Terminal I/O for Massively Parallel Systems

Dror G. Feitelson
IBM T. J. Watson Research Center
P. O. Box 218, Yorktown Heights, NY 10598

Abstract

To be useful, terminal I/O on massively parallel MIMD machines must be able to differentiate between the I/O streams from different tasks. This is done in the Vulcan terminal I/O facility by providing a special control panel, which allows an independent window to be opened for each task. The controls look like LEDs, being color coded to indicate status (e.g. output is available or the task is waiting for input). Additional LEDs are provided as a new form of output, allowing the application to report status visually rather than by using text output. This is useful during program development and debugging. A derivative of the devices reported here has been incorporated in the AIX Parallel Environment on the new IBM SP1 multicomputer.

1 Introduction

Video data terminals are ill-suited to serve as I/O devices for massively parallel computers, because they cannot display text from hundreds of processing elements (PEs) effectively. But this is just a symptom of the real problem, which is that human users cannot digest such an influx of data (left of Fig. 1). Humans by nature can only concentrate on one data stream at a time. Therefore many multiprocessor users append PE identification tags to their output commands, and then spend considerable time sifting through the data to find the output from one specific PE in which they are interested. Regrettably, it is difficult to create a mental picture of the status of the whole machine based on this type of output.

The dual problem of sending input to a specific PE is sometimes even worse. In some systems, there is no way to direct input from the terminal: it goes to whichever PE happened to send the first unsatisfied input request. It is then again up to the programmer to deal with the unpredictable outcome.

A legitimate question is whether terminal I/O is really needed? In uniprocessors, it is sometimes used for actual input and output from a program in production use. In multiprocessors, these functions are nearly always performed using files. But terminal I/O is important during program development and debugging, mainly to report status and show progress on all the PEs. When a new program malfunctions, nearly everyone uses simple print statements to find out what happened before invoking a debugger. In some parallel systems, adequate debuggers simply do not exist.

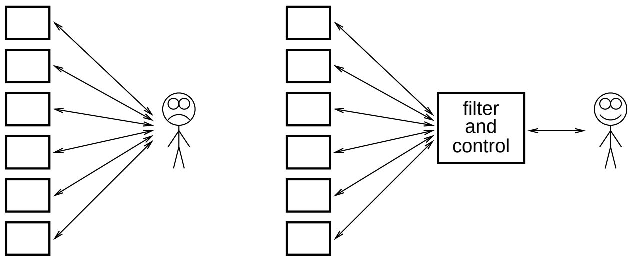


Figure 1: *The mismatch between the I/O capabilities of multiple processors and one user (left) can be rectified by a facility to filter the desired data and control the display (right).*

One solution to the problem of overwhelming users with terminal I/O is to limit the patterns of I/O that are supported. For example, the Express environment [11] encourages either common I/O, where all the PEs output and/or input exactly the same data, or interleaved I/O, where exactly one data item relating to each PE is used, and its place in the sequence is determined by the PE's number. However, forcing all programs into such patterns may not always be suitable.

We suggest another approach: instead of restricting the program, provide control over the display. This is done by inserting a terminal I/O facility that mediates between the multiprocessor and the user (Fig. 1 right). This facility is used to filter the desired I/O, and store the rest for future reference if needed. Its main attributes are the easy matching of I/O with PE, control over what I/O activity is actually observed by the user, and support for simple and easily comprehended status reporting. Such a facility has been implemented as part of the Vulcan project at IBM T. J. Watson Research Center. A derivative of this work has been incorporated in the runtime environment of the IBM SP1 product line [5], where it is called the "program marker array".

The idea is to use windows, and specifically, the X window system. In most multiprocessor systems, all the I/O is done through the same window. Some systems, e.g. PASM, allow separate windows to be opened to all the PEs [9]. We improve on this by giving the user better control over which

PEs will have windows, and by providing cues when it is beneficial to open additional windows.

In addition, we provide a new type of output for status reporting. Nearly all multiprocessors use LEDs[1] mounted on the front panel to report hardware status; the Connection Machine is especially famous for its flashing red LEDs [4], but they also proved useful on other machines, e.g. the IBM RP3 and Delta Touchstone [10]. Software versions of such displays have been used to show the output of monitoring instrumentation [3, 8, 12]. We improve on this by providing unlimited user-definable LEDs, which can be used to represent application status.

It should be noted that this terminal I/O facility is not a graphical output in the usual sense. It makes use of a graphical X terminal, but does not require any sophistication from the programmer and also does not provide support for general graphics. In particular, it is not suitable for visualizing output except perhaps in simplest of cases (e.g. the game of life). What we are dealing with is only an extension to conventional, textual terminal I/O.

2 The Vulcan Terminal I/O Facility

2.1 Vulcan architecture

The Vulcan research multiprocessor is a MIMD, distributed memory, message passing machine [13]. There are three types of nodes in the system: compute nodes, I/O nodes, and host nodes. The compute nodes may be partitioned into disjoint sets which are allocated to users upon demand. Users have exclusive use of the compute nodes in their user partition, and can use them to run parallel applications without interference from other applications belonging to other users. Each compute node executes exactly one *task*, which provides the software environment for the computation.

I/O nodes are shared by all the users. These nodes provide secondary storage services to parallel applications by allowing parallel access to multiple disks.

Host nodes are actually workstations which are connected to the Vulcan network. Users log onto the host nodes, and use them to acquire user partitions and execute parallel programs on them. The host nodes provide an interface between Vulcan and the outside world, including local area networks. In particular, they provide a gateway for terminal I/O, and link Vulcan to the user's X station (Fig. 2).

All the Vulcan nodes are connected by a high-performance multistage packet-switched network with cut-through routing. The network provides high bandwidth communication with low latency between any two nodes, and fosters the illusion of a completely-connected network. In addition, the network is used to distribute a synchronous clock signal to all the nodes.

[1] Light Emitting Diodes.

The Vulcan architecture is scalable up to a total of 32768 nodes, with a mix of compute nodes, I/O nodes, and hosts. The total computing power of a full scale machine is on the order of one TeraFLOP. A prototype with 16 compute nodes and 4 I/O nodes with 8 disks each has been constructed. The terminal I/O facility described here was implemented on a software emulator, before the prototype became available.

2.2 Design of terminal I/O facility

The Vulcan terminal I/O facility is part of the Vulcan Operating Environment [2]. It is based on an X-windows display. The heart of the terminal I/O facility is the "LED array" display. This is a rectangular window divided into subrectangles, one for each task. The number of subrectangles is equal to the size of the partition that is allocated to the user (there may be a few extra, e.g. a partition of 10 may have 12 subrectangles arranged in a 3×4 array). Each of these subrectangles may be considered as a button[2] controlling the terminal I/O facilities of its task. Placing the mouse on it and pressing the middle button causes a dedicated text window to be opened for I/O with that task. Pressing the button again causes the window to be closed. Pressing the right mouse button causes a popup to appear, displaying the task number.

The subrectangles representing the tasks are color-coded to show their I/O status. The appearance of subrectangles changing color at run time motivated the name "LED array". The following four colors are used:

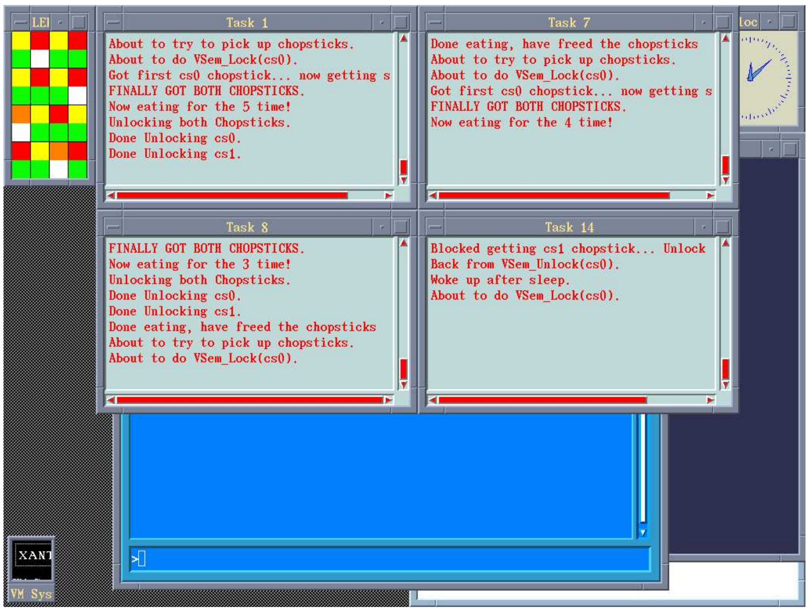
- Green: the task has sent output which has not been seen yet. The window should be opened for the output to be displayed.
- Red: the task is waiting for input. The window should be opened and some input typed in.
- White: the task currently has an open window.
- Gray: there is no I/O activity with this task.

In addition, two colors are used to indicate that this task is actually not part of the application. These are

- Black: this subrectangle does not represent any PE.
- Dark gray: this subrectangle represents a PE in the user's partition, but there is no task running on this PE.

If the system uses color coding of rectangles to indicate status, why not allow the application to do the same? To this end, each task's subrectangle is further divided into a number of squares. Each of these squares functions as an

[2] These are not implemented as X "button" widgets. Everything in Vulcan must scale up to 32K PEs, and such a number of widgets would strain any X implementation. The subrectangles are therefore simply rectangular regions in a "drawarea" widget. Using the minimal size of 2×2-pixel LEDs and 1-pixel boundaries, which is still large enough to be clicked on by the mouse, there is enough space for 64K LEDs on a small X terminal.



```
Task 1
About to try to pick up chopsticks.
About to do VSem_Lock(cs0).
Got first cs0 chopstick... now getting s
FINALLY GOT BOTH CHOPSTICKS.
Now eating for the 5 time!
Unlocking both Chopsticks.
Done Unlocking cs0.
Done Unlocking cs1.

Task 7
Done eating, have freed the chopsticks
About to try to pick up chopsticks.
About to do VSem_Lock(cs0).
Got first cs0 chopstick... now getting s
FINALLY GOT BOTH CHOPSTICKS.
Now eating for the 4 time!

Task 8
FINALLY GOT BOTH CHOPSTICKS.
Now eating for the 3 time!
Unlocking both Chopsticks.
Done Unlocking cs0.
Done Unlocking cs1.
Done eating, have freed the chopsticks
About to try to pick up chopsticks.
About to do VSem_Lock(cs0).

Task 14
Blocked getting cs1 chopstick... Unlock
Back from VSem_Unlock(cs0).
Woke up after sleep.
About to do VSem_Lock(cs0).
```

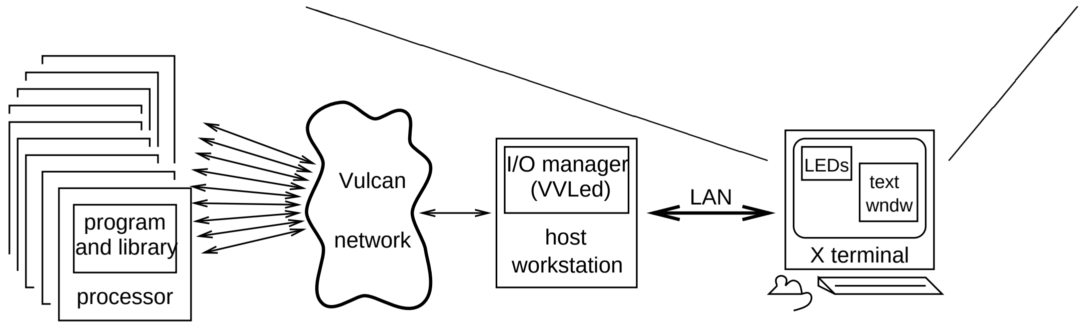


Figure 2: *Terminal I/O and the Vulcan architecture.*

independent LED. One of these LEDs is used for I/O status, as described above. The application program can set the colors of the other LEDs. For example, A particular color could indicate that the task has reached a certain point in the program. A whole spectrum of colors is provided, so LEDs changing through a sequence of colors can show an application's progress. This immediate visual feedback can be valuable in detecting endless loops and stuck situations, and providing a comforting indication of progress when the application is running properly. In many cases, this sort of output can replace voluminous text output in a highly efficient manner.

3 Implementation and Features

The terminal I/O facility is composed of two parts: a run-time library that is linked with each of the application tasks, and a separate manager, called "VVLed", which runs on the host workstation. The tasks communicate with VVLed using the message passing facilities provided by the Viper kernel, which is resident on all Vulcan nodes. VVLed communicates with the X server over a local area network using the X protocol (Fig. 2).

3.1 Program interface

The program interface to the terminal I/O library includes the following eight functions:

`VTerm_Put_Str( str )`
: Display the string in the task's window. If it does not have an open window, indicate that output is available by coloring the I/O status LED green.

`VTerm_Get_Str( buf, n )`
: Read input from the task's window into the buffer, but no more than `n` − 1 characters (the last place is kept for the terminating NULL character). If the task does not have an open window, show that it is waiting for input by coloring the I/O status LED red.

`VTerm_Open_Window()`
: Open the task's I/O window. This should be used with care, so that too many windows do not flood the screen.

`VTerm_Close_Window()`
: Close the task's I/O window. This should only be used when it is certain that the user does not need this window anymore, e.g. at the end of an input dialogue.

`VTerm_Led_On( led, color )`
: Color the specified LED in the specified color. A spectrum of 100 colors is provided, going from black through brown, green, blue, purple, red, orange, and yellow to white. In addition, there are three shades of gray. The program may also color the I/O status LED. This should only be done by programs that do not perform any text I/O.

`VTerm_Led_Off( led )`
: Turn the specified LED off, which means color it the default gray color.

`VTerm_Get_Led_Num()`
: Query the number of user-defined LEDs per task.

`VTerm_Set_Led_Num( num )`
: Set the number of user-defined LEDs per task. This causes the LED array window to resize according to the new number.

This interface provides only the most basic text I/O. However, the full capabilities of the normal formatted I/O functions are available by dividing the I/O process into two phases. For output, first format into a character array (e.g. using the `sprintf` function in C), and then output the resulting string. For input, first get a string, and then do a formatted read from it (e.g. using `sscanf`).

Alternatively, it is also possible to use the normal I/O functions (e.g. `printf` and `scanf`) directly, and not use the `VTerm_Get_Str` and `VTerm_Put_Str` functions at all. This is implemented as follows. The compilers we use are originally designed for an AIX environment. These compilers link the program's object module with runtime libraries that translate high-level language directives into low-level AIX system calls. In the case of formatted I/O, the C runtime library does all the formatting and buffering, and translates `printf`, `scanf` and the like into AIX `write` and `read` calls. The Fortran runtime library translates `reads` from unit 5 and `writes` to unit 6 into the same calls. Therefore all we have to do is catch the AIX `read` and `write` entry points.

Our versions of `read` and `write` are very simple. Their main function is to check the file descriptor that is given as a parameter. A file descriptor of 0 is standard input; the call is then turned into an input request form VVLed. A file descriptor of 1 is standard output; the call is then turned into an output message to VVLed. In all other cases, the normal function is preserved. For standard error, this results in messages from all tasks appearing in the host window. Note that we make an implicit assumption that the program does not modify its file descriptors with `dup`, `open`, or `close` calls. Of course, these calls could also be caught, and then we could follow any changes in the use of the standard file descriptors. This mechanism is used successfully to support formatted I/O in both C and Fortran. The main problem encountered is that in C `fflush(stdout)` has to be called

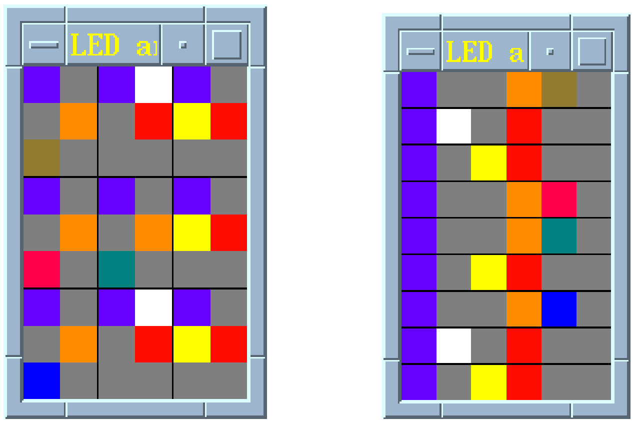


Figure 3: *A LED array for nine PEs, each with five user-defined LEDs. left: square mode, right: rows mode. The proportions of the two modes are not always the same as they are in this case.*

to cause the output to actually be sent.

3.2 VVLed and the X interface

VVLed buffers two asynchronous entities: the user and the program. It does so by polling both in turn. The user is polled by checking for any new X events. The program is polled by performing a non-blocking receive of any type of Viper message coming from any task.

When the X input focus is on the LED array window, the mouse can be used to open and close dedicated I/O windows to the different tasks. This is done by pressing the middle mouse button when the pointer is located on any LED representing the task. Pressing the right button pops up a small window identifying the task in question. Naturally, the normal X operations are also supported. The user may use the mouse to interact directly with the window manager and move, resize, or close the LED array and I/O windows.

In addition, the following commands which relate to the whole I/O system can be typed:

`m`
: Toggle the display mode. There are two display modes (Fig. 3): *square*, where the LEDs of each task are arranged in a squarish rectangle and the tasks themselves are also arranged in a squarish rectangle, and *rows*, where the LEDs of each task are arranged in a row and the tasks are arranged in a column, one above the other. If there is not enough space in the screen, more than one column may be used.

`b`
: Open a broadcast window. Text typed in this window will be sent to all the tasks. In addition, it is echoed in the private windows of all the tasks.

`c`
: Clear the text from all the windows and extinguish all the LEDs (i.e. color them gray).

`-` Close all open text windows. The text itself is saved and will be displayed again if a window is opened again (unless everything was cleared in the meantime).

`h` Display a help message.

The square display mode is better for massively parallel systems, where each task has a very small number of LEDs (e.g. only one). It allows the user to immediately pick out those tasks with abnormally colored LEDs, which indicate a problem condition. The rows mode is better for smaller machines, especially if each task has many different LEDs, because the same LEDs from all the tasks are aligned one above the other (Fig. 3 (b)). A possible third mode, which was not implemented, is to display “LED planes”. This is similar to square mode, except that the LEDs are grouped by their serial numbers rather than by the tasks to which they belong.

Some of the commands typed in the host window also affect the LED display. For example, a `reset` command causes all the text to be cleared and all the LEDs to be extinguished. A command to change the partition size resizes the LED array window accordingly.

3.3 Window placement

Opening a large number of windows to different tasks poses the question of how to handle screen real estate. One option is to leave this problem to the window manager. While this option is supported, it is not recommended: the window manager does not have enough information about the use of the different windows, so it cannot implement any sophisticated placement policy. The Motif window manager places each new window with a large overlap over the previous window, so effectively only the last window can be seen (Fig. 4).

The VVLed placement policy mimics a tiled window manager. The default window size is rather small, 400×250 pixels, so that some ten windows may be fitted into relatively small screens. Other sizes may be defined in the user’s .Xdefaults file. In any case, scroll bars are provided to scroll the text both vertically and horizontally. The LED array display is never occluded unless it fills the screen to such a large degree that there is no space for the text windows besides it. The text windows are placed side by side across the whole screen, until there is no more space. If additional windows are opened, they are placed on top of previous windows, but still do not occlude the LED array. If the LED array is moved or resized, this is taken into account when new windows are subsequently opened.

The default size of the text windows typically does not divide the available space evenly. This can be handled in either of two ways: keep all windows the same size and leave unused space around the edges, or create smaller windows if there is less space. VVLed implements both options, by allowing the user to define the minimal allowed window size in the .Xdefaults file. If the minimal size is the same as the default size, all windows will be the same. If it is smaller, then smaller windows will be created to better utilize the leftover space (Fig. 5).

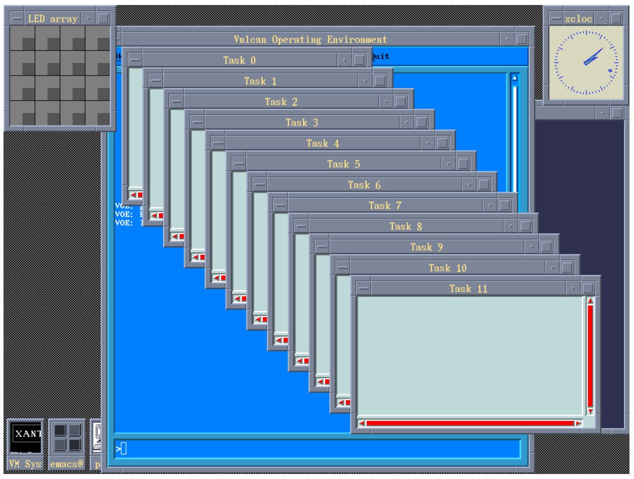


Figure 4: *Text window placement by the Motif window manager. The LED array has 4 LEDs for each of 16 tasks, in the square display mode. 3 of the LEDs are user-defined, and the fourth is used for I/O status.*

When the minimum is smaller than the normal, the system is allowed some freedom in choosing the window size. To control the window placement, the screen excluding the LED array is divided into four areas: above, to the left of, to the right of, and below the LED array. Only areas that are bigger than the minimal window size are considered. The algorithm for selecting a window size is as follows. In each dimension independently, check if there is enough place for at least *two* normal size windows. If so, use the normal size in this dimension. If not, check if there is enough space for two *minimal* size windows. If so, use half the space, and leave the other half for the next window that will be opened. If this failed too, check if there is enough space for one minimal window. If so, use the minimum between the available space and the normal window size. If not, then there is no more space in this area, so go to the next one.

This algorithm was chosen for two reasons. First, it tries to respect the user’s default size by creating normal size windows before resorting to smaller ones. Second, it allows the full screen to be utilized without leaving any unused space. To achieve this effect, the minimal size in each dimension should be one half of the normal size (the default minimal size is 200×125 pixels). Its main drawback is that the windows may have radically different sizes. An alternative algorithm is to divide the space in each dimension into one more than the number of normal size windows that fit into it. This results in all windows in the area having equal sizes.

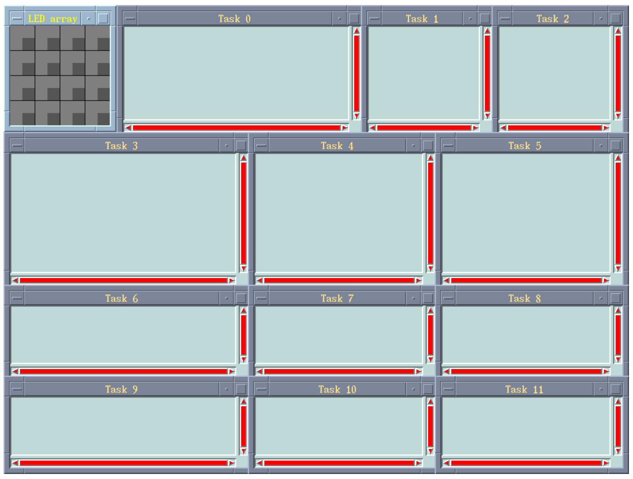


Figure 5: *By allowing windows that are as small as half the normal size in each dimension, the whole screen can be tiled.*

When a window is closed, its geometry is recorded and reused if it is subsequently opened again — users were uncomfortable with windows that come up in new locations each time they are opened. Note that by recording the geometry only when the window is closed, we take any movements and resizings performed with the mouse into account. A possible future improvement is to allow the whole setup to be recorded in a file, and reload it in subsequent sessions.

3.4 File redirection

While the VVLed facility is primarily intended to support I/O through independent windows to the different tasks, it also supports file redirection. Either standard input and/or standard output may be redirected to a file when a program is loaded. Normally, the I/O from all the tasks is interleaved into one stream in order of arrival. If the given file name ends with the suffix '`.T`', a separate file is used for each task, with the task number serving as a suffix. For standard input, the file name may also end with a '`.C`' suffix. This means that each task will receive a copy of the whole file. VVLed keeps track of the independent offsets belonging to the different tasks.

Files are only opened on demand, to reduce overhead and save space in system tables. If VVLed runs out of file descriptors, it closes all the open files and only opens them again if additional I/O operations are performed.

4 Discussion

Having described the Vulcan terminal I/O facility as implemented, we now turn to discuss its usefulness and how it can be extended to provide additional functionality.

4.1 Scrolling in time

The terminal I/O facility described here achieves a separation between the data streams from different tasks. However, this comes at the price of loosing the relative timing information that is implicit in the way these streams are interleaved when displayed in a single window. Such information may be just as important to users as the displayed text itself.

The source of the problem is that the windows representing different tasks are completely independent, and specifically, each can be scrolled independently of the others. The relative timing information can be retained by coupling the scrolling mechanisms of the different windows. Rather than having an independent vertical scroll bar for each window, we can have a single global vertical scroll bar. As each line of text is added to a window, it is tagged internally with a timestamp. The global scroll bar can then be used to scroll time backwards, always maintaining a synchronized view across all the windows. Similar mechanisms exist in several graphical display systems that show multiple views of program execution traces [8, 7].

The text windows preserve historical context in the form of previous text. It is possible to create two large windows, and compare the text that appeared in them across a certain time span. The LEDs display is inherently different from the text windows, in that it can only show a snapshot. There is no way to display how a certain LED changed colors over a time interval (this is analogous to limiting text windows to have only one line). However, the LEDs may also be linked with a global scrolling mechanism, such that when time is scrolled backwards the snapshot for that time is displayed.

While global scrolling adds relative timing information to the display, it should be noted that this timing information is not necessarily correct in absolute terms. It just reflects the sequence of events as observed by the terminal I/O facility. Nonuniform delays and message reordering in the network may cause I/O events to be displayed in a different sequence from the one in which they were generated at the tasks.

4.2 Scalability

The current design is scalable up to 32K tasks. With larger numbers, it would be impossible to represent each task by an independent LED. Luckily, users would not be able to assimilate the information in so many independent LEDs anyway. Actually, users only need to see those that are different from the general behavior. Therefore it is possible to scale to ever larger numbers by using a condensed representative panel, where each LED represents a large number of tasks, and its color is determined by the minority value rather than by the majority consensus. Thus all extraordinary tasks will show up. The user would then be able to zoom in on those tasks to investigate their status in greater detail.

Another aspect of scalability is the communication bandwidth and the processing power required by the terminal I/O facility. When operating over a network, it takes a couple of milliseconds to color a LED. Coloring each of 32K LEDs once takes an order of one minute. If 32K tasks were to generate such events at a rate of more than one per minute, the system would be flooded. Handling text I/O takes even longer, as it requires buffering and increases the chance of paging. Thus the supportable I/O rate per processor is inversely proportional to the system size, and might be quite low for large systems.

The problem of a limited I/O rate can be alleviated to some degree by a number of means. One is to postulate programming practices that limit the amount of I/O traffic. For example, a good practice is to use only the LEDs under normal conditions (to indicate execution progress), and resort to text I/O only in error handlers that are invoked when an exception condition is detected. Another is to implement a lazy display mechanism rather then an eager one. Under such a mechanism, output text would be buffered at the tasks rather than at the terminal facility. Whenever a window is opened by the user, the text for that window would be requested explicitly from the task. If no window is opened, the text would never be sent at all.

4.3 Other uses

The LED array is currently used only for terminal I/O. The same facility may be used to provide additional features. For example, additional system LEDs can be used to provide an indication of CPU utilization, cache miss ratio, or the ratio of system time to user time. Mouse control can be used for selective activation of tracing and for setting debugger breakpoints. Thus a uniform interface could be used for various instrumentation tools.

4.4 Experience so far

A number of parallel programs have used the Vulcan terminal I/O facility since its implementation. The following three case studies illuminate different aspects of the usefulness of this tool.

Dining philosophers test program

This program is a classical test case for parallel systems. In the Vulcan implementation, there is a philosopher task on each of the user's processors. Each philosopher picks up the left chopstick and then the right one; deadlock is avoided by releasing the left chopstick and doing an exponential backoff if the right one is not available.

The program includes extensive print statements which report events like chopstick acquisition, chopstick release, and backoff. Originally, all this output appeared in the host window with processor identification tags. When the stream of output stopped, users had to scroll back to ensure that all the processors had indeed terminated successfully, and that the output did not stop because of a deadlock situation.

With the new terminal I/O facility, the messages from different philosophers appear in different windows. Thus it is enough to open all the windows, and check that the last message in each one indicates successful termination. When LED status indication was added, it was no longer necessary to open the windows at all: LEDs changing color between yellow (thinking), orange (has one chopstick), and red (eating), and finally turning black (terminated), provide all the information. The screen dump shown in Fig. 2 is from a run of this program with 16 philosophers, where 4 have open windows.

Molecular dynamics application

This is a real scientific application [6] ported to the Vulcan environment, and executed on a set of interconnected RS/6000 nodes. The LED array was used to display progress according to the following color coding:

Initialization	green
Read input data	blue
Compute atomic interactions	yellow
Combine and broadcast partial results	red
Concatenate and broadcast partial results	orange
Perform graphical output (from node 0)	purple

By observing the color changes as the program ran, it was immediately obvious that computation and communication were unbalanced: the LEDs were red and orange much longer than they were yellow. This indicated that the implementation of collective communication was inefficient, and would have to be improved in order to achieve the desired speedup. A new implementation was indeed undertaken, and it performed well.

Implementation of the Vesta parallel file system

The design and implementation of the Vesta parallel file system also started as part of the Vulcan project [1]. Files and metadata in this file system are distributed across all the Vulcan I/O nodes. User applications access the files by calling library functions. These functions send messages to the appropriate I/O nodes, where they are handled by the file system code.

The Terminal I/O facility was used extensively during the early development and debugging of Vesta. During testing, the test program is loaded onto one node, and the file system code is loaded onto a few other nodes in the same user partition (rather then being loaded onto I/O nodes, which are not designed to have user interaction, and therefore do not provide terminal I/O facilities). The test program is interactive and menu-driven. It uses the I/O window to receive instructions from the tester and display responses from the file system. These responses include both data and error messages.

Each task uses a LED to represent its message-passing activity: the LED is turned yellow when sending a message, and orange when receiving (or waiting for the message to arrive). The LED is off (gray) when the task is busy. Initially, all the tasks representing I/O nodes are orange, as they await incoming instructions, and the test program is off, as it awaits user input. Once the user enters a command, the test program sends it (yellow) to the appropriate I/O node task. The I/O node task receives the command and works on it (gray); sometime later it sends a response (yellow) and then waits for the next command (orange). Some commands require the test program to communicate with more than one I/O node. by watching the pattern in which the LEDs change colors, the tester can see if the system is behaving as expected. A situation where all nodes are orange indicates deadlock with all nodes waiting for messages.

One of the available commands in the test program menu is to dump the metadata for inspection. To implement this, the test program sends a "dump" message to all the I/O nodes. Each I/O node then uses its I/O window to dump its metadata. This allows the tester to conveniently inspect the metadata on each node independently, or to compare metadata on different nodes.

5 Conclusions

As massively parallel MIMD machines become more commonplace, there is increasing need for facilities that would allow users to interact with these machines effectively. One such facility is that for terminal I/O. Our design allows the user to open an independent window to each task (or processor) in the system. In addition, we support LED output which provides easily comprehensible status information about the whole machine without using text. The programming interface is trivial and does not require the programmer to write any code dealing with windows or graphics. A similar facility would be useful in any multicomputer, not just the IBM SP1.

This design has been implemented as part of the Vulcan project. Both C and Fortran environments are supported, including their various flavors of formatted I/O. The user interface is based on the X window system and the Motif widget set. In some cases this proved to be somewhat problematic, as the Motif window manager was not always happy to give up its control of the user's screen. Indeed, the majority of the code written deals with the X interface. As windowing system continue to develop and higher-level interfaces are introduced, the implementation of such I/O facilities is expected to become much simpler. This will encourage the addition of more features and possible integration with various monitoring and debugging tools.

Acknowledgements

The idea of using mouse operations on a control panel to control the opening of dedicated windows is due to Blake Fitch and Mark Giampapa. The support for formatted I/O through catching the AIX `read` and `write` calls was designed and implemented with the help of Mark Giampapa. The rows display mode was suggested by Dror Zernik. Miron Livny pointed out the time-related difference between text windows and LEDs.

References

[1] P. F. Corbett, S. J. Baylor, and D. G. Feitelson, "*Overview of the Vesta parallel file system*". In *Proc. IPPS '93 Workshop on I/O in Parallel Computer Systems*, pp. 1–16, Apr 1993. (Reprinted in *Comput. Arch. News* **21(5)**, pp. 7–14, Dec 1993).

[2] B. G. Fitch and M. E. Giampapa, "*The Vulcan Operating Environment: a brief overview and status report*". In 5th *Workshop on Use of Parallel Processors in Meteorology*, European Centre for Medium-Range Weather Forecasts, Nov 1992.

[3] R. R. Glenn and D. V. Pryor, "*Instrumentation for a massively parallel MIMD application*". *J. Parallel & Distributed Comput.* **12(3)**, pp. 223–236, Jul 1991.

[4] W. D. Hillis, "*The connection machine*". *Scientific American* **256(6)**, pp. 86–93, Jun 1987.

[5] IBM Corp., *IBM AIX Parallel Environment: Operation and Use*. Sep 1993. Order number SH26-7230-0.

[6] J. F. Janak and P. C. Pattnaik, "*Protein calculations on parallel processors. I: parallel algorithm for the potential energy*". *J. Comput. Chemistry* **13(4)**, pp. 533–538, 1992.

[7] C. Kilpatrick and K. Schwan, "*ChaosMON — application-specific monitoring and display of performance information for parallel and distributed systems*". In *Proc. ACM/ONR Workshop on Parallel & Distributed Debugging*, pp. 57–67, May 1991.

[8] D. N. Kimelman and T. A. Ngo, "*The RP3 program visualization environment*". *IBM J. Res. Dev.* **35(5/6)**, pp. 635–651, Sep/Nov 1991.

[9] J. E. Lumpp, Jr., S. A. Fineberg, W. G. Nation, T. L. Casavant, E. C. Bronson, H. J. Siegel, P. H. Pero, T. Schwederski, and D. C. Marinescu, "*CAPS: a coding aid for PASM*". *Comm. ACM* **34(11)**, pp. 104–117, Nov 1991.

[10] W. Myers, "*Faster... Caltech dedicates world's most powerful supercomputer*". *Computer* **24(7)**, pp. 96–97, Jul 1991.

[11] ParaSoft Corp., *Express C Reference Guide*. 1990.

[12] D. G. Shea, W. W. Wilcke, R. C. Booth, D. H. Brown, Z. D. Christidis, M. E. Giampapa, G. B. Irwin, T. T. Murakami, V. K. Naik, F. T. Tong, P. R. Varker, and

D. J. Zukowski, “*The IBM Victor V256 partitionable multiprocessor*”. *IBM J. Res. Dev.* **35(5/6)**, pp. 573–590, Sep/Nov 1991.

[13] C. B. Stunkel, D. G. Shea, B. Abali, M. M. Denneau, P. H. Hochschild, D. J. Joseph, B. J. Nathanson, M. Tsao, and P. R. Varker, “*Architecture and implementation of Vulcan*”. In *Intl. Parallel Processing Symp.*, Apr 1994.