IPPM: INTERACTIVE PARALLEL PROGRAM MONITOR

Robert Craig Brandis
B.A., University of Washington, 1979

A thesis submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science & Engineering

August, 1986

The thesis "IPPM: Interactive Parallel Program Monitor" by Robert Craig Brandis has been examined and approved by the following Examination Committee:

Shreekant Thakkar
Adjunct Professor
Thesis Research Advisor

Richard Kieburtz
Professor

Dan Hammerstrom
Associate Professor

Dick Hamlet
Professor

ACKNOWLEDGEMENTS

Dr. Shreekant Thakkar provided encouragement, enthusiasm and guidance for this work. His help has been greatly appreciated. Mark Grossman and Dick Hamlet helped discuss and refine some of the concepts and are both good eggs. Moggy Vanderkin helped edit the paper and put up with me during the process. I would also like to thank Dr. Dick Kieburtz and Dr. Dan Hammerstrom for contributing some key ideas.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

The tasks associated with designing and implementing parallel programs involve effectively partitioning the problem, defining an efficient control strategy and mapping the design to a particular system. The task then becomes one of analyzing the program for correctness and stepwise refinement of its performance. New tools are needed to assist the programmer with these last two stages. Metrics and methods of instrumentation are needed to help with behavior analysis (debugging) and performance analysis. First, current tools and analysis methods are reviewed, and then a set of models is proposed for analyzing parallel programs. The design of IPPM, based on these models, is then presented. IPPM is an interactive, parallel program monitor for the Intel iPSC. It gives a post-mortem view of an iPSC program based on a script of events collected during execution. A user can observe changes in program state and synchronization, select statistics, interactively filter events and time critical sequences.

Efficient multi-processing tools requires new methods of problem decomposition - and new debugging and analysis tools for developing and evaluating parallel programs[Mapl85].

1. INTRODUCTION

Writing good, working parallel programs is difficult. The common attitude that concurrent, multi-process programs are more difficult to write than single-process programs results partly from a lack of appropriate tools. The tools for initially constructing parallel programs, compilers and languages that support concurrency, are generally available. There are fewer tools available for debugging and analyzing the performance of parallel programs. Complete parallel programming and instrumentation environments are fewer still.

Parallel programs involve a new level of complexity. The ease with which a programmer understands and deals with this complexity depends on the ability of a system to supply necessary information about the behavior and performance of a program. Traditional performance and debugging tools do not provide enough information to deal with the complexity of distributed systems. A new class of high level tools is needed to facilitate efficient parallel programming.

The primary goal is to provide some general methods for instrumenting and analyzing parallel programs. To support this goal, some intermediate goals

must be met. A model of parallel computation is needed as well as methods of collecting and organizing data gathered from executing programs. The methods should be applicable to a wide range of systems and be easily implemented. Tools can then be built that are based on these methods. The tools should provide a high level, consistent view of parallel computations.

1.1. Definitions

Parallel architectures are classified as SIMD machines and MIMD machines. This work considers only algorithms for MIMD machines. Parallel algorithms for multi-processors have been classified into synchronous and asynchronous algorithms [Kung76]. In synchronous algorithms there exist processes such that some stage of the process is not activated until another process has finished a certain portion of its program. The needed timing is achieved by using various synchronization primitives. In asynchronous algorithms there is a set of global variables accessible to all processes. To insure correctness, the operations on global variables are often programmed as critical sections. Communication between processes is achieved through the global variables, or shared data. Asynchronous algorithms do not specifically wait for inputs. However they may be blocked when attempting to enter a critical section. The term "parallel algorithm" as used here applies to both synchronous and asynchronous algorithms.

2. SURVEY OF RELATED WORK

Three general categories, performance monitoring, debugging, and visual simulation, each contribute to a user's understanding of distributed programs. While usually treated as distinct fields there is definitely overlap. This is especially true of debugging and performance monitoring. Debuggers collect data to assist a programmer in insuring "correct" execution of a program. In doing so they may provide insights into performance improvements. Performance monitors may also be viewed as "performance debuggers" [Seg85]. Visual simulation improves the user's intuitive understanding of a program's behavior by using graphics to deal with the complexities of synchronization and distributed state.

2.1. Performance Monitoring

A significant early work on monitoring distributed programs was the METRIC system [McDan75]. METRIC is based on a model of programs that communicate over a local network. It is divided into three parts: *probes*, *accountants* and *analysts*. Probes are procedure calls used by the programmer to generate trace data. A probe inserted into the program source code sends data over the network to an accountant. Accountants record or filter incoming data and may reside on different machines. Analysts are collections of processes which summarize the trace data. The explicit, modular separation of data generation, selection and analysis in the design of METRIC was adopted by many later measurement tools.

A number of research efforts have explored ways to organize event data and have borrowed ideas from databases and predicate logic for use in selecting or characterizing significant events. Snodgrass uses a relational database in his distributed systems monitor [Snod82]. He defines events as objects and stores them in the database. In his model of computation, work is performed by operations on objects. Events are recovered from the database using a query language that incorporates the knowledge of time.

Bates and Wileden introduce the idea of *behavior abstraction* as a way of classifying and evaluating traces of events from the execution of a distributed program [Bates83]. The basic idea is to be able to identify a higher level abstraction from the event traces by using an *event description language*. Their intent is to formally specify a description of the execution of a program and compare it with the actual trace data. The proposal mentioned the ability to specify events hierarchically, although only a single level of event grouping was implemented. This approach introduced the need for a correct program specification.

Another approach to performance monitoring uses special hardware for data collection. A good example is the real-time event monitor built for the Stony Brook multi-computer [Kie83]. The multi-computer was instrumented to monitor user-defined events with a high degree of temporal resolution. The problem of observing the global state of a distributed system from within the system was solved by using a fast microprocessor to monitor events occurring on slower processors. By multi-plexing observation channels rapidly the monitor

can take real-time "snapshots" of part of the system state. The event data is collected in an event register, time stamped and written to a file for later analysis. A software analysis package uses the trace data to develop statistics on cpu utilization and expected wait times. The hardware-monitoring approach has the advantage of being non-intrusive, although the amount of state information special hardware is able to capture is relatively small.

Maples developed a system with hardware support for evaluating parallel programs when run on different processor topologies [Mapl83]. The MIDAS (Modular Interactive Data Analysis System) system is reconfigurable into topologies like meshes, rings and trees. MIDAS is instrumented to show when the CPUs are processing or waiting. An important feature is the ability for performance information to be fed back to the system, enabling some simple dynamic load balancing.

The distributed programs monitor (DPM) is an attempt to build a high-level, general-purpose monitor based on a model of computation by communicating processes [Mill85]. DPM views program behavior only through message communication. Interprocess communication is monitored by the operating system kernel. Messages generate traces which are sent to a filter process. The filter decides to record or discard the trace based on a user specification. Analysis of the trace data is performed by a set of tools which tabulate communication statistics. Miller introduces the idea of *paths of causality* in a distributed program that can be characterized as servers. The history of message communication within a program forms an acyclic graph. Sequences of

messages form paths through the graph. The frequency of particular sequences can be seen as branching probabilities. For example, DPM can be used to calculate the conditional probability that, given a specific input to process A, it will communicate with process C. This is a useful metric for simulations. Miller also introduces a measure for parallelism he calls the "P" factor. It is similar to the definition of efficiency as discussed by Kuck [Kuck78].

Performance monitoring and instrumentation is beginning to be seen as an integral part of parallel programming environments. Segall and Rudolph's PIE system (Programming and Instrumentation for Parallel Processing) makes performance its central issue [Seg85]:

> *Often the only reason for developing a parallel program is for real-time performance...the difficult task of performance debugging in the context of a rudimentary program environment requires an even more specialized and highly knowledgeable programmer.*

The PIE environment has three parts: a metalanguage, a program constructor and an implementation assistant. A user writes programs in the metalanguage *MP* using the program-constructor editor. Control-structure templates provided by the implementation assistant can help provide a skeleton for program control. The templates use paradigms like master-slave, recursive master-slave, and pipelined control. MP programs may be instrumented by software *sensors* embedded in the program source code. Sensors can be used to determine the time consumed by an activity, the time required to execute a block of code and the values of local variables. They can also be used to track events defined by the user. Development time information along with runtime

data is stored in a relational database. A graphic representation system extracts and presents views of the data that may be relevant to specific performance or debugging goals.

2.2. Debugging

The development of distributed debugging has evolved by extending single-process debugging techniques. An example of an early distributed debugger is the Tymshare system [Vick76]. Vickers designed a system that allowed debugging of programs written in different languages and running on different machines. The Tymshare debugger consists of modules that reside on different contributing machines. Users can set breakpoints, start and stop processes, and examine and modify the contents of local memory. Interactions between processes were not considered.

Smith's debugger for message-based communicating processes considers only interprocess activities [Smith81]. Smith uses a model of computation involving trace data for events occurring at "border crossings." A border crossing corresponds to a message being sent or received at a process "port". A user can create and manipulate interprocess events or specify how events are to be handled automatically by the use of *demons*. Demons execute when a control predicate based on a sequence of events becomes true. They can help the user manage a potentially large volume of event data.

An important problem in distributed debugging is the effect of the debugger on the executing program. Debuggers can introduce delays which can

alter a program's behavior. Schiffenbauer uses *logical clocks* rather than actual clocks to assure that his debugger simulates a valid execution of a distributed program [Schiff81].

LeBlanc and Robbin's debugger introduces animation as a tool in understanding the behavior of a distributed program [LeBl85]. They view programs at the level of process interactions. Events collected during execution become input to a graphic replay system. The programmer controls the speed of the replay and has access to the contents of messages.

The interactive distributed debugger (IDD) combines single-process debugging techniques, process interaction monitoring, an assertion language based on temporal logic and a graphic display into a multi-level debugging system [Hart85]. Harter proposes a strategy for distributed debugging based on incremental assertion generation. The assertion language is used to address program correctness. Instead of attempting to specify the exact program state at a desired breakpoint, Harter specifies a sequence of event classes and invariant conditions that should hold over intervals of program execution.

2.3. Simulation

Simulation is a method of creating and understanding distributed programs in a simple and controlled environment. The complexities of the implementation system are eliminated in order to better understand the nature of the specific problem being simulated. Many simulation tools use formalisms such as queueing models or graph models as a basis for simulating distributed

systems. Vernon's simulator uses the UCLA graph model of behavior [Vern83]. The graph model of behavior (GMB) is designed to model contention for resources, control-flow, data flow and data transformation. Computations are defined by control graphs consisting of nodes representing events and arcs that model precedence constraints among events. The computation defined by the graph is carried out by a *token machine*. Control-flow analysis on the model can detect pathologies like deadlocks and unbounded graphs. Performance estimates can also be derived from the graph.

The Performance Analysis Workstation models programs as queueing networks [Mel85]. Melamed and Morris's work is important because of its extensive use of animation. Melamed notes that queueing networks have distinct visual aspects: the dynamic flow of transactions in a network, the topology of the nodes and the temporal evolution of numerical and graphical statistics. Users of the Performance Analysis Workstation design a system by first drawing its topology on a CRT with a mouse. Numerical parameters are added from the keyboard and the model is instrumented by defining statistics to be displayed in windows. The model is then debugged by tracing its operation during animated simulation runs. During execution the flow of data messages and dynamic changes in queues and statistics are displayed. Melamed makes the point that simulations usually provide aggregate statistics at the end of a batch run. For many problems, like Monte Carlo simulations, verification is difficult using this method because short term randomness can mask an incorrect program. The evolution of statistics over the life of the

simulation can help in detecting these problems.

2.4. Summary

Each approach, debugging, performance monitoring and simulation has advantages and limitations. Performance monitors must collect data about a program's behavior and present it in a meaningful fashion. Data must be presented in such a way that the user is not overwhelmed. Methods for filtering and presenting different views of event data are necessary. Bates and Wileden's idea of behavior abstraction seems like a useful way defining levels of viewing distributed programs. Data storage and retrieval by using relational databases is another promising method but it introduces the potential problem of not being able to retrieve large amounts of data quickly. Relational databases tend to be slow with current technology.

The debugging approach deals with the same problems of data collection and display. It affords the user interactive control over the executing program at the expense of added complexity. Distributed debuggers must solve the problem of introducing delays into a computation that can alter its behavior. The higher level debuggers that view only process interactions have an intuitive appeal. Harter's debugger is important because it introduces graphic animation and execution history scrolling in a debugger.

Simulation tools seem best suited to particular kinds of problems. Simulation offers formalisms that can be used to determine both program correctness and performance. One problem with simulations is their ability to model

complicated computations at an acceptable level of detail. The work by Melamed and Morris is significant in its extensive use of animation to help user's visualize the dynamic behavior of queueing systems.

3. PERFORMANCE MEASUREMENT AND A GENERAL MODEL OF ANALYSIS

The task of understanding the behavior and performance of a parallel program poses different problems than for a sequential program. Measuring the performance of a sequential program involves well known metrics like page fault rate and counts of subroutine calls. There are no agreed upon standard metrics for characterizing the performance of parallel programs. The state of a reentrant sequential program can be characterized by the value of its program counter and a memory image of its data. The state of a parallel program requires information for each of its component processes, however, capturing distributed state information can be difficult. It is not possible to capture the global state from any point within a system when the message delays are sufficiently long to allow the processes involved to change state. One way of dealing with this problem is to view the system in terms of its activity rather than its state. If parallel programs are to function as a single program then the components have to communicate. The correctness of the program depends on the content and sequence of its messages. Program performance can be characterized by message frequency, distribution, time density and delay. Behavior abstraction can provide a general method for analyzing both behavior and performance.

3.1. Metrics

The performance of parallel programs is usually expressed in terms of the speed-up achieved over a sequential version of the same program. Consider a synchronous algorithm with k processes. Assume that it is run on a system of k identical processors and takes time U. During time U let u_i represent the total time that i processes are active ($k-i$ processes are blocked). Note that $U = u_0 + u_1 + \cdots + u_k$ and that the algorithm can be run on a single-processor of the system in at most time

$$\sum_{i=1}^{k} i u_i$$

Therefore, by using k processors, the algorithm can be sped up by a factor of

$$S_k = \frac{\sum_{i=1}^{k} i u_i}{\sum_{i=1}^{k} u_i}$$

The efficiency of an execution can then be defined as

$$E = \frac{S_k}{k}$$

3.2. A Model for Behavior and Performance Analysis

A general model for behavior and performance analysis, based on the idea of behavior abstraction, offers a method of instrumenting and analyzing parallel programs. A system based on this model would present views of data collected during program execution. The views and level of detail would depend on the current goals of the user. Such a system could fill the roles of

both a distributed debugger and a performance-analysis tool.

The general model presented here is really four models: computation, metering, analysis and display.

3.2.1. Computation model

The computation model states that processes compute and communicate. Communication is defined as causally related events occurring between processes participating in a computation. Communication may be implemented by copying part of the data space of a process into that of another process (message passing), or by access to a shared data space. Processes consist of internal events and external events. An internal event is one that is invisible from outside a process. An example of an internal event is the execution of an assignment statement.

External events are visible outside a process. Examples of external events are communication, system calls and process termination.

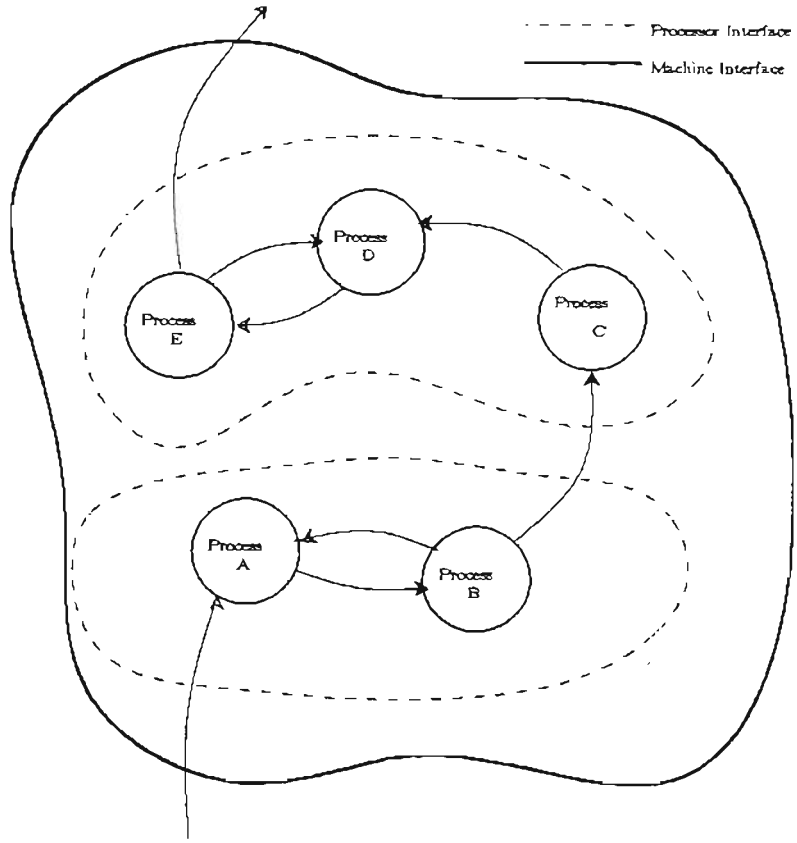


Fig 3.1 A Parallel Computation

3.2.2. Metering model

Parallel computations consist of a set of interacting processes. A process is divided into *stages* by *points of interaction* when it communicates with other processes. A process may block at the end of a stage while attempting to communicate. An execution of a process can be viewed as a state machine where states are defined as either *stages* or the wait state. External events cause transitions between states. These events can be *metered*. It is not important how metering is implemented as long as its effect on program behavior and

performance is negligible. Metering means being able to record trace data regarding external events including the time that the event occurred. This model assumes that a set of events can be selected to be metered and that it is possible to specify, to some extent, the type of data to be collected. The metering model also includes the idea of filtering traces in order to avoid overwhelming the user and the collection system.

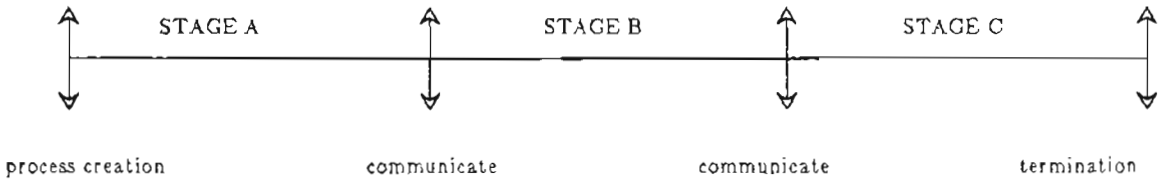


Fig 3.2 Stages of a Process

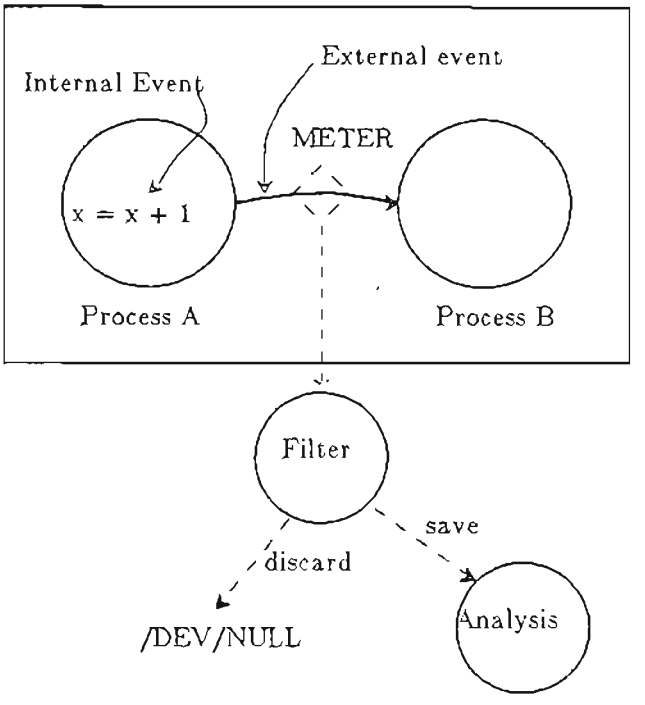


Fig 3.3 Events and the Metering Model

3.2.3. Analysis model

The trace data from each process that passes the filter provides a post-mortem view of the history of a program execution. Analysis of the data depends on the goals and views desired by the programmer. If the goal is analysis of behavior (debugging) then the sequence, type and content of the events is needed. For example, in a message-based system, the user would monitor the progress of a program until an error is found and then explore the types, sources, destinations and contents of relevant messages. An advantage of this approach to distributed debugging is that it avoids the problem of intervening in a running program and possibly introducing delays that can alter

program behavior. If the goal of the programmer is measuring performance and understanding factors that affect it, then a different view of the data is required. Performance statistics that characterize the execution of the program can be derived from the traces. Examples of these statistics are:

Event frequency (time density)

Distribution of communication among processes

Distribution of event types

Communication delay

Time spent waiting to communicate (idle time)

Lengths of communication queues

External events can also be used to determine the execution time of a parallel program. This is defined as the time at which the last process finishes. The elapsed time of a process is then defined as the sum of three quantities [Kung76]:

1. Compute time:

   This is the total processing time consumed by the *stages* of a process.

2. Blocked time:

   A process may be blocked at the end of a stage because it waits for messages in a synchronous algorithm, or to enter a critical section in an asynchronous algorithm. The blocking time of a process is the total time that the process is blocked.

3. Execution time of synchronization primitives:

   Synchronization primitives are needed for synchronizing processes and implementing critical sections.

3.2.4. The Order of Events in Distributed Systems

The analysis model requires that the order of causally related events be derivable from the traces. Because a single-process is executed sequentially, the events in it are totally ordered. In some systems with no global clock and no common memory it is may be impossible to tell which of two events occurred first. In these systems external events between multiple concurrent processes only define a partial ordering. Consider the *happened before* relation (denoted ->) defined on a set of events [Lam78]:

1. If A and B are events in the same process, and A was executed before B, then A -> B.

2. If A is the event of sending a message by one process and B is the event of receiving that message by another process, then A -> B.

3. If A -> B and B -> C then A -> C.

If two events are not related by the -> relation, then they are said to be *concurrent*. In order to determine the *happened before* relation using physical clocks, we either need a global clock or perfectly synchronized local clocks. It is possible to synchronize local clocks to a within a tolerance related to the minimum message delay [Lam78]. It requires that processes send messages to all other processes. Another way to insure the *happened-before* relation that

does not involve this system overhead is to define a *logical clock* for each process. A logical clock is a counter which is incremented between successive external events executed within a process. The logical clock has a monotonically increasing value so it assigns a unique number to every event. If event A occurred before event B in process P then $LC_i(A)<LC_i(B)$. This insures the ordering within each process. It does not insure the ordering across multiple processes. To resolve the ordering across multiple processes, processes receiving a message whose logical clock time stamp is greater than the current value of its own, must advance their clock. More precisely, if process P_i receives message event B with time stamp t, and $LC_i(B)<t$, then it should advance its clock so that $LC_i(B) = t+1$. This insures that there will be at least a partial ordering on events. Causally related events will always be correctly ordered.

3.2.5. Display model

Assuming the partial ordering of events has been accurately recorded, the *history of causality* of external events may be observed by mapping the state transitions of the processes graphically along a timeline. This is analogous to a circuit timing diagram where a portion of synchronization history is displayed linearly. Whereas the signals in a circuit timing diagram have only two states, the processes in the display diagram may have multiple states, depending on how the user has defined them.

In a circuit timing diagram, measurements may be taken between any two events along the timeline. This is because the global clock has provided a

total ordering. In the case of the display diagram, measurements along the trace of a single-process will always have meaning. Measurements between events in different processes may or may not have meaning depending on the presence of a global clock or adequately synchronized local clocks.

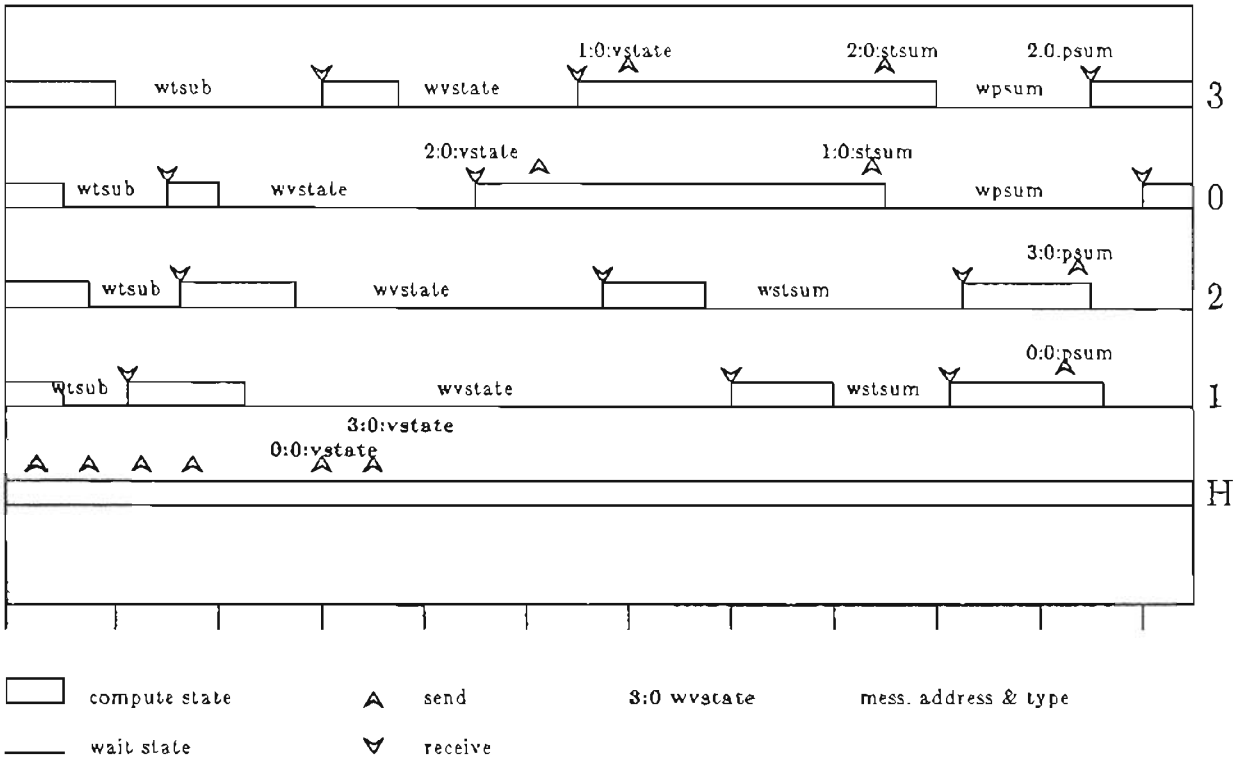


Fig 3.4 Causality History Diagram

Figure 3.4 is an example of a causality history diagram for a computation using five processes. The history of the computation may be observed by horizontally scrolling the display. In this example of a display for a message-based system, wait states are labeled with the types of messages expected. Up

arrows indicate outgoing messages and down arrows denote the receipt of a message. Message being sent are displayed with their destination address and the user-defined message type. Double lines indicate a computation stage and single lines correspond to process idle time. A user can trace the synchronization history of the program, observe relative amounts of idle time for different portions of a program, and get visual clues to possible optimizations. The display also shows some kinds of synchronization errors such as deadlock.

The display diagram provides a high level view of the history of program execution. It is only a point of departure for performance or behavior analysis. Lower level, more system dependent displays are also necessary to provide a finer grain of information. Examples of system-dependent displays are:

animation of messages

display of dynamic changes in message queue lengths

display of shared memory accesses

node topologies

histograms of system-dependent statistics

4. THE DESIGN OF IPPM

The models of measurement, analysis and display presented here could be implemented for a wide variety of parallel and distributed systems. IPPM is a performance instrumentation system based on these models and is intended to be the part of a general instrumentation environment for the Intel iPSC. IPPM enables a user to easily instrument an iPSC program, select data and events of interest and then "replay" the program's execution. IPPM can help a user visualize program synchronization, process idle time and program bottlenecks. It also functions as a simple debugger and is designed to be extendable to support views of the trace data for detailed behavior analysis. IPPM tracks and displays dynamic changes in performance statistics. Users can selectively display events related to specific processes, single step through portions of the execution, examine critical sequences, "fast scroll" history to sections of interest and then "replay" portions of the program.

4.1. Overview of the iPSC

The Intel iPSC is a MIMD distributed memory machine with a hypercube topology. Programs are partitioned into processes that are run on one of up to 128 nodes (OGC's machine presently has 32). Each node consists of an 80286 processor and 512k of RAM. There is no shared memory. Processes communicate and synchronize via buffered messages. Messages may be sent and received synchronously or asynchronously. Users have access to iPSC nodes only by communication through a *host* processor. In a typical application, a host pro-

cess sends initialization and data messages to each participating node process. The node processes work on their portion of the problem, return their results to the host and then terminate or wait for more data.

Parallel algorithms for this type of machine have control-flow structures that take advantage of the processor interconnect topology. The hypercube-connected architecture allows problems to be partioned and mapped to many topologies that are subsets of a hypercube - trees, rings, and meshes, for example. A primary reason for building IPPM was a desire for a tool that could characterize the relative performance of programs that are mapped to different topologies and control structures.

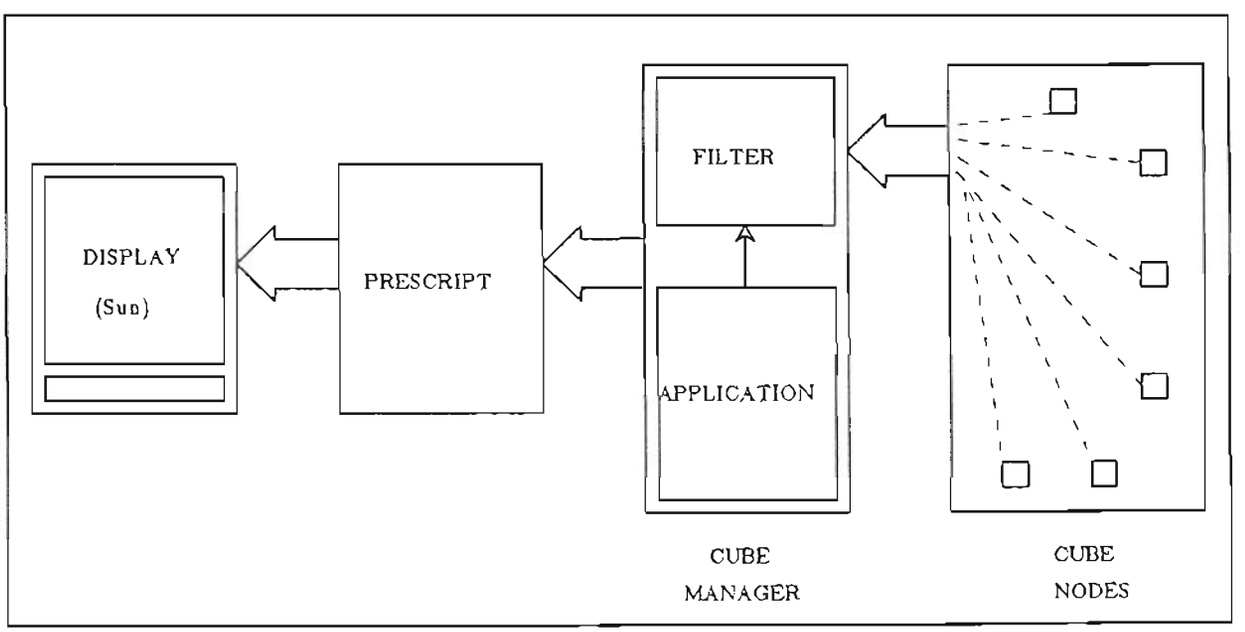


Fig 4.1 IPPM Software Architecture

4.2. IPPM Software Architecture

The IPPM software architecture follows the ideas expressed in the models described earlier. It is divided into modules for metering, filtering and display/analysis. The display portion is currently being implemented on a Sun workstation. Trace files are generated on the iPSC and then transferred to the Sun for analysis (see Fig 4.1).

4.2.1. Metering

Traces on external events for each process are collected during execution. All user processes including processes running on the host are monitored. Trace generation is relatively transparent to the user. Programmers use a library of IPPM communication primitives that in turn call the iPSC primitives. IPPM primitives increment the logical clock, build the trace, store it in the trace buffer and pass the users data to the corresponding iPSC primitive. When the trace buffer fills up it is flushed using an asynchronous send to a *filter* process running on the host.

There are nine external events that generate traces:

-starting a process
-synchronous send
-initiating a synchronous receive
-terminating a synchronous receive
-asynchronous send
-asynchronous receive
-opening a message channel
-closing a message channel
-terminating a process

All traces have a common data header:

Processor id:	the processor from which the trace came
Process id:	the id of the individual process
Timestamp:	the time on the local clock that the event occurred.
Event type:	the type of event recorded
Logical clock:	the number of the event on the logical clock

4.3. Filtering

When the application process starts on the host, a filter process is also started. The user provides a template in a start up file that the filter uses to examine and screen the incoming trace messages. A user can selectively keep traces based on the type of the event, node and process id, and the time the event occurred. For example, a template to selectively filter only synchronous sends and receives for all processes would look like:

```
etype = BREC_S , node = *, pid = *, time > 0 ;
etype = BREC_D , node = *, pid = *, time > 0 ;
etype = BSEND  , node = *, pid = *, time > 0 ;
```

The filter writes the traces that are accepted by the template to a graphic display log file on the host. A preprocessor program, *prescript*, then builds a display script from the log file. Prescript sorts the traces based on the timestamps from the logical clocks and constructs a header that includes a list of the process ids and cube nodes appearing in the script. The script can then

be used by IPPM to replay the execution and analyze the performance of the program.

4.4. Analysis

A parallel program's performance can be expressed in terms of speed-up over a sequential version of the same program. It can also be expressed simply in terms of a real-time improvement over another partition or mapping of the algorithm. The goal of instrumenting a parallel program is to be able to determine not only the time taken by the longest running process but also the distribution of work among the processes. If a problem is divided into k parts and each part is executed on one of k processors, each part taking the same amount of processing time, then perfect parallelism is achieved and the speed-up is k. However, if we know that at most half the processors are active 50% of the time then the speed up is at most $\frac{3}{4}k$. We would therefore like to know the percent of time each process spends computing. We also would like to know the distribution of compute times among the processes. If there are two processes involved in a computation and one process terminates in a minute and the other terminates in an hour, then little parallelism has been achieved.

The statistics just described give a gross view of the efficiency of a particular algorithm. At a closer level of inspection we would like to know the distribution of communication among the processes. Do some processes become information bottlenecks, forcing excessive waits by others? We would like to know the average wait times for particular events and the conditions under

which excessively long waits occur. What percent of messages are local versus remote? Messages are routed automatically on the iPSC. Message delays depend on how many intermediate nodes have to handle a packet between the source and destination. We would like to have an index of the average distance that messages are traveling. Economies can be gained by keeping message distances short. We also want to keep track of how often processes are communicating (time density) and the relative amounts of synchronous versus asynchronous communication.

4.4.1 The IPPM Display Interface

The IPPM display interface has been designed on two levels. The highest level, the system-independent display, gives the user a view of the history of causality of external events. On the lower level are system-dependent displays related to the iPSC message-based architecture. Currently the lower level has been implemented:

Message animation:

> Displays current iPSC dimension and animates message traffic between nodes.

Communication state matrix:

> Displays the current communication state of all processes. A process may be in a compute stage or waiting for a message. The type of message expected is shown and wait periods may be timed.

Message distribution:

> A histogram of communication distribution

events among processes.

Blocked time:

A histogram of accumulated idle time for each process as a percent of elapsed time.

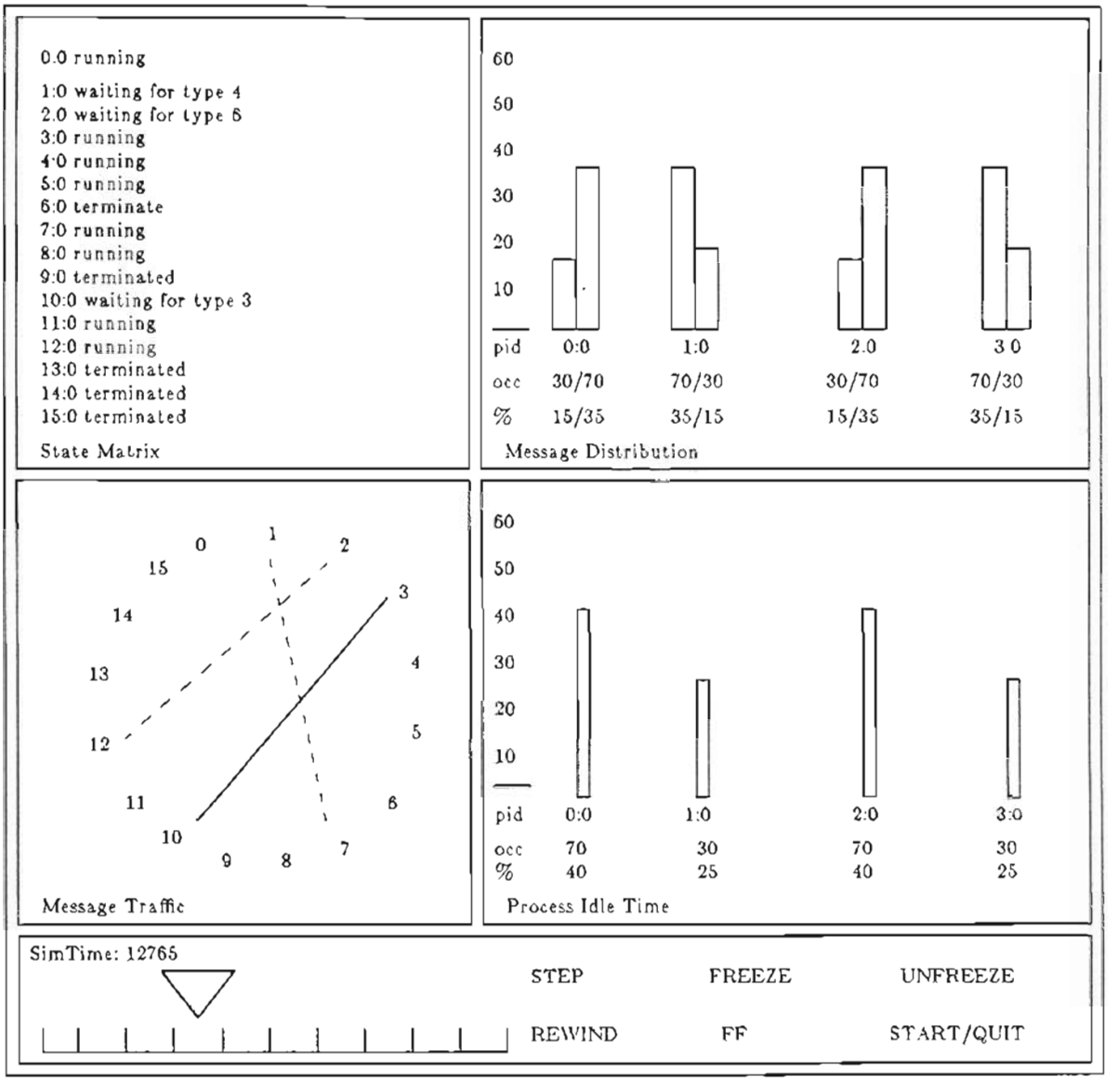


Fig 4.2 IPPM user interface

At the start of a session the user defines which processes he is interested in observing and the statistics desired. The display allows for "single stepping" through the execution or for "fast scrolling" to sequences of particular interest.

A user may also "rewind" and review portions of the execution.

The example display in figure 4.2 shows sixteen processes being tracked and four selected for statistical display. The display shows that at physical time 12765 (units of 50 milliseconds) there were two processes blocked waiting for messages. Process 1:0, process number 0 running on node 1, was waiting for a message of type 4. Process 2:0 was waiting for a message of type 6. The dashed lines in the message animation panel indicate a pending message. The solid line from process 3 to process 10 indicates that the message was received at time 12765. The message distribution display shows that processes 0:0 and 2:0 each have logged 30 sends and 70 receives at this point in the computation and these are 15% and 35%, respectively, of the total communication. The higher number of blocked receives, in comparison with the other two processes being displayed, is reflected in the amounts of process idle time. Processes 0:0 and 2:0 have each been idle 40% of the time since the start of the computation. Processes 1:0 and 3:0 have each been idle only 25% of the time. The example also shows that four processes terminated roughly one third of the way through the computation.

5. AN EXAMPLE ANALYSIS

IPPM can be used as a testbench for new algorithms or to instrument and tune existing programs. Using it has led to some general suggestions for applications. One primary motivation for studying parallel programs is to determine the suitability of the problem mapping to a particular topology and control strategy. The iPSC can emulate many topologies so it is an ideal architecture for this type of analysis.

As an example of testing a new algorithm we chose to look at a parallel implementation of Hopfield nets. The distributed Hopfield net algorithm, suggested by Dr. David Maier, uses a master-slave control method and a mesh topology. This combination is typical of a large class of matrix algorithms, although Hopfield nets have some special properties that complicate the algorithm.

5.1. An Overview of Hopfield Nets

Hopfield nets are an attempt to emulate the associative recall characteristics of neural networks [Hop82]. A Hopfield network is a recursive, asychronous, fully connected network that performs "best match" auto-association. A Hopfield network may be defined as a collection of virtual elements or "neurons", henceforth referred to as VEs. Each VE has two states: $V_i = 0$ ("not firing") and $V_i = 1$ ("firing at maximum rate"). When neuron V_i has a connection made to it from V_j, the strength of the connection is defined as T_{ij}. Connections between VEs form a directed graph described by the matrix T. The

instantaneous state of the system is specified by bit vector of N values of V_i. Networks "learn" bit vectors by adjusting the connection weights according to a learning algorithm based on a theory of neural excitation and inhibition. When the network is initialized to a random vector after "learning" several others, it will attempt to "remember" one of its learned vectors, according to the following algorithm. For each neuron V_i there is a fixed threshold U_i. Each neuron readjusts its state randomly in time but with a mean attempt rate W, setting

$$V_i = 1 \quad \text{if} \quad \sum_{j \neq i} T_{ij} V_j > U_i$$

$$V_i = 0 \quad \text{if} \quad \sum_{j \neq i} T_{ij} V_j < U_i$$

In Maier's algorithm a distinction is made between virtual processing elements and physical processing elements. A single physical element (PE), or node processor, can represent many VEs. The critical data required for this algorithm is the representation of the weighted values of the connections. For a fully connected network with n VEs, n^2 values must be stored. One approach would be to store the weights and the state vector on each PE and make it responsible for a subset of the VEs. The requirement that VE updates be globally available after they occur makes this approach unreasonable. Communication would overwhelm the system. Storing n^2 values on each node is also prohibitive.

Another approach, the redundant-state algorithm, maps the problem to a set of processors connected in a mesh. Each column consists of a master PE and a set of slave PEs (see Figure 5.1). The VE state vector is partitioned and distributed in each column. Each row PE has the same portion of the state vector. The connection matrix is (non-redundantly) distributed so that each column PE is able to calculate a partial sum in parallel and return it to the master.

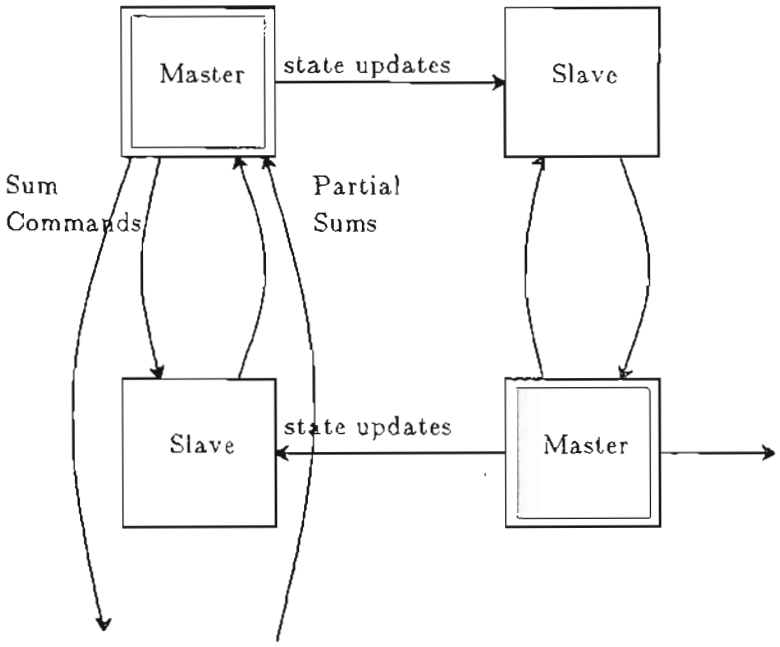


Fig 5.1 Control for Hopfield Net Algorithm

The first step is to get the control scheme working on a small mesh. Control begins with a host process sending initialization messages to each node. When the nodes are initialized the master PEs select a VE for which to sum

and send the index of the VE to the slaves in the column. Each master then calculates its own partial sum and awaits the partial sums from the slaves. If the VE changes state as a result, the update is then broadcast across the row.

IPPM message animation and state matrix displays were used to debug the initial implementation. Single stepping through the event history assured that the intended order of synchronization events actually occurred. At one point a problem causing non-termination was easily traced to a deadlock in the update cycle by examing a preliminary version of the causality history display (see Figure 5.2). The cause of deadlock at that point was an incorrect loop index.

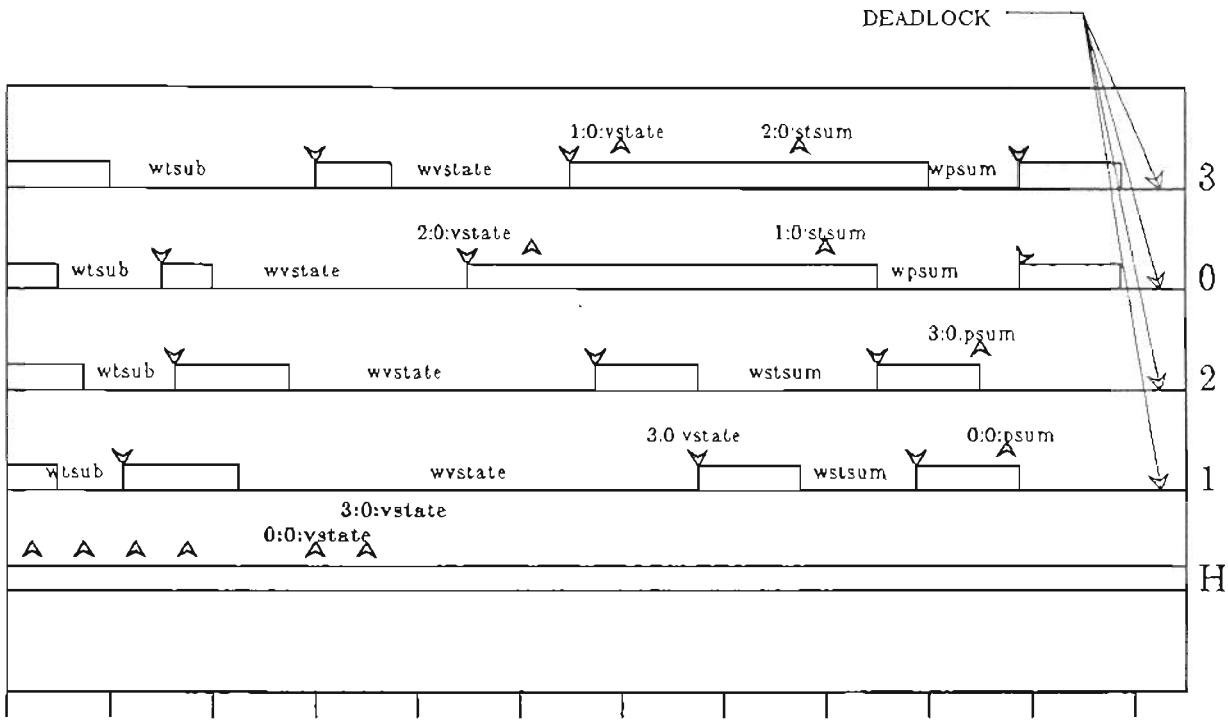
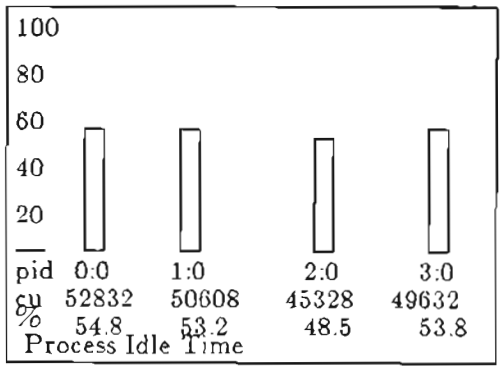


Fig 5.2 Display of a Deadlock Condition

Once control-flow is correct, the details of the application are filled in. The next step, before putting additional work into the program, is to examine preliminary performance data. The data for a 4 PE, 16 VE Hopfield net shows that the communication and synchronization time dominates the computation (see Figure 5.3). Instrumenting the VE processing loops in the master and slave processes shows that processes spend most of their time waiting for messages. CPU use is low since there is one process on each node and communication is synchronous. The time spent calculating partial sums is small compared to the time required to send the summing command, wait for the results and communicate updates. The most critical time, the time to update the network,

involves too much wait time.



System time for 400 iterations	96368 msec.
Ave. net update cycle time	224 msec.

Fig 5.3 Hopfield Algorithm Performance

There are several factors that affect the overall ratio of communication to compute time for a process. The number of summing steps for a VE increases as the square of the number of VEs in the network. Therefore the algorithm should make better use of the CPUs when working on larger networks. However, a quick analysis of available node resources shows that the storage for the connection matrix restricts the number of VEs such that the compute time required for additional VEs will not offset the necessary communication (see Figure 5.4). This will always be the case unless data is presented to

the PEs in the form of a vector rather than individual elements. A vector would reduce the total number of messages by a factor of its length. Another alternative is to pipeline the computation, or even to combine the use of vectors and pipes.

Assume 8x8mesh:

-Currently iPSC has ~256k bytes space available for connection submatrix on each node

-allow 1 byte for each table entry

-64 * 256k = 16.78 Mbyte
connection matrix matrices are symmetric and have a lot of redundancy. Assume clever storage methods - multiply by a factor of 2.5 for effective storage of 16.78M * 2.5 = 41.94M

sqrt (41.94) = 6478 VEs or ~100VEs per PE

Fig 5.4 Analysis of max VEs on iPSC

This analysis leads to a redesign of the control method. One alternative control method would require sending vectors of VE elements to the slave nodes. The internal message queueing system of the iPSC would be exploited to keep queues of vectors for the slaves to work on. Slaves would send their results as they were completed. This would complicate the job of the master PEs. Masters would now be responsible for keeping queues full and partial results straight. This is called the load balancing scheme (see Figure 5.5).

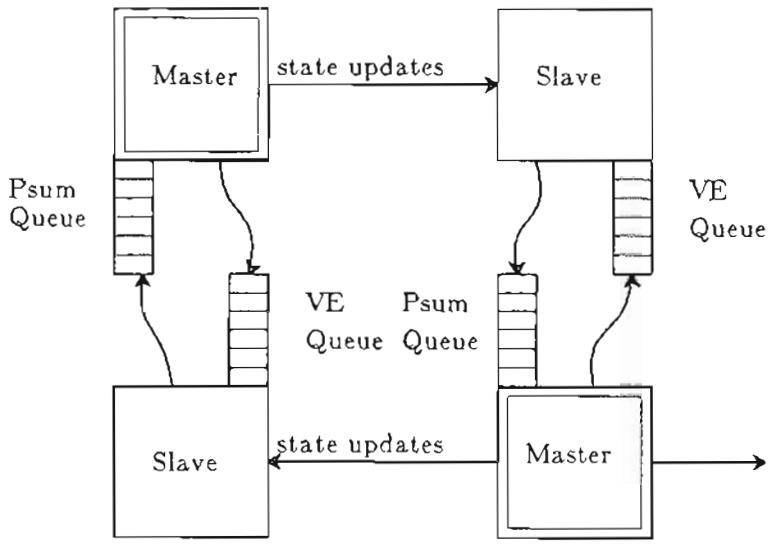


Fig 5.5 Load Balancing Scheme

A second alternative would reduce the bottleneck at the master as it waits for a partial sum from each slave. A pipelined control solution would eliminate this problem (see Figure 5.6). It also has the advantage of simplicity over the load balancing scheme. The master would compute a partial sum for a VE, or a vector of VEs, and pass the data to the first slave. The slave adds its own partial sums and either passes the data to the next slave or, if it is the last slave, forwards the finished sum to the master. The master then sends the result to the nearest row neighbor, who in turn updates its own state vector and sends the results to its neighbor. Once the pipes are full the update cycle time for the pipelined control scheme would be only the time for the last slave to add its own partial sums and forward the results. If separate channels and asynchronous messages were used, broadcast overhead could be minimized.

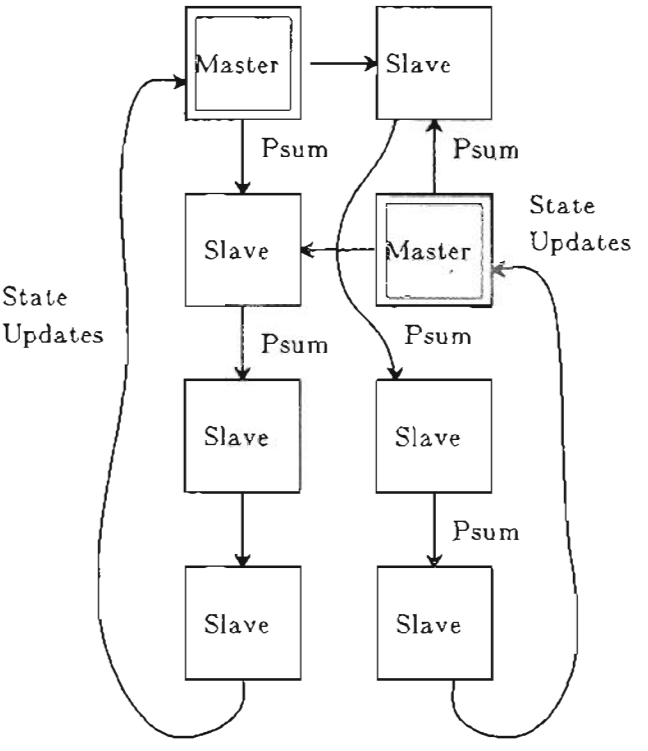
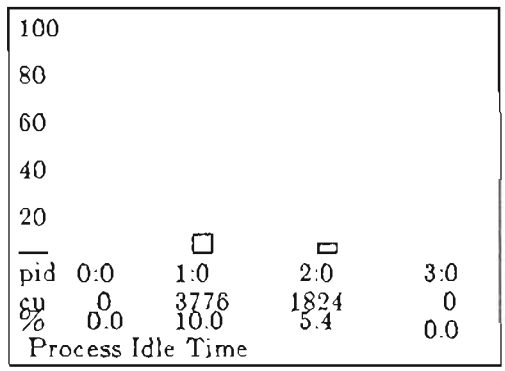


Fig 5.6 Pipeline Control Scheme

At this point we need to determine empirically if changing the control method will significantly effect the performance of the algorithm. Of the two alternatives, the most promising seems to be the pipeline scheme. It has the advantage of simplicity; each process communicates with at most two others. It also can incorporate some of the advantages of the load balancing scheme by incorporating the system queueing of messages into the pipes.

An implementation of a 16 VE / 4 PE network using the pipeline control scheme shows that the critical time period, the net update cycle has been significantly reduced (see Figure 5.7). Eliminating the wait cycle at the master by pipelining the computation has given us a 2.6x speed up over the original

design. Process idle time is now acceptably low. A significant advantage of the pipeline control over the other two is that it should also scale up to larger meshes without greatly increasing penalties. It is now apparent that, within the limits of this particular problem on the iPSC, an acceptable algorithm has been found. IPPM can now be used to scale up and fine tune the final version of the Hopfield network program.



System time for 400 iterations	37168 msec.
Ave. net update cycle time	80 msec.

Fig 5.7 Pipeline Control Algorithm Performance

5.2. Summary of the Hopfield algorithm analysis

When a parallel algorithm is first designed, performance obstacles may not be obvious in the context of a particular system. In the case of the Hopfield algorithm for the iPSC, the performance tradeoff involves finding a control

method that keeps relatively expensive communication from interfering with the progress of the computation. IPPM allowed us to quickly debug and implement an initial algorithm for a small node configuration. It was then used as a testbench to analyze the performance and determine the bottlenecks prior to spending time scaling the program up. The initial algorithm was found to be dominated by communication and synchronization time. The bottlenecks were identified and two alternatives were proposed. The more promising alternative was implemented for the same size problem and its performance was measured. This second solution was found to solve a number of problems that plagued the first solution. This entire process was eased by the use of the design, debug, and test aid IPPM.

6. Summary

Understanding the behavior of parallel programs, even small ones, can be very difficult. The added complexities of asynchrony, delay and indeterminacy associated with parallel programming combine to make the programmer's job difficult. A new class of tools is needed to assist the programmer who must confront these problems. Parallel programming may be seen as a process of several distinct steps. First the application must be examined for potential parallelism and partitioned accordingly. Next a control strategy must be defined and then the design must be mapped to an appropriate system. The final tasks are analyzing the program for correctness and performance and making stepwise adjustments. These last two steps, usually referred to as debugging and performance analysis, are very closely related. Depending on how data is collected and presented, debugging and performance analysis could be seen as different views of the same data.

An survey of existing work related to parallel debugging and performance monitoring shows a need for an integrated approach to these two tasks. Therefore, a set of evaluation criteria for a general behavior and performance analysis system was developed. Such a system should be based on a general model of computation that would include a wide variety of parallel systems. It should support debugging and performance monitoring and should manage the added complexity of parallel processing, preferably through data abstraction and graphic abstraction. Finally, the system should be non-intrusive. It should impose a minimal overhead on the computing resources and avoid introducing

delays that could alter program behavior. Existing related work was evaluated in light of these criteria. Most debuggers do not manage complexity and many are intrusive. Traditional state-based debugging techniques are hard to adapt to some systems where it is not possible to capture a snapshot of the global state from any point within the system. Most performance monitors lack a general model of computation and do not make the connection to debugging.

Several approaches were very promising, most notably the PIE system, although it also lacked a general model of computation. One method, *behavior abstraction* did provide a base for a general system. Behavior abstraction views a program in terms of activity rather than state. A set of models based on behavior abstraction was developed and then an architecture of a general system based on the models and meeting the criteria was proposed. A prototype system was then developed for the Intel iPSC. IPPM was used to develop and evaluate a new algorithm and evaluate the performance of several large existing programs. Initial experience with the IPPM prototype has shown that it works well as a design and test aid for parallel programs. A user can write a program, instrument it, and quickly see if it behaves as expected. He can also get an initial idea of the program's efficiency. After testing and evaluating alternative control structures and program partitions, and determining performance bottlenecks, the best solution may used as a base for a large scale implementation. IPPM makes the this step in writing good, working parallel programs easier.

There are numerous interesting areas for further research. One subject, also mentioned by several other researchers working on software tools for parallel processing, is the need to incorporate hardware monitors into parallel system architectures. The monitors should be general enough to support a variety of low-level performance monitoring tasks as well as high-level behavior analysis. Another interesting subject to explore is the design of canonical forms for graphic display of program events. One approach being implemented at OGC, is the use of the Large Grain Data Flow Model to display graphic event history. Program events are defined and displayed hierarchically as graphs and subgraphs in the Data Model and may be interactivly queried and expanded during the "replay" of the program's execution. This work supports our conclusions about behavior abstraction as a base for debugging and performance monitoring and shows the flexibility inherent in the display model. Further work is needed to verify that the models we have proposed are sufficiently general. A prototype system for a shared memory architecture would be a good way to test the models and would explore data collection methods on a tightly coupled system.

REFERENCES

[Bates83]

Bates,P., and Wileden, J., "An Approach to High-Level Debugging of Distributed Systems", *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on High-Level* Debugging", March, 1983.

[Hart85]

Hartner,P, Heimbigner, D, and King, R., "IDD: An Interactive Distributed Debugger", *Proceedings of 5th International Conference on Distributed Computing* Systems", May, 1965.

[Hop82]

Hopfield, J., "Neural Networks and Physical Systems with Emergent Collective Computational Abilities", *Proceedings of the National Academy of Sciences USA*, Vol. 79, April 1982.

[Kie83]

Kieburtz, R., Mukerji, J., Sadayappan, P., and Smith, D., "An Experimental Multicomputer with a Real-Time Event Monitor", *Oregon Graduate Center technical report CS/E - 83-004*, June, 1983.

[Kuck78]

Kuck, D., "The Structure of Computers and Computations", *John Wiley & Sons*, 1978.

[Kung76]

Kung, H., "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors", *Algorithms and Complexity*, 1976.

[Lam78]

Lamport,L., "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, July, 1978. Vol 21, Number 7

[LeBl85]

LeBlanc,R, and Robbins, A., "Event Driven Monitoring of Distributed Programs", *Proceedings of 5th International Conference on Distributed Computing* Systems", May, 1985.

[McDan75]

McDaniel, G., "METRIC: a Kernel Instrumentation System for Distributed Environments", *Proceedings of the 6th Symposium on Operating System Priciples*, November, 1975.

[Mapl85]

Maples, C., "Analyzing Software Performance in a Multiprocessor Environment", *IEEE Software*, July, 1985.

[Mel85]

Melamed,B, and Morris, R., "Visual Simulation: The Performance Analysis Workstation", *Computer*, August, 1965.

[Mill85]

Miller, B., "DPM: A Measurement System for Distributed Programs", *5th International Conference on Distributed Computing Systems*, IEEE Computer Society", May, 1985.

[Schiff81]

Schiffenbauer, R. "Interactive Debugging in a Distributed Computational Environment", *Masters Thesis, MIT*, August, 1981.

[Seg85]

.Segall, S., and Rudolph, R., "PIE: A Programming and Instrumentation Environment for Parallel Processing", *IEEE Software*, November, 1985.

[Smith85]

Smith, E., "Debugging Tools for Message Based, Communicating Processes", *Proceedings of 5th International Conference on Distributed Computing* Systems", May, 1965.

[Snod82]

Snodgrass, E., "Monitoring Distributed Systems: A Relational Approach", *Phd Dissertation, Carnegie-Mellon University*, December, 1982.

[Vick76]

Vickers, L., "The Design and Implementation of a Multi-Process, Multi-Machine, Multi-language Debugger", *Technical Report #27899, Tymshare Corp*, January, 1976.

[Ver83]

Vernon, M., Silva, E., and Estrin, G., "Performance Evaluation of Asynchronous Concurrent Systems: the UCLA Graph Model of Behavior", *Performance '88, North-Holland Publishing Co.*, January, 1983.

APPENDIX A: THE IPPM PRIMITIVE LIBRARY

IPPM C ROUTINES

Introduction

This appendix is intended as a reference section describing each of the IPPM primitives in the C system interface libraries for the cube manager and the node processes. The IPPM primitives are based on the iPSC communication primitives. In addition to communication and synchronization, they generate event traces, update the logical clocks and maintain the event buffers.

For a more complete description of the iPSC communications primitives, refer to chapter 7 of the iPSC system documentation.

The routines are documented in six parts:

1. name of the routine and a brief description
2. the calling sequence
3. the input parameters. These are required and have no default values. The exact order of the paramters must be as shown.
4. the return values or results of executing the routine
5. a description (when necessary)
6. errors that can occur through the use of this routine. All errors cause the process that originated them to stop running and cause the operating system to write an error message in the system log file. Refer to the system log file section in chapter 13 of the iPSC system manual and to the errors section of chapter 12.

IPPM C Routines
Cube Manager and Node Processes

Ginit

Ginit initiates event data collection for host and node processes.

Calling Sequence

```
int ipid;
.
.
.
Ginit(ipid);
```

Input Parameters

ipid the process id of the calling process

Return Values

None

Description

Ginit should be the first procedure call in the process "main" procedure.

Errors

None

IPPM C Routines
Cube Manager and Node Processes

Gterminate

Gterminate ends event data collection for host and node processes.

Calling Sequence

```
int ipid;
.
.
.
Gterminate(ipid);
```

Input Parameters

ipid the process id of the calling process

Return Values

None

Description

Gterminate should be the last procedure call in the process "main" procedure.

Errors

None

Gcclose

Gcclose destroys the specified communication channel created by a previous Gcopen.

Calling Sequence

```
int ci;
.
.
.
Gcclose(ci);
```

Input Parameters

ci — is the channel identifier of the channel you want to close. ci was returned by XENIX as a result of a previous "Gcopen".

Return Values

None

Description

Explicitly closes a communications channel and stores a trace in the event buffer.

Errors

EQCI - the channel identifier specified in the routine is invalid.

IPPM C Routines
Cube Manager and Node Processes

Gcopen

Creates a communication channel for a cube manager process or a node process

Calling Sequence

```
int ci;
int pid;
.
.
.
ci = Gcopen(pid);
```

Input Parameters

pid — is the user defined porcess id of the process opening the channel. The valid range is 0 - 32767 excluding 99 which is the pid of the monitor process.

Return Values

ci — is the channel identifier used by XENIX to manage the transmission and reception of messages.

Description

The maximum number of open channel allowed is 16. Ccopen causes an event trace to be stored in the event buffer.

Errors

EQNONMEM - the system has insufficient memory to create the structures necessary to manage the channel

EQPID - pid out of range

EMFILE - attempt to open more than 16 files

IPPM C Routines
Cube Manager

Grecvmsg

Grecvmsg initiates the receipt of a message from a node process or another cube manager process.

Calling Sequence

```
int ci;
int type;
char buf[n];
int len;
int cnt;
int node;
int pid;
int ipid;
.
.
.
Grecvmsg(ci,&type,buf,len,&cnt,&node,&pid,ipid);
```

Input Parameters

ci — is the channel identifier that was returned as a result of the previous "Gcopen"

len — is the number of bytes in the message buffer

Return Values

received will be stored

buf — pointer to the process message buffer where the received message is to be stored

cnt — pointer to the location where the number of message bytes is stored

node — pointer to the location where the node id of the process that sent the message is stored

pid — pointer to the location where the process id of the process that sent the message is stored

ipid — process id of the process that initiated the receive

Description

This routine causes the calling process to be blocked until a message has been received (see iPSC manual p. 7-9). Grecvmsg causes two event traces to be generated, one when the receive is initiated and another when the message actually arrives.

Errors

EQCI - the channel identifier specified in the routine is invalid.
EQBLEN - the length of the message buffer is negative

IPPM C Routines
Cube Node Processes

Grecv

Grecv initiates the receip' ' a message from another process. The user message buffer is not availabe for reuse and the return values are not updated until "status" is performed.

Calling Sequence.

```
int ci;
int type;
char buf[n];
int len;
int cnt;
int node;
int pid;
int ipid;
.
.
.
Grecv(ci,type,buf,len,&cnt,&node,&pid,&ipid);
```

Input Parameters

ci — is the channel identifier that was returned as a result of a previous "Gcopen"

type — is an integer referring to the type of message to be received. Valid range is 0..32767

len — is the length in bytes of the message buffer, "buf"

ipid — is the process id of the process initiating the receive

Return Values

A "status" must be performed to update these values

buf — is a pointer to the buffer where the received message is to be stored

cnt is a pointer to the location where the number of message bytes received is stored. See iPSC system manual p. 7-25 for restrictions.

node is a pointer to the location where the node id of the process that sent the message is stored

pid is a pointer to the location where the process id of the process that sent the message is stored.

Description

Grecv generates a trace for the event corresponding to the time the receive request is logged with the kernel.

Errors

EQCI - the channel identifier specified in the routine is invalid.

EQCLOSE - a "Gcclose" routine was invoked while a message was being sent.

EQBLEN - the length of the message buffer is negative

IPPM C Routines
Cube Node Processes

Grecvw

Grecvw initiates the receipt of a message from another process. Execution of this routine causes the calling process to be blocked until the message has been received.

Calling Sequence

```
int ci;
int type;
char buf[n];
int len;
int cnt;
int node;
int pid;
int ipid;
.
.
.
Grecvw(ci,type,buf,len,&cnt,&node,&pid,&ipid);
```

Input Parameters

ci	is the channel identifier that was returned as a result of a previous "Gcopen"
type	is an integer referring to the type of message to be received. Valid range is 0..32767
len	is the length in bytes of the message buffer, "buf"
ipid	is the process id of the process initiating the receive

Return Values

buf	is a pointer to the buffer where the received message is to be stored
cnt	is a pointer to the location where the number of message

bytes received is stored. See iPSC system manual p. 7-25 for restrictions.

node is a pointer to the location where the node id of the process that sent the message is stored

pid is a pointer to the location where the process id of the process that sent the message is stored.

Description

Grecvw generates two traces in the event buffer. The first trace is generated when the receive request is logged with the kernel. A second trace is put in the buffer when the message arrives.

Errors

EQCI - the channel identifier specified in the routine is invalid.

EQCLOSE - a "Gcclose" routine was invoked while a message was being sent.

EQBLEN - the length of the message buffer is negative

IPPM C Routines
Cube Node Processes

Gsend

Gsendmsg initiates the transmission of a message to another process. The user message buffer should not be rewritten until a "status" indicates that the send operation is complete.

Calling Sequence

```
int ci;
int type;
char buf[n];
int len;
int node;
int pid;
int ipid;
.
.
.
Gsend(ci,type,buf,len,node,pid,ipid);
```

Input Parameters

ci	is the channel identifier that was returned as a result of a previous "Gcopen".
type	is an integer value referring to the type of message to be sent
buf	pointer to the buffer that contains the message to be sent
len	number of butes to be transmitted
node	id of the node being sent the message
pid	process id of the process being sent the message. Valid range is 0..32767 for node processes and is the same for messages to the host, excluding 99 for the monitor process.
ipid	process id of the process initiating the send.

Return Values

None

Description

The return to the calling process from "Gsend" does not indicate that the message has been sent, only that the system has been notified that a message is ready for transmission. Gsend causes a trace to be stored in the event buffer corresponding to the time that the send request was logged with the kernel.

Errors

EQCI - the channel identifier specified in the routine is invalid.
EQLEN - the length of the message exceeds maximum
EQNODE - the node address referenced is invalid
EQPID - the pid referenced is out of range
EQTYPE - the type specified is not within range of 0..32767

IPPM C Routines
Cube Node Processes

Gsendw

Gsendw initiates the transmission of a message to another process. Execution of this routine causes the calling process to be blocked until the message has been sent.

Calling Sequence

```
int ci;
int type;
char buf[n];
int len;
int node;
int pid;
int ipid;
.
.
.
Gsendw(ci,type,buf,len,node,pid,ipid);
```

Input Parameters

ci	is the channel identifier that was returned as a result of a previous "Gcopen".
type	is an integer value referring to the type of message to be sent
buf	pointer to the buffer that contains the message to be sent
len	number of butes to be transmitted
node	id of the node being sent the message
pid	process id of the process being sent the message. Valid range is 0..32767 for node processes and is the same for messages to the host, excluding 99 for the monitor process.
ipid	process id of the process initiating the send.

Return Values

None

Description

Gsend causes a trace to be stored in the event buffer corresponding to the time that the send request was logged with the kernel. Another trace is generated at the time the message arrives.

Errors

EQCI - the channel identifier specified in the routine is invalid.
EQLEN - the length of the message exceeds maximum
EQNODE - the node address referenced is invalid
EQPID - the pid referenced is out of range
EQTYPE - the type specified is not within range of 0..32767

Using the IPPM primitives to generate event traces is done by linking to a monitor library. Use the following makefile as a template.

```
/*  Example makefile for linking to IPPM monitor library
 *
 *  Craig Brandis
 *  OGC
 *  7/9/86
 */

LIBS = /usr/ipsc/lib/Llibc.a /usr/ipsc/lib/Llibm.a -lm
LIBS2 = /usr/ipsc/lib/chost.a
MPATH = /usr/ogc/students/brandis/mon

HOSTOBJS = host.o

NODEOBJS = node.o

all:    host node

host: $(HOSTOBJS) ../mon/hglib.a ../mon/mon.a
        cc -Alfu -o host host.c
        /usr/ipsc/lib/cloader.a $(LIBS2)
        $(MPATH)/hglib.a $(MPATH)/mon.a


node.o: node.c ../mon/nglib.a
        cc -Alfu -K -c node.c

node:  $(NODEOBJS)
        ld -Ml -o node /usr/ipsc/lib/lcrt0.o $(NODEOBJS)
        $(LIBS) $(MPATH)/nglib.a
```

APPENDIX B: EXAMPLE PROGRAMS

Any process that uses the monitor library must initiate data collection when the process is invoked and terminate data collection prior to process termination. This is accomplished by calls to *Ginit* and *Gterminate*. In addition the host application process must start the monitor process by forking a child (see example).

It is not necessary to always use IPPM primitives instead iPSC primitives. In many cases programs have sections of code that are the limiting factors to performance. When using IPPM as a performance analysis tool it is often easiest to instrument only these portions of the program.

host program

```
/* Example host program using IPPM primitives
 *
 * Craig Brandis
 * OGC
 * 7/9/86
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/times.h>
#include "/usr/ipsc/lib/chost.def"

static char buf[100];
int globalcid;

main()
{
        int i;
        int type,cnt,snode,spid;
        char *str1 = "HELLO";
        char buf[80];
```

```
        /* load the node program onto a 2d cube  */
        for(i = 0; i < 4; i++)
                load("node",i,0);

        if(fork())      /* fork the host monitor process */
           monitor();

        /* parent process continues - open a channel */
        globalcid= copen(0);

        /* initialize event monitoring */
        Ginit(0);

        for(i = 0; i < 4; i++)
        {
           /* send a message to i:0 */
           Gsendmsg(globalcid, 0, str1, strlen(str1), 0, 0,0);

           /* receive a message from i:0 */
           Grecvmsg(globalcid, &type, buf, 80, &cnt, &snode, &spid,0);
        }


        /* close the channel */
        cclose(globalcid);

        /* terminate event data collection and flush
         * the buffers
         */
        Gterminate(0);
}
```

```
/* Example node program using IPPM primitives
 *
 * Craig Brandis
 * OGC
 * 7/9/86
 */
#include <stdio.h>
#include "/usr/ipsc/lib/cnode.def"

main()
{
        int cnt, node, hostid, pid;
        int i;
        int globalchan;
        char buf[80];

        /* initiate event data collection */
        Ginit(0);

        globalchan = copen(mypid());

        /* receive a message from host process */
        Grecvw (globalchan, 0, (char *)buf, sizeof(buf), &cnt, &node, &pid,0);

        /* send the message back to host process */
        Gsendw(globalchan, 1, (char *)buf, sizeof(buf), MANAGER, 0,0);

        cclose(globalchan);

      /* terminate event data collection */
      Gterminate(0);
}
```

APPENDIX C: IPPM USERS GUIDE

NAME

ippm - interactive parallel program monitor

SYNOPSIS

ippm | logfile initfile |

DESCRIPTION

IPPM is an interactive graphic monitor that is driven from a script of data collected during the execution of an iPSC program. It can be used as a simple debugger but its primary use is for performance analysis. It is currently configured to run on a sun workstation.

IPPM makes a distinction between *tracked* and *displayed* processes. It can track and calculate statistics for up to 32 processes, but displays dynamic changes to statistics for up to 8. These are configurable from a start-up file.

IPPM requires two input files:

logfile

a file of event data collected using the IPPM primitives on the iPSC (see appendices A and B). The default is "DLOG".

initfile

a file of initialization information in the following example format:

```
5              -- the number of processes being displayed
0 0            -- list of node ids and pids of
1 0            -- processes to be displayed in this session
2 0
3 0
-32768 0
```

the init file defaults to *.ppminit*

IPPM currently supports four *views* of the script data:

-communication state matrix

shows the current communication state of each process chosen for display

-process idle time display
shows accumulated process idle time as a percentage of elapsed time on the local clock

-communication distribution display
shows distribution of communication types among the display processes.

-*message animator*
animates message traffic

Commands are entered from by menu selection. Selection is accomplished by placing the cursor on or near the menu item associated with the desired command and pressing the left command button.

The cursor is moved by a "mouse". The mouse has three buttons - only the left and right buttons are active in IPPM. The following is a list of commands available in IPPM and an explanation of their use.

S/Q
Start or leave the current display session

FF
Fast forward to a particular set of interesting events. Use the history scroll bar to determine where in the event history to stop.

FREEZE
Freezes the display at the current event. Can be used in normal execution mode or from a fast forward mode.

UNFREEZE
Returns the history display to normal playing speed.

STEP
Single step through a series of events. Pressing the leftmost mouse button will advance the display one event.

DUMP
Make a bitmapped screen dump of the current screen.

REPLAY
Replay the current event script

SEE ALSO

prescript: preprocessor for ippm script files (Appendix C)

BUGS

Screen dumps come out in sun file format. The Sun is currently configured to use vax utilities and file format for printing raster files.

NAME

prescript - preprocessor for IPPM

SYNOPSIS

ippm infile outfile

DESCRIPTION

prescript takes a raw event data file, generated by using IPPM primitives in an iPSC program, and adds a header for the IPPM display on the sun. File arguments must be in the order shown

SEE ALSO

IPPM: interactive parallel program monitor (Appendix C)

BUGS

BIOGRAPHICAL NOTE

The author was born May 13, 1955 in Sheridan, Wyoming. He graduated from Hudson's Bay high school in Vancouver, Wash. in 1973 and then spent two years at Clark College in Vancouver in general liberal arts studies. In 1975 he enrolled in the Bachelor of Arts program in the College of Architecture at the University of Washington. During the academic year 1976-77 he was awarded a scholarship to study computer applications in land use planning at the Technical University of West Berlin. He was awarded the Bachelor of Arts *cum laude* from the University of Washington in 1979. After working for architecture and engineering firms in Bellevue and Portland for several years, the author enrolled in the Masters program in Computer Science and Engineering at OGC in 1983.