Performance Visualisation in a Portable Parallel Programming Environment

Ian Glendinning, Vladimir S. Getov, Stephen A. Hellberg, Roger W. Hockney, David J. Pritchard [a]

[a]Department of Electronics and Computer Science, The University of Southampton, Southampton, S09 5NH, United Kingdom

Abstract

In order to obtain the highest possible performance from programs running on massively parallel machines it is essential to identify precisely where and when computational resources are consumed during their execution. A number of performance visualisation tools have evolved to meet this need for particular systems but they are often not portable to other machines. We regard portability as crucial to the widespread acceptance and use of such tools, and have investigated several approaches to achieving it. Each approach has been based on the public domain ParaGraph tool, which enables trace data collected during a program's execution to be viewed from various different visual perspectives. One approach is for programs to use the portable instrumented communication library PICL, which directly generates trace data in the appropriate format. Alternatively, trace files produced by applications using other libraries can be converted into ParaGraph format using trace filter programs. In this paper we report on an implementation of PICL for transputers and on trace filters we developed for the PARMACS and Express libraries. We also describe ongoing work within the PPPE Esprit project to integrate ParaGraph into a portable parallel programming environment based on the PCTE portable common tool environment.

1. Introduction

With the announcement by both IBM and Cray of high performance computing strategies based around the development of parallel systems [15] the question is no longer one of whether or not massively parallel machines will be a major force in computing, but of how they can best be exploited. One of the key issues which must be addressed if such systems are to be made as usable as their sequential counterparts is the development of suitable programming environments. Performance monitoring takes on an increased importance in a parallel environment, since the behaviour of a program can be so much more complex than on a sequential machine. In order to obtain the optimum performance from an application program it is necessary to be aware of the program's behaviour at run-time, and tools for debugging and performance visualisation must be provided within any such environment. We believe that, as has been the case with sequential systems, portability will be crucial to the success and acceptance of these tools.

Within the Esprit-2 GENESIS project we have developed a tool based on ParaGraph - a graphical visualisation tool from Oak Ridge National Laboratory, Tennessee [8]. It is a post-execution tool for visualising the behaviour of parallel programs on message passing multiprocessor machines. Data recorded during program execution may subsequently be viewed repeatedly, from different perspectives, in order to gain insight into the execution of the program. The multiplicity of ways offered by ParaGraph to view the information is an advantage it has over other similar tools [7]. An overview of the program is given in the next section.

ParaGraph was initially written for use in conjunction with PICL, a *Portable Instrumented Communication Library* [5, 6]. The library was designed to be easily implementable on a wide range of message-passing architectures, by virtue of offering a generic form of their most common programming model. PICL was designed not only to be easily portable, but also to be *instrumented*, and can optionally generate a *trace file* containing a record for each event of interest that occurs during the execution of a program. It is this trace file that forms the input to ParaGraph. Both ParaGraph and PICL have been placed in the public domain. The relationship between them is illustrated in figure 1.

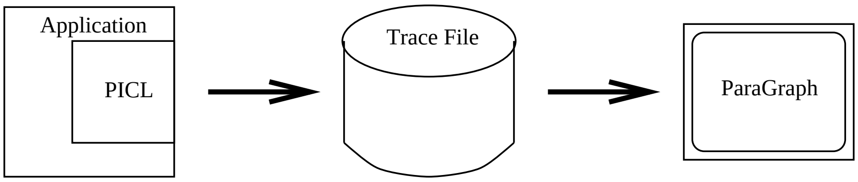

Figure 1. The Relationship between ParaGraph and PICL

PICL is described further in section 3, and section 4 describes some details of an implementation of it we undertook for transputer systems. However, we wished to be able to use ParaGraph in conjunction not only with PICL, but also with some of the other communication libraries for message passing systems that are instrumented to produce trace files. Although their trace file formats differ, there is often sufficient similarity in the kind of information recorded that the possibility exists to write *trace filter* programs to convert from one to another, possibly with some restrictions. We have demonstrated this by implementing trace filter programs to convert traces generated by the PARMACS [16] and Express [4] libraries into the format recognised by ParaGraph. Section 5 briefly discusses the filter programs and reports in detail on a case study using ParaGraph together with the PARMACS filter. The performance overhead incurred by trace generation is also examined.

Moving on to work in progress, section 6 introduces the PPPE Esprit-3 project, which aims to develp an integrated portable parallel programming environment. The focus of the project is on the integration of existing tools, and part of Southampton's contribution to the project will be to integrate ParaGraph using the PCTE database.

2. ParaGraph

A detailed description of ParaGraph can be found in [8] and also elsewhere in these proceedings [9], but we give an overview of the program here for completeness. However, since our tool was actually based on an earlier version of the public domain code, there has been some divergence between the features we support and those described in these references.

2.1. Design Objectives

The principal design objectives of the original version of ParaGraph were ease of understanding, ease of use and portability. The approach to achieve ease of use has been to make the meanings of the displays as self-evident and consistent as possible and, in particular, to provide a wide variety of them, offering different perspectives. There are two aspects to ease of use. The first is the use of an interactive graphical user interface that is mouse and menu-oriented, and the second is that the user's application program does not have to be extensively modified to obtain trace data, so long as a suitably instrumented communication library is used. There are also two aspects to portability. Firstly the graphical package itself should be portable. ParaGraph is based on the X Window System, which is supported on a wide variety of workstations. The program relies only on the standard Xlib library, which is available with any installation. Output to grey scale or monochrome monitors is supported, although it is not as effective as colour. The second aspect of portability is that the package should be capable of displaying the execution behaviour of programs running on different types of parallel machines. A degree of such portablility is achieved by using PICL, and this has been extended with the development of the trace filter programs described later.

2.2. Displays

ParaGraph is menu driven with most user input provided by mouse operation, although some options require a small amount of keyboard input. The user selects how the trace file is to be visualised and preprocessing of the trace file determines all relevant parameters automatically (eg. timescale, number of processors) before 'playback' of the trace commences. These values may be overridden by the user, however. Many different display views are supported, and any combination of them may be viewed at the same time. While some performance visualisation packages provide graphical tools to browse through static snapshots of the performance data, a more dynamic approach was adopted in the design of ParaGraph, which captures a sense of motion and evolvement by giving a re-enactment of program execution. Figure 2 illustrates a selection of displays, which are explained below:

- *Feynman*

  We have found this to be the most useful of the displays for getting an overall impression of the state of execution of the program. The thread of execution of each processor is represented by a horizontal line, drawn from left to right, which changes colour to indicate whether the processor is active, idle, or waiting for a send or receive to complete. In addition, message transfers are represented by diagonal lines from source to destination. The points where the communication lines meet the processor lines mark the send and receive times. This display shows particularly clearly if particular processors are idle because they are blocked waiting for a message to be sent or received. The sample trace depicted in figure 2 illustrates the point. After about 200 time units, the execution becomes sequentialized, as a single message is passed from one processor to the next, while the other processors wait. (The waiting is indicated by a colour change in the horizontal lines, which is more obvious on a colour display.)

- *Animation*

  Processing nodes are depicted as circles, communications between nodes are represented by lines and the status of each node is displayed by a colour which may be either busy, idle, sending or receiving. The line is drawn when there is a send and deleted when it is received. At the conclusion of the simulation all the lines of communication that were used are displayed, indicating the logical communication links between the processors.

- *Message Lengths*

  Messages are represented on a coloured square in a grid whose rows and columns correspond to sending and receiving processors. The length in bytes of each message is indicated by the colour of this square which is related to an accompanying legend. Each square displays data currently in transit, and is cleared once the data has been received. At the end of the simulation the display shows the cumulative communication volume for the entire run between pairs of processors. The colour that represents messages of a given size may be modified by the user via keyboard input in the appropriate window of the legend.

- *Kiviat*

  A geometric representation of individual processor utilization and overall load balance. Each processor is depicted as a spoke of a wheel. The fractional utilization of each processor corresponds a point on the spoke, with the hub of the wheel representing zero (completely idle) and the edge of the spoke representing one (completely busy). Taken together all these points on the spokes determine the vertices of a polygon whose size and shape indicates the load balance and utilization of the system. There is also a high water mark indicating the maximum utilization so far achieved.

- *Utilization*

  A scrolling display of the total number of processors that are busy as a function of time. The same information can be obtained from the Feynman diagram, by adding up the number of busy processors at any given time, but it is usefully summarised here.

- *Clock*

  Indicates current time in the simulation.

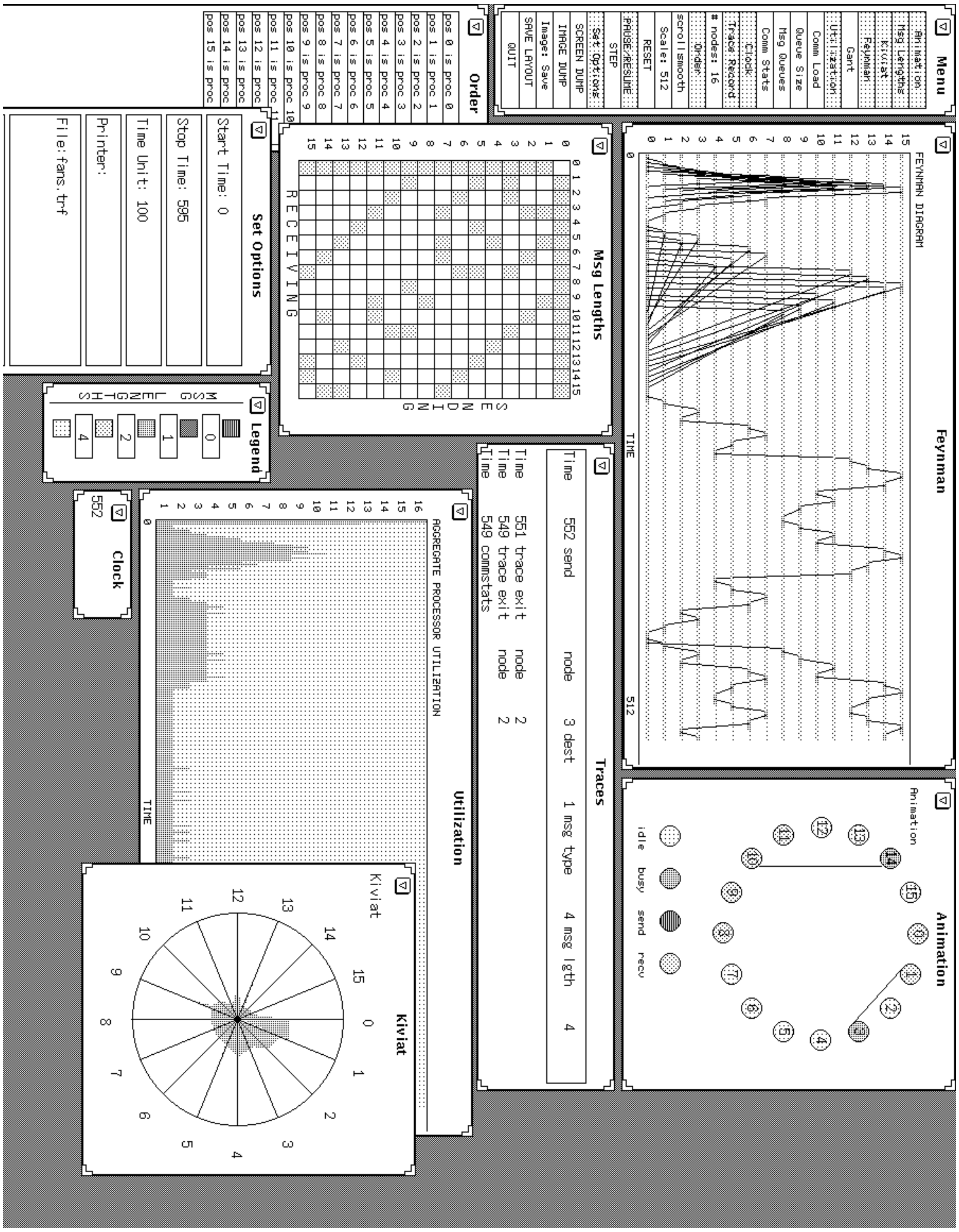

Figure 2. A Selection of ParaGraph Displays

- *Trace Record*

  The current trace record being processed and optionally, by expanding the window, past trace records.

- *Order*

  Enables the user to specify the ordering of processes in the displays, or to select a subset of the processes to be displayed, using keyboard input in the appropriate box. Automatic reordering ensures that a given process is not multiply displayed.

- *Set Options*

  Enables the user to modify the start and end times, the time step, the printer for screen dumps, and the trace and image dump file names.

- *Menu*

  This is ParaGraph's main menu. Display windows may be selected or de-selected by toggling their corresponding buttons, and replay of the trace may be started, stopped or single-stepped. A number of display options may be adjusted, and screendumps may be taken.

2.3. Modifications

Many modifications were made to ParaGraph, including the removal of a number of bugs. The graphical interface was enhanced by improving the resizing of windows and providing backing store, when available, to retain information when a window is obscured and re-exposed. Changes in the accessing of X events and trace records, so that priority is given to any user interaction, have led to an improved user response time that permits pausing in the middle of scrolling trace records. Buffering was also added to record multiple sends from a process before they are received and error detection was added in time stamping to detect receives before data is sent, which can occur due to incorrect clock alignment on processors. Other features added include a simple X window save and load to user-specified files, display of processor state in the Feynman diagram, display of processes waiting to send and receive in the Animation diagram and output of trace record histories (in a verbose form) by expanding the trace file window. To reduce the bias towards machines with hypercube connectivity, the selection of a gray coding option was replaced with the ability to select any ordering and any number of processors (rather than just a power of two).

The most significant modification made to ParaGraph was, however, the enhancement of its repertoire of recognised trace records to include blocking sends, which are not part of the PICL model of communication, while retaining the ability to recognise standard PICL format traces. This extension was found necessary in order to properly visualise traces produced by other libraries that include blocking sends in their communication model. The new type of event involved adding a "sending" processor state analagous the the existing "receiving" state, represented visually by different colours on the display.

2.4. The Trace File

In order to achieve hardware and software independence, the interface to tools such as ParaGraph must be abstracted away from the operations supported by particular systems. The trace file provides just such an abstraction, and the set of allowed trace records and their meanings can be thought of as defining the operations permitted on an abstract machine.

Table 1 summarises the trace records recognised by ParaGraph. All types of events have a time stamp indicating when they were generated. This is represented as two integers (seconds followed by microseconds since tracing was enabled) and an integer identifying the processor which generated it. The *type* indicates the message type and is user defined and *lth* is the message length in bytes. The record types defined form a superset of those recognized by the original version of ParaGraph, synchronous sends having been added. PICL generates other record types but they are ignored by our tool.

Table 1
Trace file records

Trace Record						meaning
trace_start	clock 0 0	node #				start of tracing
message	clock # #	node #	"message"			user message
trace_mark	clock # #	node #	type #			user defined event
trace_exit	clock # #	node #				end of trace execution
compstats	clock # #	node #	busy # #	idle # #		cumulative busy & idle times
send	clock # #	node #	to #	type #	lth #	send message & continue
send_blocking	clock # #	node #	to #	type #	lth #	send msg & await receipt
send_waking	clock # #	node #				continue - message received
receive	clock # #	node #	from #	type #	lth #	receive waiting message
recv_blocking	clock # #	node #	type #			wait for message
recv_waking	clock # #	node #	from #	type #	lth #	continue - message received

3. PICL

PICL is a portable instrumented communication library designed to provide portability, ease of programming, and execution tracing in parallel programs [5, 6]. The library is written in C, but may also be called from Fortran via a set of interface routines. This section describes the features of the "original" version of PICL, and the next section describes our implementation of it for transputer based systems. Other implementations of this version of the library exist for the Intel iPSC/2, Intel iPSC/860, and nCUBE/3200 families of hypercube multiprocessors and for the Cogent multiprocessor workstation. A newer version with additional features and implementations is now also available from Oak Ridge.

3.1. Programming Model

The PICL programming model can be regarded as a generic form of the message-passing programming environment typically found on distributed-memory processors. It assumes a set of autonomous processors, each possessing a fixed amount of memory to which no other processor has access. Processors share data by passing messages to each other. A program issuing a send command is idle (or *blocked*) until the message data has been copied to a system buffer. The program then resumes execution, while the data transfer to the receiving processor's system buffer proceeds asynchronously. A processor issuing a receive command is blocked until a message satisfying the request has arrived and been copied from the system buffer into the user's data area. Thus the programming style is *asynchronous* rather than synchronous, which compensates to some extent for the limitation that only one process is allowed per processor.

Messages are allowed to be sent between arbitrary pairs of processors, irrespective of the underlying hardware communication network. Of course the time required to send a message will depend on the actual network, and the user will need to be aware of such dependencies in order to write efficient code.

The model distinguishes one processor, the host, from the rest, which are called nodes. Only the host has access to operating system services such as I/O to disk, screen and keyboard. The host is responsible for explicitly allocating nodes and loading programs onto them. Although they may all be different, a typical application consists of one program that runs on the host and another that runs on each of the nodes.

3.2. Programming Interface

The routines that form the programming interface to PICL fall into three distinct sets: a set of low-level generic communication and system primitives, a set of high-level global communication routines, and a set of routines for invoking and controlling the execution tracing facility. The twelve low-level primitives that implement the basic programming model are listed in table 2.

Table 2
Low-Level PICL Routines

`open0:`	Open interprocessor communication.
`close0:`	Close interprocessor communication.
`load0:`	Load a node program.
`sync0:`	Synchronize the processors.
`send0:`	Send a message.
`recv0:`	Receive a message.
`probe0:`	Check for an arriving message.
`recvinfo0:`	Return information about the most recently received or probed for message.
`message0:`	Print a message on the standard output device.
`who0:`	Return processor ID number, host ID number, and number of node processors allocated.
`clock0:`	Read the system clock.
`check0:`	Enable or disable parameter checking.

In addition to the low-level communication primitives, PICL provides a set of high-level communication routines, such as broadcast, global maximum and barrier sychronization, that have proven useful in the development of application programs. These routines, which are built on top of the low-level routines, can help the novice avoid common synchronization and programming errors, and save programming time for experienced users. They are summarised in table 3.

Table 3
High-Level PICL Routines

`setarc0:`	Specify the underlying topology to be used by the high-level communication routines.
`getarc0:`	Return the parameters used in the most recent call to the `setarc0` routine.
`barrier0:`	Synchronize the processors.
`bcast0:`	Broadcast the elements of a vector from one processor to all other processors.
`bcast1:`	Broadcast the elements of a vector, taking advantage of pipelining during the broadcast.
`gand0:`	Compute the element-wise global "AND" of a distributed set of vectors.
`gcomb0:`	Compute the element-wise global function of a distributed set of vectors where the function is a user-supplied routine.
`gmax0:`	Compute the element-wise global maximum of a distributed set of vectors.
`gmin0:`	Compute the element-wise global minimum of a distributed set of vectors.
`gor0:`	Compute the element-wise global "OR" of a distributed set of vectors.
`gprod0:`	Compute the element-wise global product of a distributed set of vectors.
`gsum0:`	Compute the element-wise global sum of a distributed set of vectors.
`gxor0:`	Compute the element-wise global exclusive "OR" of a distributed set of vectors.
`ginv0:`	Compute the inverse Gray code transformation of an integer.
`gray0:`	Compute the Gray code transformation of an integer.

The third set of routines provided by PICL offers control over the amount and type of tracing. Trace data may optionally be produced by code within the low-level primitive and/or high-level communication routines. Table 4 shows the routines available.

4. A Transputer Implementation of PICL

Bare transputers provide only nearest-neighbour bi-directional communication over 4 links, which presents a problem when it comes to implementing a communication library

Table 4
PICL Routines for Tracing

`traceenable:`	Enable tracing and specify the name of the trace file.
`tracehost:`	Begin tracing on the host.
`tracenode:`	Begin tracing on a node processor.
`tracelevel:`	Specify the amount and type of trace information.
`traceinfo:`	Return the current `tracelevel` specification.
`tracemark:`	Generate a user-typed trace record.
`tracemsg:`	Write a line of text directly into the trace file.
`traceflush:`	Send trace information to the trace file *now*.
`traceexit:`	Stop tracing.

like PICL, which allows messages to be passed between arbitrary pairs of processors. Clearly some message routing software is required. Rather than implement a router from scratch, we based our implementation of PICL on VCR, Southampton's Virtual Channel Router [2]. VCR provides 'virtual' channels, multiplexed over transputer links, that can connect pairs of processes placed on arbitrary processors, and therefore makes all-to-all communication possible. VCR presents a truly distributed channel connected system, but is still some way from the support necessary for the PICL model of communication. Notably the mailbox is missing. This is the usual collection point for messages, forming the interface between the communication medium and the processes. The mailbox is implemented in an **occam** harness that is attached to the PICL program. The following subsections describe the overall structure of PICL, how VCR is used, and the details of the harness. We then conclude this section by reporting on the current status of our PICL implementation and discussing some of the areas of difficulty that we encountered in producing it.

4.1. The Structure of PICL

PICL consists of two layers of code, a machine dependent layer, and a machine independent layer. The machine independent layer provides the interface functions outlined in the previous section. They make calls to the machine dependent routines, which encapsulate all of the machine specific calls. Not all of the underlying calls in the lowest layer of PICL involve communication - many service routines are provided to effect the status of 'global' variables that store the state of PICL and involve no communication.

In implementing PICL for a new machine, all that should be required is to rewrite the machine dependent routines in terms of the calls available on the new target. The machine independent layer consists of two identifiable sets of calls that provide high and low level functions. The high level calls are written in terms of the lower level calls, hence comprehensive testing at the low level should be sufficient to ensure correct operation of the higher level calls.

4.2. VCR

In our implementation of PICL the machine dependent operations are written in terms of VCR calls. The implementation of VCR is fully embedded within the **occam** language, preserving **occam** syntax and semantics, but removing the restriction imposed by the transputer's limited number of physical links. The harness routines are written in **occam**, which is the native language for expressing parallelism on the transputer, and was developed jointly with it, making it very efficient. In the process of forming an executable PICL program we actually build a mixed language application although the user may never need to be concerned with this. The linker sees the compiled C program as if it were encapsulated within the **occam** harness.

4.3. The PICL Harness

The harness is formed from host and node components, with the host acting as a master that must always be present, and as a gateway to the user through an interface to the system hosting the transputer network. The rest of the distributed harness is composed of replicated instances of the basic node component. These components are each formed from a number of **occam** programs. There is an instance of a common buffer program for host and node components, a program forming each of the host and node mailboxes, and a further two programs that form the C interface attached to the respective C programs. Communications from within the PICL program are implemented by a pair of channels passed through the C interface to the PICL program providing bi-directional communication with the mailbox serving that virtual processor. Internal codes are sent across these channels to select different types of transaction, such as send, receive, probe etc. The mailbox is served by the VCR kernel that supports the routing of the messages through the transputer hardware links. The creation of the network of dynamic virtual channels and the remote procedure call used in process creation are all done using VCR. An overall view of the harness structure is shown in figure 3. The buffers, which are not shown explicitly, are within the receive() processes.

The host part of the harness is formed of four basic components. At the centre is the memory buffer used to store the messages, and various routines that add remove, and test the buffer in various ways. Then there is a server process that performs all the transactions on the contents of the buffer. The buffer server processes requests from two sources; any incoming traffic from the node-node interconnect, and requests routed from the PICL program acting on the mailbox message buffer. A second dedicated server processes the PICL program's requests, either routing the transaction request on the buffer to the process serving the buffer, or by injecting (sending) the message to the interconnect. The final component is the **occam** process which supports the C interface to which our PICL host program is attached.

The node part of the harness is composed of similar elements, only differentiated in function along the lines of master-slave. Upon the initial loading of the transputer network resource, a node is loaded with a component of the communication harness. This program connects the node to the network by dynamically creating four virtual channels. Two channel ends are passed to form the connection to the node interconnect (basically attaching to the ends of the array of channels that forms the communication medium), across which it then can receive the node program under PICL control. The remaining two channels are passed through the C interface to the user's PICL node program. Once the code of the node program is sent, the user process is started on the node, at which time the thread of control passes to the PICL program. Finally on terminating the C node program, it disposes of the dynamic channels created (that is the two connecting to

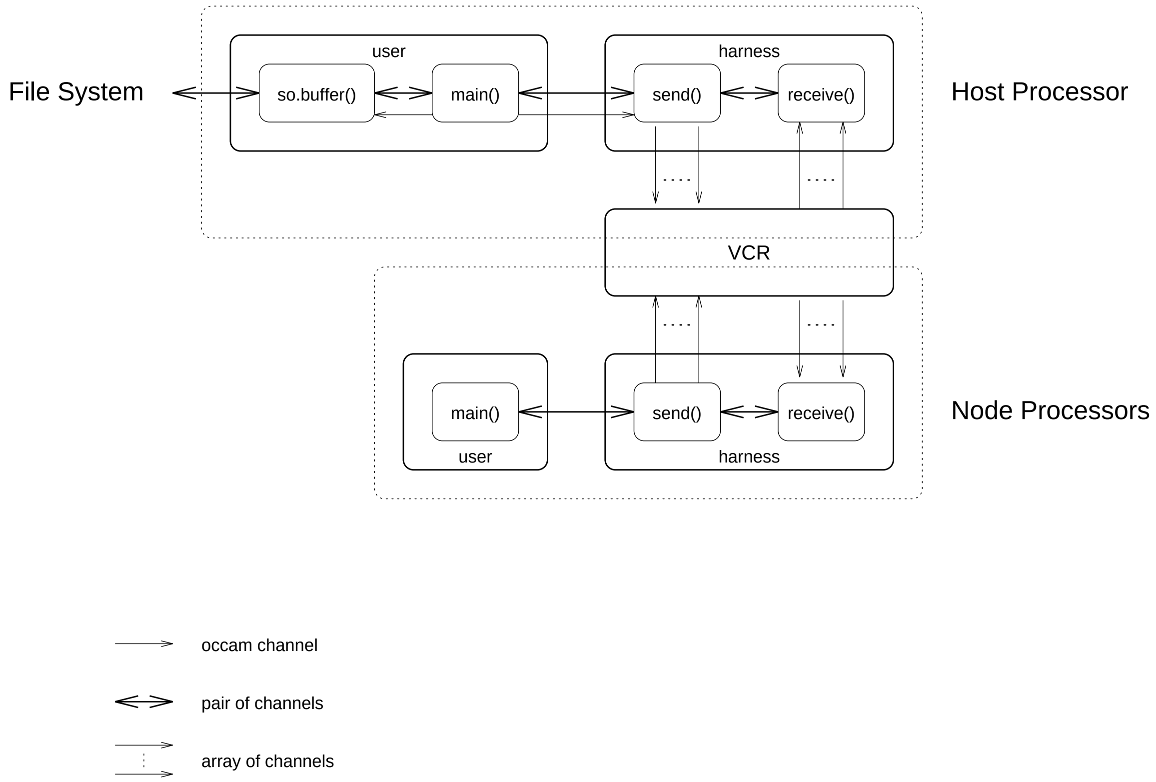

Figure 3. Simplified view of harness processes

the C node program, and the two that connect it to the node interconnect).

4.4. Implementation Status

We now have working implementations of most of the examples given in the PICL literature (adjusted in some cases so as not to incur a data size problem in communication), plus some others that have been created in the course of testing. Simple programs (utilising low level functions only) have been proven and stable, providing correct, and deterministic behaviour from these codes. The work has proven the viability of running PICL on VCR on transputers, and even provided sufficient workspace for small applications on systems with only 256 kilobytes of memory per node. In short, we have produced a usable, functioning, albeit basic, implementation of PICL. In the course of the work we encountered one or two areas of particular difficulty, not all of which have yet been completely resolved. They are discussed below.

4.4.1. Time

This is an area of obvious implementation dependence. Many of PICL's 'magic numbers' were locked away amid the large amount of variable state that is stored, and underlies PICL. Many of these numbers had to be sorted out to be correct for the transputer implementation, as many were carried over from an iPSC860 version of PICL which we modelled this implementation on. This accomplished, the time dependent aspects of

PICL were mainly straightened out. Tracing, and clock routines now return sensible figures (usually!). However there is still some more to be done here, since it currently will not handle the complete cycle of the clock correctly (i.e. overflow), and would therefore return rubbish if an application had a long lifetime - more work needs to be done here in the near future.

4.4.2. Termination

One of the most major 'quirks' encountered affected the termination of PICL programs, where a host program was permitted to terminate prior to the nodes. This proved to be a problem for our implementation because it was only while the host program was active that it retained the services of VCR, and indeed it used the termination of the host to signal to the file server that it had finished it's use of the transputer resource. Resources that are claimed by the host process to run 'child' node processes are deemed to be dependent on the 'parent' host process and have no lifetime after the host process has finished. It was therefore required to insert code to ensure that termination of the node processes completes prior to the host being allowed to terminate and subsequently release the resources, and hence provide acceptable program behaviour. The node status check was achieved by examining the state stored in the node processor state variable array maintained on the host about the current state of the nodes. Programs that then did not feature an inherent synchronisation at the end of the application by virtue of returning trace data, were then required to wait for the signal to the host that a node was then inactive (i.e. had no program awaiting host service, or any loaded program in an unfinished state).

4.4.3. Routines employing global communication

At present there is an unknown problem with the high level routines which causes them to deadlock, rendering them effectively unusable. It is suspected that they do not fail because they are wrong, but for a more subtle reason associated with the implementation. However, it has proved a difficult task to trace the source of the problem. There exists a restriction within the current implementation that imposes a limit of 255 bytes per message, and we suspect that this may account for the failure of the global routines, since underlying every global function is the global combine call, which concatenates the data to form messages of increasing length.

5. Trace Filters

Many other packages generate trace data and it is possible in many cases to devise transformations that will convert it into the syntax described previously. A trivial example is that PICL can generate two formats of trace file that are semantically equivalent; verbose and compact, both ASCII. The verbose format is easier to comprehend, where each trace record is identified by a keyword and all numeric data preceded by a descriptive keyword. The compact format contains just an integer representing the keyword and the numerical values as before, with no descriptive keywords. More complex transformations may in general need to combine more than one record from the input trace to generate each output record.

The PARMACS [16] and Express [4] packages were selected as test cases to investigate the feasibility of developing trace filters in practice. EXPRESS generates three files: a profile of program functions called, a profile of the communication functions called, and an event profile. The event profile contains all the information required to allow it to be mapped into ParaGraph format. PARMACS does not generate all the information required, message lengths for example, but enough is produced to generate some of the trace records previously described. Typically this provides enough information on the execution of the system to make monitoring useful.

Implementation of both filters proved to be easier than had been expected. In the case of Express it is was found that the implemenation of the filter took significantly less time to complete than that of an implementaition of PICL for the same target hardware (transputers), despite having to reverse-engineer the trace file format. The trace filters extend the use of ParaGraph to include the following combinations of hardware and software:

- PARMACS
  - Cray Y-MP
  - Intel iPSC/2
  - Intel iPSC/860
  - nCUBE 2
  - SUPRENUM
  - Meiko (Transputer & i860 versions of CSTools)
  - Transputers
- Express
  - Transputers (PC hosted)

Because of its central role in the GENESIS and PPPE Esprit projects in which we have been involved, we have made more use of the PARMACS filter than of the Express filter. We report below on experiences using ParaGraph with a real application, in conjunction with PARMACS, and briefly examine the performance overhead incurred when tracing is enabled.

5.1. A Case Study Using Paragraph with PARMACS

The Paragraph performance analysis tool has been successfully used to discover the source of poor performance of the Genesis LPM1 benchmark [11] on the Intel iPSC860 and Touchstone Delta computers.

5.1.1. The LPM1 Benchmark

This benchmark is the time dependent simulation of an electronic device using a particle-mesh or PIC-type algorithm. It is a two-dimensional (r, z) geometry with the fields computed on a regular mesh (33x75). The fields computed are E_r, E_z, and B_ϕ(into the mesh). The electron distribution in the device is represented by a group of 500 to 1000 simulated electron clouds each of which has its positions and velocity stored. The timestep proceeds as follows:

1. The particle coordinates are inspected, and the current of each particle is assigned to the mesh points in its neighbourhood.
2. The fields are advanced for a short timestep according to Maxwell's equations.
3. Each particle is accelerated according to Newton's Laws with the new field values obtained by inspecting the mesh values near each particle.
4. Injection takes place along the cathode of the device according to the value of the normal E-field.

This is called a local particle-mesh simulation because the timescale is such that only neighbouring (i.e. local) field values are required during the update of the field equations. In contrast a global simulation might require the solution of, say, Poisson's equation over the whole mesh. Because of the locality of data a geometric subdivision of the problem between the nodes of a transputer network (or other distributed-memory computer such as the Intel iPSC/860) is natural. The existing distributed-memory implementation takes this approach, and performs a one-dimension domain decomposition. For p processors, the device is divided into p approximately equal slices in the z-direction, and each slice is assigned to a different processor. Each processor is responsible for computing the fields, and moving the particles in its region of space. When particles move out of this region, their coordinates are transferred to the correct neighbouring processor, which computes their subsequent motion. The timestep is such that a particle will move no further than to a neighbouring processor in one timestep. The processors are therefore configured as a chain, and during the timestep loop, communication is only required between neighbouring processors in the chain. The communication comprises an exchange of the edge values of the fields and currents, together with the coordinates of any particles that are moving between processors. The standard benchmark is defined as the simulation of one nanosecond of device time using a (33,75) mesh and starting from an empty device. During this time electrons are emitted from the cathode starting at the left of the device, and the electron clouds gradually fill the device from left to right. After one nanosecond the device is about half filled with electrons. Larger problem sizes are introduced by using more mesh-points in the z-direction, and computing on a $(33, 75\alpha)$ mesh, where α is the problem-size factor (=1,2,4,8). The charge on each particle (which represents a simulated cloud of electrons) is divided by α, so that the number of particles is also approximately multiplied by α as the problem size increases.

5.1.2. Benchmark Results

Figure 4 shows the benchmark performance, $R_B(\alpha; p)$, as a function of the number pf processors, p. The curves are for four constant problem sizes specified by $\alpha = 1, 2, 4, 8$. The data at the bottom starting at about 0.27Mflop/s shows the normal scaling behaviour of this benchmark, and were measured on the Parsys Supernode SN1000 with up to 60 INMOS T800 25MHz transputers. For a single processor the four problem sizes generate closely the same benchmark Mflop/s , whereas for larger numbers of processors the larger problem size generates more benchmark Mflop/s , because of the greater computational intensity of the calculation. The computational intensity is defined as the ratio of arithmetic operations to words communicated [12, 13]. The performance for all

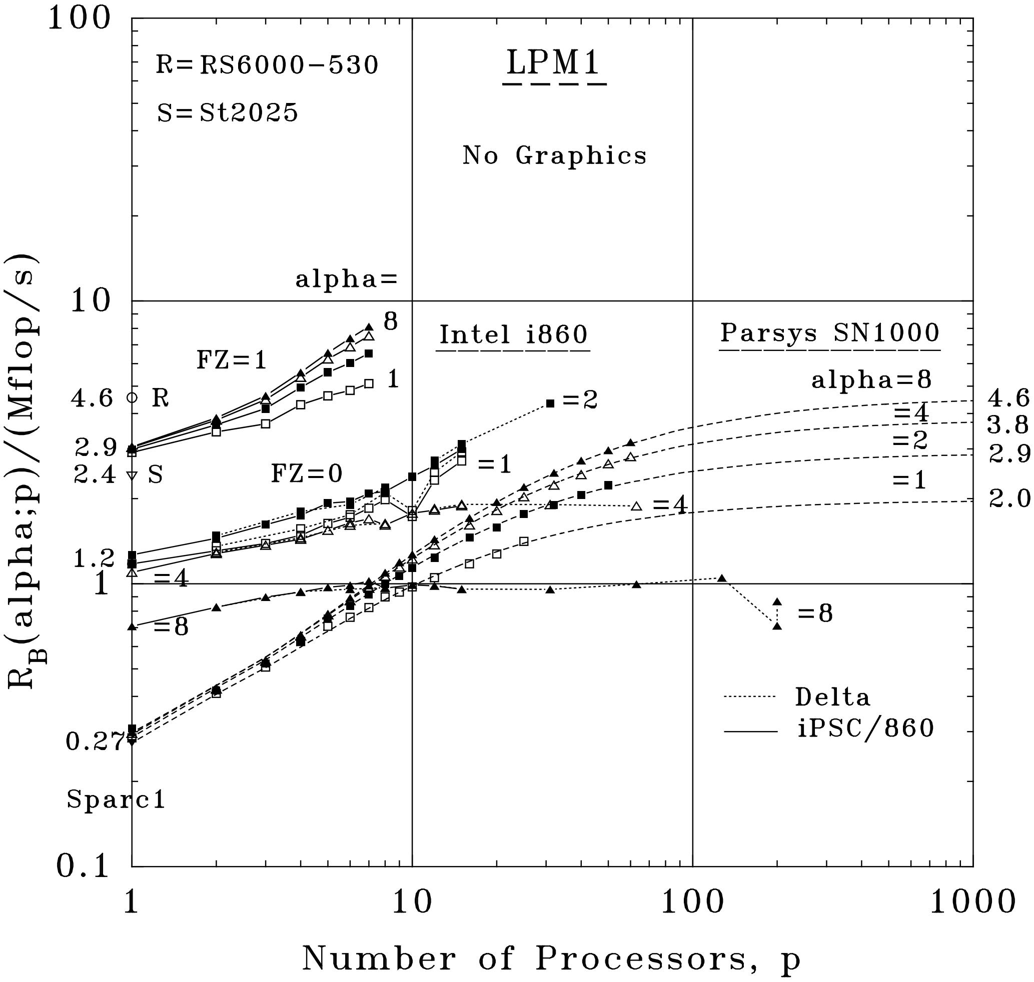

Figure 4. Benchmark performance of the LPM1 benchmark for four problem sizes as a function of the number of processors used, on a Parsys SN1000 Supernode with 25 MHz T800 transputers, the Intel iPSC/860 and Touchstone Delta. α is the problem-size factor. FZ is the value of the *flush zero* flag in the floating-point status register of the Intel i860 chip. FZ=0 generates a system trap on underflow, FZ=1 replaces the underflowed value by zero, without generating a trap.

problem sizes shows Amdahl saturation as the number of processors increases, as shown by the dotted lines which are a theoretical extrapolation from the measured values up to 60 processors [11]. The saturation arises from all terms in the timing formula which are independent of p, in particular message startup times, and any code that is repeated in all processors (i.e. unparallelised code outside of the parallelised DO-loops).

The performance curves starting around 1 Mflop/s for one processor show the result of running the same LPM1 benchmark on the Intel iPSC/860 (the solid lines up to 15 processors), and the Touchstone Delta (the dotted lines up to 200 processors). The data is almost indistinguishable where it overlaps. The scaling behaviour is abysmal, and we actually see a speedup of unity for 200 processors on the Touchstone Delta for the largest problem size (which should give the best result). This is my (rwh's) entry for the worst speedup prize. Something is clearly going terribly wrong when this benchmark is run on the Intel iPSC/860 and Touchstone Delta. Since these computers both use the same arithmetic chip (the original i860), but have quite different communication networks, hardware and properties [14], the almost identical bad behaviour of the two computers suggests that the problem arises in the arithmetic and not in the communications. It was therefore imperative to discover, by using performance tools, what was going wrong, and this is where Paragraph proved its usefulness.

5.1.3. ParaGraph Identifies an Anomaly

The LPM1 benchmark was run under PARMACS for the simple case of 3-processors and $\alpha = 1$, and the trace files were generated for input to the Paragraph performance analysis tool. We found the Feynman-diagram the most useful display, and figure 5 shows the timing for the electric-field update and the electric-field exchange in the first step of the simulation. The electric-field update occurs during the period of processor activity after each processor has engaged in its first four communication events. The electric-field exchange is represented by the two subsequent communication events. The problem is divided into three slices of 25 mesh points each, and given to processors 1, 2, and 3 which are translated to a linear chain of physically connected processors on the Intel iPSC/860 hypercube. The processor numbering that we use here is that used by Paragraph and differs from the numbering of the processors on the hypercube. There are two data exchanges close together just before the field update in which processes 1 and 2 exchange values in parallel (i.e. simultaneously) with processes 2 and 3. These show up as two pairs of crosses at the left of the diagram. The field update is also seen to take place in parallel in the three processors, and to take closely the same time in each processor, because each is calculating the same number (=25) of mesh points. In contrast figure 6 shows the same section of the timestep at the end of the run (step=104). The two exchanges just prior to the field update are seen as before at the left of the figure. Processors 1 and 2 then compute for roughly the same amount of time as in step 1 (actually about 30 percent longer). Processor 3, however, takes 4.6 times longer than the same calculation took in step 1. This results in a distortion of the following field-exchange (XE2), and substantial idle time in processors 1 and 2 as they wait for processor 3 to finish. This lengthening of the compute time in processor 3 is observed for many timesteps near the end of the run, and the idle wait time associated with it could explain some or all of the poor performance.

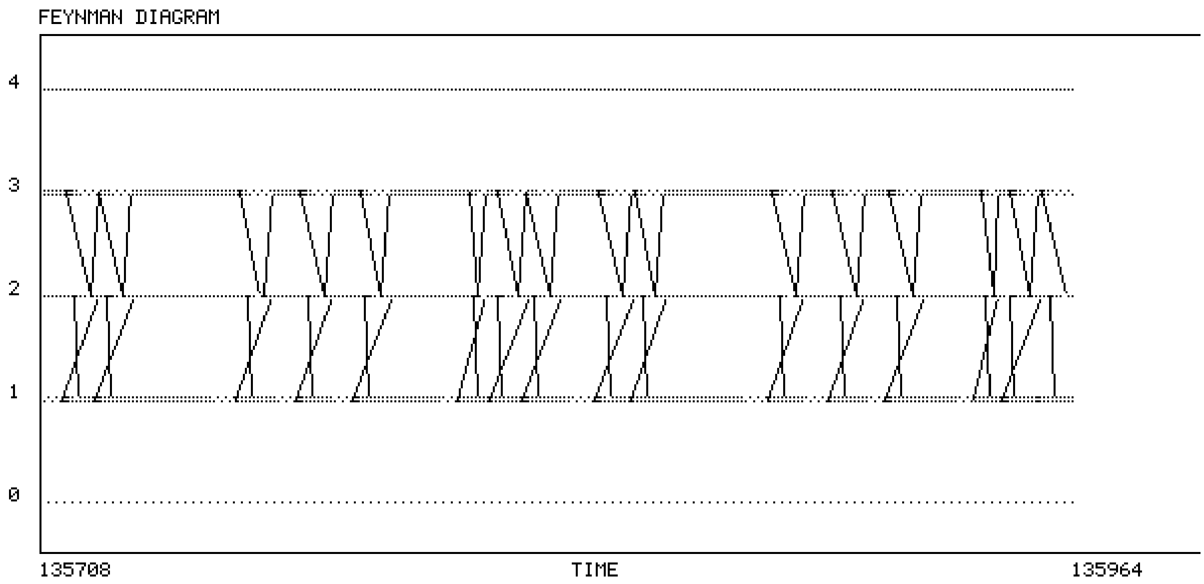

Figure 5. Feynman diagram of the communication and calculation of the LPM1 benchmark on the iPSC/860 with the *flush zero* bit FZ=0, showing the electric field update and electric field exchange in the first timestep of a 3-processor run. Processor 0 (the host) and processor 4 do not take part in the MIMD simulation.

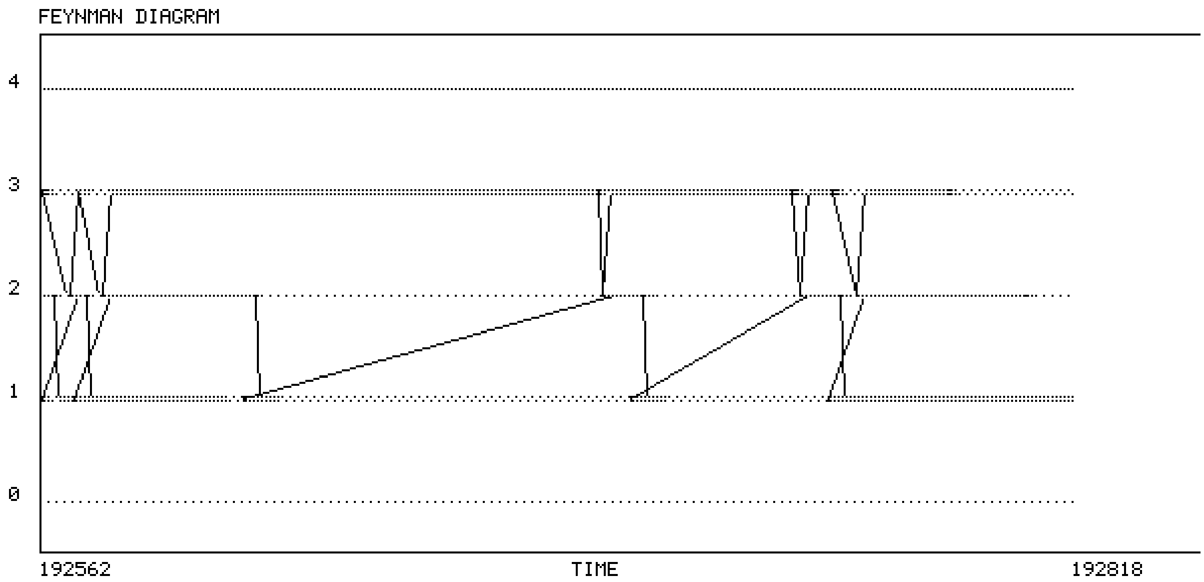

Figure 6. Feynman diagram of step 104 of the LPM1 benchmark run on the iPSC/860 with the *flush zero* bit FZ=0. Notation and scale the same as figure 2.

5.1.4. Establishing the Cause of the Anomaly

Paragraph as presently implemented through the PARMACS communication macros, is unable to provide further information as to why the same calculation in processor 3 takes almost 5 times longer in step 104 than it does in step 1. The E-field update takes place in two DO-loops, and there is also intermediate calculation and a call to the system clock. Because PARMACS/Paragraph does not provide the facility of inserting user-defined events within the user code, we were unable to trace exactly where within the user code, the extra calculation time was occurring. Such events, if they had been available, would give rise to corresponding markers or time stamps in the Paragraph trace file and output diagrams, thus enabling a detailed investigation of the timing within the user code.

However, using the Parasoft Express performance analysis tool included in the Intel iPSC/860 software, we were able to obtain an Express 'e-tool' trace of the same 3-processor run. This showed the identical anomalous timing phenomenon, but provided additional information of the entry and exit times of all user subroutine calls. Express also allows the insertion of user-defined events including the output of the values of selected user variables at the time of the event. With this facility we were able to show that the extra computation time was occurring in both DO-loops, and that the intermediate calculation and the call to the system clock took negligible time. This still left the question as to why the same DO-loop takes longer to compute in processor 3 than it does in processors 1 and 2, and much longer than it did during the first timestep?

We are indebted to Mark Pinches of the Southampton Parallel Application Centre for suggesting that floating-point exceptions might be occurring during execution, and that such traps to a system handling routine would lengthen the computation time considerably. This had been the explanation of strange timing behaviour in other applications run on the iPSC/860 at the centre. No such exceptions had been reported during the running of the program, but we learned that the iPSC/860 operating system does not report any exceptions to the user, not even floating-point overflow! However a C-routine, traps.c, callable from Fortran, is available that reports the number of all exceptions generated up to the time of the call. Using this routine we found that about 25,000 floating-point underflow traps were generated in processor 3 over the complete run of 104 steps, whereas processor 1 generated 190 traps and processor 2 about 5,000. Furthermore no underflow traps occurred at all on any processor before step 19. Thus the observed difference in timings could all be explained by the servicing of underflow traps taking, as they do, a certain time. The underflowed numbers also lead to 'inexact' and 'source' exceptions, but no other exceptions were reported.

5.1.5. The Remedy

The generation of floating-point underflow during the execution of a program does not necessarily indicate an error in the calculation, but only that a number smaller in magnitude than the smallest representable number, has been computed. The LPM1 benchmark uses single-precision arithmetic, and the i860 uses IEEE floating-point format. This means that underflow occurs for numbers smaller in magnitude than about 10^{-38} which can quite legitimately by replaced by an exact zero. It is not difficult to produce such numbers when taking the difference of nearby (and therefore similar) field values during the field-update step of the benchmark. The i860 offers two alternative actions if underflow occurs. If the *flush zero* bit (FZ) of the floating-point status register (FSR) is zero, a result exception is generated and the user program is suspended whilst the system handling routine is executed. If, however, FZ=1 and underflow occurs, the result is set to zero, and no trap due to underflow occurs. It is this second option that we clearly wish to use in the LPM1 benchmark (and probably most others). Unfortunately the Intel iPSC/860 software appears to have FZ=0 as the default, so that all floating-point underflows cause a trap and waste time. This seems particularly pointless because, having wasted the time in the trap, the system doesn't even use that time to report to the user that a trap has occurred.

Fortunately there exists a simple C-routine, flush.c, that forces FZ=1 when it is called, and thereby disables floating-point underflow traps. If this is linked into the Fortran link step and called as the first statement of the program, then no underflow traps will be generated. A call to TRAPS as the last statement of the program, will report if any other exceptions, such as overflow, which we do want to know about, have occurred during execution.

Figure 4 also shows the benchmark performance obtained with the underflow trap disabled by setting FZ=1. The 1-processor performance has been improved by a factor 2.4 to 2.9 Mflop/s for all problem sizes. The performance then scales similarly to that of the Parsys SN1000 up to 7 processors, which were the most available at the University of Southampton Parallel Application Centre (PAC). The curves suggest that improved performance will be obtained well into the tens of processors. We now find that the largest problem ($\alpha = 8$) generates the most Mflop/s as is expected in normal scaling behaviour, and the performance for 7 processors has improved by a factor of about nine. All the weird and anomalous scaling behaviour which was observed in the first benchmark runs with FZ=0 has disappeared.

5.1.6. Remarks

It is also worth saying that the hopelessly poor parallel performance of the Intel iPSC/860 and Touchstone Delta on this realistic electronic device simulation benchmark, was entirely due to a single bit being set to zero instead of one. No doubt this phenomenon of floating-point underflow destroying both good serial and parallel performance will occur in other user programs. Intel would be well advised to consider whether they should change their system default to FZ=1 rather than FZ=0, particularly since their system does not report such traps to the user even when it bothers to generate them. This unwise choice of default can unnecessarily bring their computers and the i860 chip into disrepute.

5.2. The Performance Impact of Monitoring

While performance monitoring tools incontestably offer advantages for debugging and optimization of parallel programs, they can lead to a certain degradation of system's performance. Benchmark measurements were made for PARMACS, with and without trace generation, in order to assess the magnitude of performance degradation caused by the addition of performance monitoring. The measurements were taken using the COMMS1 benchmark from the GENESIS distributed-memory benchmark suite [1] on the iPSC/860 machine at Daresbury. This machine has Intel's Microcomputer/386 as a front-end with the AT & T UNIX System V/386, Release 3.2 and the 40 MHz i860's as node processors. The Fortran compiler was the iPSC Fortran Compiler Release 2.0, which incorporates the Portland Group products.

The COMMS1 benchmark performs a single message 'ping-pong' between two neighbouring nodes. The transfer time is in fact half of the total time, measured by the benchmark for different message lengths. The measurements were taken from PARMACS, which results in about 4-10% overhead in the start-up time if comparing it with the native Intel communications [14]. This is actually the *price of portability* on iPSC/860, which one should pay in order to have a portable program across different distributed-memory computers using PARMACS macros. Two series of experiments were performed regarding the performance monitoring overhead - one with and one without a trace generation (Table 5.2).

Table 5
Start-up time in μs and bandwidth in *Mbyte/s* for different experiments on iPSC/860 using COMMS1

Msg. type	Native message-passing		PARMACS macros		PARMACS + trace generation	
	Start-up	Bandwidth	Start-up	Bandwidth	Start-up	Bandwidth
Short	76.0	2.40	85.3	2.12	157.1	2.32
Long	205.0	2.80	212.8	2.80	285.1	2.80

The influence of performance monitoring tools on the system performance is a function of message length (see Fig. 7). The results show a constant increase of $72\mu s$ in the start-up time when the trace generation is switched on. This constant increase is about half of the whole transfer time for very short messages, while it can be neglected for long messages. This corresponds to 47% and 11% of the total communication time, for message lengths of 1 and 1000 bytes respectively.

This overhead arises because of the additional software code for trace generation inserted in the communication subroutines calls. Every example was repeated 100 times. The time resolution (1 ms) is still somewhat rough, especially for short messages, but reasonable enough for comparison. This is most likely the reason for a slight decrease in the transfer time for short messages.

It should be mentioned, however, that the overheads reported above are relatively very small in comparison with the time for writing the trace file ($>$ 2 Mbyte long in this experiment) on a disk at the end of the execution, which is the major time-consuming operation for the trace generation. The run-time overheads, however, are much more important, as they give an idea of the possible alterations to the communication/calculation ratio, the communication's overlapping, etc. of a given parallel program, caused by the performance monitoring tools.

6. A Portable Parallel Programming Environment

So far in this paper we have addressed only the portability aspect of a parallel programming environment, but another important requirement is that of tool integration.

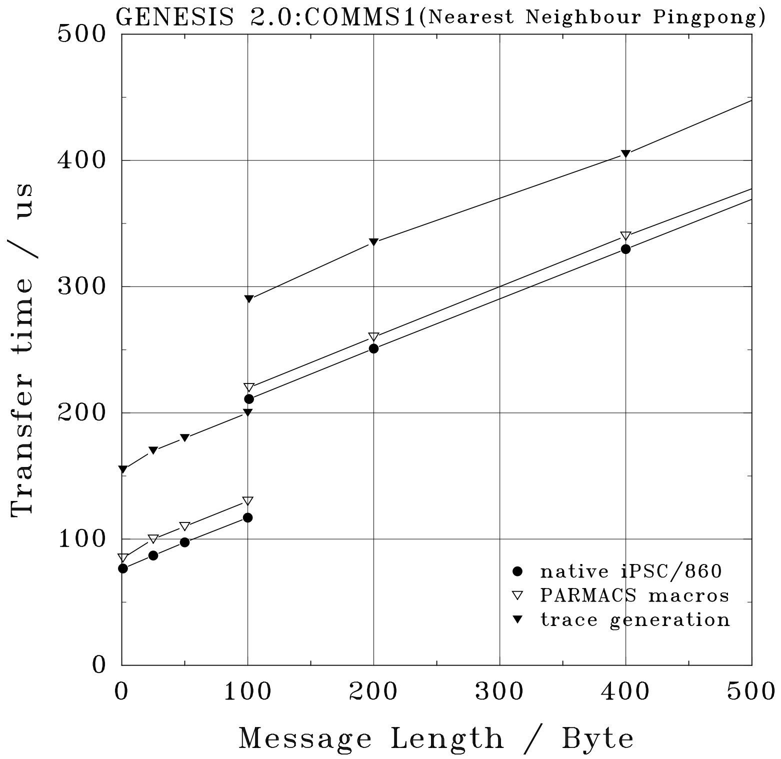

Figure 7. The time for a single message transfer, as a function of the message length on iPSC/860.

In this section we introduce the PPPE project, which is focusing on the integration of existing tools rather than the development of new ones. One of our activities within the project will be to integrate our version of ParaGraph into the PCTE portable common tool environment.

6.1. The PPPE Project

PPPE (Portable Parallel Programming Environment) is Esprit project 6643. It started in July 1992 and will last for three years. The project aims to deliver a well-integrated production quality toolset whose purpose is to simplify the development of large scale scientific and engineering applications for massively parallel distributed memory architectures. The toolset will comprise:

- An explicit message passing system, a distributed object and workload management system and an event monitoring system. Collectively these components are referred to as the run-time system, and they execute on the target supercomputer systems, namely Meiko, Intel and IBM machines.

- A High Performance Fortran transformation system, a HPF symbolic debugger, a post-mortem performance analyser, and a static estimator. Collectively these components are referred to as the cross development tools, and they execute in a distributed environment, namely Sun SPARC based and RS/6000 workstations.

The project has its origins in the Genesis and PUMA Esprit-2 projects, and the software tools and applications developed in them will be used as a base platform for the software development within this project. The toolset will be portable. The portability layer for the cross development tools will be the PCTE (Portable Common Tool Environment) database system [17, 3] and the portability layer for the run-time system will be the PARMACS message-passing macros [10, 16].

It recognised that internationally accepted standards which define stable interfaces between sub-systems are the key to assuring both portability and inter-operability:

- Language standards form the interface between applications and their human developers. The primary application programming language standard of this project will be Fortran. Fortran 77 will be adopted as a base-line, while aiming to support Fortan 90 and to participate in the development of evolving standards such as Fortran-D, Vienna Fortran and High Performance Fortran.

- The run-time system forms the interface between an application and the platform on which it runs. This project will use PARMACS as its base-level run-time and message passing interface. Developments in message passing standards will be tracked and included into the definition and implementation of the run-time system.

- Data interchange standards define the interfaces between programming tools. The project will use PCTE to define and implement inter-tool interfaces. New tools will themselves be written in ANSI-C, using the POSIX standard as the operating system interface. The low-level graphical interface will be the X-Window System, however all graphical tools will be written with the Motif library to assure a common look and feel.

The major partners in the project are Meiko, Emeraude, GMD, the University of Vienna and PALLAS, with Meiko coordinating. Associate partners include applications developers, manufacturers and the University of Southampton. Southampton will be involved in the integration of the cross development tools into PCTE, the development of a Parallel Performance Estimator, the evaluation of the tools, and in liaison, dissemination and standards activities.

6.2. PCTE and it's Role in PPPE

PCTE is a *tool support interface* that, with the addition of appropriate tools, provides an *environment* for the development and maintenance of software applications. The PCTE interfaces provide:

- A database called the *object base*, that is transparently distributed across a network of workstations, and structured so it reflects the software development activity it supports.

- the PCTE *Object Management System* (OMS) that accesses and manages the object base in response to requests from users and tools.

- A distributed architecture based on a network of workstations.
- An advanced user interface, that allows the user to view data from various sources at the same time, in different windows. It utilises the object-oriented technique of handling screen elements. A standard set of operations for user interaction supports a uniform method of interaction

Within the PPPE project PCTE will be the framework which will support the Cross Development Environment on Unix workstations. Emeraude and Southampton will be jointly responsble for defining the PPPE Schema Defintion Sets (SDSs) that will capture the inter-relationships between the tools.

6.3. The Integration of ParaGraph

In order to evaluate the complexity of the PCTE integration task, the existing ParaGraph tool will be integrated, in the following stages:

1. PCTE Encapsulation: No need for recompilation. A PCTE process transfers the information (objects, links and attributes) to Unix, the tool is executed on Unix, and the output is transfered from Unix to PCTE
2. PCTE Rehosting: The tool is recompiled in order to run on PCTE. Unix-like I/O libraries (open, scanf, printf...) allow the tool to operate on object contents instead of Unix files. Minimal source code modification involves changing Unix file pathnames to PCTE object pathnames.
3. PCTE Exploitation: More modification to the source code to preform real PCTE calls to browse and update the objects, links and attributes defined by the PPPE SDSs.

7. Conclusion and Future Work

We have seen that tools such as Paragraph help to understand the behaviour of code and to locate possible performance bottlenecks, with only a moderate performance overhead. Furthermore, the use of portable libraries and trace filters allows such tools to be largely independent of hardware platforms, although it would be desirable to standardize the models that trace files are based on as far as possible. Complementary to the portability aspect of tools in a parallel programming environment is that of tool integration, which is now an area of active investigation. Looking ahead, it is clear that for massively parallel systems many of the graphical representations we currently use will not scale well, and the user's ability to understand such volumes of information will soon degrade. Monitoring tools for such large scale parallel systems will require new ways of displaying the information at a higher level of abstraction.

References

1 C. Addison, V. Getov, A. Hey, R. Hockney, I. Wolton, *The GENESIS Distributed-Memory Benchmarks*, Proceedings of the conference *Parallel Processors - Benchmarking and Assessment*, Teddington, March 1992.

2 M. Debbage, M. Hill, D.A. Nicole, *Virtual Channel Router Version 2.0 User Guide*, University of Southampton, October 1991.

3 *The Emeraude Environment Guide*, GIE Emeraude, 68 Route de Versailles, 78430 Louveciennes, France.

4 *Express User's Guide, Version 3.0*, Parasoft Corporation, 2500, E.Foothill Blvd., Pasadena, CA 91107, USA.

5 G.A. Geist, M.T. Heath, B.W. Peyton, P.H. Worley, *PICL: A portable instrumented communication library, C reference manual*, tech. rep., Oak Ridge National Laboratory. ORNL/TM-11130, 1990.

6 G.A. Geist, M.T. Heath, B.W. Peyton, P.H. Worley, *A Users' guide to PICL, A portable instrumented communication library*, tech. rep., Oak Ridge National Laboratory. ORNL/TM-11616, 1991.

7 I. Glendinning, M.J. Gorrod, S.A. Hellberg, P.A. Shallow, *Tools for Parallel High Performance Systems – Comparative Evaluation*, Southampton Novel Architecture Research Centre tech. rep. SNARC 92-01, January 1992.

8 M.T. Heath, J.A. Etheridge, *Visualizing Performance of Parallel Programs*, tech. rep., Oak Ridge National Laboratory. ORNL report TM-11813, 1991.

9 M.T. Heath, *Performance Visualisation with ParaGraph*, in *Proceedings of the Workshop on Monitoring and Visualization of Parallel Processing Systems*, edited by G. Kotsis and G. Haring, Elsevier.

10 R. Hempel, *The ANL/GMD Macros (PARMACS) in FORTRAN for Portable Parallel Programming using the Message Passing Programming Model, User's Guide and Reference Manual, Version 5.1*, Gesellschaft für Mathematik und Datenverarbeitung mbH, Postfach 1240, 5205 St. Augustin 1, Germany.

11 R.W. Hockney, *A framework for benchmark performance analysis*, Supercomputer **48**, IX-2, 9-22 (1992).

12 R.W. Hockney and I.J. Curington, $f_{\frac{1}{2}}$*: a parameter to characterise memory and communication bottlenecks*, Parallel Computing **10**, 277-286 (1989).

13 R.W. Hockney and C.R. Jesshope, *Parallel Computers 2: Architecture, Programming and Algorithms*, (Adam Hilger/IOP Publishing, Bristol & New York, 1988). Distributed in the USA by the American Institute of Physics, c/o AIDC, 64 Depot Road, Colchester, VT 05445.

14 R.W. Hockney and E.A. Carmona, *Comparison of communications on the Intel iPSC/860 and Touchstone Delta*, Parallel Computing, to appear (1992).

15 *Editorial*, Parallelogram - The International Journal of High Performance Computing, ISSN 0953 7252, Issue No. 44, March/April 1992.

16 *PARMACS Performance Analysis Tools Documentation*, PALLAS GmbH, Hermülheimerstraße 10, D-5040 Brühl, Germany.

17 *PCTE, A Basis for a Portable Common Tool Environment, Functional Specification Version 1.5*, Commission of the European Communities, DG XIII/A4 ESPRIT, 25 rue Archimède, 1040-Brussels, Belgium.