2) array3[pt3++] = array2[pt2++];

        size3[0] =  pt3-1;
        array3[0] = pt3-1;

        return(array3);
}

compare(i1,i2)
int *i1;
```

```
int *i2;
{
  if (i1[0] < i2[0]) return(-1);
  if (i1[0] > i2[0]) return(1);
  return(0); /* == */
}


/* WAIT_AND_MERGE */
/* this function will wait for a data packet and merge it with its current list */

int *wait_and_merge(f_array,data_count)
int f_array[];
int *data_count;
{
        int *temp_array;
        int buffer_size;
        struct user_packet *buff;
        int *data_list;
        int from,type;

        buff = (struct user_packet *)malloc(MAXSIZE);
        if (!buff) {
                WRITE("Wait and Merge: Couldn't allocate (buff) space\n");
                WRITE("Free space = %u \n",coreleft());
                WRITE("Exiting\n");
                exit(-1);
        }


        from = -1;
        type = -1;
        location = WAITING;
        Receive_Packet(&from, &type, buff, &buffer_size);
        location = MERGING;
        data_list = (int *)buff->Data;

        if (debug) {
                WRITE("Wait_and_Merge: got list FROM= %d     SIZE= %d    ",
                                        from,data_list[0]);
                if (type == SORTED) WRITE("SORTED\n");
                else WRITE("NOT_SORTED\n");
        }

        if (type == NOT_SORTED) {
                /* sort it first */
                qsort(&data_list[1],data_list[0],sizeof(int),compare);
        }

        temp_array = merge_sort(f_array, data_list, data_count);
        free(buff);
        return(temp_array);
}

/*************** SEND_DATA ***************************/
/* this function will send off a data packet to "send_to" */
send_data(vid,array,type)
int vid;
int *array;
int type;
{
        int send_count;
        int retval;

        location = SENDING;
        send_count = 0;
        retval = -1;
        while (retval AND (send_count<5))           {
                if (debug) WRITE("Send packet to VID = %d    SIZE = %d\n",
                                        vid, array[0]);
                Send_Packet(vid,type,(char *)array,(sizeof(int)*(array[0]+1)),
                                        &retval);
                send_count++;
        }
        if (retval) {
                if (debug)
                        WRITE("Send to VID = %d failed, sending to vid = 0\n",vid);
```

```
                Send_Packet(0,type,(char *)array,(sizeof(int)*(array[0]+1)),
                                        &retval);
        }


}

/******************** CONVERT_TIME *************************/
long convert_time(buf)
struct time *buf;
{
        long tmp;

        tmp = (long)buf->ti_hund +
                (long)((long)buf->ti_sec  * 100) +
                (long)((long)buf->ti_min  * 6000) +
                (long)((long)buf->ti_hour * 360000) ;
        return(tmp);
}


dump_location()
{

        switch (location) {

                case SORTED:
                        WRITE("location = SORTED\n");
                        break;

                case NOT_SORTED:
                        WRITE("location = NOT_SORTED\n");
                        break;

                case NOTHING:
                        WRITE("location = NOTHING\n");
                        break;

                case WAITING:
                        WRITE("location = WAITING\n");
                        break;

                case MERGING:
                        WRITE("location = MERGING\n");
                        break;

                case SORTING:
                        WRITE("location = SORTING\n");
                        break;

                case SENDING:
                        WRITE("location = SENDING\n");
                        break;

                default: WRITE("location = NOT DEFINED\n");
        }
}
```