

MESSAGE QUEUE MANAGER

by

Don Y. Fang

A Project Report

submitted to

Department of Computer Science

Oregon State University

in partial fulfillment of the
requirements for the degree of

Master of Science

Major Professor: Dr. Cherri M. Pancake

Completed August 1996

ABSTRACT

The conventional way of debugging is to examine the program's state at various points during execution. With the addition of message passing as a means of communication and synchronization in message passing programs, the state of pending message operations also needs to be examined. Current parallel debuggers or analysis tools offer little information on the status of message operations. When they do, the information presented is voluminous and hard to assimilate.

The *Message Queue Manager* (MQM), a Parallel Tools Consortium project, provides a simple and direct way to determine the status of pending message operations during program execution. It has been designed and developed as a modular component that can be incorporated into a parallel debugger or other analysis tools on a variety of hardware platforms. MQM obtains a snapshot of message queues from the underlying message passing system (using a standard interface at some breakpoint or other pause in execution) and presents the data graphically. Two levels of information are available -- an overview gives an abstraction of message queue lengths, while a detailed view shows pending message operations for specific processes. The overview can display a large number of processes in a reasonably small screen, accommodating massively parallel programs. Several filtering mechanisms allow the user easily control how much information is shown.

Table of Contents

1. INTRODUCTION	1
1.1 Overview of Message Passing.....	1
1.2 Problems in Message Passing.....	4
1.3 The Message Queue Manager.....	8
1.4 Organization of The Report.....	10
2. RELATED WORK	11
2.1 Trace-based Tools.....	11
2.1.1 <i>ParaGraph</i>	12
2.1.2 <i>etool</i>	16
2.1.3 <i>Upshot</i>	17
2.1.4 <i>VISPAT</i>	19
2.1.5 <i>AIMS / CXtrace</i>	20
2.1.6 <i>Xab</i>	22
2.1.7 <i>Radar</i>	23
2.1.8 <i>Belvedere and Ariadne</i>	24
2.1.9 <i>Pablo</i>	25
2.2 Interactive (Break-Point Style) Tools.....	26
2.2.1 <i>ndb</i>	27
2.2.2 <i>IPD / XIPD</i>	28
2.3 Summary.....	29
3. SAMPLE USER'S SESSIONS WITH MQM	31
3.1 The Example Program.....	31
3.2 When Execution Seems to "Hang".....	34
3.3 Improving Program Efficiency.....	39
4. MQM USER'S MANUAL	42
4.1 Invocation.....	42
4.2 Overview Display.....	42
4.2.1 <i>Main Viewing Area</i>	43
4.2.2 <i>Menu Bar</i>	43
4.2.3 <i>Queue Display Options</i>	44
4.2.4 <i>Summary of Message Filters and Message Area</i>	45
4.2.5 <i>Window Control Buttons</i>	45
4.3 Process-level Display.....	45
4.4 Change Filters Dialog.....	46
4.4.1 <i>Message Source and Destination Filters</i>	47
4.4.2 <i>Message Type Filter</i>	48
4.5 Process Subset Display.....	48
4.5.1 <i>Selecting A Subset of Processes</i>	48
4.5.2 <i>Process Subset Display</i>	50
4.6 Summary of Tool Functions.....	50
5. IMPLEMENTATION OF MQM	52
5.1 Structure of the GUI.....	53
5.2 Structure of the Query Manager.....	55
5.3 File Structure of MQM.....	58
5.4 Design and Implementation Decisions.....	59
6. CONCLUSIONS	63

6.1 Future Work.....64

BIBLIOGRAPHY.....66

Appendix A - CONCEPTUAL MESSAGE OPERATIONS70

Appendix B - STANDARD QUERY FORMAT.....73

List of Figures

Figure 1.1 User-based message passing taxonomy	3
Figure 1.2 (a) Example message passing program, and (b),(c) its two possible executions	5
Figure 1.3 Message passing execution deadlocked.....	6
Figure 1.4 Message path of send and receive operations	7
Figure 2.1 ParaGraph Communication Traffic display.....	13
Figure 2.2 ParaGraph Spacetime Diagram	13
Figure 2.3 ParaGraph Message Queues display	14
Figure 2.4 ParaGraph Communication Matrix display.....	14
Figure 2.5 ParaGraph Animation display	14
Figure 2.6 ParaGraph Node Data display.....	15
Figure 2.7 etool's graphical user interface	17
Figure 2.8 Upshot.....	18
Figure 2.9 VISPAT Navigation display	20
Figure 2.10 VISPAT Animation display.....	20
Figure 2.11 AIMS Overview display	21
Figure 2.12 AIMS Boxes displays	22
Figure 2.13 Xab's display.....	23
Figure 2.14 Radar's view of a message being sent	24
Figure 2.15 Belvedere's snapshot of message passing.....	24
Figure 2.16 Visualizations of message passing constructed using Pablo.....	26
Figure 2.17 ndb's "show queues" command	27
Figure 2.18. XIPD traceback window	28
Figure 2.19. XIPD pending sends dialog	29
Figure 2.20. XIPD pending receives dialog	29
Figure 3.1 (a) SOR master program, (b) SOR worker program	32
Figure 3.2 Data sharing among the processes of the example program	33
Figure 3.3 Order of computation of the example program.....	34
Figure 3.4 MQM Overview upon invocation.....	35
Figure 3.5 Overview with more message queues selected.....	36
Figure 3.6 Process-level Display of process 2:0.....	37
Figure 3.7 The filter dialog	38
Figure 3.8 Overview with filters set.....	38
Figure 3.9 MQM Overview with new snapshot.....	39
Figure 3.10 Process-level Display of process 2:0.....	40
Figure 4.1 Components of Overview display.....	42
Figure 4.2 File menu	43
Figure 4.3 Window menu.....	43
Figure 4.4 Help menu.....	43
Figure 4.5 Components of Process-level Display	46
Figure 4.6 Change Filters Dialog	47
Figure 4.7 MQM in process subset selection mode	49
Figure 4.8 Selecting a subset of processes.....	49
Figure 4.9 Components of Process Subset Display	50
Figure 5.1 Structure of MQM.....	52
Figure 5.2 Structure of the GUI.....	54
Figure 5.3 Data structure for message queue contents.....	56
Figure 5.4 Data structure containing message queue lengths	56
Figure 5.5 Structure of the Query Manager.....	57
Figure 5.6 File structure of MQM.....	58

List of Tables

Table 4.1 Summary of Queue Display Options 44

1. INTRODUCTION

The Message Queue Manager (MQM) is an X Window System-based tool for debugging message and fine tuning the performance of message passing programs. It was designed as a component for incorporation into parallel debuggers, performance tools, and message passing systems on various hardware platforms. MQM is a Parallel Tools Consortium project [3]. Oregon State University's participation was sponsored by Lawrence Livermore National Laboratory and Intel Corporation.

1.1 Overview of Message Passing

A parallel program is a collection of processes or programs that cooperate with each other to satisfy a given specification [4]. In order for these processes to work cooperatively toward some final result, they must interact with one another. Process interaction takes place, for instance, when data from one process has to be shared with others, or when the progress of one process is dependent on the progress of another, in which case some sort of synchronization between them is required. Interaction among processes adds a new dimension to computer programming, and is a common property of parallel programs [19]. As a result, the parallel programmer has something extra to worry about namely, the interaction and relative timing of individual processes [30].

There are many ways to develop parallel programs, depending on what types of parallel operations are available. The conceptual view of the operations available is called a programming model [8]. A programming model does not include the specific syntax of a particular language or library, and is independent of the underlying hardware that supports it. Message passing, data parallelism, shared memory, and remote memory operations are all examples of parallel programming models.

In the message passing model, all process interactions are done by passing "messages" from one process to another. Each of the cooperating processes has only local memory, and does computation on its own using its local data [8,19]. When data needs to be transferred from the local memory of one process to the local memory of another, it is first formulated as a message. Processes communicate with each other

by sending and receiving these messages. Different types of message operations are provided so that processes can interact in various ways.

Message passing, like any other parallel programming model, can be implemented on any parallel computer, but it fits well on multicomputer architectures, where separate processors attached to their local memories are connected by some communication network. Since this multicomputer architecture encompasses most of today's parallel supercomputers and workstation clusters, message passing has become a widely used programming model [8].

Current message passing systems fall into two basic categories: proprietary and third-party systems. Proprietary message passing systems are typically implemented on just one manufacturer's machines, while third-party systems are developed by independent groups and usually ported to a number of machine types. Although each message passing system has its own set of specific operations, we found that message operations can generally be categorized into several conceptual operations (see appendix A). We summarize these conceptual operations in Figure 1.1.

The most basic message passing operations are *point-to-point communications*. This type of communication is initiated by either a source process executing a send operation, or a destination process posting a receive operation. Both operations are required to complete the communication. The operations are called point-to-point because messages are passed between just two processes.

Both send and receive operations can be categorized as blocking or non-blocking operations [24,29]. In a blocking operation, the process waits, or blocks, until the operation completes. In a non-blocking operation, the process continues execution after the operation is initiated, without waiting for it to complete; a separate call is then required to determine when the non-blocking operation has actually completed. Non-blocking operations can improve performance by making it possible to overlap computation with the completion of message operations. For example, a non-blocking receive allows some alternate computation to be performed by the receiver if the message has not yet arrived.

All send operations can further be categorized as acknowledged or unacknowledged [24,29]. An acknowledged send completes only when the sender is notified of the receiver's acceptance of the message. In comparison, the completion of an unacknowledged send indicates that the message has been

released to the message passing system and the sender's buffer can be reused, but it does not say anything about the status of the receiver process. A send and a receive can also be combined, so that the two processes involved can exchange data with a single operation.

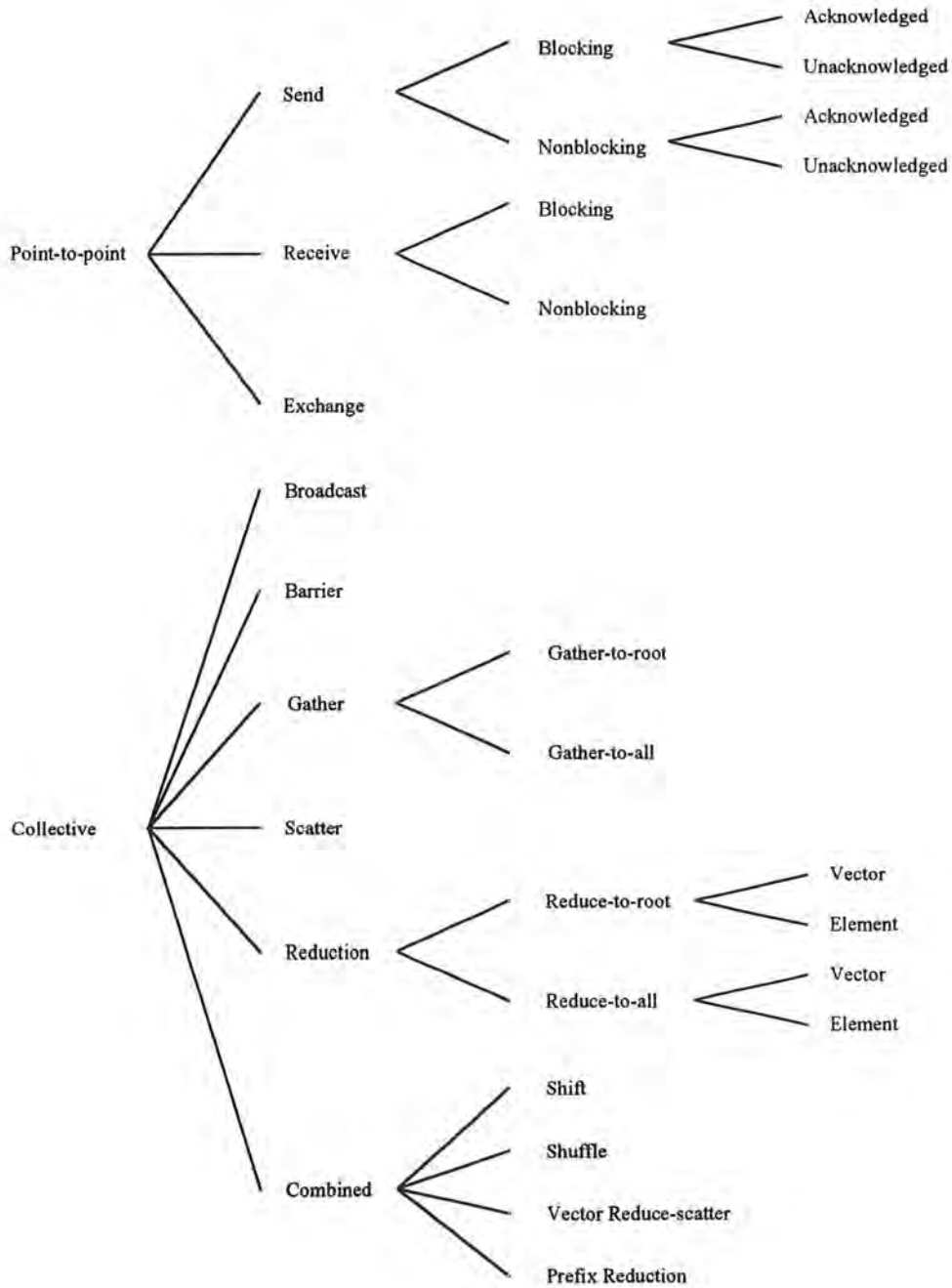


Figure 1.1 User-based message passing taxonomy

In contrast to point-to-point operations, *collective operations* involve a group of processes. Although these may be simulated by multiple point-to-point message operations, the collective operations are more convenient to use and usually are good targets for optimization on a particular architecture [24]. The distinction between blocking and non-blocking operations also applies to collective operations, even though most message passing systems provide only blocking versions.

Broadcasting allows a process to send the same message to a set of processes or all processes in one step. A barrier prevents processes from proceeding further until all processes have reached that point in execution. A gather operation collects local data from each process and stores them in collected form. In gather-to-root operations, only one process receives the resulting collection, while in gather-to-all operations, all processes get copies. A scatter operation is the inverse of a gather-to-root operation. The root process subdivides the data, and sends a distinct subset to each process in the group.

Reduce operations apply some global mathematical function such as summation, multiplication, bitwise operations, or some user-defined function across data local to each process. As in gathering, the result of a reduction can be sent to just the root process (reduce-to-root), or to every process in the group (reduce-to-all). Reduce operations can be performed on vectors, as well as single data elements.

Finally, some collective operations have the effect of combining other operations for programming convenience. In a so-called shift operation, each process sends a message to another process while receiving a message from a third process. This shifts a set of data along a chain of processes. In a shuffle operation, each process receives data from each of the other processes (process i receives the i -th data element sent from process j , placing it in the j -th block of some local array). A vector-reduce-scatter operation is equivalent to performing a reduce operation on vectors, then scattering the result to all processes. In prefix reduction, each process ends up with the result of performing a global mathematical function across all processes that are lower in rank.

1.2 Problems in Message Passing

The concept of message passing sounds simple, but it is as complex as any other parallel programming model because of the extra “dimension” of process interaction. Although its advantages

make it popular, message passing also has significant shortcomings. The most obvious is that the programmer is responsible for many low-level programming details to ensure the correctness of process interaction [29]. Consider, for example, the four types of send operations. For every message to be sent, the programmer must choose a send operation appropriate to the situation; a matching receive must also be performed, by a different process and perhaps in an entire different source program. Message passing programming is prone to errors, since many things can go wrong if the programmer is not careful about placing message passing routine calls. Two classic examples are race conditions and deadlocks.

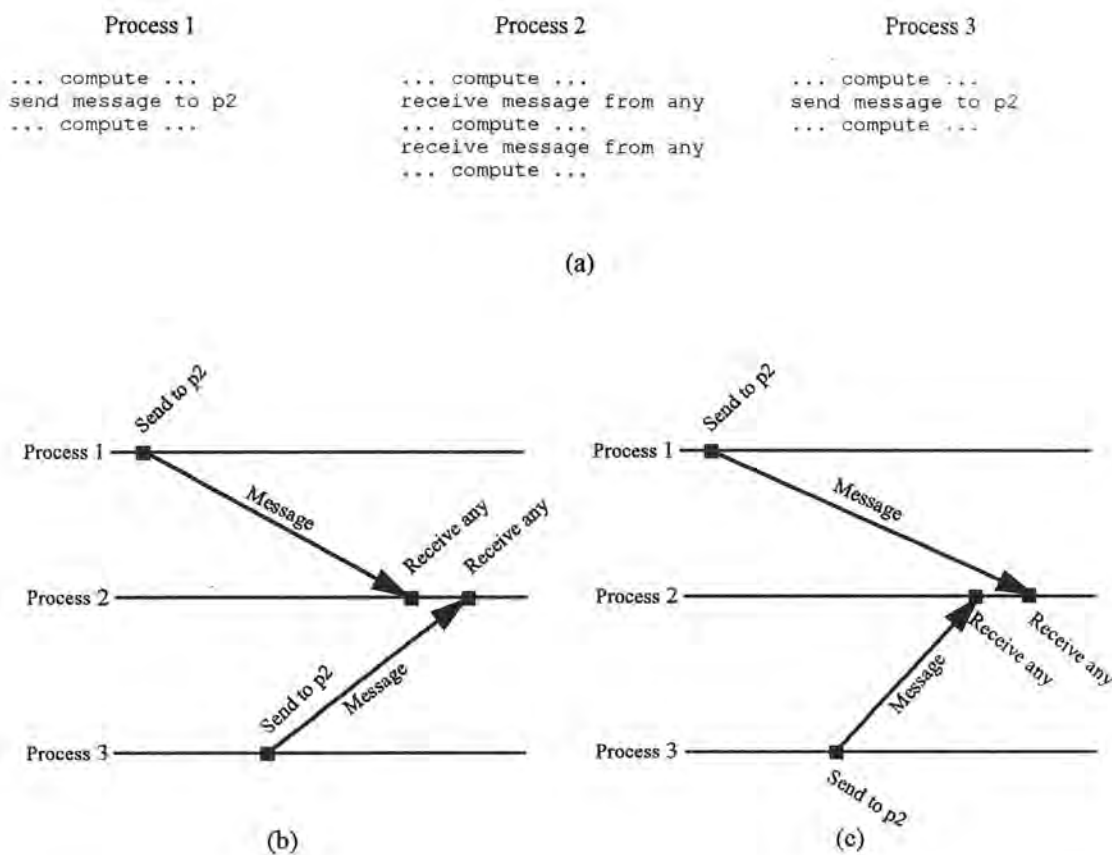


Figure 1.2 (a) Example message passing program, and (b),(c) its two possible executions

Messages race when some receive operation could accept any of several sends [26]. For example, in Figure 1.2(a), process 2 will receive two messages, while both process 1 and process 3 will each send one. Either message could arrive at process 2 first and be accepted by the first receive operation as illustrated in Figure 1.2(b) and (c). The order of message arrival can be uncertain, due to variations in

message latency, process scheduling, and network connection speed. Although, in some cases, a message race condition need not result in problems, it could cause the program to generate incorrect results, or even crash, if a certain order of message arrival is required. The indeterminacy in this example can be avoided if the programmer specifies explicitly the sources and/or types of the messages to be received.

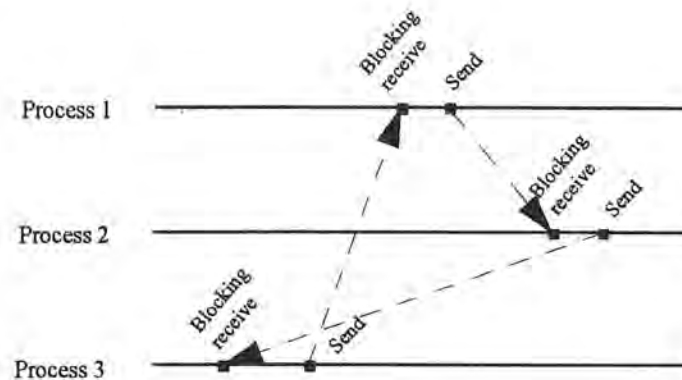


Figure 1.3 Message passing execution deadlocked

A program execution becomes deadlocked when processes wait for messages that will never arrive.

Figure 1.3 shows a trivial example. Each process is waiting to receive a message from another process, but none of the receives can complete since the messages that would satisfy them will not be sent until after the receives have completed. One way to avoid this situation is to use non-blocking receives; but the most obvious solution is to make sure that data is sent before posting a related receive.

Careless or incorrect use of message routines not only risks some serious problems, but also can cause a message passing program to be very inefficient [19]. For example, blocking operations and barriers can be used to synchronize processes, making sure that data is available before the computation that requires it; however, these operations force some processes to wait for others. Program performance can be severely degraded if the wait is prolonged. The problem for the programmer is how to order message passing and computation routines so as to minimize the wait time without altering data dependencies. Frequent messages also result in performance degradation because of the high overhead of message passing (e.g., message startup cost, data buffering at message passing system level). For this

reason, it is not efficient to send frequent small pieces of data; instead, such data should be combined and sent together. Of course, when the data to be sent affects the execution of other processes, there is a tradeoff between reducing message overhead and minimizing process wait time.

When programming errors are suspected, the program must be debugged. The classic approach in debugging serial programs is to examine the program's state (e.g., data values, location of instruction counter) at various points during execution [21]. However, in a message passing environment, the status of message-passing operations also forms a part of program state and must also be examined.

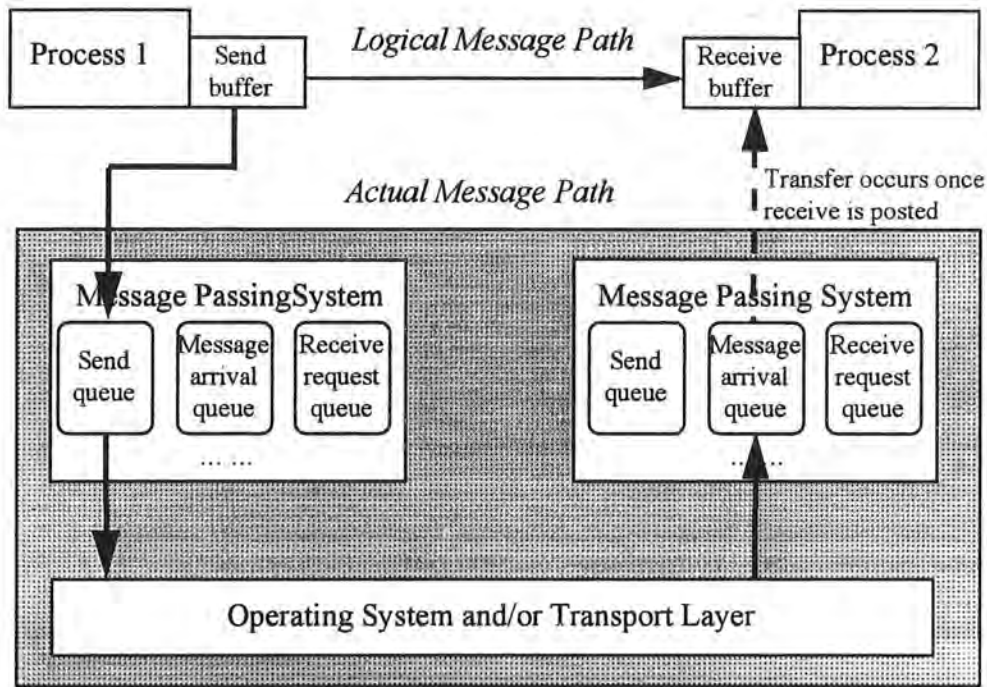


Figure 1.4 Message path of send and receive operations

The fact that a message has left the sender's buffer does not mean that the receiver has actually gotten it (see Figure 1.4). The message must first be queued into the message passing system and transferred across the network. Once it arrives at the destination processor, if the matching receive has not been posted, it will be queued again in the message arrival queue. The message will be copied into the user-level receive buffer only after a matching receive has been posted. On the other hand, if a receive is posted before the satisfying message arrives, the receive request will be queued instead; now when the

message arrives, it can be copied into the receive buffer without being queued first. (The number and type of buffers and queues actually varies from one system to another.) The procedure may fail at any of these steps if the communication is incorrect, or it may remain incomplete. However, the program is not able to detect whether a message has arrived unless it is actually received, since it is restricted to "logical message path" shown in the figure. If an error occurs, there is just not enough information for the programmer to easily locate the problem. Without being able to observe more about the "actual message path", debugging message-passing programs is very difficult.

1.3 The Message Queue Manager

MQM provides a straightforward way for the user to determine the status of message operations. MQM takes a snapshot of current message queues upon the user's request, usually during the debugging process. The contents of all message queues are displayed, allowing the inspection of pending message operations. By providing detailed information on the message queues, MQM makes it possible for the user to peek into the "actual message path" when messages are being passed. MQM collects all necessary data automatically, allowing the user to concentrate on finding problems.

Graphical representation has been demonstrated to be an efficient way to convey large quantities of information [17,31]. MQM's graphical data representation also has the advantage of scalability; it scales well with the number of processes involved in large message passing programs. With its hierarchical displays, MQM can easily represent messages in programs involving hundreds or even a few thousands of processes, within a reasonable amount of screen "real estate". It also allows the user to view only a subset of processes. Its scalability relieves the user from constant window scrolling in order to locate a small piece of data among the potentially huge amount of information associated with hundreds of processes. Popup windows help the user compare message queues from logically distant processes.

Filtering mechanisms allow the user to screen voluminous information. The scope of displayed data can be restricted to certain operations, processes, sources/destinations, user-defined message types, or any combination of these attributes. Multiple combinations can be displayed simultaneously in different windows for better observation of related information with finer details.

MQM obtains the message queue data from a parallel debugger, performance monitor, or the message passing system. It was designed to work best with a breakpoint style tool that provides comprehensive program execution controls. With this support, the program can be paused upon the user's request or at user-defined breakpoints. Then, the user can examine the current state of the message queues, just like a debugger supports examination of other program state information. The problems associated with techniques for debugging messages using trace monitoring (see Section 2) are avoided.

MQM can also be used as a performance tuning tool. Although MQM does not offer support that is as comprehensive as some trace-based tools (see Section 2), it does provide essential information for fine-tuning message passing operations. The same detailed information on the status of message operations that helps in debugging also helps tune the performance of these operations. This is specially true for so-called tuning *in the small* (i.e., the detection, diagnosis, and remedying of a specific instance of inefficiency).

MQM must be incorporated into a run-time environment (parallel debugger, performance monitor, or simply the message passing system). Since it only communicates with that software, MQM is essentially independent of the underlying hardware platform. It is also largely independent of the message passing system, although it does rely on the availability of status data from at least one of the queues (sends, message arrivals, or receive requests). MQM has already been incorporated into Intel's Interactive Parallel Debugger (IPD), which works with Intel's NX message passing system on the Intel Paragon and iPSC machines. The integration process proved to be smooth and took only a couple of days. If a message passing system does not support certain message queues, displaying of these queues can be easily disabled by clearing a couple of flags in MQM's header file.

The functionality of MQM and the appearance of the tool were established by a process of user-centered design [27] carried out by Dr. Pancake. Users were involved early in the design stage and throughout the entire development process, with a working group was formed to obtain user input on design issues. The design of the tool went through many iterations. We first started with a "paper prototype" to establish what functionality should be included. This was refined so users could identify specific features desired for the interface. The users' involvement continues through the stages of

implementation and testing to ensure that MQM met their needs and preferences. Because MQM meets users' needs and preferences, it has been included in Guidelines for Writing System Software and Tools Requirement for Parallel and Clustered Computers [28], a list of software needs compiled by a national body of users.

1.4 Organization of The Report

This section provided some background information and an introduction to MQM. Section 2 reviews related tools and research, examining features to help in debugging message passing operations. Section 3 takes the reader through a sample user session to demonstrate the features of MQM. A user's manual for the tool is presented in Section 4. Section 5 discusses the major design decisions in MQM's evolution, and describes its implementation. The report concludes with Section 6, which assesses the tool's limitation and suggests future enhancements.

2. RELATED WORK

We studied a number of existing parallel tools to see what kind of support is available for debugging and tuning the performance of message passing. We found that some of the tools may be used to help determining the status of messages, but that support is inadequate in all cases. In the subsections below, we look at trace-based and interactive tools' visualization methods.

2.1 Trace-based Tools

Program debugging with a trace-based tool typically includes three stages: instrumentation, monitoring, and post-mortem analysis. Instrumentation statements must be inserted into the source code by the programmer, the compiler, or the tool. During program execution, instrumentation routines monitor the state of the program, and record run-time data in an execution history, or trace. The trace information can then be used by post-mortem tools to analyze the behavior of program execution in many different ways.

In spite of their unique approaches to trace analysis, all trace-based tools come with some common technical difficulties. Source code instrumentation requires extra steps by the user. Even when instrumentation statements are inserted automatically, the user is required to choose what files and constructs should be monitored. Furthermore, the insertion of instrumentation statements alters sensitive timing sequences, which may decrease the chance of repeating errors, or new errors may be introduced [35].

With trace-based tools, the trace can be played back many times after execution, but the user cannot interact with or control the executing program. If the user wants to analyze a different program run, or to record a different type of data, re-instrumentation and re-execution are required. Also, a large storage space is necessary for trace information, since the output volume increases rapidly with the number of processes and the frequency of taking snapshots [35]. CPU time and memory traffic increase as well due to the overhead of monitoring, and may further interfere with the regular execution sequencing [35].

While most trace-based tools include all three components (instrumentation, monitoring, and post-mortem analysis), the post-mortem component can be independent, relying on traces generated by other instrumentation and monitoring tools. The most common post-mortem analyses are static displays, animations, and statistical summaries [17]. A static display presents a graphical overview of execution. In this type of display, time appears on one axis, and some aspect of the execution occupies the other. A typical example is the time-process diagram in which processes fill one axis, so that the temporal relationship among executing processes is presented. In contrast, an animation displays a sequence of snapshots as execution advances. A statistical summary is a tabular or graphical summary of performance, usually showing minimum, maximum, and mean values for the program as a whole.

Each of the following subsections describes a trace-based tool, discussing the post-mortem analysis features that could help the user understanding a message passing program.

2.1.1 ParaGraph [9]

ParaGraph is one of the earliest and best known performance visualization tools for message passing programs. Although it could be employed for real time visualization, ParaGraph has been exclusively used in post-mortem fashion. Various types of post-mortem visualization are offered simultaneously; each gives a unique visual perspective of the program performance. A global control panel controls the replay of program execution. Intel has adapted ParaGraph into their Paragon Performance Monitoring Environment because of ParaGraph's widespread use [34].

ParaGraph provides multiple displays in four categories: CPU utilization, communication, user-defined tasks, and miscellaneous. As the name suggests, only the communication displays deal with message passing. Note that all the displays in ParaGraph deal with nodes, or physical processors, instead of logical processes.

The Communication Traffic display (see Figure 2.1) is a static display that plots the count or the volume of all pending messages during execution. Total traffic can refer to the accumulation of messages for all processors, or for just a single processor.

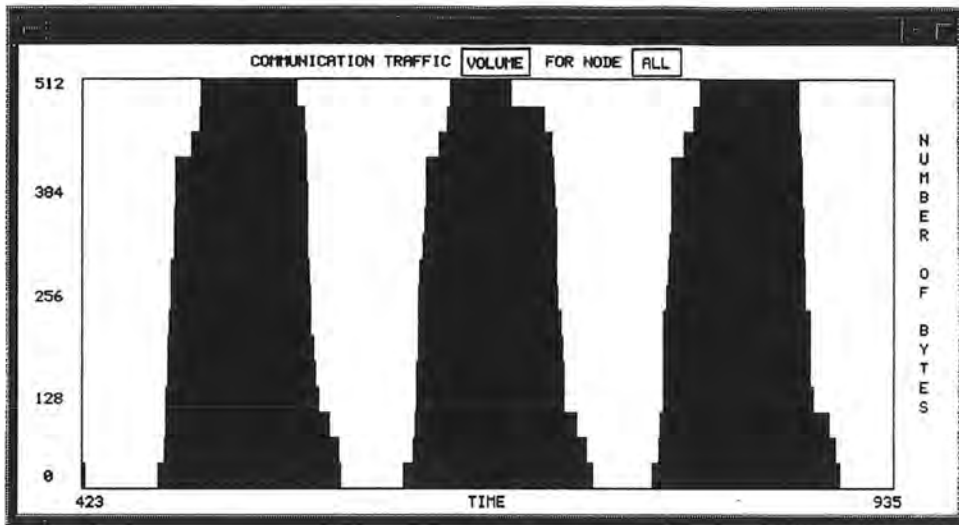


Figure 2.1 ParaGraph Communication Traffic display

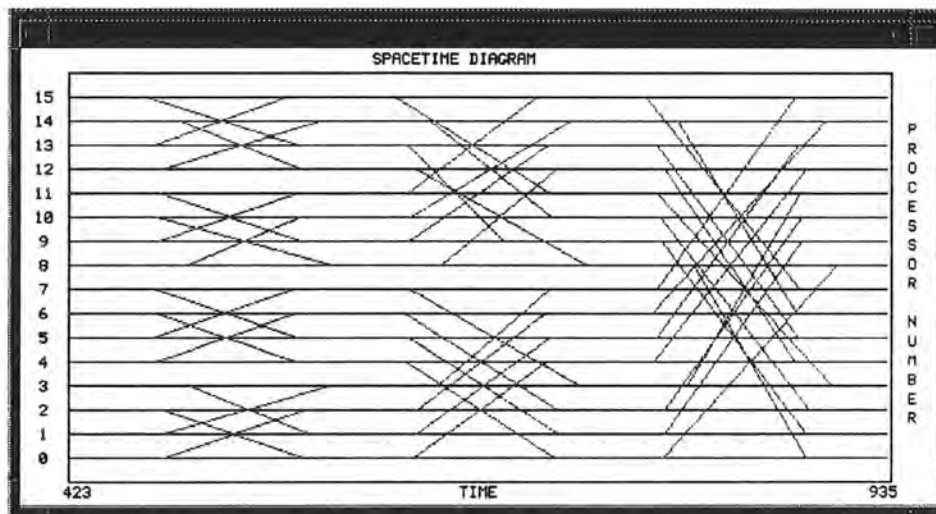


Figure 2.2 ParaGraph Spacetime Diagram

The Spacetime Diagram is also a static display, comparing processor activity (busy or idle) with communication among processors during execution (see Figure 2.2). Process activity is indicated by horizontal lines, one for each processor. A solid line means the corresponding processor is busy; a blank line means it is idle. Messages between processors are depicted by slanted lines between the sending and receiving processors' activity lines. The message lines are color coded to indicate message type. However, no message line is drawn until it has been received, which means the user cannot observe the send event if execution halted before the message was received.

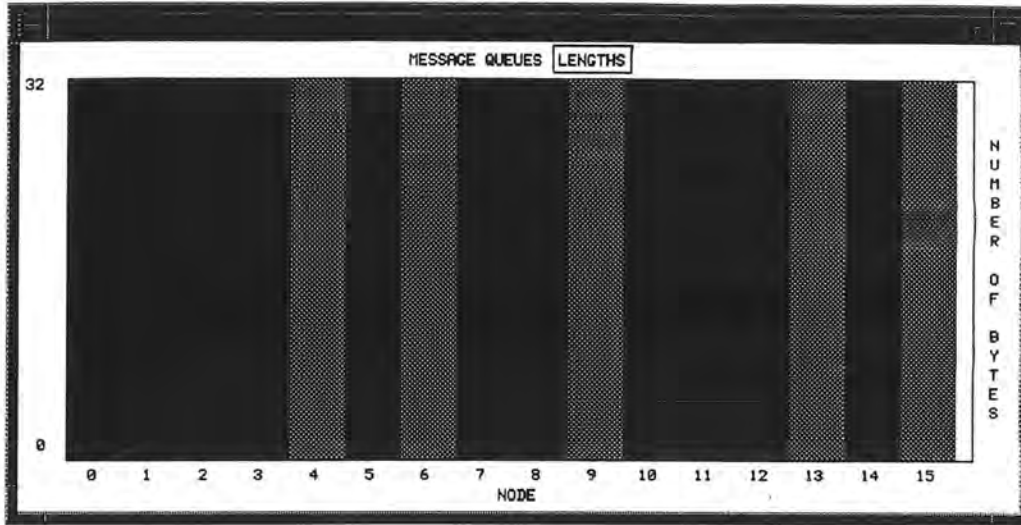


Figure 2.3 ParaGraph Message Queues display

The Message Queues display animates the number, or the total length in bytes, of all pending messages (see Figure 2.3). The height of each vertical bar represents the current count, lengths of any messages waiting in the corresponding processor's queue of arrived messages. The heights vary with time, as the queues change.

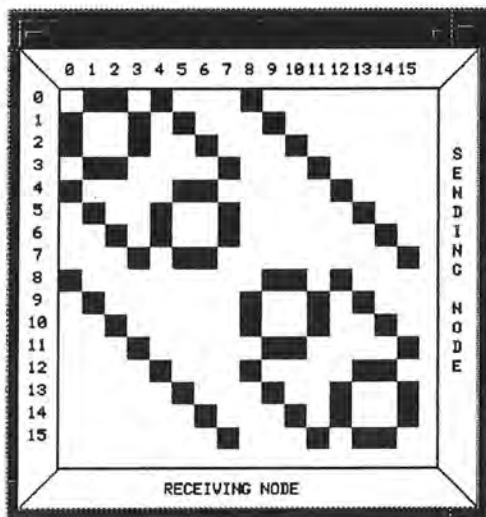


Figure 2.4 ParaGraph Communication Matrix display

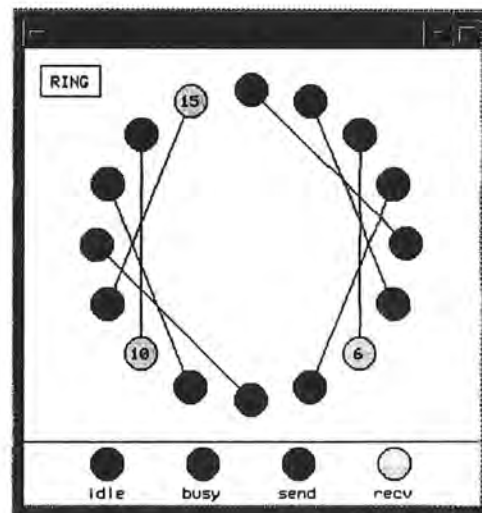


Figure 2.5 ParaGraph Animation display

The Communication Matrix display is also animated (see Figure 2.4). Messages are represented by squares in a two-dimensional array, whose rows and columns correspond to the sending and receiving

processors for each message. Each square is colored when a message is sent and erased when it is received. The color changes for different message types.

The Animation display is similar, showing messages being exchanged among processors (see Figure 2.5). Circles in the graph represent processors, and lines between circles symbolize messages being passed. The colors of the circles indicate processor status (busy, idle, send, or receive). A line is drawn and erased as a message is sent and received.

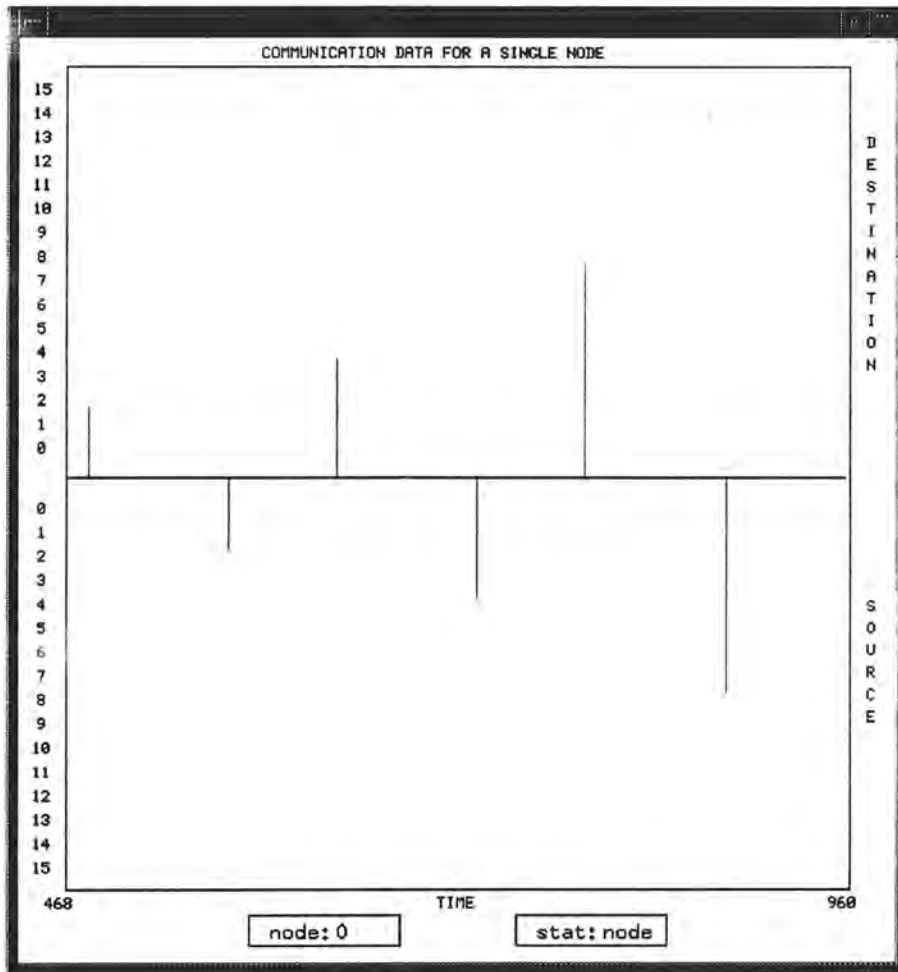


Figure 2.6 ParaGraph Node Data display

The Node Data display is a static display showing detailed communication data for any single processor (see Figure 2.6). To change the selected processor, the user clicks on the “node” box until the desired processor number appears. One of four pieces of message information can be shown in the display: source / destination, type, length, or distance traveled. The selected data is shown for each

message sent to or from the chosen processor. The content can be changed by clicking on the "stat" box, in a way similar to choosing a processor with the "node" box.

ParaGraph carries all the common disadvantages of trace-based tools. Communication information is scattered in many windows. It is not detailed enough to really determine the status of a message. For example, the Message Queues display shows the number of messages that have arrived, but the user does not know which ones they are. Nor is information provided on whether a receive operation has been posted. None of the communication displays is suitable for displaying programs with a large number of processors (scrolling is definitely needed). ParaGraph relies on tracefiles produced by PICL (Portable Instrumented Communication Library); the Intel version reads traces in the Pablo Self-Defining Data Format. The dependency on particular instrumentation libraries limits its portability.

2.1.2 etool [25]

etool is one of the three performance monitor profiling tools provided on nCUBE2 supercomputers. It is an event profiler that details activities for every processor and displays them side by side. Only desired events are actually monitored. After collecting event profiling data, etool analyzes data for display with either a graphical or tabular interface. The tabular interface summarizes each subroutine's execution statistically.

The graphical interface is a static time-process diagram (see Figure 2.7). The presence/absence of plain horizontal line indicates computing activity for the corresponding processor, while colored and shaded boxes indicate user and system events. These events are defined and indexed in the legend. The user can see when a message passing operation happened and can compare operations on different processors. Detailed event information, including the exact time of occurrence and some user-defined data, can be obtained via menu selections. Processor activity lines and events can be added or removed from the display so that information can be reviewed selectively.

etool provides event filtering. However, the only message information produced automatically is the timing of message operations. Important message information (source/destination, type, length, etc.) cannot be obtained unless the user manually inserts each request for printing such information during

instrumentation. Nor does the display scale well to programs using a large number of processors. Finally, etool only works on nCUBE machines.

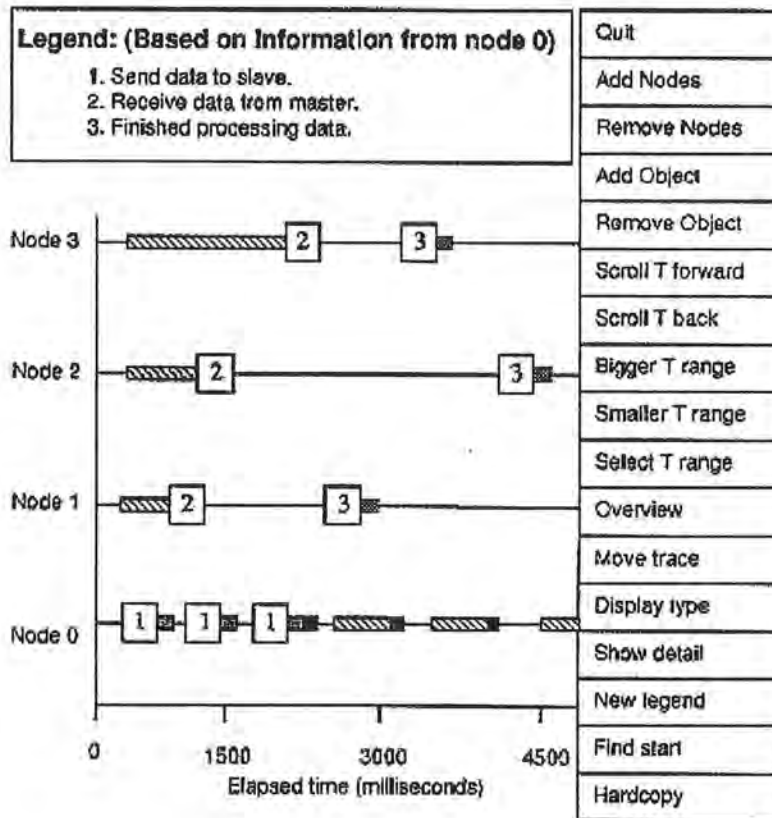


Figure 2.7 etool's graphical user interface

2.1.3 Upshot [10]

Upshot is another performance visualization tool for parallel programs. It does not include a monitoring component, but the trace can be generated from Strand, ALOG, PCN, or P4 programs. The user hand-codes calls to appropriate logging routines (similar to ParaGraph's mechanism). Upshot's visualization is similar to etool's, but provides greater depth and generality. The basic unit in Upshot is the process, not the processor (as in the tools just discussed).

Upshot shows both events and user-defined states (see Figure 2.8). States are defined in a "statefile", by specifying a state name, the corresponding entry and exit events, and colors to be used. These states are indicated by colored bars on the horizontal process lines, while events are represented by

indexed boxes. The meanings of process state bars and event boxes are listed on each side of the display. Clicking on an event box causes a data box to be popped up next to the event box, containing the event type, process identification, time when the event occurred, and a short user-defined string (all of which were recorded in the trace).

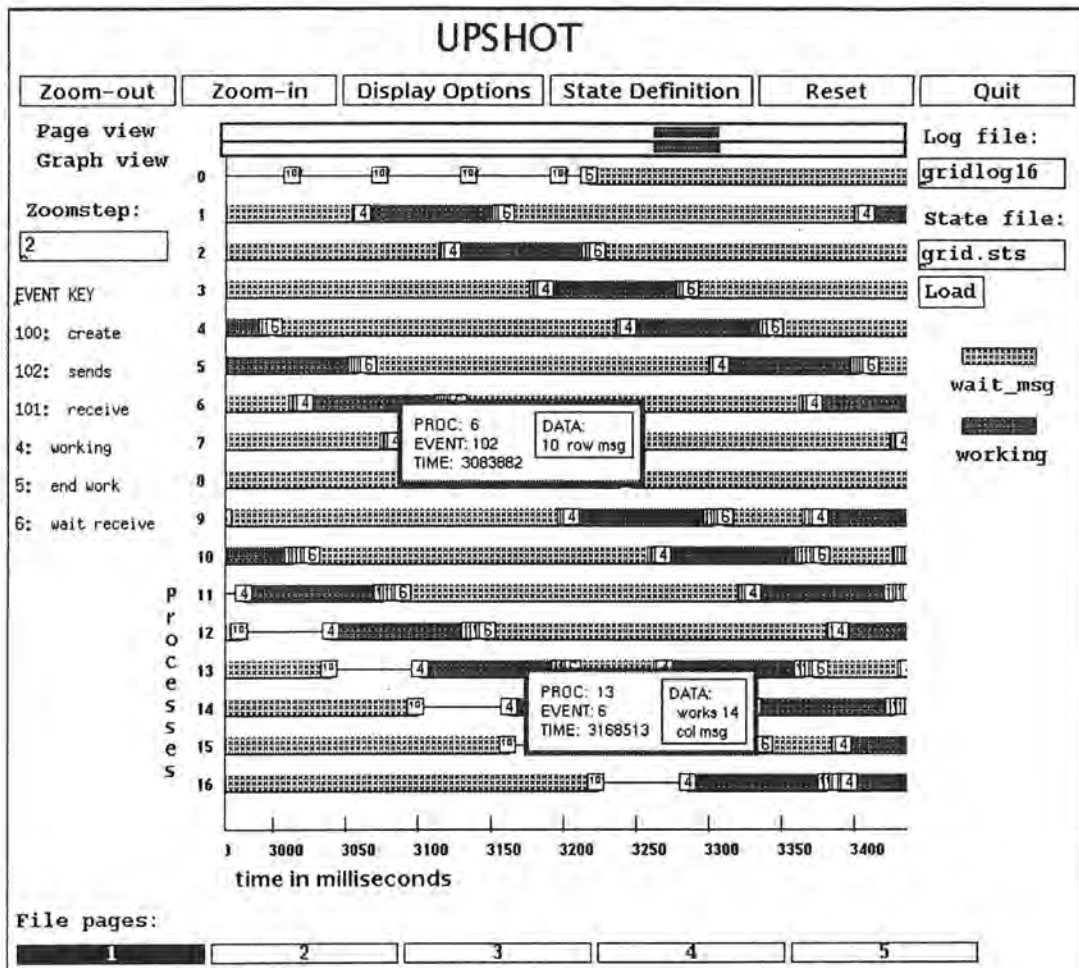


Figure 2.8 Upshot

If a trace is long, Upshot divides the display into several “pages”, each with roughly the same number of events. The user can choose which page to view by clicking on page numbers at the bottom, then scroll through the selected page using the scrollbars at the top. Upshot also provides zooming, so that process bars can be stretched to separate closely-spaced events, or reduced to hold more events on the screen.

Upshot has all the disadvantages of a trace-based tool. However, it is platform independent as long as the proper trace format is generated. Message information is still very limited (must fit into a user-defined string of 12 or fewer characters), and must be furnished as part of the manual instrumentation. Although Upshot provides an interesting scrolling and zooming mechanism, it does not scale well to large numbers of processes or events.

2.1.4 VISPAT [12]

VISPAT (VISualization for Performance Analysis and Tuning) is another trace-based tool for parallel program performance analysis and tuning. It is somewhat similar to ParaGraph, but with more depth of information and far fewer visualization displays. VISPAT processes and visualizes traces produced by instrumented programs written with the Common High-Level Interface to Message Passing (CHIMP) and the Parallel Utilities Library (PUL) built on top of it.. Four types of data visualization are provided, and more can be added. Of the four, only the Navigation and Animation displays contain information on the status of message passing operations.

The Navigation display is a time-process diagram (see Figure 2.9), with the user defining the process states. The visualization differs from other time-process diagrams in that it shows events (including calls of library functions and programmer-defined logical parts of the source code) hierarchically, reflecting the hierarchical structure of the program. Events, represented by horizontal bars, can thus be expanded into sub-events. Particular events can also be filtered from the display. With this combination, the user can control not only which events are visible, but also the amount of detail shown. The Navigation display also provides two display modes for non-blocking communications. In the first mode, such an operation finishes when the function initiating it returns; in second, the operation must truly have completed.

The Animation display (see Figure 2.10) presents a two-dimensional image of program communications. Each circle represents a process, numbered by its identifier. Incoming arrowheads indicate where processes have initiated a blocking receive. Outgoing arrowheads indicate initiation of blocking sends. The presence of an arrow body shows that a communication is in progress.

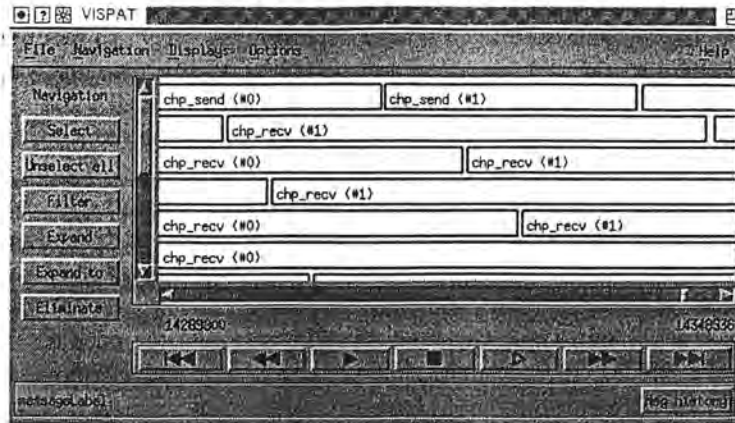


Figure 2.9 VISPAT Navigation display

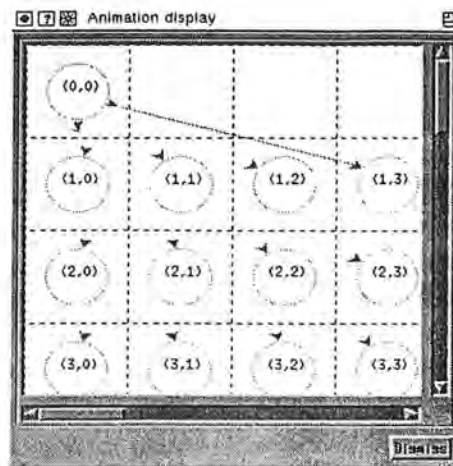


Figure 2.10 VISPAT Animation display

Although VISPAT improves on earlier trace-based tools, it still does not provide complete information on the status of message operations. For example, the Animation display does not tell the user the status of non-blocking operations. Neither display scales well to large programs, although the hierarchical event structure is very useful. VISPAT works only with the CHIMP and PUL.

2.1.5 AIMS [22] / CXtrace [5]

AIMS includes the typical trace-based components: instrumentation interface, run-time performance-monitoring library, and visualization/analysis tools. However, it provides a graphical instrumentation tool that allows the user to select desired files and constructs, and have them

instrumented automatically. Four types of displays are included in the visualization tool. CONVEX adapted AIMS on their machines as CXtrace, but only uses one of AIMS's visualizations; they developed their own textual performance summary as well.

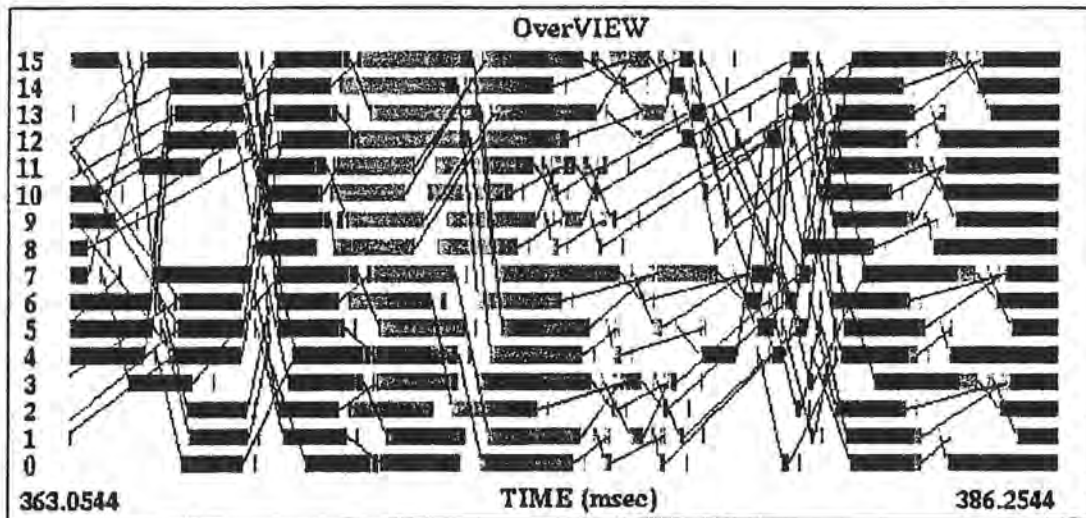


Figure 2.11 AIMS Overview display

AIMS Overview is another time-process diagram (see Figure 2.11), but it does a better job on describing processor interactions and offers more complete event information than previous time-process diagrams. A horizontal bar indicates that the processor is executing, where different colors represent different source code construct; a blank space means the processor is blocked waiting for something to happen. A thin line linking different processor bars simulates a message being sent and received. The time of send/receive, message source and destination, message type, and size are displayed if the user clicks on the message line. The source code corresponding to a message can be seen using mouse button and key combinations.

Two other displays (Figure 2.12) show the status of individual processors. Each node represents a process; its identifier is shown at the top of the box. Colors in the middle rectangle represent the process' state (busy, blocked, or flushing); the name of the source program routine is at the bottom. The relative number of pending messages is implied by the height of the bar in the left column, while processor utilization is described by both the height and color of the bar in the right column. Lines or arrows

between boxes indicate where messages are waiting to be received. Other displays show the cumulative volume (in bytes) or count of pending messages.

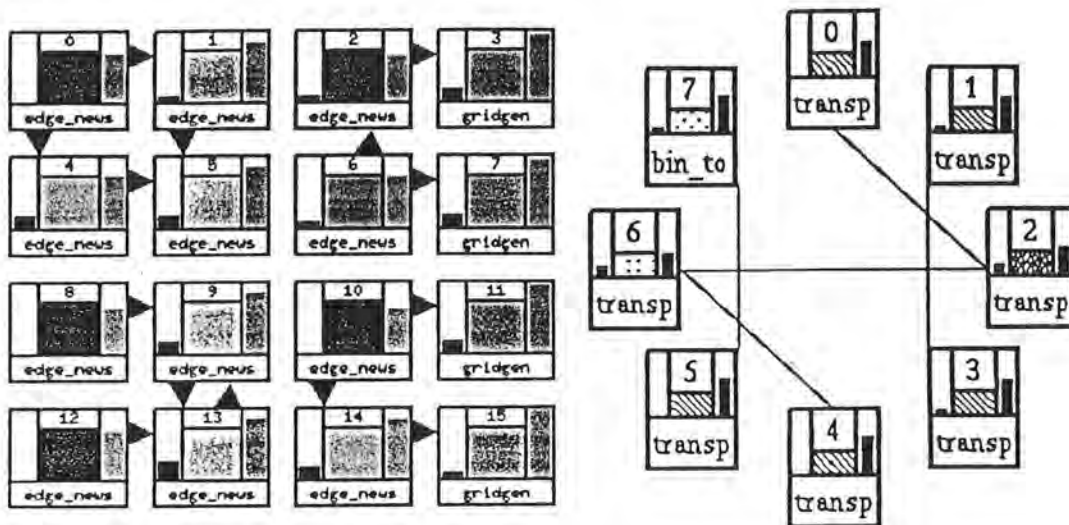


Figure 2.12 AIMS Boxes displays

AIMS and CXtrace exhibit all the disadvantages of trace-based tools. In the Overview, the user does not know the status of messages until they have been received. In the Boxes views, the user cannot tell what messages are pending receipt. The displays are not scaleable to a large number of processors. Although both tools can visualize trace files on any workstations, AIMS only works with NX programs, while CXtrace only works with ConvexPVM programs.

2.1.6 Xab [2]

The structure of Xab is similar to trace-based tools -- it has an instrumentor, a monitoring library, and a visualization tool. However, as an option to recording trace data in a file, the data can be piped to the visualization tool as it is generated, yielding a quasi-real time representation of the program execution. Instrumentation is through an instrumented version of the PVM library; the monitoring program runs as a PVM process and gathers monitor events in the form of PVM messages. Figure 2.13 shows an example of Xab's interface. The display provides a brief descriptions of each event that occurs. A typical message description includes the host name, event name, message type, a unique serial number, message size, and an icon indicating the event type.

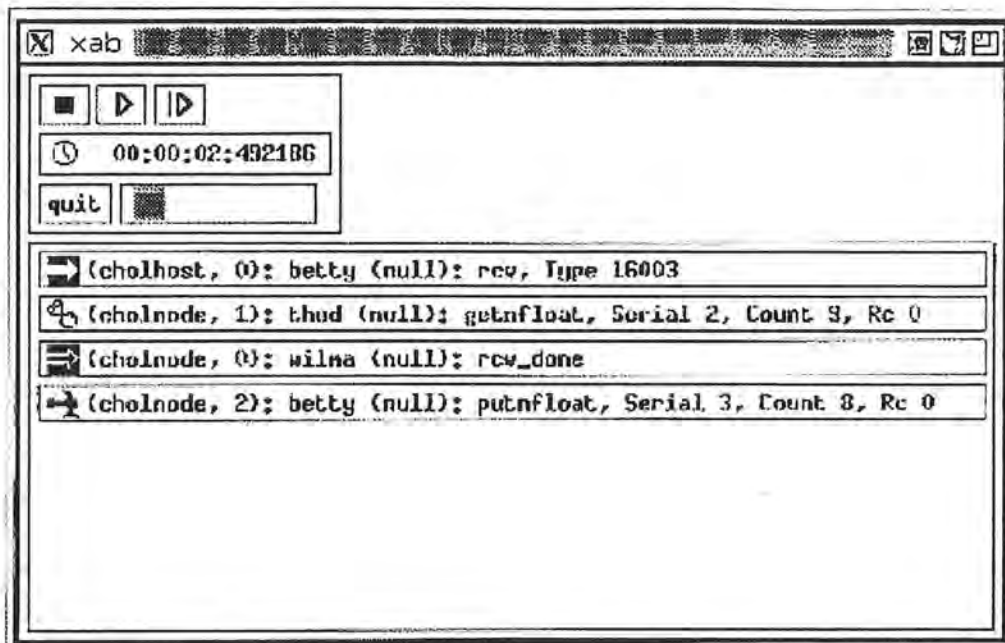


Figure 2.13 Xab's display

Xab yields an textual listing of events when animating a full program execution. Only row event information is provided, and there is no status information on pending messages. Information is displayed as the event happens, so it is not well organized. With a high volume of messages, the display can be extremely hard to read, clearly lacks scalability. Finally, Xab supports only PVM.

2.1.7 Radar [18]

Radar is a post-mortem visualization tool. It runs on PERQ computers, and supports programming in Pronet. Radar is used to provide a graphical replay of the execution from the records generated by Alsten, an event monitor. It is another trace-based tool, but Radar's visualization style is unique.

In Figure 2.14, each process is represented by a rectangular box. The process identification and the numbers of input and output ports are indicated inside the box. In this particular case, the dotted line animates a message being sent from process B to process A. The animation occupies 3 seconds, since the actual sending is too fast to be displayed. Once a message arrives, it queues up at the destination process and a plus sign is displayed. However, there is no way to find out whether a particular receive operation

has been posted, nor is the display scalable to a large number of processes. Radar has not been ported to any generally available systems.

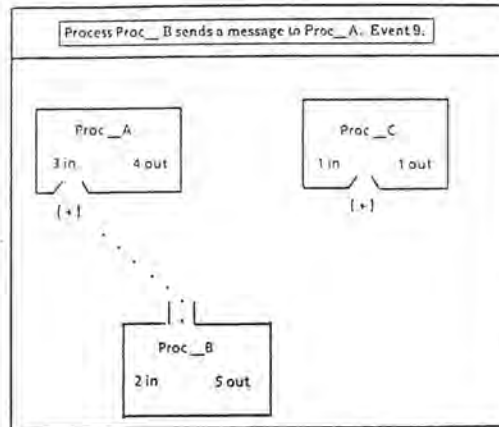


Figure 2.14 Radar's view of a message being sent

2.1.8 Belvedere [13] and Ariadne [6]

Belvedere is a special trace-based tool. It is a part of the Simple Simon Programming Environment, which mainly is a multiprocessor simulator. Event records are automatically generated by the simulator, and treated as a relational database so that queries can be issued to obtain selective information. The selected event records can then be animated to show the patterns of interprocess communications.

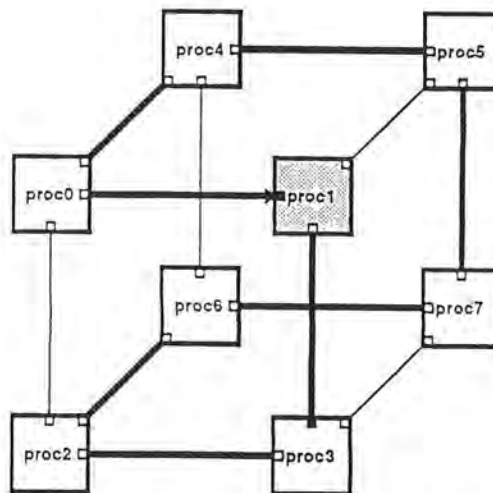


Figure 2.15 Belvedere's snapshot of message passing

In Belvedere's animated display (see Figure 2.15), each box represents a processor, with small squares inside the box representing its ports. A thin line connecting a pair of processor boxes signifies a connection between them, specified by the user. Message passing is indicated by highlighting. A highlighted arrow indicates a send operation, while a highlighted port shows a receive operation. The number of arrowheads refers to the number of messages pending receipt. In its normal mode, highlighting lasts only as long as the events themselves. Belvedere also provides a traced animation where highlighting persists for a longer time.

The diagram gives an overview of the communication status, and is useful for recognizing communication patterns. However, it cannot display information from a large number of processes; even a moderate number presents problems. Currently, Belvedere only works on trace-data produced by a low-level multiprocessor simulator.

Ariadne is a post-mortem modeling language that also uses the trace generated by Simon's simulator. It provides a formal way to define patterns of events, and issue instructions similar to queries to find matches in the trace. The user uses the language to organize and display desired event data. Although Ariadne is more concise than many other modeling language, the language is still complex, and all the disadvantages of textual data presentation persist.

2.1.9 Pablo [1,33]

The Pablo software environment contains source code instrumentation, data capture, and performance analysis components. The instrumentation software can automatically adjust monitoring frequency if the overhead is higher than a specified level. The data capture software records data in SDDF (Self-Defining Data Format) format and computes a performance summary.

The Pablo performance analysis software consists of several independent, though related, components, all of which share data in SDDF format. These include: Graphical Programming, HPF Analysis, Statistical Analysis, I/O Analysis, and Virtual Environment. Graphical Programming is the component of interest here. It allows the user to construct a data analysis graph by interactively selecting, connecting, and configuring performance data analysis and presentation modules.

Figure 2.16 gives an example visualizations built using Pablo facilities. The data analysis graph in the background specifies visually what analysis will be carried out and how results should be displayed. The root indicates the input trace (must be in SDDF format). Each node below reduces or transforms performance data (still in SDDF format) as it flows through the graph. Each leaf is a graphical representation module that displays the reduced data graphically. The six diagrams in the foreground correspond to the six leaves in the graph. Like ParaGraph, Pablo can provide many visual perspectives of the same data. However, there is a distinction. Since Pablo is a tool that “builds” graphs, any diagram can be used to present any set of data; in ParaGraph, though, each of the diagrams is pre-defined. The tradeoff is that the user must learn how to visually program what he/she wants to see.

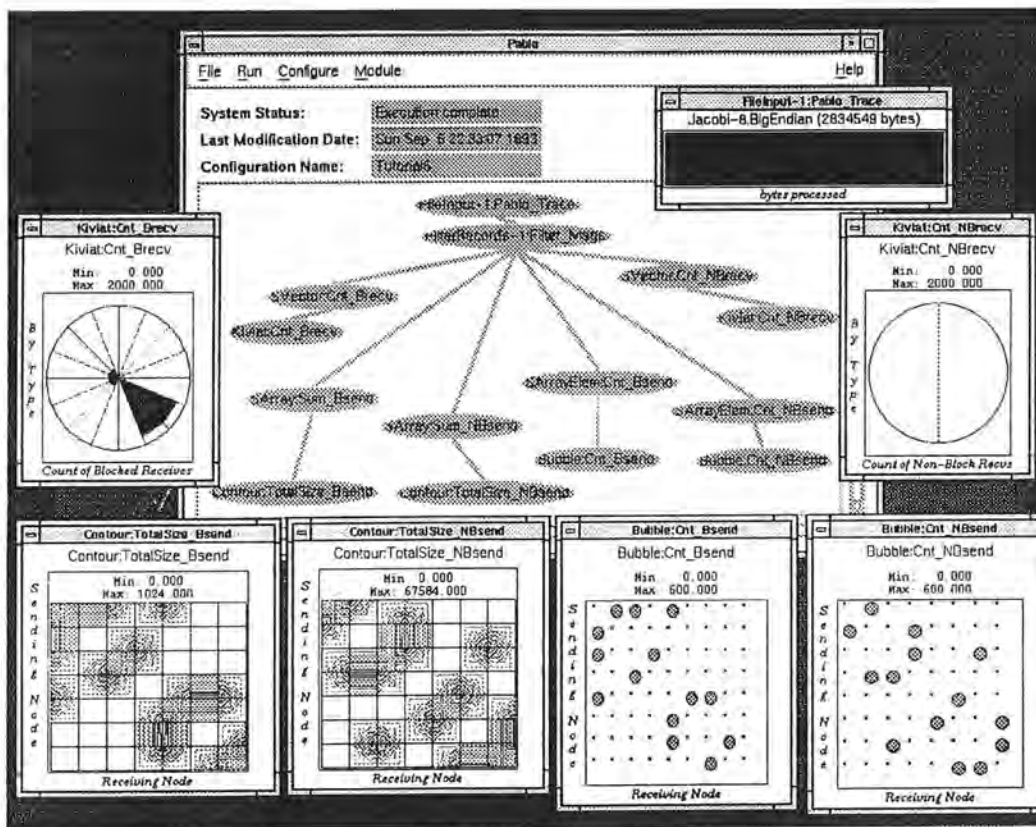


Figure 2.16 Visualizations of message passing constructed using Pablo

Pablo provides great flexibility in visualizing data, but emphasizes performance. There is no support for determining the status of message operations. Also, although data analysis modules can be

used to reduce the volume of data, the diagrams still do not scale well to a large number of processes. Pablo could work with any message passing system, but requires that SDDF format be used for the event trace.

2.2 Interactive (Break-Point Style) Tools

With an interactive, or break-point style tool, debugging can be performed during program execution. The user either sets breakpoints, at which the execution will stop automatically, or manually pauses execution to examine the current state of the program. Execution control over the program must be provided to make any interactive tool useful.

With interactive tools, it is difficult to get an overview of program execution (as in a time-process diagram), but the user gets the benefit of being able to inspect program state closely at any point during the execution. Unlike trace-based tools, no instrumentation is required from the user. However, execution is slower, and if multiple runs are necessary to locate errors, the same message passing sequence cannot be guaranteed to repeat due to the non-deterministic nature of parallel programs [21].

2.2.1 ndb [25]

ndb is nCUBE's interactive symbolic debugger. It supports both machine and source level debugging. The "show queues" command lists all messages that are pending receipt, including source, type, length, data (in hex), and memory address of each (see Figure 2.17).

```
> show queues
Node 0:
  src 1 type 124 pack 1 length 4 (d0 0f 49 40) @0x800e453c
  src 2 type 124 pack 1 length 4 (d0 0f 49 40) @0x800e4968
  src 3 type 124 pack 1 length 4 (d0 0f 49 40) @0x800e4d94
Node 1: No messages
Node 2: No messages
Node 3: No messages
>
```

Figure 2.17 ndb's "show queues" command

With **ndb**, it is easy to find out what messages are pending receipt. However, no other information on message passing such as pending send and receive operations, is provided. The output is textual, and there are no filtering mechanisms. This makes the data difficult to read, especially when many processes are involved. Finally, **ndb** only works on nCUBE machines.

2.2.2 IPD [14] / XIPD [16]

Intel's IPD (Interactive Parallel Debugger) is a source-level debugger for parallel programs running on the Intel iPSC/860 and Paragon system. XIPD is an X Window System interface to IPD. XIPD shows current processor activity graphically, using icons, and can immediately point to the location of the source code that is currently being executed. The user has full control over program execution, can display and modify variable values, and can view information on the status of message operations.

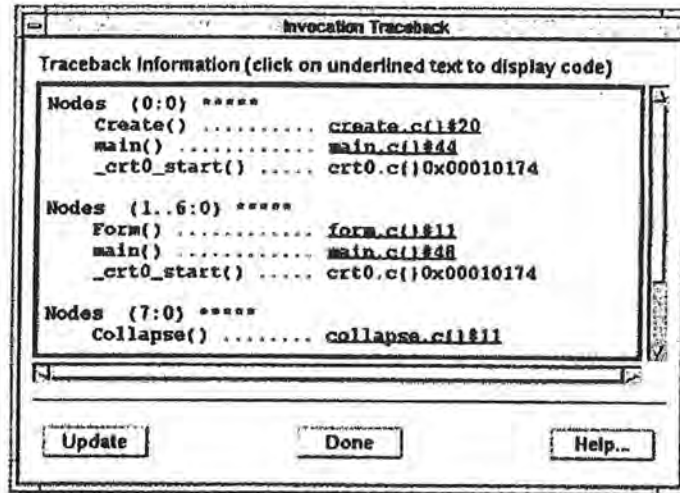


Figure 2.18. XIPD traceback window

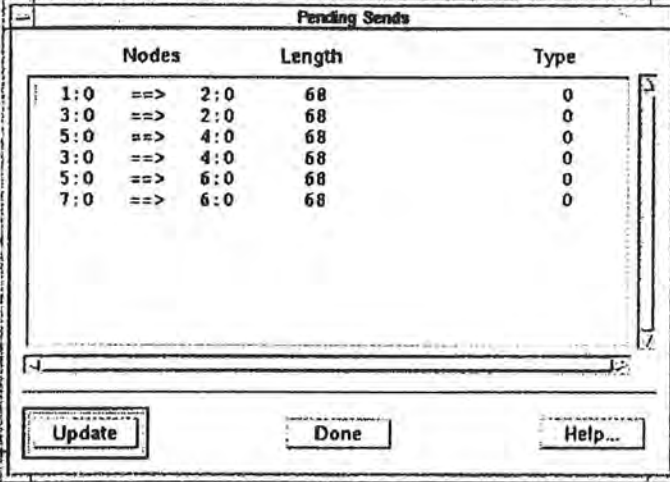
XIPD's traceback window (see Figure 2.18) displays the output of IPD's **frame** command. All routine calls on each processor's stack are listed, including message passing routines and their source code locations. If the source location is underlined, clicking on it will display the code.

The pending sends dialog (see Figure 2.19) shows the source, destination, length, and type of all messages pending receipt, in the same format as IPD's **msgqueue** command. The pending receives dialog (see Figure 2.20) shows any processors that have posted receive operations, as well as the lengths and types of the messages to be received (identical to IPD's **recvqueue** command).

With IPD or XIPD, the user can find out the status of pending messages and pending receive operations, but not the status of pending send operations. Although XIPD has a graphical interface, the contents of traceback window, pending sends dialog, and pending receives dialog are textual in nature.

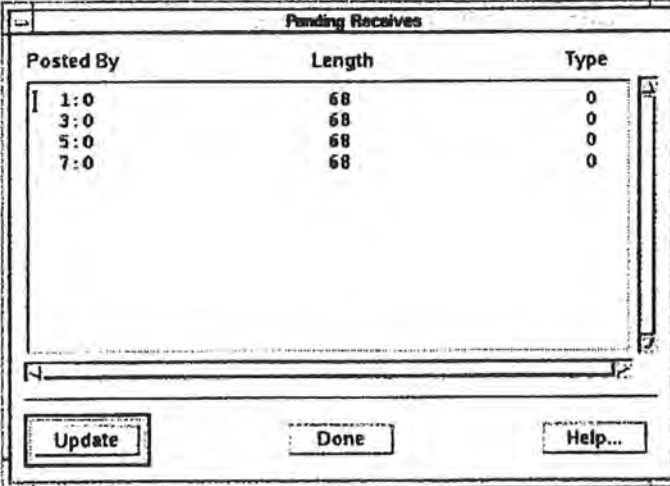
Some filtering is provided, but not filtering by message type. None of the three windows is scaleable.

Finally, IPD and XIPD are platform dependent; they only run on Intel parallel computers.



Nodes	Length	Type
1:0 ==> 2:0	68	0
3:0 ==> 2:0	68	0
5:0 ==> 4:0	68	0
3:0 ==> 4:0	68	0
5:0 ==> 6:0	68	0
7:0 ==> 6:0	68	0

Figure 2.19. XIPD pending sends dialog



Posted By	Length	Type
1:0	68	0
3:0	68	0
5:0	68	0
7:0	68	0

Figure 2.20. XIPD pending receives dialog

2.3 Summary

In the previous subsections, we discussed several debugging and performance tools for message passing programs. Although all the tools discussed can be very helpful in various areas of message

passing programming, none of them provides sufficient information for the user to determine the status of message passing operations. Simple questions, such as

- what messages have arrived at processor i , and are waiting to be received?
- what messages process j is waiting to receive?
- whether the message that process k is expecting has been sent?

cannot be answered completely by any of the tools mentioned. This finding alone supports the need for a dedicated message passing debugging and performance tuning tool like MQM.

Meanwhile, from the survey, we learned a couple of things. First, an animated display is better suited to convey message information than a static display. A static display is very useful in describing a program's run-time behavior over the entire period of the execution, but it cannot offer a detailed description of what is going on at a particular moment because of limits on screen space. A snapshot, on the other hand, uses the extra dimension of time to represent transitory details. Second, we will be better off acquiring message information dynamically as it is needed, rather than storing an entire trace with frequent monitoring, if an overview summary of the execution is not required.

During the study, some common shortcomings among existing tools caught our attention. Most of the tools surveyed run on certain types of hardware platforms, or work only with some particular message passing system. Scalability is a big issue, too. Most visualization displays cannot handle programs involving hundreds of processes in a reasonable screen space, which makes it difficult to form an overall picture of the program status. Filtering mechanisms reduce the amount of data shown and help the user identify important information. Many tools do offer some sort of data filtering, but the support is generally limited.

During the developing of MQM, we kept in mind what we have learned, while trying to overcome, or at least address, as many common problem areas as possible. As a result, MQM is portable to any hardware platform running the X Window System, and can be adapted to any message passing system, debugger, or performance tool if message queue data is accessible. All message related information is accommodated. The scalability issue is addressed in MQM, and a comprehensive filtering mechanism is included.

3. SAMPLE USER'S SESSIONS WITH MQM

This section demonstrates by examples how MQM can be used to debug and fine-tune message passing programs. The first subsection explains the example message passing program we use. The next two subsections take the reader through a couple of MQM sample sessions that debug and fine-tune the example program.

3.1 The Example Program

Many scientific and engineering problems can be characterized as systems of linear equations. Iterative methods are commonly used to solve these linear systems on computers, especially for large sparse linear systems. With iterative methods, the solutions (the values of the unknowns) are initially estimated. Since each unknown in a linear system can be expressed in terms of other unknowns (e.g., $X_i = (b_i - \sum_{j \neq i} a_j X_j) / a_i$), substituting the initial values of the unknowns into the expressions yields a new value for each unknown. The new values are then substituted back into the expressions for a new round of computation. The accuracy of the unknowns' new values improves after each round of substitution and computation. The iteration continues until the unknowns' new values obtained in the current iteration and their old values from the last iteration converge, or their differences are less than the specified criterion.

Successive Overrelaxation (SOR) is one of the iterative methods. It accelerates the rate of convergence with two improvements. First, SOR always substitutes the latest computed values into the expressions; if these unknowns have been computed earlier in the current iteration, those values will be used instead of the ones from the previous iteration. Second, SOR always takes into account the unknown's previous value, assigning it a new value that incorporates both the expression's new result and the unknown's old value. Details of SOR and other iterative methods can be found in [32,36]. In this section, we use the SOR pseudocode from [11] to illustrate the usefulness of MQM. Figure 3.1 shows a slightly modified version of that program using master-worker model.

```

real A(n,n)          /* nxn unknowns in a matrix with initial estimated values */
int nproc           /* number of worker processes */
spawn(nproc)       /* spawn nproc worker processes */
scatter()           /* distributes n/nproc rows of the initial values to each worker
gather()           process */
gather()           /* collects worker processes' results and store them into A */

```

(a)

```

int m = n/nproc;    /* each worker gets n/nproc rows to be
                    stored in B */
real c              /* convergence criterion */
real B(m+2,n)      /* allocates two extra rows (B(0), B(m+1))
                    for storing adjacent rows from neighbors */
receive_scatter()  /* receives m rows of initial values and
                    store them in B(1) to B(m) */

myid = myproc()
while (max>c) do
  if (myid>0) send(myid-1,B(1)) /* start SOR */
  /* send the values in top row to the lower-
  ranked neighbor (except for first worker) */
  /* receive the data sent by my neighbor */

  if (myid<nproc-1) recv(myid+1,B(m+1))
  do i=1,n-2
    if (myid > 0) recv(myid-1,B(0,i)) /* blocking receive a newly computed
    bottom-row unknown from the lower-ranked
    neighbor; the first worker is never blocked */

    do j=2,m-1
      tmp=B(j,i)
      B(j,i)=0.7*(0.25*(B(j-1,i)
        +B(j,i-1)+B(j+1,i)+B(j,i+1)))
        +0.3*B(j,i) /* assigns 70% of neighbors' average and
        30% of its old value */
      diff=abs(B(j,i)-tmp) /* difference between old and new value */
      if (diff>max) max=diff /* find the local maximum difference */
    enddo
    if (myid<nproc-1) send(myid+1,B(m,i)) /* send next process a bottom-row unknown's
    newly-computed value */

  enddo
  barrier /* synchronize all worker processes */
  max=reduce(max) /* reduce to find the global maximum
  difference */
enddo /* end SOR */
gather() /* send master process the final result */

```

(b)

Figure 3.1 (a) SOR master program, (b) SOR worker program

In our example, we have nxn unknowns arranged in a matrix. A master process equally distributes rows of estimated initial values of these unknowns to the worker processes, and collects their results. The worker processes receive their shares of the initial values and start performing SOR. Each non-boundary unknown is expressed as the average of its four neighboring unknowns. During each iteration, the four neighboring unknowns are substituted into the expression, with the new value based 70% on the result and 30% on the old value.

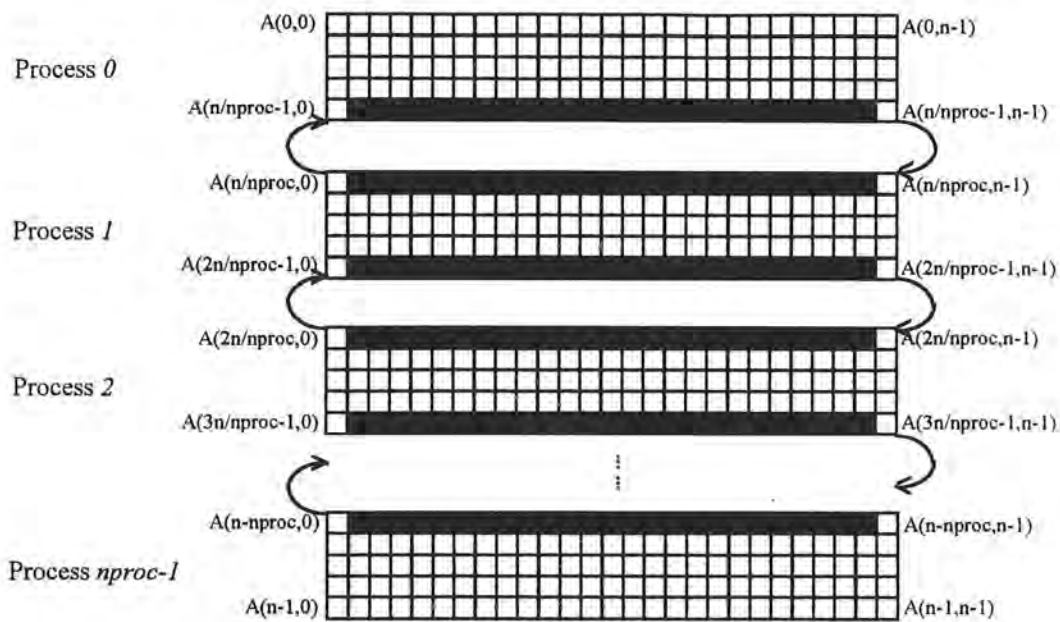


Figure 3.2 Data sharing among the processes of the example program

Since data is now stored on different worker processes, the values of unknowns along the boundaries (shaded elements in Figure 3.2) need to be shared with neighboring processes. In Figure 3.2., the arrows indicate sharing of data with adjacent processes. The first sharing (arrows on the left) occurs at the beginning of each iteration. Due to SOR's rule of using last-computed values, the other data sharing (arrows on the right) has to be postponed; since we always start the computation from the upper-left corner, no unknown can be computed until its left and upper neighboring unknowns are available. Thus, all processes except *Process 0* have to wait for the updated data from their neighbors before they can compute the new result.

The order of computation, row-wise or column-wise, does not change the result of the program, but it affects the program's performance dramatically. If the program is executed row-wise, sending the bottom row to the next processes cannot take place until most of the local computations have been completed. This results in poor overlapping of computation with communication -- the execution becomes virtually serial. If the execution order is column-wise instead, as in our program, some of the bottom-row unknowns are updated much earlier. These unknowns' latest values can be sent to the next process immediately to improve parallelization. Figure 3.3 shows the execution order of our example program

during each iteration. Instead of sending the entire updated bottom row at once, each element is sent as soon as it has been updated. As a result, each process lags its neighbor by one column plus the communication time, and the execution becomes pipelined. Of course, our program may be further optimized by combining the computation of multiple columns, in cases where communication overhead is relatively high. (This optimization is left out in our program for simplicity.)

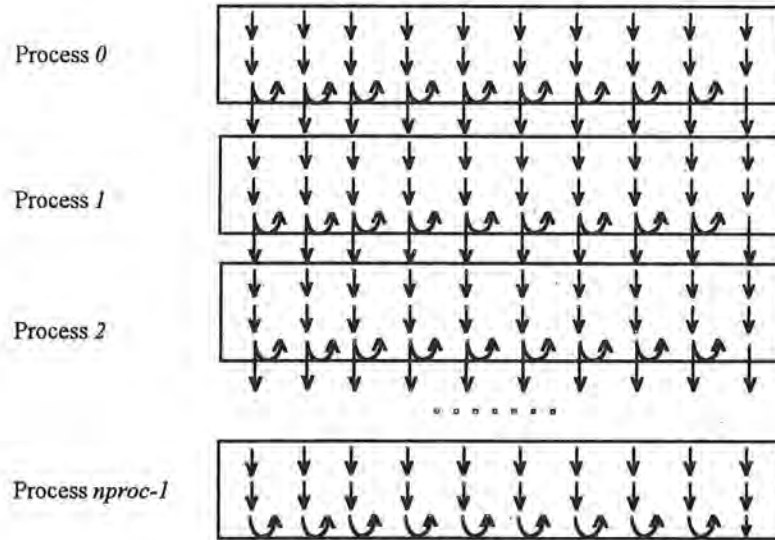


Figure 3.3 Order of computation of the example program

At the end of each iteration, the maximum local differences between old values and new values are compared with those of other processes. If the global maximum difference is less than the threshold criterion, the latest values are returned to the master process as the final solution. The overhead of global communication can also be reduced by computing the global maximum less often, say every several iterations. (Again, this optimization is not included in our program.)

3.2 When Execution Seems to “Hang”

Consider the situation where our SOR program has been executing for some time, but nothing seems to be happening. The user suspects that the execution is somehow “hung”, but there is no real indication of what is happening. We can use MQM to find out. Note that MQM must be integrated into a debugger, a performance tool, or a message passing system for underlying functionality; that tool’s

execution control will be used to pause or halt execution. MQM is then launched, and its Overview window is displayed (see Figure 3.4). The grid of colored squares in the center of the window represent the processes. When the cursor is positioned over any square, it is outlined, and the corresponding process number is shown in the message area at the bottom. The color scale on the right tells the user the meanings of the colors. Each represents a range of how many message operations are pending in that process' message queues. A series of *Queue Display Options* to the left of the window allows the user to include or exclude particular message queues in determining the count of pending message operations. The default selection used when MQM is invoked, is to show only receive operations (i.e., the process has blocked on a receive).

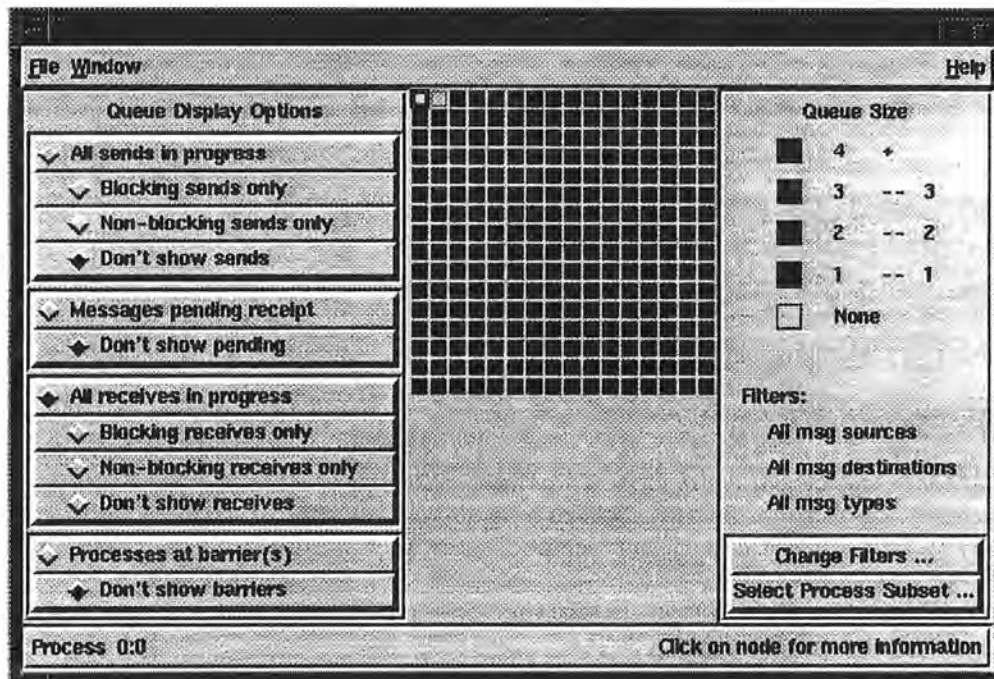


Figure 3.4 MQM Overview upon invocation

In this example, there are 256 squares, corresponding to processes running on each of the 256 nodes of an Intel Paragon computer. Since the Paragon uses logical processor/process numbering conventions, the master process corresponds to process 0:0, and the rest of the processes (1:0 to 255:0) are workers. With the default settings, we are observing all processes and all receive operations. Note that all but the master process and process 1:0 (first worker process) are waiting to receive a message. This

situation indicates two possible scenarios. Either no messages matching the receives have been sent, or the messages are still in transit. To establish which of the two has occurred, we need to see if there are any messages in transit that match the receive statements. We click on “All sends” to see if there are messages currently being sent, but nothing on the display changes.

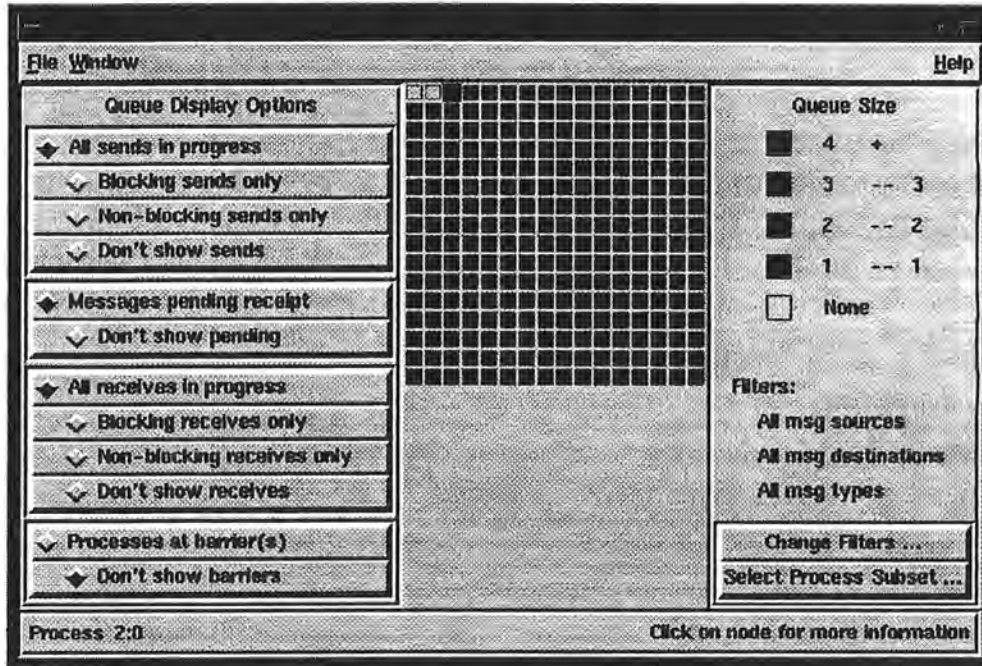


Figure 3.5 Overview with more message queues selected

We then include “Messages pending receipt”, and process 2:0 changes color, showing that a large number of messages are queued, waiting to be received (see Figure 3.5). To investigate why there are so many pending messages on this one worker process, we first want to know exactly which messages they are. To see process 2:0’s message queues, we click on the square representing the process (see Figure 3.6).

A Process-level Display window pops up. At the top of the window, we see that this process is blocked on a receive operation. In the middle of the window, three stacks are shown representing the receive request queue, message arrival queue, and send queue, respectively. Each tile in a stack indicates a pending message or operation. The two fields of the tile identify the other process involved in this point-to-point communication, and the user-defined numerical message type associated with the

communication. In Figure 3.6, we see one receive operation that is waiting for a message of type 100. The tile appears red because it is a blocking operation; non-blocking operations are blue.

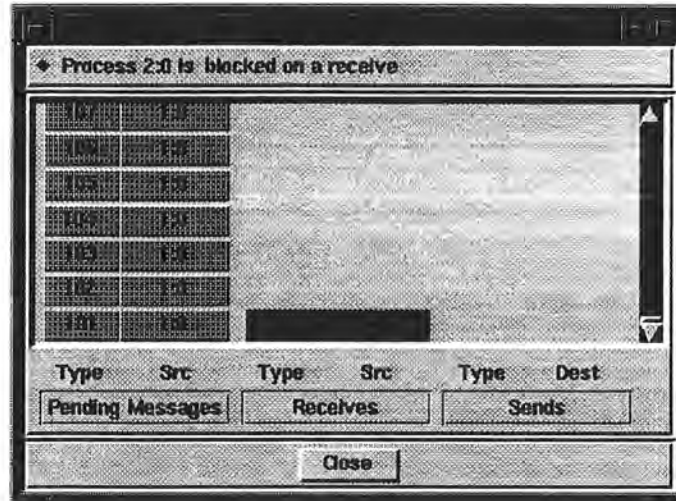


Figure 3.6 Process-level Display of process 2:0

We note that process 2:0 has many messages waiting to be received, but to continue executing, the process must first receive a message of type 100. To see if any messages of this type exist anywhere in the system, we use the filters. This is accomplished by de-selecting receives and clicking on the "Change filters ..." button (see Figure 3.7). We select 100 as the message type and click on "Apply". This has the effect of removing all colors from the Overview (see Figure 3.8), indicating that there is no message of type 100 in transit. Since the program has been executing for quite some time, it is unlikely that a message of type 100 will ever be sent.

To determine where the program is stuck, we search the source code for receive statements naming message type 100. The search leads us to the second receive statement, which should accept a message containing the updated value of a bottom-row unknown (from the lower-ranked neighboring process). We verify that this is the problem by going back to process 2:0's Process-level Display, where it can be seen that all the pending messages are coming from process 1:0 (its lower-ranked neighbor). Since process 2:0 is not accepting any of these messages, something must be wrong with either the receive statement or the send statement.

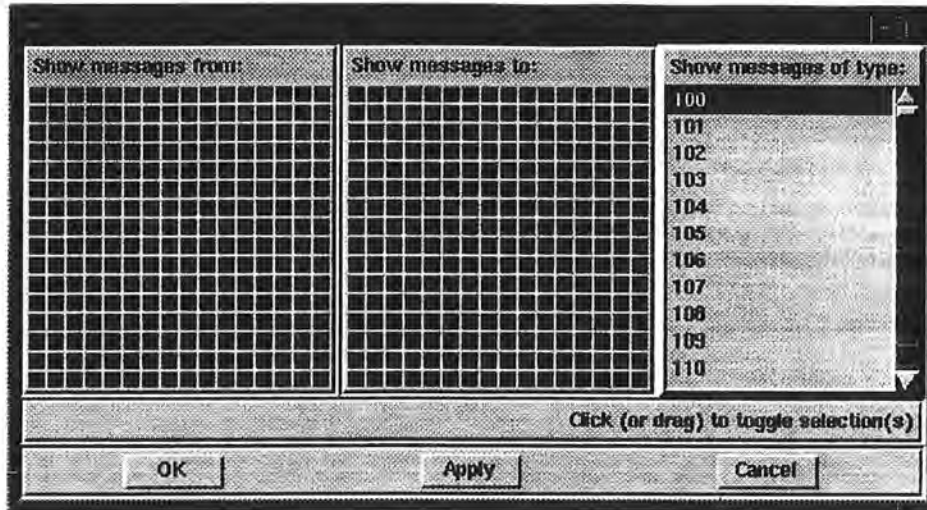


Figure 3.7 The filter dialog

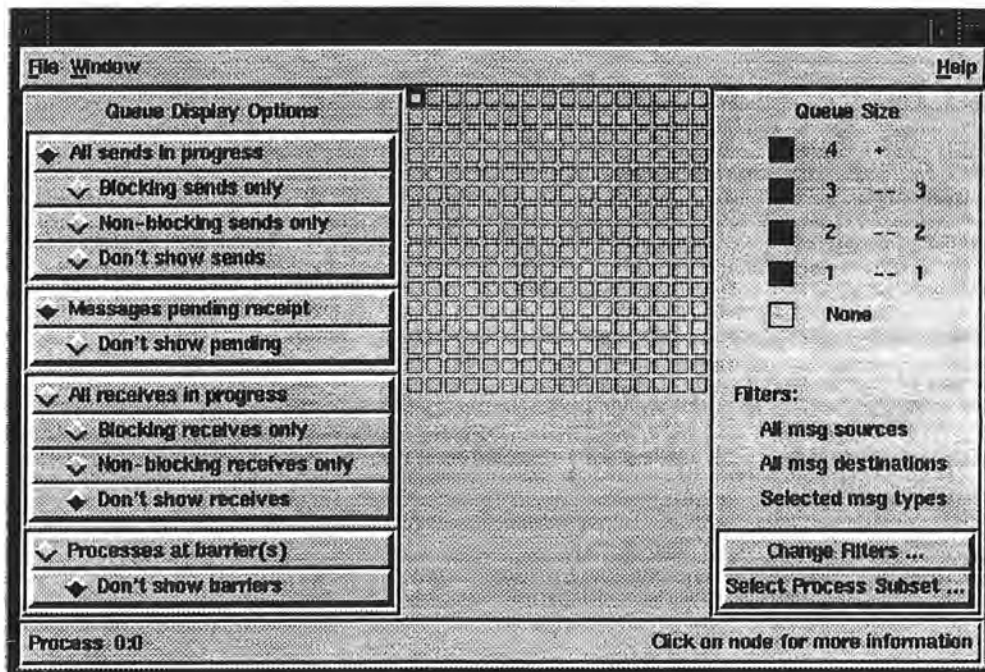


Figure 3.8 Overview with filters set

By examining the source code closely, we conclude that the receive statement names wrong message type; as a result, the receive is not satisfied by the pending message sent by process 1:0. This is true for other processes, too. However, since process 2:0 has not yet received any data, it can not start

computing, not to mention sending data to its neighbor. Because the communications form a pipeline, none of the 253 other worker processes can begin computing.

3.3 Improving Program Efficiency

After fixing the erroneous message type, all subsequent executions of the program seem to be fine. To be sure, we can stop the program anywhere during execution to observe the behavior of message passing using MQM. Figure 3.9 shows one of the snapshots. Although the execution does not “hang” any more, process 2:0 still has a large number of pending message operations. In addition, some of the other processes now have a few more pending operations.

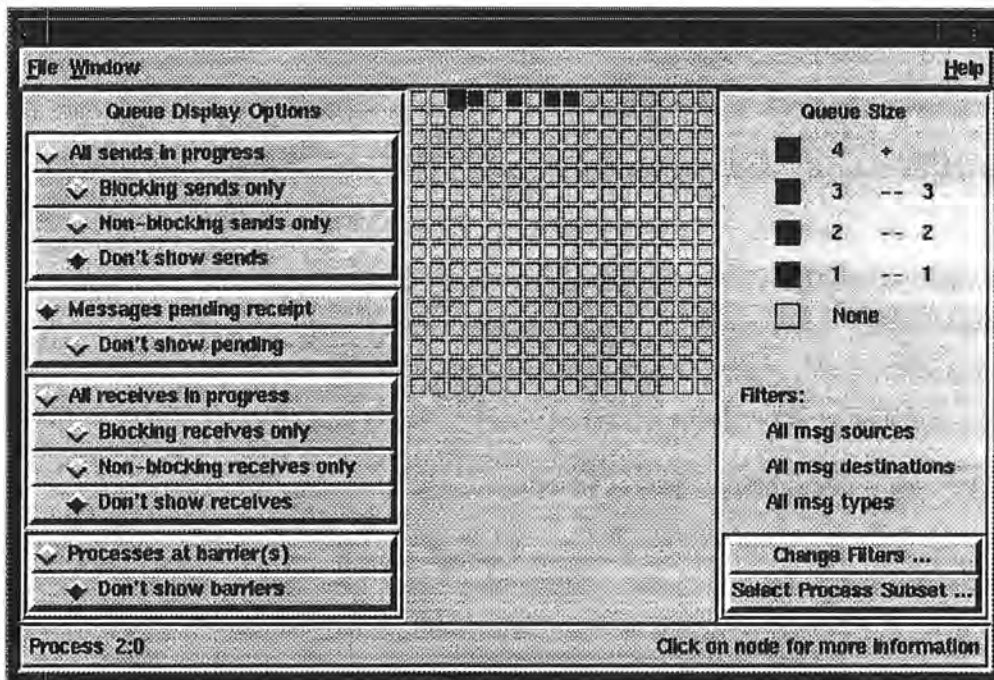


Figure 3.9 MQM Overview with new snapshot

We want to look at process 2:0's queues again (see Figure 3.10) to find out why. The display shows that a lot of messages from its lower-ranked neighbor (process 1:0) have been queued, and that a message is being sent to its higher-ranked process (process 3:0). From the types of pending messages, we see that process 1:0 has finished computing all columns (since all messages containing the last row

unknowns have arrived ^a). We can also see that process 2:0 has just completed computing the ninth column (this is message 109, so it has been incremented nine times, once for each column). The process-level displays of other processes reveal that only a few messages are waiting to be received elsewhere.

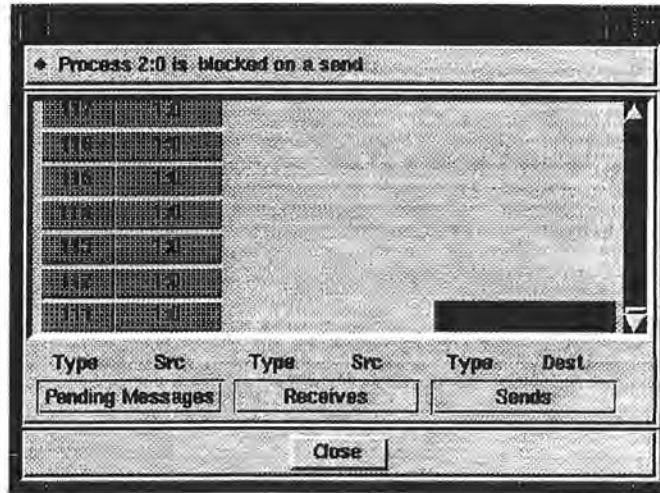


Figure 3.10 Process-level Display of process 2:0

What this information tells us is that process 1:0 (the first worker) is executing much faster than the rest. Recall that the only difference between the first worker and the rest is that process 1:0's computation is not dependent on data from other processes. The other workers, however, must perform a receive for each column they compute. The discrepancy in execution time suggests that the higher-ranked processes are spending a lot of time waiting for data from their neighbors. This means communication overhead is too high for the amount of computation being done between messages. Thus, it is advisable to combine the computations of multiple columns, as mentioned earlier. Although MQM cannot tell the

^a The program increments the message type each time a message is sent, starting from 101, so we can tell at a glance what each message contains.

exact number of columns that should be combined, it can be used effectively to gain a rough idea of whether or not the current configuration generates too much overhead.

4. MQM USER'S MANUAL

This section describes in detail the features of MQM. Each display is described in a separate subsection. A quick-reference summary of tool functionality concludes the section.

4.1 Invocation

MQM can be invoked by the command `mqm` or by other means depending on how MQM is integrated into the underlying software. Since MQM does not function without the support of some run-time software, issuing the command `mqm` with no argument initiates a demonstration of MQM, using sample data.

4.2 Overview Display

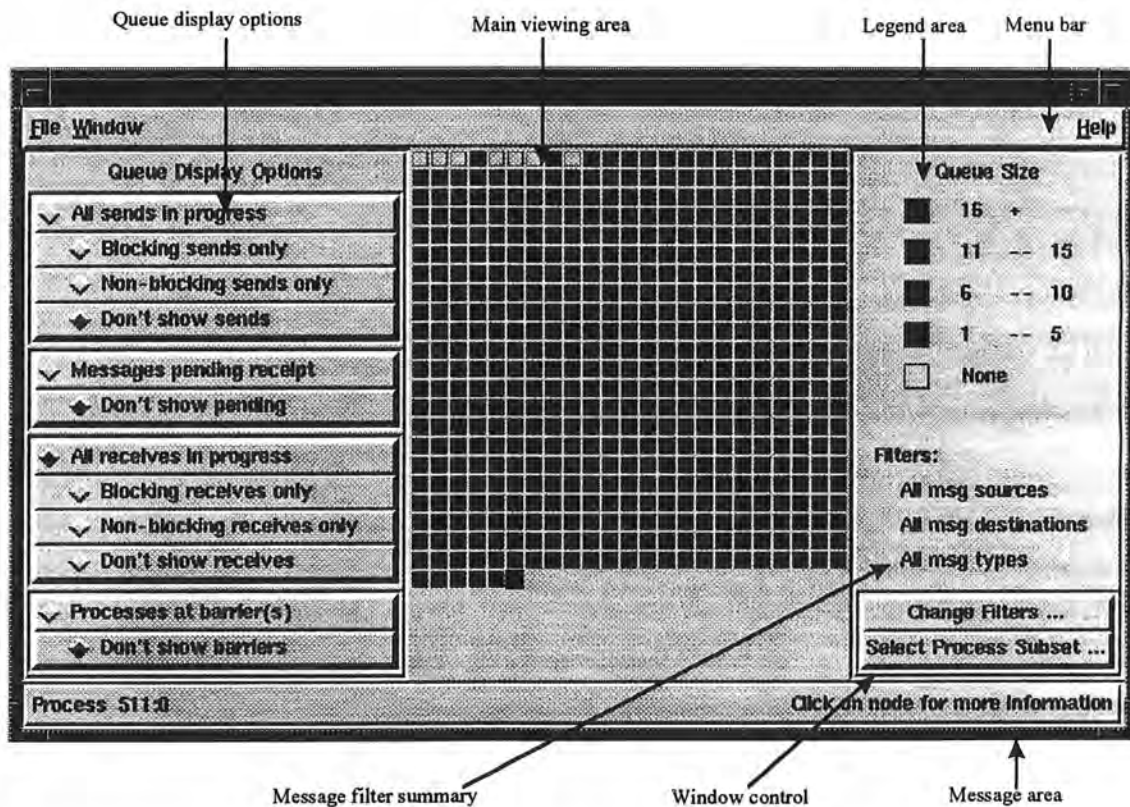


Figure 4.1 Components of Overview display

When MQM is initiated, the Overview Display appears (see Figure 4.1). This presents a graphical depiction of the sizes of all message queues (for each process). A menu bar is located at the top of the window. Queue display options are a series of radio buttons located at the left. To the right, a colored legend is shown, as well as a summary of the current settings of all message filters. Two buttons control the Change Filters Dialog and Process Subset Display; these appear immediately below the summary. A message area is located at the bottom of the display. The following subsections discuss these components in detail.

4.2.1 Main Viewing Area

The main viewing area contains a number of squares, each representing a process involved in program execution. The squares are numbered logically from left to right (i.e., row-wise ordering). The size of the squares varies according to the number of processes displayed. When fewer than 64 processes are involved, the squares are larger for better visibility. If the number of processes is more than 64, the squares become smaller so that more can fit on the screen. The squares are colored according to the sizes of the corresponding processes' message queues. The legend area indicates what numerical range of queue sizes each color represents. The numerical range of each color is equally divided based on the largest queue sizes. It is calculated upon startup and re-computed when program execution state changes.

When the cursor is positioned over any square, that square is outlined and the corresponding process number is shown in the message area. Clicking on any square will pop up the Process-level Display for that process.

4.2.2 Menu Bar



Figure 4.2 File menu

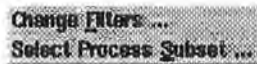


Figure 4.3 Window menu



Figure 4.4 Help menu

Three menus are present in the menu bar: File, Window, and Help. The File Menu (Figure 4.2) has only one item, to exit. The Window Menu (Figure 4.3) contains two items controlling the Change

Filters Dialog and Process Subset Display windows; these menu items have the same effect as the two window control buttons. The Help Menu has four items as shown in Figure 4.4. About MQM displays release information. The Quick Reference provides a summary of functionality. Queue Display Option and Message Filter explains the meaning of the options and filters and describes how to change their settings.

4.2.3 Queue Display Options

<i>Queue Display Option</i>	<i>Explanation</i>
All sends in progress	Include all messages that are being sent by both blocking and non-blocking sends. A send operation contributes to the total reported for the source (sending) process.
Blocking sends only	Include all messages that are being sent by blocking sends; exclude non-blocking sends. (There can be at most one blocking send in progress for a process at any moment.)
Non-blocking sends only	Include all messages that are being sent by non-blocking sends; exclude blocking sends.
Don't show sends	Exclude all messages being sent.
Message pending receipt	Include all messages that have arrived at the destinations and are waiting to be received. Pending message operations contribute to the total reported for the destination (receiving) processes.
Don't show pending	Exclude all messages that are awaiting receipt.
All receives in progress	Include all receives that have been posted, but not yet completed. A receive operation contributes to the total reported for the destination process.
Blocking receive only	Include blocking receives that have been posted; exclude non-blocking receives. (There can be at most one blocking receive posted by a process at any moment.)
Non-blocking receives only	Include non-blocking receives that have been posted; exclude blocking receives.
Don't show receives	Exclude all receives, both blocking and non-blocking.
Processes at barrier(s)	Include one barrier operation in the total count for each process that is currently waiting at a barrier. (Each process can be at one barrier at any moment; different processes can be at different barriers.)
Don't show barrier(s)	Exclude all barrier operations.

Table 4.1 Summary of Queue Display Options

The queue display options allow the user to include (or exclude) certain types of message queues in the total queue sizes shown in the main viewing area. Four categories of message queues are supported in the current version: send, pending message, receive, and barrier. (We assume that a barrier operation is queued before it is completed.) Four groups of radio buttons are used to control the inclusion/exclusion

of those queues. Send and receive queues are further sub-divided into blocking and non-blocking operations. The effect of each radio button is explained in Table 4.1.

The default option upon invocation is “All receives in progress”. That is, all sends, pending messages, and barriers are excluded from the counts. Since some message passing systems do not support all these types of message operations, the queue display options may not include all the radio buttons described above, but the format and functionality remain the same.

4.2.4 Summary of Message Filters and Message Area

Two areas in the Overview Display provide visual feedback to the user. A message area at the bottom of the display shows the process number of currently selected node, as well as general information on how to proceed.

The message filter summary describes the filters (see Section 0), using with descriptors “all/selected” to indicate the setting. The first indicates that all message sources (or destinations/types) have been included in the display. With “selected” filters, only the message sources/destinations/types specified by the user have been counted. The default settings are “All msg source”, “All msg destination”, and “All msg types”.

4.2.5 Window Control Buttons

Clicking the “Change Filters ...” button will open a dialog which allows the user to change message filters (see Section 0). Clicking the “Select Process Subset ...” button allows the user to select a subset of processes to be viewed in a Process Subset Display (see Section 4.3).

4.3 Process-level Display

The Process-level Display shows the actual contents of a process’s message queues. It is invoked by clicking on any process square in the Overview or Process Subset Displays. Figure 4.5 shows the components of a process subset display.

At the top of the display is a message indicating whether the process is currently blocked, and what type of operation is responsible. The "Close" button at the bottom dismisses the window. The contents of message queues are shown in the center area.

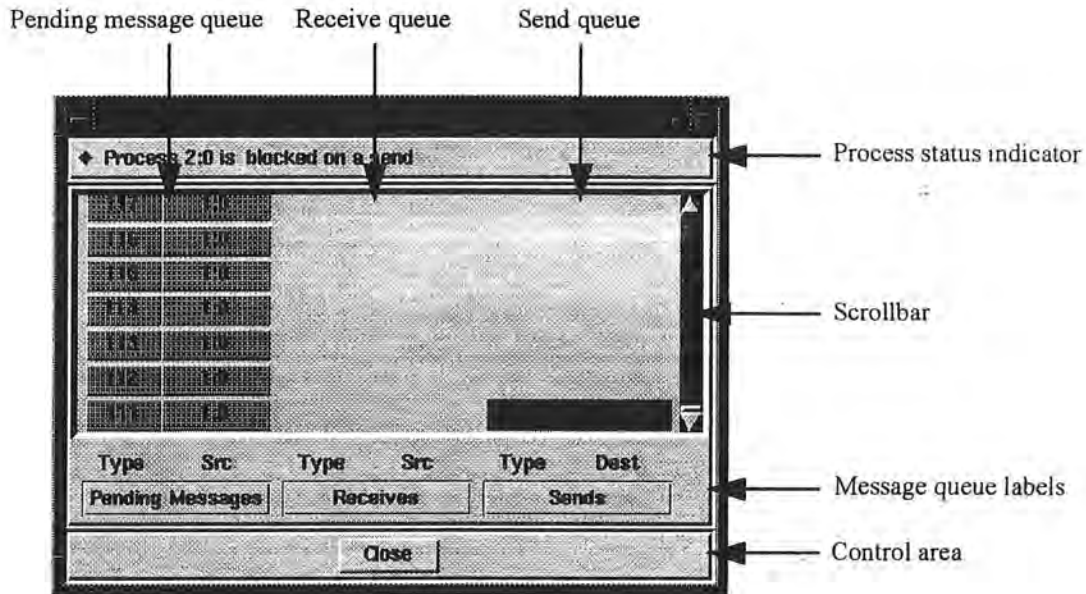


Figure 4.5 Components of Process-level Display

Three message queues are shown: messages pending receipt, receives, and sends, respectively. Barriers are not shown graphically, since there could not be more than one barrier active for a process at any moment. If the process is at a barrier, this is indicated by the message at the top of the window.

The queues contain multiple items, each represent a message operation. At the left of an item is shown its user-defined message type. The field on the right identifies the other participant of the operation (e.g., for a message pending receipt, the process number of the message source appears on the right). All items in the queues have a blue background, except for (at most) one blocking operation which has a red background.

4.4 Change Filters Dialog

The Change Filters dialog is invoked with the "Change Filters ..." button on either the Overview or Process Subset Display. Figure 4.6 shows the names of components of the dialog. The title bar of the dialog indicates to which window these filters apply.

The dialog is sub-divided into three areas, each representing a filter: source, destination, and user-defined message type, respectively. Each is explained below. The filters are activated when the user clicks either the “OK” or the “Apply” button. The difference between the two is that the dialog stays on the screen with “Apply”, while it is dismissed with “OK”. The user can also undo any changes made to the filters by choosing the “Cancel” button. The dialog is dismissed and the display is not changed in any way.

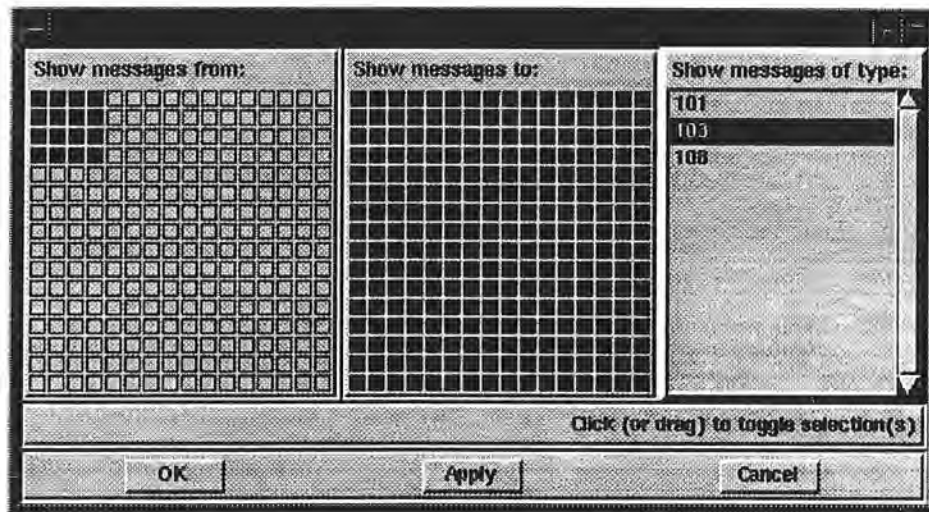


Figure 4.6 Change Filters Dialog

4.4.1 Message Source and Destination Filters

The first two filters resemble the grid from the Overview Display, except that every square is either a single color or blank. Each square still represents a process and its number is shown in the message area.

Selecting a square in the source (destination) filter means that messages whose sources (destinations) match that process should be included in the queue counts shown in the main display area. To de-select processes that are already selected (filled squares), the user clicks or rubber-bands those squares.

4.4.2 Message Type Filter

The area to the right of the dialog has a scrollable list of user-defined message types. The types shown in the list reflect the current settings of the queue display options and message source/destination filters. That is, only the types associated with operations fulfilling both the display options and source/destination filter settings are listed. (Consequently, if the options or filters are changed, some message types may no longer be valid and are removed from the list). To select or de-select message types, the user clicks or drags through the desired message types.

4.5 Process Subset Display

The Process Subset Display is invoked by clicking on the "Select Process Subset ..." button in either the Overview Display or any existing Process Subset Display. It has the same functionality of the Overview Display, but shows just a subset of processes so that the user can focus on specific message patterns.

4.6 Selecting A Subset of Processes

Process Subset Display window first appears (see Figure 4.7), all menus and buttons are desensitized, with the exception of the "Close" button in the new window. A message in the window instructs the user how to select a subset. This indicates that MQM is in *subset selection* mode. The user can only select a process subset or cancel the selection procedure by clicking the "Close" button. Selecting process subset takes place in the original display window (where "Select Process Subset ..." was clicked). The user clicks or rubber-bands the squares representing the desired processes.

When one or more processes have been selected, the reminder message goes away and the selected nodes are copied into the new subset window, maintaining their relative screen positions (see Figure 4.8). Meanwhile, the "Done Selecting" and "Delete Node" buttons in the new subset window become sensitized. If a mistake has been made, the user can remove any node from the subset by clicking the "Delete Node" button. The pointer changes shape, and clicking or rubber-banding the squares in the

new subset window removes the nodes from the subset. When the subset is complete, the user clicks "Done Selecting" button, and members of the subset can no longer be changed.

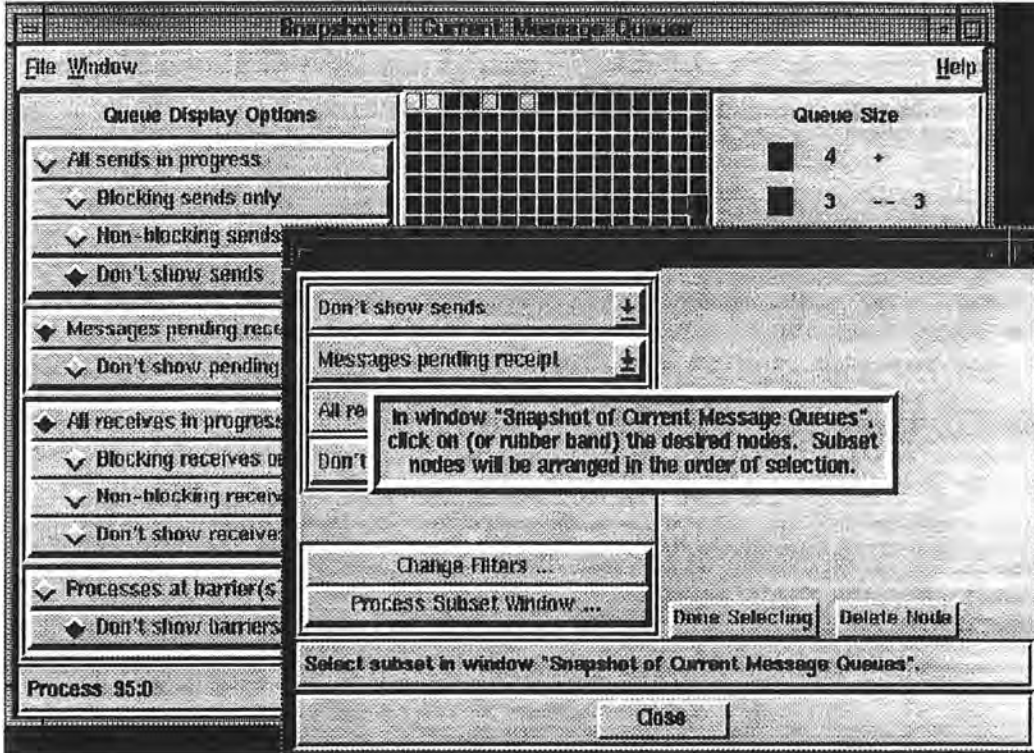


Figure 4.7 MQM in process subset selection mode

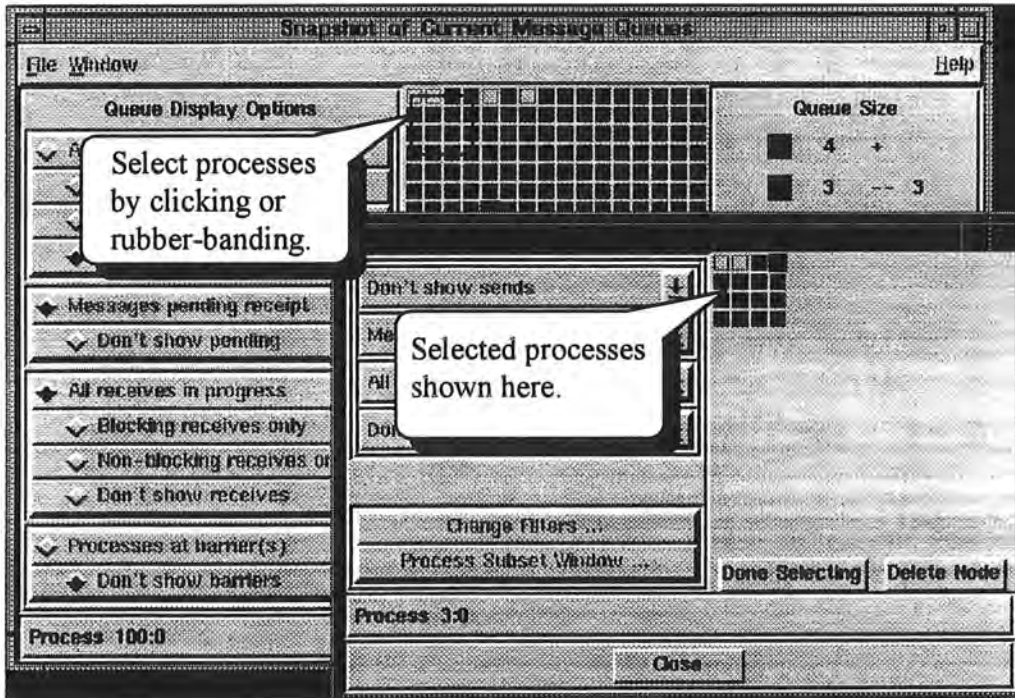


Figure 4.8 Selecting a subset of processes

4.6.1 Process Subset Display

Subset selection mode ends when the “Done Selecting” button has been clicked, and all normal buttons and menus become available. Figure 4.9 shows the new Process Subset window. The “Done Selecting” and “Delete Node” buttons are gone. Squares in the main viewing area are compacted in the order of selection.

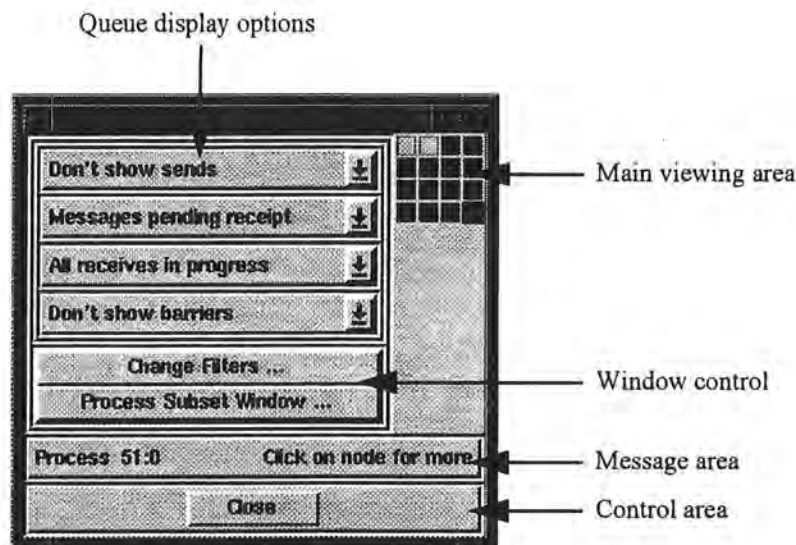


Figure 4.9 Components of Process Subset Display

Although the Process Subset Display is similar in functionality to in Overview Display, it has a slightly different layout. There is no menu bar, color keys, or filter summary. The queue display options appears not as radio buttons, but as pull down menus. These changes reduce the size of the window, but all functionality is the same. Each process subset has its own set of filters, and can be further reduced in size by creating a new subset.

4.7 Summary of Tool Functions

The following list is a quick reference to the functions supported by MQM.

- **To see the identification of the process represented by a square,** position the pointer over the square. The process ID appears in the message area (refer to Section 4.2.1).

- **To find out which processes are engaged in a particular type of message operation** (e.g. blocking receives), select the radio light on the main window that corresponds to that operations, and set all other categories to “Don’t show” (refer to Section 4.2.3).
- **To locate processes that are currently busiest in terms of message operations**, choose the radio lights on the main window that correspond to “All sends”, “Messages pending receipt”, “All receives”, and “Processes at barriers”. Look for the squares colored in the “hottest” color (refer to Section 4.2.3).
- **To find all processes that are blocked** due to some type of message operation, make sure the radio lights for “Blocking sends only”, “Don’t show pending”, “blocking receives only”, and “Processes at barriers” are selected. If the process’ square in the grid appears colored, it is blocked (refer to Section 4.2.1).
- **To view the contents of a process’ message queues**, click on the square corresponding to that process from the colored grid. If the square is not colored, the queues are empty (refer to Section 4.2.1).
- **To find out which operation is responsible for blocking a particular process**, click the process square to bring up the Process-level Display and check the Process Status Indicator at the top (refer to Section 4.3).
- **To find all message operations that are from/to a specific source/destination**, click the “Change Filters ...” button to bring up the Change Filters Dialog. Use the mouse to select which source/destination(s) you are interested in. Click “OK”, and the Overview Display will change to show those operations (refer to Section 0).
- **To find message operations corresponding to a particular set of user-defined message types**, click the “Change Filters ...” button to bring up the Change Filters Dialog. Use the mouse to select which type(s) you are interested in. Click “OK”, and the Overview Display will change to show those operations (refer to Section 0).
- **To view only a subset of processes**, click the “Select Process Subset ...” button. A new window will appear, giving detailed directions (refer to Section 4.6).

5. IMPLEMENTATION OF MQM

MQM consists of two major components: a back end that obtains and manages message queue data, and a graphical front end that presents a snapshot of message queues. The user manipulates the graphical presentation of message queue data by applying a combination of display options and filters, or viewing the data at finer levels of detail. When the user requests a different data presentation, the front end sends a request to the back end, the Query Manager. If the data needed for the new presentation is already available at the Query Manager, it is extracted and sent to the GUI for displaying. Otherwise, the Query Manager forwards queries to the underlying software with which MQM is integrated.

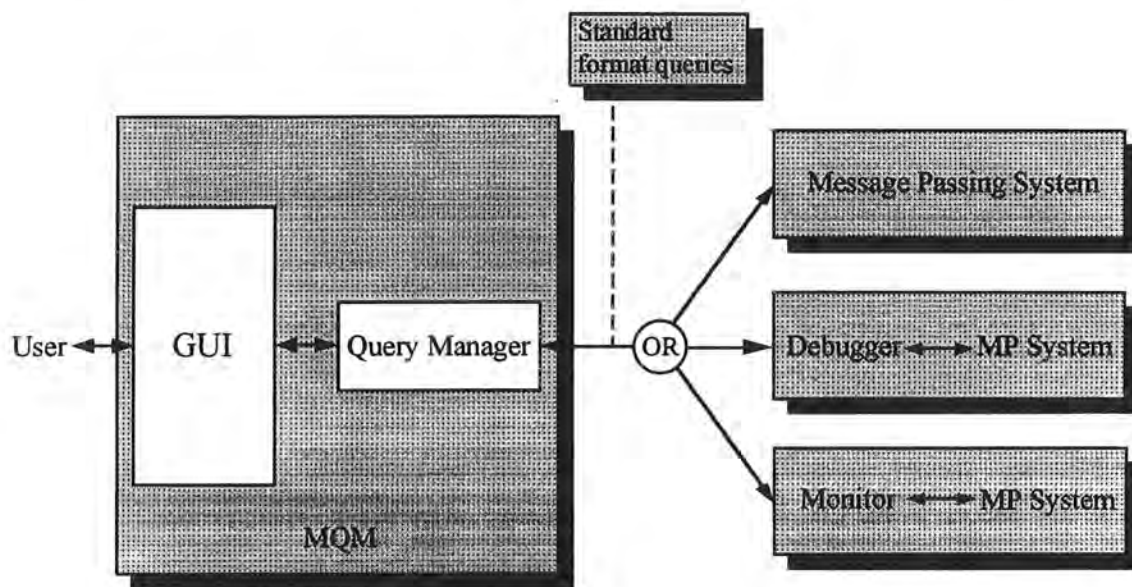


Figure 5.1 Structure of MQM

Figure 5.1 shows the structure of MQM and how it obtains message queue data. The rectangles on the right represent the kinds of software that can service MQM. Message queue data is obtained from the software using standard format query routines, so that the interface layer can remain the same regardless of the system implementation. That is, MQM defines the format of the query routines, but does not enforce any specific low-level implementation. (The queries and their formats are listed in Appendix B.)

The following two subsections describe the implementation of MQM's graphical user interface and Query Manager. The third subsection shows the file structure of MQM's distribution package. A final subsection explains some of the key decisions in MQM's design and implementation.

5.1 Structure of the GUI

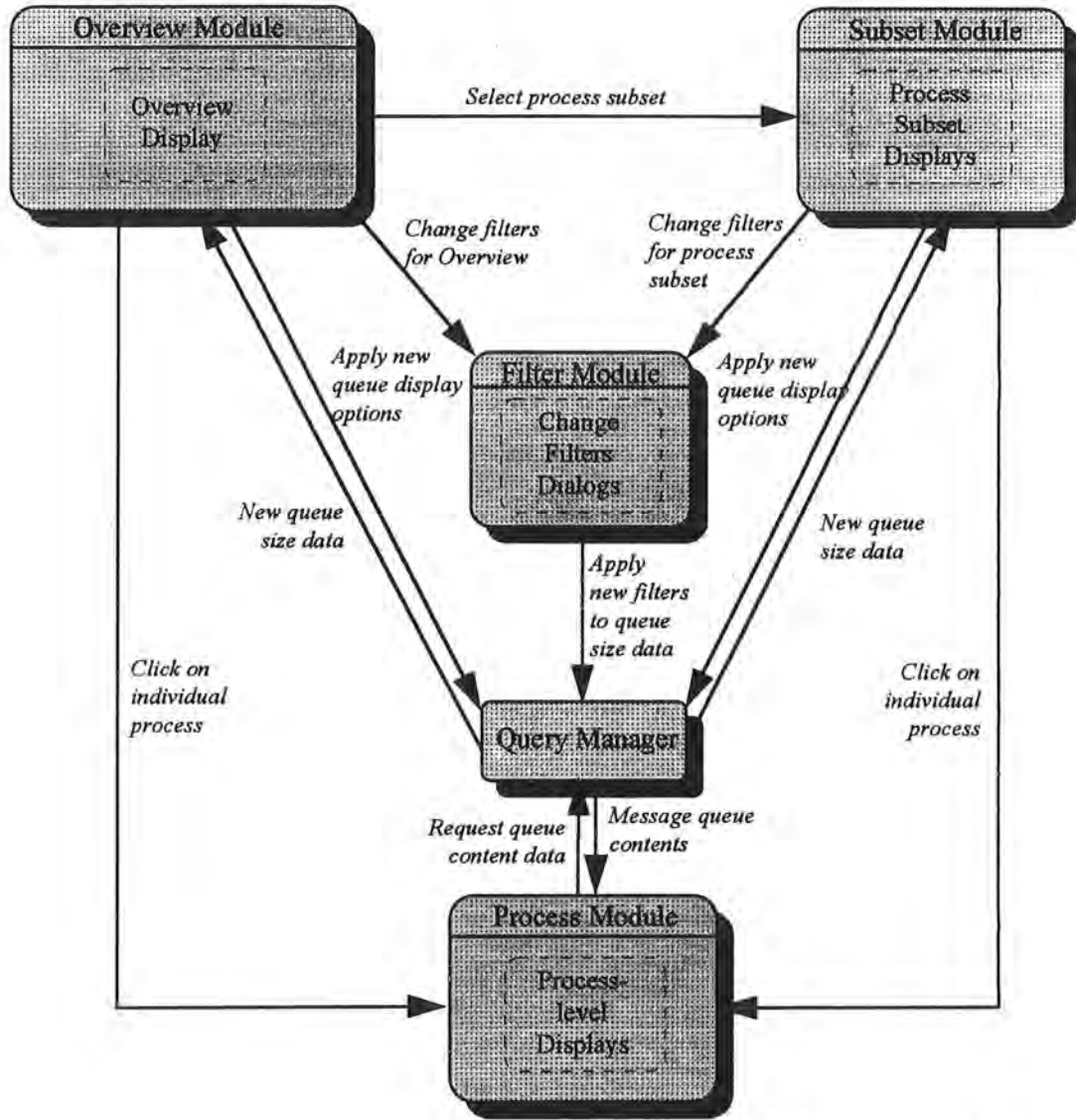
Upon invocation, the GUI presents the total size of each process's receive queues in the Overview display, subject to the default settings (exclude sends, messages pending receipt, and barriers). The user can then specify certain message operations to be included or excluded from the display. The user can also view just a subset of all processes, or the contents of individual process message queue. The GUI is implemented in Tcl/Tk. Figure 5.2 gives an overview of the structure of the GUI, showing the interactions between modules.

The *Overview Module* handles everything related to the Overview display, from the creation and displaying of the window to the handling of interactions between the user and the display. Each process is represented as a square on the screen, colored according to the number of pending message operations in the process's queues. This module also controls the *queue display option*, allowing the user to choose which types of message queues should be included in the total count. When the user changes the queue display option, (or when message filters change), the module requests the Query Manager for updated message queue sizes and re-colors the squares in the Overview accordingly.

The *Process Subset Module* is responsible for everything related to Process Subset displays, including the mechanism for selecting a subset of processes. Each subset display is similar to the Overview display, except that it is restricted to just some of the processes. The user can select a new subset from any existing process display. All functionality from the Overview Module applies to the Process Subset displays; in addition, the Process Subset Module is capable of controlling multiple displays.

The *Filter Module* controls the Change Filters dialog and activates the changes. The user can change any of three filters: message source, message destination, and message type. A change in a filter causes the Query Manager to re-count the pending message operations, eliminating any that don't match

the source/destination processes and types. Each process display (Overview and Process Subset) has its own set of filters; changes in the filters affect only the associated display. After the Query Manager re-counts the message queue sizes, this module will notify the corresponding process display to re-color the process squares using new size data.



** The Update Module is not shown here.*

Figure 5.2 Structure of the GUI

The *Process Module* is responsible for showing Process-level displays. It requests the complete contents of individual process's message queues from the Query Manager, and displays them by queue category. It also indicates whether the process is blocked and the operation that blocks it.

The *Update Module* (not shown in the figure) forces update of all existing displays. It is designed to be invoked only when the execution state (i.e., the current state of the messaging system) changes. It works in either *data-push* or *data-pull* mode to maximize flexibility of integration. In data-push mode, which we recommend, the underlying software forces an update of the displays whenever a change in execution state is detected. If MQM is integrated into an interactive debugger, for example, stopping program execution at a different point causes the debugger to notify the Query Manager that new data is available. The Query Manager then asks the Update Module to force update of all existing displays. In data-pull mode, on the other hand, the Update Module is activated by a user action, usually after execution has been stopped at a different point. This mode uses the mechanism described previously, where display modules actively request new information from the Query Manager in response to user actions.

5.2 Structure of the Query Manager

MQM presents two types of message queue data: sizes and contents. The Query Manger obtains information on content through the query operations. Different queries return the contents of different types of message queues (e.g., non-blocking send queue, message arrival queue, blocking receive queue, etc.). The results of these queries are lists of message descriptors, which consist of message sources, destinations, types, and lengths. The only exception is the barrier query, which returns a list of the processes that are blocked at barriers.

The Query Manager organizes the descriptors by process, using the data structure shown in Figure 5.3. One instance of the data structure is maintained for each type of message queue. Each record of the table has three entries: a process number, a query flag, and a pointer to a list of message descriptors. The query flag indicates whether the contents of the corresponding process's queue have been queried; it differentiates whether the process has not been queried, or has been queried but has no pending operations. The pointer itself references a two-dimensional linked list of message descriptors, corresponding to message operations initiated by this process. As shown in the rectangular box in Figure 5.3, descriptors for operations of the same message type form a list (horizontal links in the figure), with

the message type and the size of the list stored at the beginning of the list. The heads of these linked lists are linked to form the two-dimensional list (vertical links). With this data structure, there is no need to examine every descriptor's message type each time the message type filters are changed.

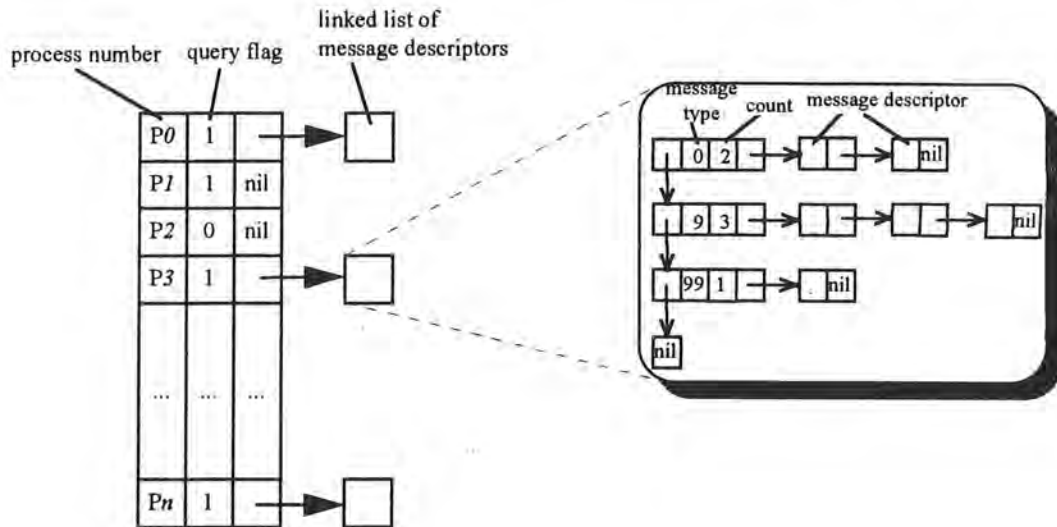


Figure 5.3 Data structure for message queue contents

The Query Manager maintains a separate table indicating the size of each message queue (see Figure 5.4). Anticipating that the queue display option is likely to be changed frequently, the table allows MQM to re-calculate the queue sizes without accessing the actual queue contents. The data in the table reflects the current filter settings. (Thus, it is necessary to access the content data structure whenever the message filters are changed.) The Query Manager keeps one table for the Overview and one for every Process Subset display, to allow different filter settings for each display.

	Blocking Send Count	Non-blocking Send Count	Pending Message Count	Blocking Receive Count	Non-blocking Receive Count	At Barriers
P1						
P2						
...						
Pn						

Figure 5.4 Data structure containing message queue lengths

When the Query Manager gets a request from the GUI, it firsts checks whether message queue size or content data is needed. If the latter, it accesses the message operation descriptors. If the query flags are not set for the processes whose queue contents are requested, the Query Manager issues a query to the low-level system, then stores the result in the content data structure and returns the requested data to the GUI. On the other hand, if queue size data are requested, the appropriate size table is accessed. If data is available in the table and the filters have not been changed, it will be returned immediately. Otherwise, the Query Manager must access the content data to calculate the sizes. These re-calculations are also required when the information in the table is no longer valid, such as when the underlying system has signaled a change in queue state. Once the size tables have up-to-date data, the Query Manager returns the requested sums to the GUI.

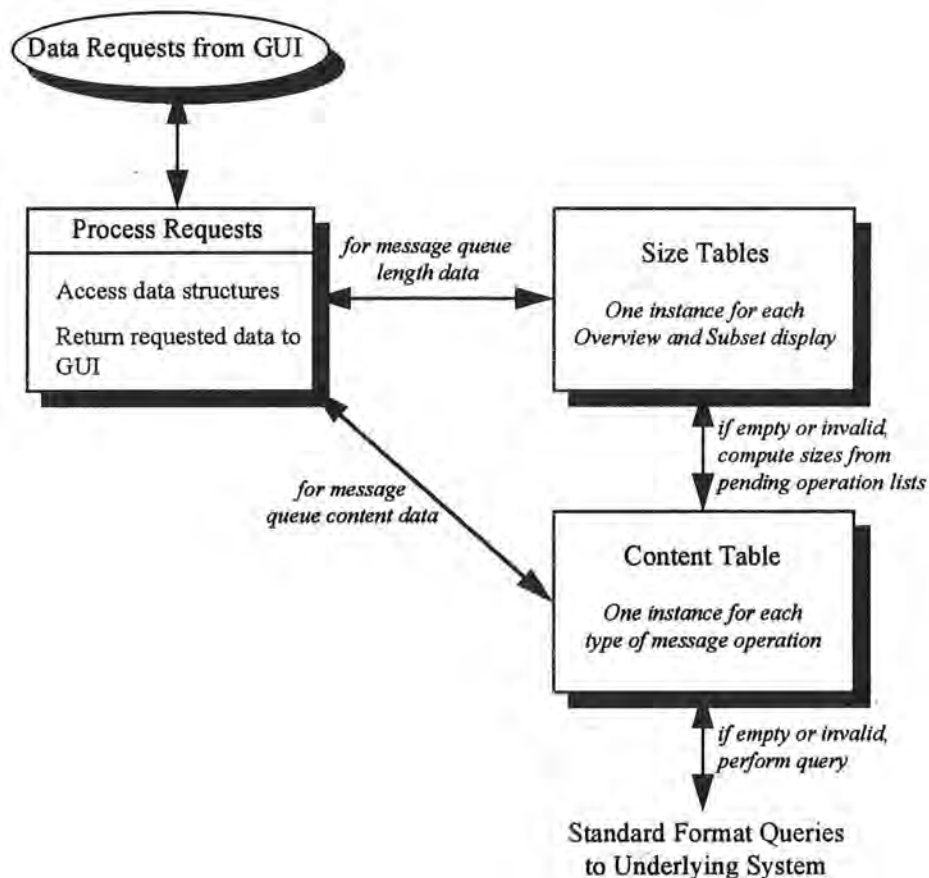


Figure 5.5 Structure of the Query Manager

5.3 File Structure of MQM

Figure 5.6 shows the file structure of MQM's distribution. The GUI and Query Manager occupy two sub-directories of the general directory, `mqm`; a third directory stores bitmap images. The `gui` sub-directory includes all the GUI modules and a few related files not mentioned in Section 5.1. File `mqm-defaults` sets the default values for display variables (such as colors, widget border widths, etc.). File `mqm_main` initiates the GUI. File `mqm_node`, `mqm_state`, and `mqm_utilities` contain utilities that are shared by the other modules. During implementation, a Tk widget called *OptionBox* was created to be used in Process Subset displays; it is located in the `gui` sub-directory, `OptionBox`. The `query_manager` sub-directory contains the C programs for the Query Manager. File `main.c` initiates the application, and defines the routines that bridge the GUI and Query Manager. These routines are implemented in `query.c`. File `utility.c` contains utility routines shared by other routines. Standard query routines are only used in these utility routines. Thus, `utility.c` is the file to modify to implement standard queries. Since MQM cannot run without an underlying system that responds to standard queries, the file also includes sample routines that generate random message queue data for demonstration purposes.

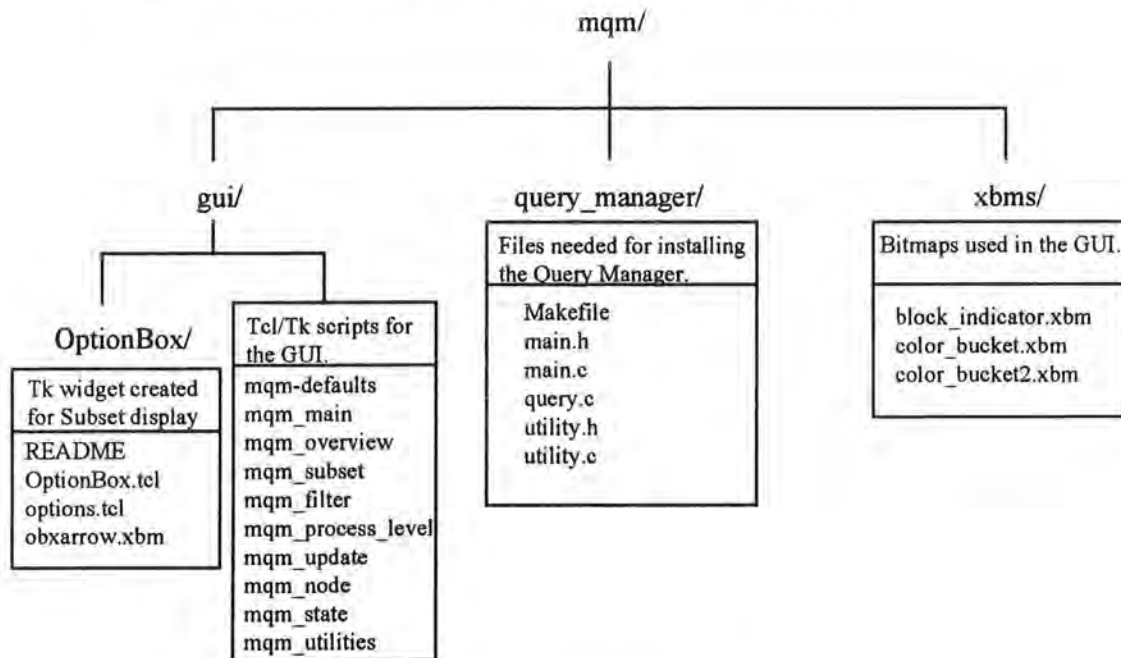


Figure 5.6 File structure of MQM

5.4 Design and Implementation Decisions

Each subsection below discusses a specific design decision encountered during development of MQM. It describes the rationale for the decision, and alternative designs that were considered in making the decision.

Multi-level data presentation. One of MQM's goals is to present message queue data in a reasonable amount of screen space, without sacrificing details. Since message queue data is potentially voluminous, the data presented has to be selective to fit the screen. To compensate for the loss of detail this entails, detailed information is shown separately at the user's request. In the Overview, MQM presents the total message queue size for each process. If this information does not satisfy the user, Process-level displays can be launched to show details of individual message queues. The requirement of selective data presentation also leads to filtering mechanisms which include the queue display option, message filters, and process subsets.

Using "queue display option" to reduce volume of data shown. With the queue display option, the user can include or exclude certain message queues when calculating the total queue size for each process. The Overview uses this concept to provide access to details without increasing the requirement for screen space. The option categories reflect the message passing taxonomy described in Section 1. Since the taxonomy is based on user experiences, the query display option is likely to make a good match with the user's mental model, and can be applied to any message passing system.

When to activate queue display option. The user clicks on a radio button to change the option setting. The change takes effect immediately. When the user wants to select or de-select multiple types of message queues, the Overview display must re-draw itself multiple times. We thought about allowing the user to make multiple changes on queue display options before actually applying the changes. This way, the Overview only needs to update itself once; it also eliminates wasted queries and re-displays caused by user mistakes. However, allowing multiple changes at once would require a sequence of user operations, as well as an extra mechanism in the interface. The user would have to choose the queue display option,

then click on some button to have the change take effect. It could confuse the user if he/she is not aware of the sequence (for example, the user may wonder why nothing appears to happen when he/she changes the option). Therefore, we abandoned the idea of allowing changing multiple queue display options at once to avoid complicating the interface.

Controlling three filters with one dialog. There are three independent message filters: source, destination, and message type. Originally, we had each filter in a separate dialog so that the user changed filters one at a time. It became obvious that this solution involved many windows, especially when each Process Subset window and the Overview display had their own sets of filters. Therefore, we combined the three filters in a single dialog. Although the resulting dialog became larger, fewer steps are needed to make changes to multiple filters. The user also remains fully aware of the settings of all three filters.

What message types to list in the filter dialog. Originally, the list of message types in this dialog showed all user-defined message types known to MQM (i.e., from any message queue). However, this approach was confusing, since the user could select/de-select a message type yet have no effect on the process display. This mechanism was changed to reflect the "current state" of queue display options or message source/destination filters change; only the message types associated with the current display state are listed. Although the new approach requires access to the message descriptor lists, it guarantees that the process display will change visibly when the new message type filter is applied.

Message guiding process subset selection. To select a subset of processes, the user must click or rubber-band the desired group processes in the previous display window. In an informal user test, many users did not understand how to choose the subset. It became obvious that this two-step operation is not easily understood. Thus, we decided to display a message that guides the user through the selection process. The message was initially displayed at the bottom of the main window (where the other messages are displayed), but it did not catch the users' attention. Consequently, we moved the message to the middle of the new window, where it was effective in catching attention.

Modality of process subset selection. When the user is selecting a process subset, MQM becomes modal -- that is, the user can do nothing else until the subset has been selected. The selection mode is important because clicking on a process under normal conditions will launch the Process-level display, but clicking on a process is also the most straightforward way of subsetting. Thus, modality distinguishes the different meanings of clicking. There are some disadvantages associated with the selection mode. First, more user operations are required (selection mode must be entered and exited). The mode also restricts the freedom of doing different operations in user preferred order, so mode reminder messages are definitely needed. Despite the disadvantages, we chose to use the mode so that process selection would be straightforward. Of course, visual reminders are used (such as de-sensitizing buttons that do not work in selection mode).

Changing radio button to selection buttons in Process Subset displays. The functionality of queue display options are available in Process Subset displays, too. However, the full set of radio buttons takes too much screen space proportionally, since the process grid in the subset window is usually far smaller than the one in the Overview. One alternative for radio buttons are Motif's "option buttons", but our informal user tests showed that the Motif version was confusing and unfamiliar to users. Thus, we decided to implement our own Tk widget (named option box) based on more the familiar selection button available on PCs.

Absence of a color scale in Process Subset display. The color scale is not repeated in Process Subset windows for three reasons. First, the color scale in the startup display applies to all displays. Second, omitting the scale saves screen space. Last, it helps to differentiates subset windows from the Overview window.

Process status indicator in popup window. The process status indicator in the Process-level display tells the user immediately if the process is blocked and which type of operation blocks it. Without the indicator, the user must look for the occurrence of a blocking operation in the queues, which might

require window scrolling. If the process is blocked at a barrier, this is the only way in this window the user knows it, since barrier operations are not queued.

6. CONCLUSIONS

Programming in message passing is error prone, since the programmer has to worry about not only the extra “dimension” of process interaction, but also low-level programming details. When an error occurs, the ability to examine the status of message-passing operations is essential for debugging. The same status information can also be useful for fine-tuning program performance. However, extensive status information is not available to the programmer, even with the help of current parallel debugging and tuning tools. None of the tools we surveyed focuses on providing such information. Moreover, most of them do not work on multiple hardware platforms or message passing systems; they also have trouble handling programs with hundreds of processes.

MQM is a simple tool that focuses on a single function. It provides comprehensive information on the status of pending message-passing operations. The queue sizes of each process are presented in an overview, where potentially problematic processes are color-highlighted. The user can zoom in on any processes for a closer examination of the contents of their message queues. This arrangement of different levels of details allows more processes to be displayed within a limited screen space, without sacrificing details. To provide more flexibility, MQM offers three categories of filtering mechanisms so that undesired information does not distract the user’s attention. Since users were involved throughout the tools development, MQM functionality directly addresses users’ needs and preferences. One substantiation of this is that a national body of users chose to include MQM as a recommended requirement of parallel computers [28].

MQM is designed to be incorporated into a run-time environment, and obtains message data from it. When integrated into multiple systems, its look and feel remain consistent across the platforms, and it becomes largely independent of the message passing systems.

With the help of Don Breazeal, Karla Callaghan, and Eric Richards at Intel’s SSD division, MQM has been successfully integrated into Intel’s Interactive Parallel Debugger (IPD). The main purpose of the integration was to prove that MQM works, and that the integration process itself is straightforward, as long as message queue data is accessible. At the most recent meeting of Parallel Tools

Consortium, Intel announced that they were going to formally integrate MQM into XIPD, the graphical version of IPD.

NASA Ames has also been very interested in integrating MQM into their AIMS monitoring system [22]. They performed preliminary trials to establish the feasibility of acquiring the needed information on message queues through the monitor. All problems appear to be resolvable. Jerry Yan (at Ames) recently stated that NASA were planned to begin the actual integration in summer, 1996.

6.1 Future Work

Several features can be added to MQM to enhance its functionality and usability. Users at Parallel Tools Consortium Third Annual Meeting (June 1996) suggested that it is desirable to be able to review earlier execution data, even after the fact. This could be easily achieved by allowing the user to save the message-passing status data in a file, and re-display the data later.

To provide more comprehensive message passing information, MQM needs to show the actual data carried by a message. Such data must be displayed in a format that can be easily understood. It could be added to Process-level display, where the contents of message queues are shown. For example, when an item from the send or pending message queue is pointed by the cursor, a small box containing message data could be shown next to the item.

In the original project proposal, MQM was supposed to provide graphical support for modification of message passing state, so that the programmer will be able to correct or disable the effects of erroneous message operations without modifying the source code, re-compiling, re-linking, and re-executing [3]. "Adding" messages dynamically is unrealistic due to constraints on buffer space and queuing mechanisms. However, user surveys showed that facility to "delete" operations from the message system would be satisfactory. This feature was not implemented in the current version of MQM, but it needs to be addressed. The graphical support is easy to implement, but the underlying system would have to support deletions from process buffer space.

To extend the usefulness of the tool, MQM needs to support group operations. This is not difficult to accommodate on the graphical interface, but the run-time system must be able to report status

information on those operations. At the present time, most run-time systems appear to be incapable of distinguishing a collective operation from a stream of point-to-point operations.

After MQM was completed, one thing we wish that we had done is to have fewer MQM-related windows. User responses to the Lightweight Corefile Browser (LCB) revealed that they do not like to work with many windows [20]. One possibility is to integrate the Overview display with the Process Subset displays, since they have similar functionality. The process grid of the subset could be shown in the main viewing area of the Overview display. The main viewing area could then have multiple layers, each showing one subset (or the entire set) of processes. A button could be created to control each layer, so clicking on a button would bring the corresponding layer to the top. The user might also be able to choose to view a subset in a separate window, like the subset window in the current version. Similar modifications would need to be made to the filter dialog so that filters for all process subsets could be controlled through one dialog.

Finally, if MQM's graphical interface is implemented in a compiled language, instead of Tcl/Tk, tool execution would be much faster. The implementation of some features (e.g., message filters) would also be simpler, since Tcl/Tk is not convenient for managing data.

BIBLIOGRAPHY

- [1] Ruth A. Ayt. An Informal Guide to Using Pablo. University of Illinois, Department of Computer Science, Urbana, Illinois, January 1995.
- [2] Adam Louis Beguelin. Xab: A Tool for Monitoring PVM Programs. In *Workshop on Heterogeneous Processing*, pages 92-97, Los Alamitos, California, April 1993.
- [3] Don Breazeal, Cherri Pancake, and Jerry Yan. March 8, 1994. MQM: A Tool for Graphical Management of Application Message Passing. The Parallel Tools Consortium [Online]. Available: <http://www-ptools.llnl.gov/projects/mqm/proposal.txt.html>.
- [4] Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming*. Jones and Bartlett Publishers, Boston, 1992.
- [5] CONVEX Computer Corporation. *CXtrace User's Guide for X2.1.0.4*. CONVEX Press, February 1994.
- [6] Janice Cuny, George Forman, Alfred Hough, Joydip Kundu, Calvin Lin, Lawrence Snyder, and David Stemple. The Ariadne Debugger: Scaleable Application of Event-Based Abstraction. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85-95, San Diego, California, May 1993.
- [7] Al Geist, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee, May, 1994. Updated version available at <http://www.epm.ornl.gov/pvm>.
- [8] William Gropp, Ewing, Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, Cambridge, Massachusetts, 1994.
- [9] Michael T. Heath and Jennifer Etheridge Finger. *ParaGraph: A Tool for Visualizing Performance of Parallel Programs*. Oak Ridge National Laboratory, June 1995.

- [10] Virginia Herrarte and Ewing Lusk. Studying Parallel Program Behavior with Upshot. Technical Report ANL-91/15, Argonne National Laboratory, Argonne IL, 1991.
- [11] Seema Hiranandani, Ken Kennedy, Chau-Wen Tseng, and Scott Warren. The D Editor: A New Interactive Parallel Programming Tool. In *Proceedings of Supercomputing '94*, pages 733-742, Washington DC, Nov. 14-18, 1994.
- [12] Anna Hondroudakes and Rob Procter. The Design of a Tool for Parallel Program Performance Analysis and Tuning. In K. M. Decker and R. M. Rehmman, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 321-332. Birkhauser Verlag, Basel, 1994.
- [13] Alfred A. Hough and Janice E. Cuny. Belvedere: Prototype of a Pattern-Oriented Debugger for Highly Parallel Computation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 735-738, University Park PA, 1987.
- [14] Intel Corporation. *Paragon Application Tools User's Guide*. Technical publication, order no. 312545-002, Intel Supercomputer Systems Division, Beaverton, Oregon, October, 1993.
- [15] Intel Corporation. *iPSC/860 C System Calls Reference Manual*. Technical publication, order no. 312233-001, Intel Supercomputer Systems Division, Beaverton, Oregon, March 1992.
- [16] Intel Corporation. *iPSC/860 Manual: Interactive Parallel Debugger Manual*. Technical publication, order No. 312353-002, Intel Supercomputer Systems Division, Beaverton, Oregon, April 1991.
- [17] Arie E. Kaufman. Visualization. *Computer*, 27(7)18-88, July, 1994.
- [18] Richard J. LeBlanc and Arnold D. Robbins. Event-Driven Monitoring of Distributed Programs. In *Proceedings of the Fifth International Conference on Distributed Computing Systems*, pages 515-522, Denver CO, May 1985.
- [19] Bruce P. Lester. *The Art of Parallel Programming*. Prentice Hall, Englewood Cliffs, New Jersey, 1993.
- [20] Manjunath Maddarange Gowda. Lightweight Corefile Browser. Technical report 94-80-17, Department of Computer Science, Oregon State University, Corvallis OR, 1995.

- [21] Charles E. McDowell and David P. Helmbold. Debugging Concurrent Programs. *ACM Computing Surveys*, 21(4):593-622, December 1989.
- [22] Pankaj Mehra, Sekhar Sarukkai, Melisa Schmidt, Cathy Schulbach, Jerry Yan. The Automated Instrumentation and Monitoring System (AIMS): Reference Manual version 2.2. Technical Report MS 269-3, NASA Ames Research Center, Moffett Field CA, May 1994.
- [23] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, June, 1995. Updated version available at <http://www.mcs.anl.gov/mpi>.
- [24] The MPI Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*, pages 878-883, Portland, Oregon, 1993.
- [25] nCube Corporation. *nCube 2 Programmer's Guide, v2.0*. Beaverton OR, December 1990.
- [26] Robert H. B. Netzer and Barton P. Miller. Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs. In *Proceedings of Supercomputing '92*, 81(2):502-511, Minneapolis, Minnesota, 1992.
- [27] Cherri M. Pancake. Improving the Usability of Numerical Software through User-Centered Design. In B. Ford and J. Rice, editors, *The Quality of Numerical Software: Assessment and Enhancement*. Chapman & Hall, 1996.
- [28] Cherri M. Pancake, Bruce Blaylock, and Robert Ferraro. Guidelines for Writing System Software and Tools Requirements for Parallel and Clustered Computers. Technical Report 95-80-11, Department of Computer Science, Oregon State University, Corvallis, OR, November 1995.
- [29] Cherri M. Pancake. Multithreaded Languages for Scientific and Technical Computing. In *Proceedings of the IEEE*, pages 288-304, February, 1993.
- [30] Cherri Pancake and Donna Bergmark. Do Parallel Languages Respond to the Needs of Scientific Programmers? *Computer*, pages 12-23, December 1990.
- [31] Cherri M. Pancake. Graphical Support for Parallel Debugging. In *Software for Parallel Computation*, pages 216-228. Springer-Verlag, Berlin, 1989.
- [32] Michael J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, 1994.

- [33] Daniel A. Reed, Ruth A. Aydt, Tara M. Madhyastha, Roger J. Noe, Keith A. Shields, Bradely W. Schwartz. An Overview of the Pablo Performance Analysis Environment. University of Illinois. Department of Computer Science, Urbana, Illinois, 1992.
- [34] B. Ries, R. Anderson, W. Auld, D. Breazeal, K. Callaghan, E. Richards, and W. Smith. The Paragon Performance Monitoring Environment. In *The Proceedings of Supercomputing '93*, pages 850-859, Portland, Oregon, November 1993.
- [35] Sue Utter-Honig and Cherri M. Pancake. Advances in Parallel Debuggers: New Approaches to Visualization. In Harry W. Tyrer, editor, *Advances in Distributed and Parallel Processing Volume One: System Paradigms and Methods*, pages 35-70, Ablex Publishing, Norwood, N.J., 1994.
- [36] Eric F. Van De Velde. *Concurrent Scientific Computing*. Springer-Verlag, New York, 1994.

Appendix A CONCEPTUAL MESSAGE OPERATIONS

Even though each message passing system has its own variations of message operations, they fall into a series of conceptual operations, as demonstrated here. Figure 1.1 (in the text) diagrams the interrelationships among them. Section A.1 explains the conceptual operations. The message routines provided by PVM, NX, and MPI are categorized in Section A.2 to provide examples of how arbitrary message passing libraries can be mapped to them.

A.1 Conceptual Categories

Point-to-point communication

<i>Operation Categories</i>	<i>Explanations</i>
Blocking Acknowledged Send	Control returns to invoker once send buffer can be reused; completion of send operation indicates that matching receive operation has at least started to receive message.
Blocking Unacknowledged Send	Returns once send buffer can be reused; completion of send operation gives no indication that matching receive has started.
Non-blocking Acknowledged Send	Will return before data is copied out of the send buffer. Program must query status before re-using buffer; status of "complete" indicates that matching receive operation has at least started to receive the message.
Non-blocking Unacknowledged Send	Will return before data is copied out of send buffer. Must query status before re-using buffer; status of "complete" gives no indication that matching receive has started.
Blocking Receive	Returns once message has been copied into receive buffer.
Non-blocking Receive	Will return before a message has been stored into the receive buffer; must query status before using buffer data.
Exchange	Combines send and receive; so the two processes exchange data.

Collective communication

<i>Operation Categories</i>	<i>Explanations</i>
Broadcast	Sends a message to multiple processes or all processes.
Barrier	No process can proceed until all processes have arrived at barrier; hence all processes are synchronized.
Gather-to-root	Collects data from all processes and stores at one (root) process.
Gather-to-all	Same as <i>Gather-to-root</i> , except collected data is sent to all processes.
Scatter	Sub-divides data on root process and sends distinct subset to each process in the group.
Reduce-to-root	Performs a global function on local data across all processes, and stores results on root process.

<i>Operation Categories</i>	<i>Explanations</i>
Reduce-to-all	Same as <i>reduce-to-root</i> , except results are sent to all processes.
Shift	Each process sends data to another process, and receives data from a third process.
Shuffle	Each process sends distinct data to every process in the group and receives data from all others.
Vector Reduce-scatter	Performs a vector reduction, then scatters the result to all processes.
Prefix Reduction	Performs a reduce function across data of all “lower ranked” processes and stores at next-higher process.

A.2 Examples from Three Message Passing Systems

Point-to-point communication

Operations	<i>PVM [7]</i>	<i>NX [15]</i>	<i>MPI [23]</i>
Blocking Acknowledged Send			mpi_ssend
Blocking Unacknowledged Send	pvm_send ^b pvm_psend	csend	mpi_send ^c mpi_bsend mpi_rsend ^d
Non-blocking Acknowledged send			mpi_issend mpi_start ^e mpi_start_all ^e
Non-blocking Unacknowledged Send		isend hsend ^f	mpi_isend ^c mpi_ibsend mpi_irsend ^d mpi_start ^e mpi_start_all ^e
Blocking receive	pvm_rcv ^b pvm_precv	crecv	mpi_rcv
Non-blocking receive	pvm_nrcv ^g pvm_trecv ^h	irecv hrecv ^f	mpi_irecv mpi_start ^e mpi_start_all ^e
Exchange		csendrecv ⁱ isendrecv ^{ij} hsendrecv ^{ij}	mpi_sendrecv ^k mpi_sendrecv_replace ^{k,l}

^b Must pack data into active send buffer or unpack from receive buffer separately.

^c Message may or may not be buffered.

^d Not guaranteed to be successful if matching receive is not posted before send.

^e Used in conjunction with an initialization routine (mpi_send_init, mpi_bsend_init, mpi_ssend_init, mpi_rsend_init, mpi_rcv_init).

^f Executes a user-written exception handler upon completion.

^g Receives message if already arrived; returns otherwise. Multiple calls maybe needed to receive message.

^h Receives message if already arrived; otherwise, returns when times out. Multiple calls may be needed to receive message.

ⁱ Sends a message to another process and receives a message from it.

Collective communication

Operations	<i>PVM</i>	<i>NX</i>	<i>MPI</i>
Broadcast	pvm_bcast pvm_mcast	csend ^m hsend ^{l,m} isend ^m gsendx ⁿ	mpi_bcast
Barrier	pvm_barrier	gsync setiomode ^o	mpi_barrier
Gather-to-root			mpi_gather mpi_gatherv ^p
Gather-to-all		gcol ^{p,q} gcolx ^p	mpi_allgather mpi_allgatherv ^p
Scatter			mpi_scatter mpi_scatterv ^p
Reduce-to-root	pvm_reduce		mpi_reduce
Reduce-to-all		gxand gxor gxhigh gxlow gxprod gxsum gxxor gopf	mpi_allreduce
Shuffle			mpi_alltoall mpi_alltoallv ^p
Vector Reduce-scatter			mpi_reduce_scatter ^p
Prefix Reduction			mpi_scan

^j Non-blocking send and receive.

^k It is an exchange operation when only two processes are involved.

^l Same buffer is used for both send and receive.

^m Sends a message to the processes with the same process id on every processor when a negative processor number is specified.

ⁿ Sends vector only.

^o Performs a global synchronization when sets the I/O mode.

^p Allows collection and/or distribution of varying amount of data from and/or to each process.

^q Used when the sizes of data to be gathered on some processes is unknown.

Appendix B STANDARD QUERY FORMAT

This appendix lists all query routines used by the current version of MQM. Information on how to invoke the Update Module is included for implementing "data-push" update mode (*active updating*) when execution state changes (see Section 5.1). Also included in this appendix is the definition of data types used by those query routines.

B.1 Query Routine Format

All queries return an integer status code. Arguments marked "IN" are used but not updated by the query routines. Arguments marked "OUT" are updated. Arguments marked "INOUT" are both used and updated. The list does not include gather, scatter, reduction, or more complex collective operations, since these operations are not supported by the current version of MQM. Note that we attempt to distinguish for which processes a query is meaningful; the message operations inside brackets are qualified by "src" and "dest" to indicate to which participant the query applies.

1. MQM_GetProcessList

Returns process count and a list of all processes in the application.

Arguments: OUT: MQM_process_list_size
 MQM_process_list

- | | | |
|----|----------------------------|---|
| 2. | MQM_WaitingMsgs | [dest: all send and combined send-receive operations] |
| 3. | MQM_IncompleteSends | [src: non-blocking sends] |
| 4. | MQM_WaitingSends | [src: blocking sends] |
| 5. | MQM_IncompleteRecvs | [dest: non-blocking receives] |
| 6. | MQM_WaitingRecvs | [dest: blocking receives] |

Returns count and list of descriptors for all messages satisfying the appropriate condition:

WaitingMsgs - messages queued to be received;
IncompleteSends - messages being sent by non-blocking operations;
WaitingSends - messages being sent by blocking operations;
IncompleteRecvs - receives posted by non-blocking operations;
WaitingRecvs - receives posted by blocking operations;

The queries can be restricted to a subset of processes, by specifying their count/list in the first two arguments. Otherwise, MQM_ALL is used as the first argument to signify all processes, while a null pointer can be used for the second argument. If the OUT arguments return as MQM_NONE and MQM_NULL_LIST, respectively, no processes were found to satisfy the query.

Arguments: IN: MQM_process_list_size
MQM_process_list
OUT: MQM_desc_list_size
MQM_desc_list

7. **MQM_BarrierArrived** [barriers]

Returns number and list of processes that are currently blocked at any barrier. Can be restricted to a process subset as in (2)-(6).

Arguments: IN: MQM_process_list_size
MQM_process_list
OUT: MQM_process_list_size
MQM_process_list

B.2 Definition of Data Types

A. Pre-defined constants

```
#define MQM_NONE 0  
Constant indicating that a subsequent list argument is empty  
  
#define MQM_ALL -1  
Wild card constant indicating "all possible elements"  
  
#define MQM_NULL_LIST NULL  
Null pointer for an empty list
```

B. Processes

```
char * MQM_process  
A process id  
  
MQM_process * MQM_process_list  
Array of MQM_process  
  
int MQM_process_list_size  
Number of MQM_process in MQM_process_list
```

C. User-defined Message Tags

```
int MQM_msgtag  
User-defined message tag  
  
MQM_msgtag * MQM_msgtag_list  
Array of MQM_msgtag  
  
int MQM_msgtag_list_size  
Number of MQM_msgtag in MQM_msgtag_list.
```

D. Message Descriptor


```

typedef struct {  MQM_process      src
                 MQM_process      dest
                 MQM_msgtag        tag
                 MQM_msg_size      size
                 } MQM_msg_desc

```

Message descriptor indicating source, destination, tag, and the size of the messages. Fields can be 0 or NULL if the element doesn't apply to a particular message operation. Wild card constant can be applied to src, dest, and tag.

```

MQM_msg_desc * MQM_desc_list
    Array of MQM_msg_desc

```

```

int MQM_desc_list_size
    Number of MQM_msg_desc in MQM_desc_list

```

B.3 Active Updating Mechanism

With active updating, whenever a change in the state of one or more message queues is detected (e.g., at quiescent moments under control of an interactive debugger), the underlying software can force MQM to update its displays.

The update is handled by the Update Module as long as the ACTIVE_UPDATE variable in file `gui/mqm_main` has been set to 1 (the default value). Then, any call to the C routine `MQM_ActiveUpdate` (in file `query_manager/main.c`) will cause the Update Module to perform the display update.