FORTRAN 77 PROGRAMMING OF PARALLEL COMPUTERS

by

R.J. ALLAN, SERC Daresbury Laboratory

NOVEMBER, 1989

Science and Engineering Research Council
DARESBURY LABORATORY
Daresbury, Warrington WA4 4AD
FORTRAN77 Programming of Parallel Computers

R.J. Allan 29/6/89
Advanced Research Computing Group
Daresbury Laboratory
S.E.R.C.
Daresbury
Warrington, WA4 4AD
U.K.

Abstract

A review is given of some of the implementations of the FORTRAN programming language on coarse and medium-grain MIMD processors, which have come to be known as multicomputers. Some general notes on operating systems and algorithms strategy for scientific computing are included. Information is provided on using the parallel computing facilities at Daresbury.

Most FORTRAN implementations consist of language extensions for inter-processor or inter-task communications. These are dealt with explicitly and example programs are shown. A new occam-2 harness for the Meiko Computing Surface is described.

Outline of Review

I Introduction

II Operating Environments

UNIX
Trilium, TransTech Niche etc.
CE/AR (COSMOS, Intel iPSC etc.)
Helios (Perihelion, Atari etc.)
Occam TDS (Inmos, Meiko, TransTech, Parsys etc.)

Threads and Lightweight Processes (3L etc.)

Seminodes and Ports

III Parallel Programming

Algorithms, FORTRAN 8X, numerical libraries etc.
Debugging
Balancing, optimisation and performance monitoring

IV Programming Environment (FORTRAN)

Helios
Intel iPSC/2
Meiko Computing Surface (and the Fortnet harness)
MCUBE
TransTech NTP1000
Parsys SN1000
3L, Parallel FORTRAN

References

Appendices

Timing intrinsic FORTRAN routines
Whetstone Benchmark
Communications Benchmark

I.1.1

I Introduction

Some of the material of this review was presented in the TCS Division seminar series on 3rd October 1986, and is intended in part as an introductory guide to programming at Daresbury Laboratory (DL) in the FORTRAN language [1]. In further part the material has been collected for my own personal use during evaluation of MIMD systems and as a repository to share with colleagues. Many statements in the text are personal views and are not considered to represent ARGC policy, or policy of any of the companies mentioned. Many of the names included in the text are protected by registration or trade marks. All information in the text is public knowledge as far as I am aware, any inaccuracies or misconceptions are entirely due to my own fault and are in now way deliberately misleading (standard disclaimer).

The reason for using P77 as opposed to or the new parallel syntaxes such as Concurrent Pascal [11,12,22], Ada [2], Concurrent Pascal [3], 55GAL [4] etc.) is the massive investment in code developed for scientific applications over the last 20 years. Mathematical subroutine and program libraries (such as those maintained by NAG, CPC, Argonne and others) make rapid implementation and exploitation of new methods in physics, chemistry and engineering possible. Without that infrastructure the basis of modern scientific computing, and much of the impetus for developing codes on, and using supercomputers would be lost. Another important area in which we have seen major advances in recent years is in debugging and analysing complex code. The presence therefore, of very good P77 programming tools provides an incentive to the academic scientific community in exploiting parallelism.

What is parallel computing? For the purpose of this review the definition is restricted to the use of N processors and M tasks (in other words N interconnected sequential processors) which execute simultaneously to produce a useful result. This is multiple instruction multiple data (MIMD) processing as opposed to single instruction multiple data (SIMD e.g. as on the distributed array processor, DAP [1]). Tasks may be partitioned, independent work on different data, independent work on the same data, or concurrent work with inter-task communications. More discussion of the approaches is deferred to section VII.1. No consideration is given to the debate over hybrid or distributed systems, the present discussion being restricted to distributed memories. Some comparison and analogies are hinted at.

What characteristics of a programming language are required for a MIMD computer? The processes require the ability to synchronise under program control; this may be for data transmission. The term “blocking” will be used for this synchronisation - it ensures that the processor is ready to receive the correct message and block incorrect ones. A more restricted terminology used in the literature means that the task is blocked from continuing its execution until the transfer completes. This will effectively also be the case in our implementation as seen in section IV.3. In the protocol of communicating sequential processes (CSP) [6] all communication is blocked by definition. The reason for blocking is simply to allow different amounts of work to be done by each processor, essentially the argument that this causes load imbalance and therefore waste of resources is valid, it is no worse in practice than not fully utilising a vector processor in sequential code. Some redundancy of operation may indeed be an aid to conceiving an efficient parallel implementation.
and losses are then outweighed by gains. Blocking or CSP protocol is quite different in strategy from buffered protocol (of which the prime example is UNIX [7]). It is more primitive and we shall see that the latter might be implemented in terms of the former.

In modern data transfer in memory may be unblocked and synchronous or asynchronous protocol. All processors want to be able to access files and devices (often attached to the front-end computer unless a distributed file system is available) to reduce processing. It is reduce calls to message passing routines. One reason for wanting to reduce message passing is that it is slow. As modern processor chips increase in performance, communication becomes a real bottleneck, just as in a sequential machine where memory access rate dictates the ultimate performance. This is the reason for favouring MIMD architecture in the first place, but careful programming is needed to make full use of all available resources.

What other features might be useful? Firstly the program might want information about itself, such as the processor (node) address upon which it runs, the configuration of its hardware environment and its process id (pid) in UNIX. The ability to load or spawn new processes or threads and kill old ones is also useful. Messages must be checked and corrected if in error (wrong message or wrong address) and perhaps queued in buffers. We would ideally like to be able to automatically configure programs for a given machine topology or vice versa. That would balance the workload in the optimum way for a given arrangement of processors, or suggest how a machine should be built optimally for a specific application. More discussion of load balancing and performance monitoring is given in section III.3. It is a current research topic. Automatic routing of messages is essential to high-level applications.

Areas of other current and future research include the following. Implementation of both data-topology-dependent and topology-independent optimised mathematical algorithms, and machine topology variable under program control (access to hardware switches). There is a clear need for concurrent debuggers to check programs and data flow, which would ideally be able to place processes so that trial configurations can be made. Graphical interfaces will be important in this area. Design of mathematical software packages in nature to those protocols to accommodate all sequential machines is in progress. Implementation of shared-memory emulators on MIMD machines to facilitate porting of programs will be useful. There is however as yet no generally accepted convention to write such general data, and many strategies can be conceived and must be tested in practice. A few thoughts on parallel subroutine library implementation are given in section III.1.

In the next sections a brief description of some of the operating systems available on parallel machines is given. This will help to understand what the FORTRAN77 language implementations can be capable of, and how they work.

II Parallel Operating Environments

This section gives a brief introduction to how some of the most common parallel environments operate, but is by no means exhaustive. A moderate understanding of the main features should give insight into what a parallel program is capable of doing within its environment, how access is gained to the hardware, and to what extent knowledge of the hardware is required. Leaving aside the fine-tuning issues, parallelism is put explicitly into the code by writing separate host and node programs. The operating systems essentially provide only a way to access a set of kernels which transfer data as messages between processes, either concurrent in one node or in networked nodes.

Access to system kernels is via a set of higher-level library routine or server calls, which package the messages using a suitable protocol depending on their length and destination, automatically route them through the system ensuring minimum disturbance to other processes, and prevent errors from occurring. Errors may be of the type that a message arrives that isn't wanted yet (or indeed ever). There are two approaches to this problem; the first is to queue the message, so that the user's program may later check for its presence - a strategy that will work well for either synchronous or asynchronous transfer, although for the latter it is essential. If there is an error in the program the message queue may eventually overflow and the job fail requiring a post-mortem trace of the queue build up. The second approach is to signify an error to the sending process and not to route the message as described in section IV.3. In section III.3 it is assumed that any system can be designed to be fault tolerant to a large extent. Most systems described in this document, which are based on UNIX-like kernels, use the first approach. Our own work (described in section IV.3) uses the second approach and contains an error-handling capability to avoid the need to have the system flush “lost” messages and to resolve blocking. Program errors are assumed to be the cause of one class of lost messages and are flagged immediately such as in a run time trace.

A description of parallel communications on the CRAY, IBM, ETA and FPS-720 and in the SPLIB (standard parallel library) is given by David Smilling and Geerd-R. Hoffmann [II.1.1]. Their analysis is somewhat different from the one presented here.

II.1 Parallel processing in UNIX

One of the most important parts of 4.2BSD UNIX and the proposed POSIX standard [II.1.7] is the interprocess communication (ipc) facility. In UNIX version 7 (the previous standard discussed in many textbooks) the only way that processes could communicate was by pipes. Not only is this difficult for pipes when distributed in a distributed environment, but the scope they offer is limited. UNIX 4.2BSD contains the implementation of sockets through which processes may rendezvous to transfer data via a file-system name, essentially on an machine or via a network.

Three types of sockets exist, and they may be used to connect either processes executing in the same host or processes in different hosts. A client-server model is often used and many UNIX facilities are available as servers to which a process connects by a socket, albeit embedded in another system library call. The three types of socket are: first, a socket stream which is connection-oriented, reliable, and half duplex; second, a datagram with no record boundaries, and in usually used inside a single host. Pipes are implemented using a pair of stream sockets. A second
type, the datagram socket, is used over a network and closely models a packet-switched network protocol such as Ethernet. It supports bidirectional flow of messages that are however not guaranteed to be sequenced, reliable or unduplicated, and may therefore be duplicated or out of order upon arrival. Record boundaries are however preserved. Finally a raw socket is provided to give access to underlying protocol, it should not normally be used in applications.

To communicate between processes a socket must first be created, then bound to a name (either a file name, or network name) and may then be connected to and used for data transfer. Again in the UNIX model clients connect while servers listen and accept connections before providing a facility.

Synopsis of UNIX ipcall calls.

Establishing and using synchronous blocked sockets.

socket(domain, type, protocol) -- create a socket, in this call values of domain of 'AF_INET' and 'AF_UNIX' are for transfer over a network or inside a machine, values of type of 'SOCK_STREAM', 'SOCK_DGRAM' or 'SOCK_RAW' are for a stream socket a datagram socket or a raw socket respectively. If protocol=0 a suitable protocol is chosen by UNIX, alternatively the call

socket,proto=socketname(tcp) -- may be used to get a protocol number

bind(s, name, len) -- assign a name to a socket
connect(s, server, serverlen) -- rendezvous with process server
using a stream socket
listen(s, n) -- wait for up to n connection requests from clients

accept(s, from, fromlen) -- accept a connection when requested by a client. A blocking call which returns the new socket and description upon receipt of connection.

write(s, buf, len) -- write to socket s with established connection to a stream socket
read(s, buf, len) -- read from socket s
send(s, buf, len, flags) -- send to socket s, like write
recv(s, buf, len, flags) -- receive from socket s, like read
shutdown(s, how) -- shut down connection prior to closing
close(s) -- discard a socket
sendto(s, buf, len, flags, to, tolen) -- send data to a datagram socket
recvfrom(s, buf, len, flags, from, fromlen) -- receive data from a datagram socket
select(nfds, readfds, writefds, exceptfds, timeout) --
multiplex I/O requests between multiple sockets or files

Other calls are available to specify service addresses on a network:

gethostbyname(... -- return a hostent structure from
/etc/hosts

gethostbyaddr(... -- map host address into a hostent structure

gethostent(...) --
getbyname(... -- as above for networks

getbynumber(...) --
gethostent(...) --
getprotobyname(...) -- as above for protocols

getbynumber(...) --

Miscellaneous calls:

bcopy(s1, s2, n) -- copy n bytes from s1 to s2
bzero(base, n) -- fill n bytes from base with zeros

htons(val) -- convert long 32b from host to net byte order
ntohs(val) -- convert long 32b from net to host byte order

If a machine has UNIX implemented it is clearly possible to set up and talk to communications servers and thus send data between processes on the same or different nodes. This is done by a kernel and we will see this in the next few paragraphs. For this reason it has been outlined in some detail here. Similar calls for instance are used by Transtech in their RHEBE PRO system which also uses UNIX with the Trillium ipcall layers for underlying communications servers (see next section) and by Mekko in their SUN CStools. A version of UNIX called AXIS is used on the NCUBE machine (section IV.4), with a kernel on the nodes called VERTEK. The Intel hypercube has implemented the IX/2 kernel on the nodes and a Unix host operating system. On the Parsys SN1000 there is a system called IDRIS which adheres to the POSIX standard. Parsys offer only these socket calls as a means of communication in F77.
II.2 Trillium (lately called trillium)

Both trillium and trillium are wild flowers native to the New York region, and when it was found that the former name was already licensed for use by another company's product the Cornell Theory Center rapidly changed it. However trillium is now international jargon for their multicomputer operating environment.

Trillium ([1]) was originally designed for the transputer-based FPS T-series host by a Gould UTX-32 and other connected UNIX-based machines on a network. This explains its use on the Transeact NT1000 platform developed by MICHE Technology with its transputer array and SUN 3.5 front end, and on the Topologix boards.

The operating system has two fundamental layers, the local ipc layer, which is a message rendez-vous kernel to pass messages between blocked processes in a node and which is tightly coupled to the underlying UNIX, and the remote ipc layer which loosely follows the OSI network standard to pass messages between nodes. The former is always present, whereas the latter, which is called from the application code, is loaded as required from a library. Within the latter layer are four more layers called physical, data-link, network and transport. These respectively transfer to physical ports, adjacent nodes via a protecting process, any node on the system with routing information transparent to other processes, and any other node with a protection protocol which checks system status to ensure reliable delivery. Because of the way transputers are linked there is a higher overhead in using the network and transport calls. More information and the system subroutine calls are given in section IV.b.

II.3 CE/RK

The Cosmic Environment and Reactive Kernel were developed by the research group at CalTech for use on the Cosmic Cube medium-grain multicomputer which they designed. The multiple-process message-passing environment is configured on both network hosts and attached arrays with internal data transfer between nodes.

The CE host runtime system handles message passing between programs on a collection of networked UNIX hosts and also allows them to allocate and interface to one or more arrays. It provides a set of daemon processes (servers), utility programs and runtime libraries to do this. The RK node operating system supports multiprogramming, message-driven process scheduling, storage management and system calls on each node.

A message can be any size from zero to the maximum available memory. Messages of different lengths and destination have different protocols, but are handled in an identical way by the programer. Messages are queued if necessary in either the sender, the receiver or in transit. An arbitrary delay is therefore introduced, but ordering is preserved between pairs of processes.

A process group within the CE system is the complete set of node and host processes that make up a computational job. CE establishes and maps these processes onto the multicomputer nodes, and handles communication between them. Each process has an id consisting of an ordered pair (node: pid). Node parameters range from 0 to N-1 and N is the host. The mynode(), myhost() and mypid() functions give information about them (see section IV.2). Messages are sent between processes according to the structure within the process group. Processes have reference to other processes so that a complete map can be maintained and new paths discovered, whilst flow control is applied to preserve ordering.

Explicit spawning and killing of processes is permitted, for instance the call

spawn(filename, node, pid, mode)

The writers of RK have been careful to consider the requirement for fast process creation in multicomputers. They therefore share code segments between processes created from the same program in the string filename and cache the initial data segment as subsequent creations using filename need not access the file system again.
II.4 Helios

Helios is the name of an operating environment developed by Perihelion Software Limited [1]. It is designed to run on transputers with at least 256 kbytes of memory each, and can cope with an arbitrary interconnection topology. The hardware for which it was developed is the Atari A3000 system. Perihelion also produce their own transputer cards, and Helios can be run on other systems such as the TransTech NT1000, Parytek, Helico etc.

Helios is a completely distributed operating system, with each task containing a nucleus which manages processes under its control and their relationship to other processes. It can therefore be used in an embedded system (cf. section IV.1). The nucleus consists of a kernel, a system library, a loader and a process manager.

The current user interface gives a command line similar to the UNIX C-shell coupled to an implementation of X-windows VI [2]. Commands piped together on the command line may be distributed over several processors by the system to run concurrently.

The basic unit manipulated by Helios is a task. It is a program in a known state of execution containing a number of concurrent processes and environmental data such as files, memory, and other resources. The only means of communicating between tasks is by messages. Message passing primitives are implemented in the Helios kernel. A message is sent upon request to a server.

Distributed programming under Helios entails putting a single task on each processor (although there may be others from other users too). A blueprint file contains a description of how the tasks are related and is read by a server which matches the requested configuration to available CPU and memory resources.

The system is designed as a client-server model, where each processor has a number of servers (perhaps different ones) and a name server telling where they and others can be found. Tasks also running concurrently on these processors request services such as file handlers, window managers, date servers, spoolers and so on, using a general server protocol. Servers are designed to have no state so that they are unaffected by hardware faults and can be brought up on any available node to balance loading. Memory and processor management is carried out by assigning a list of capabilities to each client. All objects (such as files, directories, tasks, processors) have an associated access matrix telling what operations may be performed on them. Only a client having the correct capability may carry out an operation. An encryption scheme is used for additional security.

II.5.1 Occam and the Transputer Development System

Occam

Occam-2 is the language of the transputer and is an implementation of the CSP protocol [I.1.6] and a development of the experimental programming language EPL of D. May [1]. It enables a system to be described as a collection of concurrent processes, which communicate with each other and with peripheral devices through channels. Each transputer assigns local memory space for a number of processes executing with local variables. The only connection between them is by calling procedures which pass arguments, global variables and channel communication. It might be argued that if processes share global variables they are actually the same process. Only three primitive operations are combined to form an Occam program; these are

\[
\text{-- assign the result of evaluating expression } e \text{ to variable } v
\]

\[
\text{c} ! e \text{ -- output the result of evaluating expression } e \text{ to channel } c
\]

\[
\text{c} ? v \text{ -- input variable } v \text{ from channel } c
\]

Processes may be combined into constructs in either sequential SEQ, parallel PAR or alternative ALT fashion. Whilst the first two constructs should be self-explanatory, the last one is novel. It indicates that the first of the ready processes listed together under the ALT should be executed; useful if one requires to wait for input from more than one source. It will be seen that this is a fundamental feature of the CSP model and is also provided in a number of higher-level implementations of FORTRAN. Other more conventional language constructs such as IF, WHILE and CASE are also provided and replicators may be applied to all constructs. Occam is reentrant.

Occam processes are connected by channels which are strictly defined. Communication is synchronous and does not occur until both the sending and receiving processes are ready. The Transputer chip executes concurrent processes, and the programming model for on-chip concurrency and distributed processes are identical. Processes are executed, if they are ready, with round-robin scheduling. Unready processes are swapped out (they might for instance be awaiting input from another descheduled process). Soft channels connecting processes in a single transputer are stored in memory so that data is not lost as processes environments are swapped. There is some similarity to the UNIX sockets. The channels connecting processes on different transputers do however require to be mapped onto hardware links. This is done at configuration time along with placement of processes onto transputers. Since the connecting processes are on different chips, data is passed when they are both scheduled and ready to transfer.

The form of an Occam-2 program is specified by a syntax rather similar in style to C (at least it looks to a FORTRAN programmer). Structure is however maintained strictly by indentation in units of two spaces, no semi-colons being needed to terminate lines. The only lines not indented relative to one another are those belonging to the list of items, such as SEQ, PAR, ALT, IF, CASE or WHILE construct. Indentation represents hierarchy. The language facilities are very limited, there is for instance (as in C) no default i/o specified and a number of libraries are now available which have implemented the procedures.
Utilities will be contained in the manual for the transputer array in use, which should be consulted. See also section IV.3 below. A set of Emacs macros is available from the Occam Users' Group [R.1.2] to simulate the TDS editor on any UNIX system: it is not restricted to use on occam programs.

Harnesses

Transputer arrays like the Meiko and Inmos systems rely traditionally on occam harnesses to carry out communications between processors (but see section IV.3). That means linking the FORTRAN77 object code to the occam and supplying subroutine libraries to handle message passing. Configuration is done in occam. This is really no different from other operating systems that we have described, which are mostly written in C, but the current user interface is poor so that the programmer must be to some extent familiar with the occam environment. This is not liked, and steps are being taken to change it so that the occam environment with embedded foreign languages (F77 or C) may continue to compete with hypercubes like the Intel or NOVEX, or indeed with transputer arrays like the Trans tech NTP1000 and Meiko Octocell system have more sophisticated operating systems.

Some of our own work on occam harnesses for the Meiko is described in section IV.3.

II.5.2

II.5.3

Utilities are accessed by moving the cursor onto the appropriate line and pressing a special key sequence, after which a new menu appears in the editor window.

More information about the folding editor environment and
II.6 Threads and lightweight processes

The ability to produce or spawn new concurrent processes has already been mentioned in the discussion of CEB/RK and is also available in the Mcube operating system. It is also the single most important feature of UNIX with its fork and join strategy for producing child processes from parents. Some approaches to parallelization embody particularly efficient methods to create or spawn processes which consequently, having low overheads, might be of short duration. These are referred to as threads or as lightweight processes. They are processes in modula-2 or coroutines in some other languages. They share code, heap, static and external data memory with all other threads created by the same task. We will later examine in detail the 3L FORTRAN compiler which allows explicit control of processing threads.

If such a strategy is implemented then algorithms may be developed which are characterised by replication. The divide and conquer and merge sort techniques are examples, illustrated in section III.1.1. Some of these methods are particularly useful on shared-memory multiprocessors, where replication of processes does not necessarily demand replication of data and inherent message-passing complexity. Work is being done in Argonne National Lab. to exploit these methods in numerical analysis algorithms. One might envisage using threads as a way to replicate processes within a single processor of a MIMD machine, and use its local memory in a shared fashion. This would in fact yield emulation of one type of hybrid architecture. This is done for instance in the 3L parallel system where FORTRAN common space on a single processor is common to all threads in the same process task. Semaphore functions are used to prevent interference between threads sharing the same data.

A disadvantage of threads in a MIMD architecture is that they may not be moved to another processor from their invoking task, whereas a normal process may be reconfigured.

II.7 Semaphores and ports

One protocol to communicate data is using blocked communications of the CSP type, or some variation of that with buffers. A different protocol is to use ports as a place to put data which can then be read by another process at a later date. Ports are used by 3L and by Melko, see section IV. The actual port (a buffer) may be anywhere in the system and is akin to a shared file with multiple access. In the general sense this involves difficulties such as those met in shared-memory machines or in multi-access relational databases, and mostly port usage is restricted to one process writing and others reading, or vice versa with the port being a FIFO buffer so no messages are lost. This is the UNIX method. The same problem arises with shared resources such as disks. Since concurrent file systems are now becoming available (e.g. [IV.2.6]) the question is non-trivial.

One way to handle shared resources is by the use of semaphores [1.2]. These are used to indicate to the operating system when a process has finished with a resource and it is free to be used by another process. They must be included by the programmer in his code, but are implemented in the operating system in a special way, so that they are not themselves subject to the same difficulties.
III.1 Parallel Algorithms for FORTRAN

This section is divided into three parts: III.1.1 is about algorithms, methodology and ideas. III.1.2 is about FORTRAN and possible language extensions and inadequacies. III.1.3 is about mathematical subroutine libraries which are available for MIMD computers.

III.1.1 Parallel Algorithms

Parallel algorithms can be classified in three types: simple ones, which are coarse-grained and use the inherent symmetry of the physical problem with no communication overheads, almost a definition, and could be called symmetry parallelism for obvious reasons, coarse and fine-grained ones in which the problem is mapped onto the machine in a geometric way so that only nearest-neighbour communications are needed in the main (geometric parallelism), and difficult ones which are fine-grained, necessarily involve a lot of communications, and try to solve a single mathematical operation in a distributed way, e.g. multiplication of large matrices.

Since this text presents only an overview, and most newcomers to parallel processing find it easier to think in terms of the symmetry of their problems, I concentrate on the first approach in paragraphs 1-3 and the rest of the document.

I again split this section into five smaller pieces: 1) Farming algorithms which might be either the data-farming or task-farming kind, 2) Pipeline and conveyor-belt algorithms, 3) other geometric algorithms, 4) binary-tree and recursive-spawning algorithms, e.g. the merge sort method, and finally 5) more general techniques.

1) Farming-type algorithms.

Farming methods are of two types: a) in which the same program is put on all nodes and is expected to run many times with a stack of input data, different data being sent to each occurrence of the waiting program and b) with a set of tasks that require to be carried out either on the same or different data so that many different programs need to be run. I shall refer to the former as data farming, and the latter as task farming.

Examples of data farming are given in sections IV.3 and IV.4 and are described in detail there. The idea is quite simple and trivial to implement. A set of n processors all hold identical copies of a program, or different programs. A host process, often called the master, sends these data to work on and waits for them to reply. When they have replied more data is sent until it is all used up. In that way load balancing is quite automatic, especially if all n processes are used.

Task farming is equally simple, but requires on the master to be able to dynamically load tasks onto the nodes via the operating system (The usual way to do this is to have a file containing a list of independent jobs which constitute the program, but which might be run concurrently with only limited synchronisation through data passed via the master. The master loads the jobs in turn onto available nodes and carries out data forwarding. When a node has finished its appointed task it must signal its readiness to receive a new one, which is the next in the list. So on until the list is used up and the program finishes. This is the prototype of a
parallel batch queue.

An intermediate style of programming, which may be considered
medium-grained but occurs very frequently as a result of unrolling
(splitting) inner loops in a sequential program, is described as
follows.

i) split the inner loop and send out independent tasks for each
value of an index reflecting progress through the loops as long as
no processors are waiting to return results, otherwise go to (iii)
ii) receive all results which are pending and then go to (ii)
(iii) receive all remaining results and shut down nodes.

An example of this coding is shown below. During loop
splitting, a problem of data presentation on the screen is often
encountered. Output during cycle (i), which consists of header
information, must be stored in a buffer to be retrieved and
displayed only during cycles (ii) and (iii) along with the other
data to which it belongs.

host program

SUBROUTINE DIFFKS
...
C workspace for header information
character*132 head(50)
counter for number of headers per task
dimension ihed(nloop)
common/talk/inode,nnode,ihost
khead = 1
ihed(id)=1
READ(5,...
call crcvc(-1,ms,4)
C LOOP OVER COLLISION PROCESSES
...
C start of sending loop, phase (i)
proc=0
nse=1
nrecv=1
70 READ(5,...
go to 30
C send data to idle processor
call csend(i,nse,4,iproc,0)
...
C internal write of header information to be later output with
C results
write(head(khead),....
khead=khead+1
910 VMD=*18386.1
C if (nmod(10.0,99) go to 70
nse=1
iproc=proc+1
C starting point in vector head for first header of next task
head(nse)=khead
if(iproc.ge.nnode)iproc=0
C receive cross sections back in correct order, phase (ii)
85 iret=iproc(nrecv)
if(iret.ne.1)then
...
go to 70
end if
...
call recv(nrecv,bmpi,ms,8)
call recv(nrecv,bmpi,ms,8)
nrecv=nrecv+1
c print header information and then results from this proc
k=nrecv/2
do 50 =ihead(k),ihead(k+1)-1
write(6,'(a132)')head(1)
50 continue
... goto 86
30 continue
C receive remaining results phase (iii)
if(nrecv.eq.nsent*2-1)then
call recv(nrecv,bmpi,ms,8)
nrecv=nrecv+1
call recv(nrecv,bmpi,ms,8)
nrecv=nrecv+1
c print header information and then results
k=nrecv/2
do 60 =ihead(k),ihead(k+1)-1
write(6,'(a132)')head(1)
60 continue
... nsent=nsent-1
goto 30
end if
C shut down nodes
do 31 i=0,nnode-1
call csend(i,nse,4,1,0)
... 31 continue
END

node program

SUBROUTINE DIFFKS
...
integer buffer
common/workspace/buffer(2000)
common/talk/inode,nnode,ihost
READ(5,...
read(2,...
if(inode.eq.0)call csend(1,ns,4,ihost,0)
...
70 continue
call recv(1,nsent,4)
nretns=nsent*2-1
call recv(1,kf,4)
call recv(1,kf,4)
if(kf.eq.0.000.and.kf.eq.0) return
call recv(1,mdc,4)
call recv(1,weight,8)
71 do 25 i=nsent+1,nsent+nsent+1
nbyts=8*nbyts
read(2,...
c compute intensive routine
CALL EIKON(2A,2B,ZET1,ZET2,RPHAs,V,VMU,MDG,ANG,MINE,JPB,BMP)
...
IF(WTGT.LT.0.998)GOTO 95
call csend(nretn,bmpt2,mang*8,ihost,0)
nretn=nretn+1
79 call csend(nretn,bmpt,mang*8,ihost,0)
95 rewind(7)
... goto 70
END

The above example is taken from a real application of an atomic collision package on the Intel [5].

Another example of unrolling a loop is provided by the following multi-dimensional integral code. The simplicity of the code derives from the fact that all nodes have read or received data from the host so they know all values of the loop indices and are able to execute synchronously with the host.

host program
...
knode=nnodes-1
kout=3
lword=4
do 999 j=1,ng2
... do 777 j=1,ng1
... do 666 i=1,nn1
... do 8 j=1,ng
kout=kout+1
iproc=mod(kout,knode)+2
call check(iproc)
call receive(iproc,nn*1word,dsdo)
...
5 866 ...
777 ...
999 ...
end

node program
...
node=mynode
...
c start of loops here
...
if((node.eq.mod(kout,knode)+2)) then
do 1 i=1,nn
...
1 continue
... call wait(1)
call send(1,nn*1word,dsdo)
end if
5 continue
666 continue

It can be seen that this method of distributing work is quite automatic and deadlock will not occur unless the nodes cease to return a result. This can happen e.g. on the Meiko, for reasons of numerical error (overflow) and is hard to debug. The above example is taken from a real application of an electron-atom scattering program on the Meiko using the Fortnet harness [6].

2) Pipeline and conveyor-belt algorithms.

Pipeline algorithms work in the same way as a vector pipeline processor, but each step in the pipe is considerably more complicated. Operation. A number of tasks are joined one after another, input being from the one previous and output to the next in line. A master process sends a continuous stream of data down the line, and might even dynamically load tasks. This is the first kind of algorithms we have set (and the simplest) which explicitly involves inter-node communications, albeit to nearest neighbors. Many extensions of this idea are possible, such as nearest-neighbor communications over a grid topology.

This kind of technique is ideal for applications in which the program may be split into a number of almost independent tasks, but in which the overall program must be executed many times. Note that data dependency might also be allowed through common disk file storage with semaphore messages passed to signal access clearance.

3) Geometric algorithms

The geometric loop algorithm is useful if a program is modelling interactions between objects (either physical or symbolic), and all pairs have to interact. Data describing the objects can be passed around the processors so that eventually each object's description will be passed to every other. True systolic algorithms would involve synchronising this data transfer with, for instance, every processor sending data one way to its neighbor around a ring or a mesh in a series of regular pulses spaced by computation. Other ways to do regular communication in which all processors perform on an equal footing could be envisaged. See reference [1].

4) Binary-tree and recursive-spawning algorithms.

It seems that there is a class of problems which are amenable to the divide-and-conquer strategy. I refer to their solution generically as binary-tree algorithms, although that might not be wholly accurate.

I illustrate the method with the merge sort technique for sorting a large number of data elements into order in an uncountable set. Another sorting program, illustrated in section IV.3 assumes a uniform distribution within a bounded uncountable set to achieve good balance - we shall not do that here. The method consists of subdividing the set arbitrarily until the maximum 2^n processors is reached, sorting the subsets (a combinatorial problem of N/2^n complexity rather than N) and subsequently merging pairwise in n steps to obtain the completely ordered list.

This might be achieved using fork and join procedures if such were available. I give an example, but untested, where these
are not used, but the node programs are aware that they belong to 2^n
clones. The pattern of events is then to send a processor's ordered
data to its left nearest neighbour, a distance 2^n away, which
receives it and merges it with its own data. Then send the
data to the neighbour 2^n away and merge, and iterate from 1=1,n.
The last step takes all the data onto the leftmost processor and the
final merge sorts it completely.
A redundancy of operation is involved, for example in the code
the last step involves 2^n(1-1) processors doing worthless merges.
There is also in the example code a redundancy of storage which is
much more serious, but I have left it in so as not to overcomplicate
the illustration. Of course both these points are extremely serious
on shared-resource machines where more than one person's task, and
if I were to make my processors wait instead of computing, they
could be used by someone else.

```
program merge : Fortran-like coding syntax

c node program of merge/sort routine, recursive binary tree
dimension a enormouse, b enormouse
integer seg, seg0, seg1
common /task/ inode, nsend(10), nw, nwait
data ndm/power/

nnode=2**ndim ! number of nodes
call receive(0, word, seg)
call receive(0, seg*word, a(1)) ! get share of data from host
call sort(seg, a(1)) ! sort this data
now merge first to left nearest neighbour, then to p=2^n;
c n=0, ndim
   do 5 j=0, ndim-1
      n2^n
      if((inode.e, nnode-n) then
         call receive(inode+n, word, seg)
         call receive(inode+n, seg*word, a(seg))
      end if
   end if
   if(inode.gt.n) then
      call send(inode-n, word, seg)
      call send(inode-n, seg*word, a(1))
   end if
5 c merge sort two vectors into b
   if(a, k=seg+1 : 100
   if(a(j), a(k)<seg then
      j=j+1
   l=j+1
   a(l)=a(k)
   k=k+1
   else if(a(k), a(j)<seg then
      j=j+1
   b(l)=a(j)
   k=k+1
   else if(a(k), |seg*seg then
      j=j+1
   b(l)=a(j)
   k=k+1
   end if
   if(j.lt. seg and k.lt. seg*seg) goto :10
   if(j, .eq. seg
      do 10 m=j, seg
         l=l+1
      end if
   10 b(l)=a(m)
   else if(k, l. eq. seg*seg)
      do 20 m=k, seg*seg
         l=l+1
      end if
   20 b(l)=a(m)
```

end if

```
call send(0, seg*word, a)
call receive(0, word, seg)
c all used, do next dimension
   do 25 i=1, seg
      25 continue
```

5 continue

c all merged, send from proc to host
   if(inode.eq.1) call send(0, seg*word, a)
c this is real parallel programming isn't it!
end
```

5) General algorithms.

Simple sequential processes are of the type, get one data
element, do operation, store result. This was generalised for
mathematical processes to, get as much data as required for
operation, do operation, store result. For most of these operations
were carried out at once in SIMD fashion. This assumes that the
data and results are to be in the same memory
available to the processor (albeit several processors with
individual jobs to do as in a pipeline). However data transmission
problems do arise in virtual memory where slow data access from
disk is countered by introduction of cache and paging devices.

These problems are the same, but highlighted, in a MIMD parallel
machine since data management is essentially left to the programmer.
By that I mean if data is in the local memory of one processor and
is required by another for some operation it must be sent
explicitly. This is hard work, requires imagination, and is the
central issue of general parallel computer algorithms design.

Consider for instance a scenario in which the entire memory
of the machine, of which a segment on each processor, is used for
the storage of program variables, which may span segment, that is
processor boundaries. Offsets can be assigned, and a table can be
provided to search for individual array elements. An operation, such
as multiplication, with two operands and a result requires the
cooperation of all processors to do the gather-scatter part, but for
only one vector multiplication of perhaps just a few elements. Hence
the scenario begins to look sequential if the number of elements is
small, and the application will therefore run slowly. This
emphasises that, on parallel machines it is necessary to scale up the
problem rather than just subdividing it. A further discussion of such implicit
shared memory simulation is given in [II.1.2].

This discussion brings into question the usual philosophy of
FORTRAN subroutine libraries, which require to have data put into
them, and results stored after some complicated operation. In
general the data is not in the optimum place for the operation and
all processors must cooperate to get it. This eventually leads to
sequentialisation unless one is careful. There is as yet no protocol
for the storage and retrieval of data on a parallel machine by which
this problem may be overcome. I stress that local-memory machines on
which general algorithmic parallelism is implemented will always run
into sequentialisation through file or memory contention or violation of data
integrity (outside the CSP methodology), if a naive approach is employed. New research work done in the field of
multi-access databases will hopefully indicate answers to the
problem of contention and transmission of data for both shared and
distributed-memory architectures. Semaphores are a convenient way in
some applications.

I illustrate the above ideas in a very naive form, with the

III.1.6

III.1.7
following program that uses the entire computing resource of an nprocessor machine to transpose a large matrix, of which seg(1) elements are stored on each node i in contiguous fashion.

```fortran
subroutine mtrans(idim,a)
parameter (nnode=nprocs)
integer seg,procs,offs
dimension a(*)
common/proclist/procs(nnode),offs(nnode),seg(nnode)
common/talk/inode,nend(10),nw,nwait
external equal
do i=2,idim-1
    do j=i+1,idim
        c find out on which processor element a(i,j) of the full matrix
        c resides, and put its number into procs(1)
        call use(idim,i,j,1);
        call use(idim,i,j,2)
        if(inode.eq.procs(1).or.inode.eq.procs(2))then
            procs(3)=procs(1)
        call vcopy(a(offs(2)),1,b,1,3,equal)
        call vcopy(a(offs(2)),1,b,1,3,equal)
        end if
        end do
    end do
end
```
very good communications interface to the operating kernel, better than the simple procedural interfaces on other systems in which libraries are employed.

III.1.3 Numerical subroutine libraries for parallel machines

Eispack

A parallel version of the Eispack library for matrix eigenvalue problems is being prepared for the Intel hypercube. So far only the central part of the library is available, that is the computation of all eigenvalues and eigenvectors of a dense, real, symmetric matrix. Calculations are distributed, and the innermost loops of the algorithms access columns of the matrix. A matrix of order N is distributed over n processors by storing N/n columns per processor. There may be a small load imbalance if N is not a multiple of n.

The programs use Householder similarity reduction to tridiagonal form, followed by bisections on each processor and "perfect-shift" tridiagonal QR iterations with accumulation of the distributed eigenvalue matrix.

Lincube

The Lincube library developed from Linpack for the Intel hypercube analyzes and solves systems of linear algebraic equations involving dense matrices. Data is again distributed columnwise as in Eispack.

NAG

A NAG library implementation and general study has been provided under SERC EMR and EEC Esprit 1085 projects by the group at Liverpool University [3]. A small set of fully-documented parallel library routines has been provided [4]. Routines are written in OCCAM and are assumed to be callable from a serial FORTRAN (or other "alien" language) source running on the host processor.

Data is distributed as necessary, some planning has however been done for the case of already-distributed data. A rectangular grid topology has been adopted as the standard case at present, although the authors of the report [3] consider it to be a temporary measure. The library is not reconfigurable.

Some discussion in [3] focuses on the program structure and notes the "inside-out" environment inherent in multi-processor OCCAM. I do not consider it to be uniquely a feature of OCCAM, although it is very much emphasized in that language with the need to write explicit harness code (see section IV.3 for instance). It seems rather to be deeply engrafted in one way of parallel coding which I labelled above "skeleton" coding.

A further development is the possibility to dynamically load routines under control of a "library eegener". Routines are loaded when requested by the host program by a set of buffer processes which are transparent to the user. Only one routine is loaded on to the array at a time.

SUPRENUM

The project recognizes the need for standard communications libraries [II.1.2] both to simplify programming and for portability. The portability issue means that standard libraries should be implemented on all machines. The major thrust of the library so far has been in grid topologies for finite-element type calculations. Since the routines were developed on an IFSGO/1 hypercube the library is available through the Intel Users' Club.

Summary

Clearly the above libraries have been developed for cases in which the array is viewed as an attached processor pool. They cannot be called from completely distributed programs, unless run on attached subarrays or an attached array mapped onto the original processor and executing concurrent code (probably causing a logical bottleneck and contention as in shared-memory machines).
III.2 Debugging parallel executing programs

Our discussion is limited to MIMD architectures with partitioned local memory. A debugger must therefore be able to analyse the following:

1) Sequential code of a process or task
2) Messages between processes in send or receive operations
3) Dynamic process movement in load or kill operations, and relocate operations in the future
4) Analyse and change variables in local store
5) Analyse and change contents of messages
6) Analyse and change configuration of active processes (not presently available)

This assumes that the software debugs sequential processes, messages and tasks, which are collections of sequential processes.

The debugging procedure is in essence familiar, and similar to that for normal sequential code. Debugging of UNIX processes is equivalent. An awareness must be maintained of what processes are active, their source code (so that it can be pointed to for symbol manipulation), and their interconnection and dependency (through messages).

A set of problems specific to concurrent processes has been defined by Snelling and Hoffman ([1.1.1]). They are as follows.

The "stamped" effect which occurs when evidence of an error in one process is lost because another process.

The "bytander" effect which occurs when a process fails because its data was corrupted by another process.

The "deadlock" effect when several processes stop waiting for messages. This is usually easy to correct.

The "irreproducibility" effect in which a program may give different results each time it is run because processes are executed in a different order. This may, or may not be a problem. It makes convergence techniques hard to parallelise.

The "Heisenbery" effect, in which addition of the debugging processes to the task remove the apparent problem or shift it to another area.

A good debugger should enable these problems to be sorted out.

Debugging is inherently difficult because of the non-deterministic nature of concurrent execution, itself responsible for the irreproducible behaviour of programs.

The present concurrent debuggers provide a capability to:

Control program execution, by initiating, suspending and restarting execution of processes through breakpoints and tracepoints which may, conveniently, be conditional. Points may be set on user-defined object points to in the source listing, such as variables, statements, labels, subprocesses or any other program entity.

Access data and structures, to retrieve information when a program is suspended, modify program entities, read message status and contents, and continue execution after making repairs.

Load processes. A user may wish to modify the program structure by loading or killing processes. This is not easy as their relation to other processes relies on ability to communicate - a feature built in to the compiled code. Nevertheless some flexibility can be allowed by careful programming. Most use of these commands are made

when initially loading the task to be debugged onto the machine. A particularly important aspect of this on parallel machines is the ability to load tasks onto different, or the same processor i.e. to configure the job. That makes it possible to experiment with arrangements for load balancing.

Control the debugger, with execution of command scripts and logging of sessions (in case of accidental typing mistakes), calling UNIX services, defining aliases and getting help.

To our knowledge the DECON concurrent debugger for the Intel iPSC is the first one satisfying the requirement to debug and modify messages [1.2]. Nevertheless it does not allow much flexibility in configuration, except in the initial loading as mentioned above. It furthermore fails to debug code which is loaded under program control, since the code starts executing immediately and no breakpoints can be set.

The use made by NICHE Technology (Transtech) and Melko of the SUN dbx tool allows a complete parallel program to be compiled on the SUN and analysed. message passing and process control being done by the trillium-like PRT operating environment or SUN cStools interface. Fully debugged processes can then be recompiled for the array.

Another debugger is available for the NCUBE machine, which has the capability to analyse messages and is otherwise rather like dbx on conventional systems.
III.3 Load Balancing and Graphical Optimisation Methods

Only a few hints on what is possible, and some examples of implementation tools can be given.

Analysis of mathematical algorithms is the first step to optimisation. In sequential machines considerations are the number of floating-point operations performed, amount of memory used, and total memory access time. In MIMD machines we must add to this list memory access time to another node. This is a complex quantity since it involves sending and receiving operations as well as data transmission on a link. Sending and receiving between concurrent processes on a node is another problem requiring separate consideration. In both cases, processes must synchronise, and the inevitable waiting time must be accounted for in a detailed analysis. This analysis should be performed for different configurations to find an optimum, and different algorithms compared. An additional serious complication is the non-deterministic behaviour.

Whilst this procedure might be feasible for a given mathematical operation, such as Gaussian elimination in a matrix, and has indeed been carried out [1] it is not possible in more sophisticated tasks where data dependency is involved. In these cases the only solution, unless we resort to actual programming, which is likely to be expensive and tedious, is to use simulation tools. A sequential block of code of the traditional type might be analysed for floating-point operation counts and memory access as usual, and interacting blocks of code analysed for data dependency in decision making and message passing. This information is then used to program a simulator which will model the passage of computational activity through a processor array. Different configurations might be tried to see the effect and the overall performance estimated.

An example of this technique is the TRANSIM simulation tool and dynamic profiler package written in C and developed under the Alvey project [2]. A similar tool to Schmid [3] is available for placing and analysing activity on shared-memory machines. The packages run on a SUN workstation, and depict the processor array with colouring to indicate computation rate. Hot bottlenecks and cold areas of inactivity can be identified easily and corrected by relocating the processes.

Interesting wave motion of computational activity has been found in moderately complex codes. This unexpected nonlinear behaviour emphasizes the importance of such analysis exercises, and also the inherent difficulty of parallel programming.

A static profiler has also been developed under the Parsifal project [4]. A high priority timer runs in parallel with the application and a mapping to source code instruction pointers enable runtime information to be gathered.

IV Programming Environment (FORTRAN)

This section is divided into seven parts which present the FORTRAN77 programming environment on different multicomputers. These are mostly libraries of communications routines which may be linked to the user's sequential code and loaded onto the nodes. The routines interact with the operating system kernels as described above.

Of particular importance to Daresbury Laboratory are the Intel and Meiko multicomputers which are therefore described in greater detail.

IV.1 Helios

The following notes on the Helios programming environment are a summary of the presentation at the Helios developers' workshop held in 1986. The original overheads were for the C programming language, so the examples have been translated and may not correspond exactly with the commercially available package.

Inter-process communication is by message passing using links if the processes are on different transputers. This is transparent to the user and insensitive to the network topology because logically created ports and surrogate ports are used as intermediaires. Processes can therefore be placed anywhere, and a table is kept with their location and capability. The overall system should be fault tolerant because processes can be relocated. Helios works as a client-server model the distributed servers are transparent, can be relocated, and all have the same protocol. Port servers send a message at the request of a client (a process). A program running on the host computer is also a Helios node and supports a file server process plus screen, keyboard, and port servers.

Messages, ports, lists, semaphores and other objects have a specific data structure which is described in the Helios documentation.

List of routines available in Helios

Message-passing calls

port=NewPort() -- locate, allocate, and initialise a free slot in the port table
 ierr=FreePort(port) -- release the given port and invalidate its port descriptor
 ierr=PutMsg(scbl) -- send the message desribed in the message control block (scbl)
 ierr=GetMsg(scbl) -- receive a message into the scb
 ierr=AbortPort(port) -- abort any exchanges on this port
 ierr=SendExcept(port, code) -- send an exception message to the given port

List handling routines. Head and tail are pointers to a process list. Access to the transputer's process list is provided by these operations.

InitList(list) -- initialise the list to empty
AddHead(list, node) -- add the node to the head of the list
AddTail(list, node) -- add the node to the tail of the list
RemHead(list) -- remove the head node of the list and
return it, or NULL if the list is empty
node=ResTail(list) -- remove the tail node of the list and
return it, or NULL if the list is empty
Remove(node) -- remove the node from whatever list it is in. It
must be in a list
PreInsert(next, node) -- insert the node immediately before
next in its list. Next may point to the central field in a list
structure, in which case node becomes the new tail item
PostInsert(prev, node) -- insert the node immediately after
prev in its list. Prev may be a list structure, in which case the
node is added to the head of the list.
WalkList(list, function)
node=SearchList(list, function)

Semaphores are used to synchronize between processes sharing
memory on the same transputer. This is also a feature of the 3L
FORTRAN system and other shared-memory operating systems.

 ierr=InitSemaphore(semaphore, count) -- initialize the fields
 of the semaphore and set the count with the value given
 ierr=Wait(semaphore) -- perform usual semaphore wait operation
 on the semaphore
 ierr=Signal(semaphore) -- perform usual semaphore signal
 operation on the semaphore
 ierr=TestSemaphore(semaphore)

Terminal i/o. Interactive streams are defined to be windows,
consoles and serial or parallel ports. Functions are available to
tcontrol stream attributes and i/o events on streams.

 port=EnableEvent(stream, mask)
 ierr=Acknowledge(port, counter) -- send an acknowledge signal
 ierr=NegAcknowledge(port, counter) -- send a negative
 acknowledge signal, e.g. if an event is lost
 HandleEvent(eventbuf[])
 ierr=SetEvent(handler) -- install the given event handler in
 the event chain
 ierr=RemoveEvent(handler) -- remove the event handler from the
 event chain

Other Kernel routines handle tasks, links, memory and
miscellaneous functions

 ierr=TaskInit(task)
 ierr=KillTask(task)
 ierr=CallException(task, signal)
 ierr=BootLink(link, image, confg, confsize)
 ierr=EnableLink(link)
 ierr=AllocLink(link)
 ierr=FreeLink(link)
 ierr=Reconfigure(linkconf)
 ierr=LinkDate(link, linkinfo)
 iblock=AllocMem(size, pool)
 ierr=FreeMem(iblock)
 InitPool(pool)
 FreePool(pool)
 iblock=AllocFast(size, pool)
 irecall=InPool(addr, pool)
 Terminate()
 Delay(microseconds)
on the Convex C2 or SUN 3.0 machines. Remote login to the Intel is possible for external users from the Convex through the DL Ethernet or from the SUN for internal users. I will assume that you are working from the Convex and that a shell has been opened in genmas so that files may be held and edited on the C2 and run on the Intel after copying with the remote protocol software. Direct remote hosting of the Intel SRM is also possible and the following notes should be interpreted accordingly. We also plan to directly access the Convex filing system from the Intel through NFS.

The following sequence of commands can be issued

```
login ipac
logon: <id>
password: <password>
```

This establishes a session on the SRM and the root directory is /usr/user/<id>. We assume that files host.f and node.f have been created (using genmas for instance) on the Convex. They are copied to the SRM by issuing the command

```rcp cxa:convex path/host.f host.f```

A host program is only needed if host routines are to be run, and if communications to the host is required e.g. for farming type applications. Compute-intensive tasks should NOT be run on the SRM as that would cause serious overloading in a multi-user environment! The SRM is redundant if only file i/o is needed on the nodes and only node.f need be written. However I continue to assume that a fully parallel program is being developed with both host and node processes. A useful makefile containing the compiler switches to use is as follows

```all: host node (or vnode)
host: host.f ftime.o ftime.o -o host host.f ftime.o -host
ftime.o: ftime.c ! a mail c timing routine cc -c ftime.c
node: node.f f77 -o node node.f -node
vnode: vnode.f f77 -o vnode -x vnode.f -single -lxvec -lx -node```

Compilation may be found faster if the -X302 option is used instead of -vx in cases where that works. The -single option can be replaced by -double (for double precision intrinsic functions, the default) or -both. The -lxvec option loads the correct version of the VecLib library. A -g option may be put on the compiler to produce object code with information for DECON, and -v for verbose compilation.

The makefile yields executable modules host and node, or host and vnode. Quite a long wait is needed for complicated vector code due to the slow link editor. This again emphasises the need to carefully name the SRM resources which is why editing on the Convex is suggested and not using compute-intensive tools.

A subcube of the 32-node array can now be allocated, with redirected i/o if required, by issuing the command

```gencube [-c <cubename>] -n n [-<input file>] [-<output file>]```

n is the number of nodes in the subcube, which must be a power of 2. If such a cube is available it will be allocated, an error message is printed if the request is not possible.

Information about currently allocated cubes and their name is obtained with

```cubeinfo -a```

The node processes are loaded onto the currently allocated cube with the command

```load [-c <cubename>] node <pid>```

where <pid> is a number used as the process id for communications. It is selected by the current user and should not be the same as any other processes on the same cube; if omitted zero, i.e. used. The process starts execution immediately and will continue until either it waits for host communications, it finishes (or crashes) or it is aborted by the command

```killcube [-c <cubename>] [-<pid>]```

The host process is started simply by typing its name host.

The cube is released for other users by typing

```relibcube [-c <cubename>]```

If the same is omitted the current (last referenced) cube is released and all processes on it are killed. The current cube is changed with the command

```allocate [-c <cubename>]```

An alternative way to redirect output is to use

```newserver [-c <cubename>] [-<input file>] [-<output file>]```

It is difficult to get information about programs running on the cube, they tend to look dead unless the files are being updated.

The IPSO/2 simulator

We have put a copy of the UNIX simulator in cxa:/priv1/wab/sim. You can link to it by typing

```ln -s /priv1/wab/sim/bisimlib.a
ln -s /priv1/wab/sim/bisim```

in the directory you want to use.

A host and node program are compiled for the Convex (upon which the simulator software executes).

```fc -c host.f node.f```

The object files produced are then linked to the simulator.
The link control file is read into the linker alongside the vector subroutines and directs the relevant variables to be placed on the VX node as follows.

```
f77 -c -sx vxh.f
vxh -single -o vxh vxh.o
```

The link control file is read into the linker alongside the vector subroutines and directs the relevant variables to be placed on the VX node as follows.

```
f77 -c -sx vnode.f
f77 -v vnode -sx vxh vnode.o -single -lvxexvec -lvx -node
```

Concurrent file system

An introduction to the Intel concurrent I/O facilities has been given by D. Matty [6]. It is possible to access files on the directory `/cfs/<id>` from the nodes, but not from the SRM. For this reason a node-based subset of UNIX has been provided which has the capability to copy files from SRF to cfs file systems, and to inspect files. It can be inconvenient to access cfs files from the host process. We have therefore written a subroutine library and parallel-executing node file server process to enable this to be done by message passing.

A cfs file and directory might be created as follows

```
getcube -t 1
ssh
```

### IV.2.5

### IV.2.6
attachcube(subname) -- attach a cube and make it the current cube

cprone(type) -- wait for a message to arrive
crecv(type, buf, len) -- receive a message of len bytes and
wait for completion
csend(type, buf, len, node, pid) -- send a message of len bytes
and wait for completion
cubeinfo(cf, numcubes, global) -- obtain information about
allocated cubes, see [3]
flick() -- relinquish processor to another process
flushmsg(type, node, pid) -- flush specified message from
system
getcube(subname, subtype, xname, keep) -- allocate a cube
gray3() -- return the binary-reflected Gray code for an integer
-- return the inverse of gray. These calls are used
when designing the shortest actual path between nodes.
handler(type, ifunc) -- provide user-written exception handler
infocount() -- returns information about a pending or received
message
infonode() infoipid() infotype()
ctoch(v, s, n) ctofn(v, s, n)
htocf(v, s, n) htocf(v, s, n)
htoci(v, s, n) ctohs(v, s, n)
htocs(v, s, n) htoce(v, s, n)
hrecv(type, buf, len, procedure) -- interrupt-driver receive

with handler procedure

iproc(type) -- determine whether a message of a selected
message is pending
id=irecv(type, buf, len) -- receive a message asynchronously,
without waiting for completion
id=isend(type, buf, len, node, pid) -- send a message
asynchronously. The above calls are used to overlap computation and
communication.
killicube(node, pid) -- terminate and clear out a process
killicproc(node, pid) -- terminate a process
killoplog() -- terminate a syslog process
led(i, state) -- turn the node board's green led on or off
load(filename, node, pid) -- load a node process
masktrap(mask) -- return the time in milliseconds
mscancel(id) -- cancel an operation with identification id

returned by a call to irecv or isend
msgasend() -- determine whether a send or receive operation
has completed
msgawait(id) -- wait for completion of a send or receive
operation

myhost() -- obtain node id of the host machine
node = mynode() -- obtain the node id of the calling process
pid = mypid() -- obtain the process id of the calling process
newserver(subname) -- start a new file server for the
specified cube

nindex() -- obtain dimension of hypercube
node = numnodes() -- obtain the number of nodes in the cube

2^n

rcube(subname) -- release a cube
netpfid(plu) -- set process id of a host process, must be done
before loading node programs
setwsyslog(statts) -- start a syslog program
traplock(mask) -- block selected traps
trapset(mask) -- sets a new mask value
waitall(node, pid) -- wait for all the specified processes to
complete.

waitone(node, pid, cnode, epid, ccode) -- wait for a specified
process to complete

cread(lu, buf, nbytes) -- synchronous byte read from a
concurrent file

cwrite(lu, buf, nbytes) -- synchronous byte write to a
concurrent file

istat=nextnode(1d) -- has an asynchronous read/write operation
terminated?

istat=1node(lu) -- find node of a cfs file
lowait(id) -- wait for termination of an asynchronous
read/write operation

lu = iread(lu, buf, nbytes) -- do asynchronous read from file
lu = wwrite(lu, buf, nbytes) -- do asynchronous write to file
istat = lslinkfile(lu) -- is file pointer at EOF?

inode = lsseek(file, offset, iwhence) -- move file pointer in a file
bytes = leize(file, offset, iwhence) -- size of file from given
position

istat = restrict(sv, file, list) -- restrict distribution of
concurrent files to a particular disk volume

setmode(file, node) -- set file pointer mode for multiple access
of cfs file on file

Example program

The following example is a farming-type application. The nodes, of
which there can be any number, first read the file rate2.dat and
a few lines from the file rate2.go. All nodes read all the data as
they have independent file pointers (its UNIX don't forget). Then
the host program skips a few lines and waits for the nodes to
request data. Upon receiving a request it reads the next line of
file rate2.go and transmits its contents to the requesting node. In
that way a different line of data is sent to each node. This carries
on, with each node requesting a new line of data when it has
finished its work with the old one, until all lines are used. At
that point the host program waits to terminate, but it can't because
it would then kill off the node processes that are still working
on the last data read. This illustrates that starting and finishing
parallel programs is often the most tricky job.

c program rate2
c IWGO host program for parallel execution farming data from file
c rate2.go

implicit real*8(a-h, o-z)
logical list
character*6 ctype
open(2, file='rate2', statts='OLD')
c jump over data lines in this file not required on the host
read(2, '((4.1)') dumb, dumb, dumb
read(2, '(11)') next
read(2, '11)') list
write(5, l'11 ** * program rate2 *****')

IV.2.7 IV.2.8
c ask from keyboard
    write(6,'('' number of nodes ? ''')
    read(5,'(a2)')ctype
    c get a cube and release it when this task finishes
    call getcube('rate2',ctype,'.0')
    c set a process id for communications, can be any number
    call setpid(0)
    c load the node process contained in file rate2.out
    call load('rate2.out','i',0)
    c start clock
    start=real(mclock())/1000.0
    c node requests data
    5 call crecv(1,buf,0)
    c find out which node and what pid
    inode=infonode()
    ipid=infopid()
    c read temperature from file
    read(2,'(f10.5)')temp
    if(temp.1.1.0e-6)goto 10
    c send data to inode
    call csend(2,temp,8,inode,ipid)
    goto 5
10 finish=real(mclock())/1000.0
    time=finish-start
    write(6,'('' host execution time ''g12.5,'' seconds '')time
    c How do I know that all the processes have finished? (left as an
    c exercise for the student)
    c this next line kills the processes and unloads the cube....
end

C PROGRAM RATE2
C
C this is the node program for the iPS/2
C
C CALCULATES MAXWELL-BOLTZMANN AVERAGED RATE COEFFICIENTS BY
C SPLINE INTERPOLATION OF CROSS SECTIONS FOLLOWED BY ANALYTIC
C INTEGRATION OVER A SMALL INTERVAL WITH STRAIGHT-LINE FIT
C
IMPLICIT REAL*8(A-H,O-Z)
PARAMETER (INS=3)
LOGICAL LIST
DIMENSION E(50,1ms),SIGMA(50,INS),CSIGMA(50,INS),AAA(50),BBB(50),
      CCC(50),SIGA(INS),SIGH(INS),Sigs(INS),
      rate(INS),thresh(INS),nest(INS)
open(i,.file='rate218.dat',status='OLD')
open(5,.file='rate2.go',status='OLD')
READ(5,10)REM

10 FORMAT(F10.5)
READ(5,10)CONV
READ(1,'(219)')NE,NS
READ(5,10)MAX
read(5,'(8f10.5)')(thresh(j),j=1,ns)
DO I=1,NE
   READ(1,'(g12.5)')(E(1,j),(SIGMA(I,j),J=1,NS)
END DO
C loop over temperatures, data is farmed out from the host
C get host's id
 ldhost=myhost()
C tell the host I am ready for data

50 call csend(5,dua,0,ldhost,0)
C receive temperature for this loop
 call crecvv(2,temp,8)
C node just hangs if its not sent anything, i.e. at the end
... WRITE(6,45)TEMP,(RATE(I),I=1,ns,rate)
45 FORMAT('TEMPERATURE K','g12.5,' REACTION RATES CM**3/S ','
      1','6G12.5')
GOTO 50
END

The above code is a real, but very small application. The
kernel of another application in which an inner loop of a multi-
dimensional quadrature was parallelised is shown in section III.1.1
above.

The second example is again from a real code which is the host
file server mentioned above. It is an example of how to implement
the ALT functionality using the iprobe routine

10 do 300 I=1,5
   iotype=I+300
   if(iprobe(iotype).ne.0)goto(1,2,3,4,5),1
300 continue
   call flick()
1  ... <-go to 10
2  ...  goto 10
3  ...  goto 10
4  ...  goto 10
5  ...  goto 10
IV.3 Meiko Computing Surface (installed at DL summer 1987, upgraded winter 1986)

Acknowledgements: the original M10 computing surface is on loan from the SEIK/DTI Transputer Initiative loan pool at RAL. DL is an Associated Support Centre to the Initiative, and undertakes to make programs developed on the machine nationally available through the Sheffield National Transputer Support Centre Library.

Configuration of the current surface at DL. Transputer types:

1x T414 MK014 3MB byte host processor H
1x T414 MK015 128KB byte graphics processor G
1x T414 MK021 8MB byte mass-store board M
4x T800 compute nodes (1 quad board mk050) 4MB byte memory each
P1-P4
8x T800 compute nodes (2 quad boards mk009) 256 kbytes memory each
P5-P12
Electronic switching of links to reconfigure nodes
300MB byte scsi disk and 1/4" tape cartridge driven off the mk021 front end Sun tca operating SunOS 4.0

The IMST414-20 integer transputer can run at 10 mips, the 64-bit floating-point unit of the IMST800-20 can sustain 1.6 Mflops/sec (Inmos figures)

Figure 1. topology of M10 for Fortnet v2.1. T-links in dual "daisy chain".

message-passing links

filling system links

The programming environment is FORTRAN77 compiled on the Sun under SunOS 4.0 and occam 2 compiled on the Meiko host board under OPS. The pain associated with simultaneously using two operating systems is reduced on a SUN with two or more suntools windows (separate ones for each environment). The OPS must be booted in a shelltool window rather than a cshelltool window. The Meiko at Daresbury has also been provided with the MMVCS and MeikoOS operating systems, however this requires the host, mass-store (to drive the disk) and one mk050 node to be dedicated to the system leaving only 11 nodes (mostly of low memory size) to the user. Alternative software tools, called CTools, are available from Meiko for the Sun hardware, and will provide a more familiar UNIX-like environment. These are briefly described at the end of this section.

The FORTRAN77 extensions supplied by Meiko Ltd. in 1987 were minimal. They access the occam channels for data transfer and timing [11]

ib0(ichan, ibuff, isize()) -- inputs integers from ichan,
ib0(ichan, ibuff, isize()) -- outputs integers to ichan.

Corresponding routines are provided for other data types. The channels must be correctly defined in the descriptor file and in the underlying occam harness.

time(time()) -- get the current transputer clock time in 64u seconds.

Harness software for the Meiko

Some harnesses (see section II.5) have been written by Meiko Ltd. at Edinburgh for very specific jobs like task-farming applications, either single tasks from several users, or several tasks from the same user. Another harness runs in a cell type application with the possibility to swap data or cell edges, useful in OPD or lattice gauge type work.

Other harnesses are available from the Southerton Transputer Centre, via Sheffield, and also do farming applications [2-4].

The harness used at DL and Durham University is called Fortnet v2.1. It was developed by Sebastian Zureck (Cambridge), Lydias Heck (Durham) and myself. A more detailed description and published version of the code is currently being prepared for public release [6]. A version could also be made available for the Inmos TRAM in the future.

"une double postulation simultanée" Baudelaire

The original harness program at Daresbury was written entirely in occam by Sebastian Zureck during the summer of 1987. The code was subsequently simplified and partly converted to FORTRAN by Bill Purvis of DL shortly after Sep left. Since then I have taken over the work of improving and maintaining the code - the most important improvement being to implement a way of communicating READ and WRITE statements in the user process on each node with their respective files and devices on the front end machine. To do this it is necessary to develop a protocol for sending and requesting formatted (or unformatted) information to or from a particular logical unit. A more advanced protocol is now used to do full-blocked parallel SENDING and RECEIVING operations and version 2.1, written by Lydias Heck, has the full FORTRAN file system on all nodes.

A naive sorting program is illustrated here as an example. It reads data on the mass store board and partitions it into segments. Each segment is sent to, and worked on by a different processor, and the results are finally collected again on the mass store. The same slave program is now implemented on all the nodes to reduce the disk storage overheads and complication of the harness (see Fig. 1).

The slave processes (Fig. 2) consist of a buffer process (Fig. 4) to handle transmission of messages along the chain (fig. 4). One along a worker process (fig. 3) which contains the FORTRAN, READ and WRITE and SEND and RECEIVE communication is done using the protocol of fig. 5. The user's program is embedded in the FORTRAN code and should be a self-contained program with the statement "SUBROUTINE USER" at the beginning. The program and its subroutines must have been compiled on the Sun with tff77 and linked with tlink as shown below. It must fit in the memory of a single MK009 processor board (256 kbyte) if all 13 transputers are to be used, although this is clearly not a restriction on other implementations.
Subroutine calls (contained in the IOSUB library)

CALL STOP -- This call toggles the activation index for the processor in the server process. It is called by the worker process and should not normally be required by the user. Workers activate at the start of execution and deactivate when the user process terminates correctly. When all processors are deactivated the server produces a table of statistics about the entire job.
A STOP statement in the user's code should be replaced by CALL STOP
STOP
except in subroutine USER when RETURN should be used instead. The FORTRAN STOP should correctly terminate a process.

CHARACTER*132 BUFFER
CALL READ(LU,NCHAR,BUFFER)
READ(BUFFER,<fmt>/)<olist> -- This is the usual way to read a line of formatted data from LU which is attached to a Sun file or the TTY via the server. The connection between LU and the file name is made in the FORTRAN SERVER routine which runs on the first processor of the chain. It must be modified if required. This procedure gives a way of distributing data in a globally accessed file out to many processors just by reading it. Load balancing is automatically achieved if each processor reads a line of data when it is ready to work on it, as seen in the rate example in section II.2.

CHARACTER*132 BUFFER
WRITE(BUFFER,<fmt>/)<olist>
CALL WRITE(LU,NCHAR,BUFFER) -- This is the usual way to output formatted information to LU which may be attached to a Sun file or the TTY via server. Comments apply as above. These read and write subroutines bear some resemblance to the host file-server library we have implemented on the Intel, see section IV.2.

Normal FORTRAN I/O and OPEN and CLOSE statements can also be used, in which case all processors read all the data in a file which is opened by them. Care should be taken when writing to shared files opened in this way as the result will be unpredictable.

CALL WAIT(N) -- Causes the program to wait on the current processor until processor N attempts to synchronise with it. This is used as part of the blocking protocol for SEND and RECEIVE.

CALL CHECK(M) -- Checks if processor M is waiting and waits until it is to synchronise. Used with CALL WAIT above.

CALL RECEIVES(N,ISIZ,BUFFER) -- Receives ISIZ bytes into character string BUFFER from processor N assuming that they are synchronised. It waits for the data to arrive from an ocean buffer process via channels.

CALL RECEIVE (N,ISIZ,ARRAY) -- Receives data of other types as above

CALL SENDS(M,ISIZ,BUFFER) -- Sends ISIZ bytes from character string BUFFER to processor M which should previously have been synchronised by calling CHECK. This is done via an ocean buffer process and the call is to some extent asynchronous.
CALL SEND (M, ISIZ, ARRAY) -- sends data of other types as above

CALL RECANS(L, ISIZ, BUFFER) -- Receives ISIZ bytes of data into character string BUFFER from any processor that happens to send them. L is a redundant parameter on calling, but returns the address from whence the data came.

CALL RECVAF (L, ISIZ, ARRAY) -- receive other data types as above

CALL BRCASS (LIST, NPROCS, ISIZ, BUFFER) -- should broadcast ISIZ bytes from BUFFER to all the processors in the LIST(1:NPROCS). At present the implementation is slow; chances are needed in the Occam to improve it.

CALL BRCASC (LIST, NPROCS, ISIZ, ARRAY) -- broadcast other data types as above. The above four routines form part of a more general set of global communication routines which are being written for the Meiko and contained in the ISPs library.

INODE=MYNODE() -- Obtains the address of the current processor which can be used in program constructs of the form illustrated in section III.4.1

subroutines for debugging

CALL DEBUG(MODE) -- Mode is a character string which may take the values 'ON', 'OFF', and 'TOGGLE'. The three cases switch on, off and toggle the debugging mode respectively. In debugging 'ON' mode messages are printed out during the execution of each of the ISUB routines, which give information on the status of the system. This is rather like the trace command in the Intel iPSC simulator (see IV.2)

CALL STATS(MODE) -- Similar to the above. MODE is a character string which may take the values 'ON', 'OFF', 'TOGGLE' and 'PRINT'. The four cases switch on, off and toggle the debugging mode, and print currently collected results respectively. The effect of 'OFF' is also to reset the results to their initial values so 'PRINT' should be used before this is done. The statistics are collected each time a system call is made.

CALL CPU(TIME) -- This routine returns the present value of the local processor clock in seconds in double precision variable TIME.

CHARACTER*NCHAR BUFFER
CALL SUPER(NCHAR, BUFFER) -- This is the way to send a message of NCHAR characters in BUFFER to the TTY screen via the Supervisor Bus. BUFFER may be replaced by an explicit text string in quotes. This routine will not be available for the Inmos TRAN.

Changes in standard FORTRAN to use Fortnet
Six steps to parallel FORTRAN

1) replace PROGRAM statement by SUBROUTINE USER
2) replace WRITEs if necessary for common files
3) replace READs if necessary

4) replace STOP with a return out of subroutine USER to the calling system program
5) synchronise any points where inter-processor communications are necessary and use blocked SEND and RECEIVE.
6) put in debug calls!

Examples of the use of the subroutine library are shown below.

Practical hints for compiling and linking FORTRAN on the Meiko M10 at Daresbury

In order to use the Fortnet procedures described above a valid id on the Daresbury front-end Sun tcsa must be obtained. The usual way, as for the Intel, is to obtain an id on the Convex after which a remote login to the Sun can be made. Once this has been done both the FORTRAN and occam OPS environments can be used as well as Meikos. Meikos is not discussed in any detail here. An example of an include file for the linker is shown below, one of which should be created each for the master, server and worker programs. Names adopted are worker0.tpo and server.tpo which run on the main store board, and worker.tpo which runs on all compute boards. We suggest that these names are adhered to, and a new UNIX directory is created for each different program.

worker.tpo

INCLUDE /usr/meiko/ft7/fortran/hex/ft7head
PARAMETERS chan of any front. tcsa, tof, keyb, screen, user1, -
user2, report
/home/rja/export/worker
<user's main program modified as above>
{<user's subroutines>}
/home/rja/export/isubs

The directory selected should make reference to the one which contains the occam environment and the Fortnet system subroutines as in the above example. These are stored in the directory /home/rja/export/ which should be copied. When this is done the path translation file ofsttrans.trs should be modified to point at the new fortnet directory in the same way as does the reference /bob/.

bob    usr/rja/export/
user    usr/rja/export/sort/
syslib  usr/meiko/occam/system/
utlib    usr/meiko/occam/util/
complib usr/meiko/occam/library/
t4ym_objs usr/meiko/ft7/fortran/hex/

The user's main program and subroutines should be contained in files * .f (although if this is not the case only a spurious error message is printed) and each one should be compiled using the yft7 compiler as follows
tf77 +f file.f

After successful compilation, code for each node is linked with talk worker.t6 library

This has the effect of producing a library file worker.l16 which can be attached to the occam harness system. Details of how to do that are given in the next section. Note that separate files are not used
for each processor and the code contained in them is identical, therefore statements of the type

inode=makeode()
if(inode.eq.n)
...
end if

should be included if different operations are to be performed on different processors (especially for send and receive operations). Experience has shown that this restriction is necessary on larger parallel machines, and is not too difficult to work around. The INCLUDE statements in the .tpo files can refer to common source, as indeed they do for the worker and 10sub files in the system directory.

Code for the mass store board is linked from the worker0.tpo file which references part of the program responsible for data handling and control, and server which handles the front-end files. Since the mass store only has a T414, different commands have to be used as follows for the master and server:

tf77 file.f
	link worker0 library
tf77 server.f
	link server library

Using OPS to attach the FORTRAN code to the occam harness.

If the current directory is the one containing the Fortran harness (e.g. /home/rja/export/ the original one) OPS can be invoked as follows

sh -c -- if on a Sun or a Sun shelltools window or,
boot -- if on another type of terminal, these commands are
aliases which may be inspected in /home/rja/.cshrc
[root directory ?] /home/rja/fortnet2
[which terminal type ?] vt100

the screen will clear and the line
...
F /home/rja/export/level.top
will appear. One should now refer to the Meiko manual section on use of the OPS environment to find out how to edit occam programs and invoke the compile and configure utilities [5].

The actual occam code of the Fortran harness is contained in the

...PROGRAM Fortnet v2.2...

which should be opened until the separately compiled folds '...SC' are found. Inside these are references to files which are the descriptors of the user's FORTRAN source. They have the appearance:

.../bob/sub directory/worker.118
.../bob/sub directory/server.114
.../bob/sub directory/worker0.114

and should be edited to point to the correct sub directory.

Once this has been done each SC fold should be compiled using the occam-2 compiler and setting appropriate values for the options, and again whenever the FORTRAN to which it refers is re-linked. Finally the Program Fortnet can be configured and loaded onto the transputer network after using the fortnet.wir wiring file for

setting the pipeline switches. Using the run key will produce
results, or fail if problems are encountered. In the event of
failure some or all of the above steps will have to be repeated,
which after a while becomes Tedious.

Example: naive sorting program

master program

subroutine user
implicit real*4(a-h,o-z)

routine to partition and distribute data for simple parallel

workversion (1)

14/9/88

R.A.

parameter (nproc=8,maxe=500)
di=dimension x(maxx)
dimension m(maxa, nproc, nmax)
character*132 buffer
real*8 time, time0

call debug('OFF')
call write(6,37,'(''**** program sort **** Version 1.0'')')
call cpu(time0)
c read in data from file 1, all numbers stored on MK022 board
call read(1,4,buffer)
read(buffer,'(f10.5)')x(1)
c search through data for largest and smallest elements

max=x(1)
min=x(1)
do 5 i=2,n
call read(1,10,buffer)
read(buffer,'(f10.5)')x(i)
if(x(i).gt.xmax)xmax=x(i)
5 if(x(i).lt.xmin)xmin=x(i)
call cpu(time)
time=time0
write(buffer,'(g13.5)' ,seconds to read data)')time
call write(6,32,buffer)
c partition for processors into segments of width (maxa-mina)/p
width=maxa-mina/float(nproc)
write(buffer,'('' xmax, xmin, width = ',3g12.5)')
1
xmax,xmin, width call write(6,57,buffer)
c loop over data and collect it to send each segment to a processor
do 20 i=1,p
20 na(i)=1
do 15 i=1,a
15 c first processor handled separately
if(x(i).lt.xmin+width)then
a(i,na(i))=x(i)
na(i)=na(i)+1
end if

do 35 j=2,nproc-1
xna=xmin+width*float(j-1)
xmax=xmax+width*float(j)
if(x(i).ge.xmax and x(i).lt.xmax+width*float(j))
then

a(j,na(i))=x(i)
na(i)=na(i)+1

IV.3.12

IV.3.13
end if

35 continue

c last processor handled separately
if(x(i).ge.xmax-width)then
  zproc=x(i)
  nproc=x(i)
  nproc=x(i)+1
end if

15 continue

call cpu(time)
time=time+time0
write(buffer, '(g12.5, '' seconds to partition data'')' )
time = write(6, 38, buffer)

1 send data

do 25 i=nproc
  ip1=i
  call check(ip1)
  nsend=ma(i)-1
  call send(ip1, nsend)
  do 25 j=1, nsend
    call send(ip1, a(j,i))
  end do
25 continue

call cpu(time)
time=time+time0
write(buffer, '(g12.5, '' seconds to send data'')' )
time = write(6, 33, buffer)

1 receive ordered results, smallest number comes first

do 50 j=1,nproc
  ip1=j
  call check(ip1)
  nrec=ma(j)-1
  do 50 k=1,nrec
    call receive(ip1, k, buffer)
    call write(2, 10, buffer)
  end do
50 continue

call cpu(time)
time=time+time0
write(buffer, '(g12.5, '' seconds to receive data'')' )
time = write(6, 36, buffer)
end

worker program

1 slave coding of sort program version (1)
c new code 14/36/88 single file with fortran for all slaves on
compute boards:
character*32 buffer
dimension x(500)
real*8 time, time0
common/ talk, inode, nsend(10), nwait, nw

if(inode.gt.8) return

call debug('OFF')
nrec=1

call cpu(time0)
call wait(nrec)
call receive(nrec, 4, n)
write(buffer, ('' slave '', i2, '' receiving '', a(j,i) elements'')' )
inode.n

call write(6, 33, buffer)
do 5 i=1,n

5 continue

call sort(x, n) ! any sequential sorting routine can be used

call wait(nrec)
do 10 i=1,n
write(buffer, '(i10.5)''(i1)
call send(nrec, 10, buffer)
10 continue

call cpu(time)
time=time+time0
write(buffer, ('' time on slave '', i2, x, g12.5, '' seconds'')' )
inode.time = write(6, 59, buffer)
end

Future versions of Fortnet will include the following additions:
1) dynamic loading of tasks from a name file, or just by
   reference to the executable. This permits a batch queue to be easily
divided.
2) a way to kill executing tasks will be implemented
3) subroutine names will be made to resemble more those on
   the hypercube machines
4) the end processor of the chain will be given a link to the first
   for more efficient two-way message routing. This will in
generate create problems of flow control which must be avoided.
5) a message queuing system will be implemented in addition to
   the current self-directing protocol. This will enable the
   hierarchical communications calls and polling calls to be
   implemented similar to those on the hypercubes.
6) a high-level library of global and shared-memory emulation
   routines will be provided
7) run-time trace information will be displayed graphically by
   interfacing Fortnet to the Schedule package.

CStools

In the above pages we have outlined the general procedure for
accessing the GPS system on the Melko, or TBS on an Inmos system,
and indicated use of one particular harness to run parallel FORTRAN
code. It is significant that Melko Scientific Ltd. have recently
announced their CSC tools products, initially made to UNIX hosts,
but later to be available to all hosts. This range gives a different
philosophy to parallel languages as will now be described.

The means to send messages between parallel executing processes is
rather similar to the UNIX model. One process will create a named
message port, and attempt to read data from it. One or several other
processes may then send to the port. Messages are buffered and
queued in FIFO fashion and stay in order, By this means a
functionality similar to the Occam AET is provided, data will be
read from any process that is ready to send it.

Any number of ports may be accessed (limited only by the system
buffer space in memory) and the user does not in general know what
paths the data takes to arrive at its destination. A facility is
however provided to give direct link access for communication-bound
tasks. There is in this respect clear resemblance to the
hierarchical communication protocols in Unix. The designers of
CSC tools have benefitted from experience of the latter product on the
Niche system (section IV.4).

Further functionality is provided in the CSC tools environment.
/actually in the underlying Virtual Computing Surface VCS) to load
parameters using the `trun <name>` command onto a given configuration
of transputers. The processors and their processes are specified in
a file `<name>.par` where explicit links may also be specified. A
simple example is

```
par
  processor 0 master.ex8 slave1.ex8
  processor 1 slave2.ex8
  processor 2 FOR 3 slave3.ex8
endpar
```

Further instructions are of the type

```
*tnk files used to create the executable images might be
```

```
/INCLUDE /usr/mekio/fortran/hex/f77head
/INCLUDE /usr/mekio/fortran/hex/cs
```

```
master
```

which would be linked at run time with the command `link <name> t8`. In
addition to running a transputer network CTools can run a
network of UNIX processes (or one or more SUNs), giving the obvious
capability to debug with dbx tools. Again much has been learned from
earlier attempts by Niche to do this. Interface to SUN host processes
and graphics processes is clearly also possible.

Fortran subroutines in CTools

```
ldesc=cs_createPort(name) -- creates a port and returns a port
descriptor, or -1 if unsuccessful
ldesc=cs_findPort(name, iblock) -- searches for a port with
given name, waits until one exists if iblock=1
lerr=cs_send(ldesc, data, nbytes/4, iblock) -- attempts to send
data to the port described by ldesc, blocking as above
lerr=cs_recv(ldesc, data, nbytes/4) -- attempts to receive data
from ldesc and waits if none is available
```

```
cs_getInfo(nProcs, procid, localId) -- obtain information about
the array
```

```
cs_abort() -- terminates a process
```

IV.4 NCUBE

Current NCUBE products are the NCUBE ten, NCUBE seven and NCUBE
four which are manufactured by NCUBE Corporation, Beaverton, Ore.,
but are available in this country through Arrow Computer Systems
Limited of Kansas.

Hardware

The topology of the system is a hypercube.
The NCUBE ten cabinet can hold between 16 and 1024 processors
in less than 1 cubic metre. The custom VLSI processor used in the
ten is a 32-bit chip with 32 and 64 bit IEEE floating point and
error-correction memory interface and 22 independent DNA links. The
speed of a single chip is around 500 kIopsec or 2 mIop in 32 bit, or
300 kIopsec in 64 bit. Up to 500 mIopsec are available in the full
system. A node consists of one processor chip and 500 kbyte memory
on six chips although larger configurations are possible. Up to 64
nodes can be placed on a board, and up to 16 boards in the system

cabinet.

The stand-alone host is essentially an 80286/80287-based
microprocessor which runs a UNIX-style operating system called Axis.
The nodes take a kernel system called Vertex. Peripherals can be
easily added to the DMA links, and up to four disk drives are available.
Up to 8 host boards can be added giving the possibility of 64 simultaneous users with 64 Gbytes of storage. High-performance
graphics and an open-system board are other options.

The NCUBE seven and NCUBE four are smaller systems, with up to 128
and up to 4 nodes respectively. The four is a PC-AT style card which
can be plugged into a microprocessor system to run the NCUBE
software for development purposes. Up to four cards can be used
together.

Software

The Axis operating system supports an Emacs-like editor called
Nmake, the Nshell, a debugger and fast tape backup. The normal
language compilers have been extended with communication facilities
in the usual way.

Extensions to UNIX are the Ncube Network which is a networked
file system allowing a file structure to be spread across several
disk drives belonging to different physically connected systems, and
more powerful and uniform protection facilities. The hypercube may
be managed and allocated in subcubes and is otherwise similar to
other UNIX-style management mentioned here with the ability to load,
run, communicate with and debug programs in the cube.

The node operating system Vertex allows message passing,
process scheduling and debugging.

Fortran calls

Only a few of the available calls are shown to give a flavour,
rather than a complete list

```
swwrite(buffer, length, proc, type, flag) -- send vector of
```

```
swread(buffer, length, proc, type, flag) -- receive vector of
```

```
whoami(inode, proc_id, host, lorder_of_cube) -- call information
```
Other facilities and the ability to load and kill processes are rather similar to those on the Intel hypercube.

IV.6 Transtech NTP1000

Hardware

The NICHE platform (now marketed by Transtech) consists of one or more linked transputer-based cards which fit into a SUN 3.0 workstation. The cards take up 1 1/2 slots, and can be up to 3 could be put in a normal SUN. Each card has four sites into which may be put either one transputer and between 2 and 26 Mbyte memory, or 4 transputers with 2 Mbyte or less, or 8 with 32 kbyte memory. A fully populated board might therefore have 16x T680 + 32 Mbyte total memory. The system is very flexible, and separate VME card cages may be added to take more boards, the limit being governed only by expense and power supply capability. Normally a distributed system would be envisaged however with several SUN workstations having one board each for development of parallel code.

The system is electronically reconfigurable but has a hard-wired 'spine' passing through two links of every transputer. The remaining links may be wired inside a site, and sites may be wired together, allowing some flexibility.

Software

NICHE Technology took the trillium operating system (section II.2), originally designed for the FPS T20, and modified it to suit their machine. Several major parts have been re-written. The result is called the parallel runtime environment (PRE). NICHE's approach in using this environment was that it should be able to link together any number of processors of different types. In the NTP1000 these are the SUN 3.0 host and IMST580 nodes. The data transfer involves byte-swapping (as also on the Intel hypercube and Meiko) but enables programs to be developed wholly on the SUN for debugging purposes. The SUN dbx tool can be used in a window on each process of interest. Once processes are working they can be downloaded into the box with increase of performance. NICHE hoped to incorporate other types of processor into this system.

As well as the general purpose parallel FORTRAN kernel calls, software is available specifically for fararing applications with farmer, worker and gatherer tasks. This is very effective in FFT image-processing examples.

As can be imagined from reading section II.2, FORTRAN tasks talk to each other by connecting to the PRE kernel to pass messages. They may also request system services in the same way, for instance the UNIX-style file i/o. There is no occam harness!

Both synchronous MSEND and NRECV and asynchronous MTYS MSEND and NTRY RECV message passing is possible. The maximum length of message is currently 8192 bytes. The address of a target process is fixed by the transputer address plus a tag or event number. The send or receive 'statements only accept messages with the same user-supplied tag, otherwise they do not synchronise. It is important to understand that the event tag is actually sampled by the kernel, and cannot be received by a process not waiting for it. It is kept in the system until required. This is like a pid on other UNIX systems.

A message type can also be specified and the target program can choose between actions depending on that type. The message events are fixed at compile time by C-language compiler directives (trillium is written in C). For instance the top of a program might contain
The P77 compiler is by Pentasoft, and is an improved version of the one previously encountered on the T20. A list of the available subroutine calls now follows. The documentation I have seen was an early preprint, so there might be errors here:

Kernel requests

kattach -- attach process to the kernel
kdetach -- detach process from the kernel
kdoom -- doom a process (to be killed later when it makes a system request)

kexit(n) -- call kdetach and exit
kreinit(n) -- initialise data structures and call kattach
krecv -- local message receive
ksend -- local message send
kstate

tsend -- transport layer message send

Network requests

ator(host) -- absolute to relative address conversion
drecv -- datalink layer message receive
dsend -- datalink layer message send
inode = getnodeid() -- return the network id of the node
itype = getnodeid() -- return the type of node
ipid = getpid()

get_route -- given destination, find datalink to use
net = tnet(n) -- change byte order from local machine's order to PRE network (transputer) order

nrecv(event, type, length, flags, buf) -- network layer receive
nsend(node, event, type, length, flags, end) -- network layer send

rw -- reverse byte order in a 32 bit word (needed when passing messages between some machines)

h tsend -- transport layer message send
htrecv -- transport layer message send
n = nto1(n) -- change byte order from PRE network (transputer) order to local machine's order

i = ntry_recv -- asynchronous receive post, returns 0 if OK, -1 otherwise

i = ntry_send -- asynchronous send post

Utility functions

bcopy -- copy a block of memory
errno -- return the system errno variable
itimesdtimer() -- read the transputer timer; clock ticks at 64 microsecond intervals

setpri -- set process priority on a transputer

i = print(string) -- print a string on the user's screen

Low-level C library calls

fd = TOPEN(...)

ret = TREAD(fd, a, nbytes) -- read nbytes to address a
ret = TWRITE(fd, a, nbytes) -- write out nbytes from address a exactly as stored in memory

TCLOSE()
IV.6 Parsys SN1000

As far as I am aware, the Parsys transputer-based machine (SN1000 supernode), whilst configurably more versatile, is like the early Melko surface in that it has no parallel languages except Occam-2. The sequential FORTRAN77 compiler is the one supplied by Inmos or 3L. The supernode should however be able to run either the Fortnet v2.1 harness (see section IV.3) or the new 3L software (IV.7), or indeed a version of trillium (II.2).

The operating environment of the SN1000 is IDRIS, which provides a full range of POSIX standard (UNIX) commands, and enables the user to run a series of separate processes which may share the system resources. This allows coupling of F77 tasks via UNIX system requests (sockets) to the host program.

The main impact of the machine is its complete 4-fold reconfigurability and efficient engineering. As a result of the joint European ESPRIT 1085 project it has undoubtedly taught the partners a lot about parallel Occam engine design. A new project ESPRIT 2085 will look more closely at software design.

IV.7 3L Parallel FORTRAN

There are current implementations of 3L software for MICHE (Transmeta) and PC-based transputer arrays such as the Gemini system (2) and also the Melko surface.

The aim of 3L's tools is to allow concurrent transputer programs to be written without using occam. They supply high-level support for programs which exploit any number of transputers.

The software is designed for a single user, and specifically for embedded systems of transputers where there is no operating system as such. After the user program is loaded, it, together with some linked libraries, takes over the nodes. The run time library controls the transputer channel and scheduling. This results in a very low overhead of around 5 kbytes per node. Furthermore, since there is no building of communications through Occam harness multiplexors, inter-processor communication is efficient.

The 3L compiler accepts standard FORTRAN [3] and produces binary object code in Intel object file format for T414 or T800 transputers. Some extensions to sequential FORTRAN such as DO WHILE are permitted. A program is treated as a task and run in parallel with other tasks. The tasks can be run on a single processor or on multiple processors and linked via a number of input and output ports.

Concurrency features are added to the language through run time libraries as in the other products reviewed here. These extensions are rather similar to the ones implemented by Melko for their Cstools environment. As well as the ability to send and receive data on channels a functionality similar to the Occam ALT is provided by the ability to wait until one group of channels receives a message and report which it is, or to check without waiting for the presence of a message.

A further help for concurrent programming is the explicit farming software called the flood-filling configurator PCONF. A worker program, which is loaded to all available transputers, reads a work packet, processes it, and writes a reply packet back to the master. Data routing is automatic within this restricted paradigm. A normal configurator PCONF allows a general concurrent task to be built on an arbitrary array from instructions placed in an external configuration file.

A second library feature of the language is the multiple-thread facility. New execution threads may be created within a task at run time. Threads share the same code, static and heap data areas, but have their own stack. Threads are allowed to communicate across transputer channels using messages, and also use semaphore functions to flag access to shared data areas (common blocks). Ports or message buffers. Unfortunately, since FORTRAN is not reentrant, i.e. the same program unit cannot be active more than once, a subprogram can be only invoked once by a thread at any one time.

Compilation of code is done on the transputer using either the tcf or tcf commands e.g.

tcf source

which looks for a file source.tcf. This produces an object file source.bin as output. This is linked with the parallel Run time library using either the tcflink or tcflink commands. Executable files source.exe or source.d8 are produced. In MS-DOS the link line can alternatively be used for several object files.

The program is run using

cfg config source.cfg source.exe
The configuration file for a typical job is rather complex in appearance. An example is

```
processor host
processor root
wire jumper host[0] root[0]! describes hardware configuration
task one ins=2 outs=2 task two ins=2 outs=2 data=10k
task aserver ins=1 outs=1! declares tasks with the number of
input and output ports and amount of workspace
place aserver host
place one root
place two root! placement on host and first transputer
connect ? two[0] aserver[0]
connect ? two[0] one[0]
connect ? one[1] two[1]! shows how the tasks are connected
```

As will be seen below, the 3L run time library is rich and sophisticated. It comprises separate sections for control of host memory, threads, timers, semaphores, channels and processor tasks. Channels may be bound to ports of a task, as by the configurator, or freely assigned to integer words for internal communication between many threads of one task.

Before communication can be done between tasks the address of a physical channel connected to a port must first be found, then data is sent to or received on it. The actual channel address is defined by the configurator and is therefore only known at run time, whereas the port numbers and internal channel words are defined in the code.

List of FORTRAN subroutines.

```
out=F77_Chan_out_Port(k) -- finds address of channel bound to
output port k
in=F77_Chan_In_Port(k) -- finds address of channel bound to
input port k
iaddr=F77_Chan_Address(chan) -- return address of internal
channel word ichan
F77_Chan_Init(iaddr) -- initialises an internal channel word
whose iaddr for communication. All internal channel words
must be initialised. Channels bound to ports should not be
initialised.
F77_Chan_out_Buffer(buff, out) -- send lowest byte from buff on
channel out
F77_Chan_in_Buffer(buff, in) -- read a single byte from channel
in
F77_Chan_out_Word(var, out) -- send four byte var out on
channel out
F77_Chan_in_Word(var, in) -- read four byte var from channel in
F77_Chan_out_Message(nbytes, array, in) -- send a message of
nbytes on channel out
F77_Chan_in_Message(nbytes, array, in) -- wait for message on
channel in
```

```
i=F77_Chan_in_Buffer(buff, in, timeout) -- try to read during
next timeout 'ticks and return value 'false' if no byte found.
Equivalent functions are available for other message types as above.
in=F77_Alt_Wait(chan, iaddr1, ..., iaddrn) -- waits until one
of the channels has data ready to read and returns a value saying
which of the arguments it is
```

```
in=F77_Alt_Wait_Vec(chan, invvec) -- as above but the argument
list is replaced by a vector of channels
in=F77_Alt_Wait_Vec_Vec(chan, invvec) -- as above but the argument
list is replaced by a vector of channels
```

```
F77_Sema_wait(nsem, array, lcomplete) -- sends processor farm
message. Master to slaves or slaves to master
F77_Sema_wait(nsem, array, lcomplete) -- receive a
processor farm message
```

```
F77_Timer_after(timer1, timer2) -- is .true. if
timer1>timer2, otherwise .false.
F77_Timer_delay(ticks) -- causes current thread to wait for at
least ticks
```

```
F77_Timer_now() -- return current value of timer
F77_Timer_wait(ticks) -- wait until the value of the priority
time is at least time
```

```
F77_Make_out_Port() -- returns number of output ports
F77_Make_in_Port() -- returns number of input ports
ihandle=F77_Chan_reset(iaddr) -- resets a channel, and also a
link if the channel is mapped onto one. It will suspend a thread
which was communicating on the channel and return a handle to it for
later restarting.
F77_Chan_start(subroutine, iwsarray, mwidths, nthreads, arg1, ..., argn) -- starts thread
```

```
i=F77_Chan_wait(pthread) -- stops the current thread
```

```
i=F77_Chan_create(subroutine, mwidths, nthreads, arg1, ..., argn) -- attempts to create a thread using subroutine of the
same priority as the current thread and taking mwidths from the heap as workspace. This workspace is never returned, and
F77_Chan_start should be used alternatively.
```

```
F77_Chan_stop -- stops the current thread
```

```
F77_Chan_exit() -- ensures that no other threads use the run time library, waits if it is already in use. This is again
because FORTRAN is not reentrant and library routines can be invoked
only once at a time.
```

```
F77_Sema_init(mysema, val) -- initialise semaphore variable to an empty queue
```

```
F77_Sema_wait(mysema) -- choose one of the thread waiting for
semaphore to be reactivated. The value of the semaphore decreases by
one only if no threads are waiting. Only threads executing at
the same priority can synchronise with a semaphore, otherwise they
must use sema_wait on channels
F77_Sema_signal(mysema, n) -- call above routine n times
```

```
F77_Sema_wait(mysema) -- wait for semaphore, if the value is
zero it is incremented by one, otherwise it is unchanged and the
current thread is added to the list of threads and descheduled
```

```
F77_Sema_wait(mysema, n) -- call above routine n times
iaddr=F77_alloc_host_mem(nbytes) -- allocates a block at
```
least nbytes in the base memory of the host and returns its 32-bit
address
F77_free_Host_mem(jaddr) -- frees base memory
F77_Block_to_Host(inaddr, iaddr, nbytes) -- transfers nbytes of
data from transputer memory starting at inaddr to host memory
starting at imaddr
F77_Block_from_Host(outaddr, outaddr, nbytes) -- reverse of above.
F77_read_Segments(idc=block) -- reads processor segment
registers on a PC
F77_Host_interrupt(intno, lsegs, idc=segment) -- loads the
contents of DQS block into the host registers and then calls an
interrupt on a PC.

Acknowledgements

I would like particularly to thank Neil Burnett and Patrick
Evans who are now with Melko Scientific Ltd., Richard Chamberlain of
Intel Scientific Computing Ltd., and David Pinchin of Keele
University for helpful discussions and advice.

References

The numbering scheme for these references is designed to
reflect the section of text in which they were first of interest.
Not all references are explicitly used in the text but may be
required for background reading or further information. The
numbering scheme is [section . sub section . reference number].
The list is neither complete nor up to date.

Some compilations of literature are available, one such, which
has been published under the auspices of the SERC/DTI Transputer
Initiative, and is available from the Sheffield National Transputer
Centre, is [R.I.1]. Other sources of useful information are the
Ocam Users' Group [R.I.2], the Edinburgh Supercomputer Centre
Newsletter [R.I.3], the Transputer Initiative Mailshot [R.I.4] and
ourselves [R.I.5].

[R.I.1] "The programming language FORTRAN" ANSI X3.9 (1978)
[R.I.3] B.Hansen "The programming language Concurrent Pascal"
single assignment language, reference manual" Lawrence Livermore
[R.I.5] Reference to OAP
[R.I.6] CSP
[R.I.7] "IEEE 1003.1 POSIX standard draft 13" IEEE Working
Group Technical Committee on Operating Systems of the IEEE Computer
Society (New York, 1988)
libraries for parallel processing" Parallel Computing 6 (1988)
255-6
Colloquium, 30th September- 2nd October 1987, Bonn, FRG" Parallel
Computing 7 (1988) 263-499
operating system (release 1.0)" Cornell Theory Center, New York
(1987)
passing Concurrent computers" Computer August (1988) 0-24
[R.I.12] Perihelion Software Limited, 24 Brewhouse Buildings,
Charlton Trading Estate, Shepton Mallet, Somerset, BA4 5SQ.
(1987)
[R.I.14] D.May EPL
Prentice-Hall (1988)
[R.I.16] Dick Fountain "A tutorial introduction to Ocam
programming" Blackwell Scientific Publications for Inmos Limited,
Bristol (1986)
International

[II.5.5] "JMS T800 Architecture" Inmos Limited, Bristol.

Technical r report 72-TCK-006

[II.5.6] Inmos Limited "Transputer Development System" Prentice


[II.5.7] Edinburgh Harpnesses, TIN etc.


[II.7.2] A.S. Tanenbaum "Operating Systems: Design and


[II.1.1] W. Smith, D. Fincham, A. Raine...

[II.1.2] "FORTRAN 8K features that assist in the exploitation

of parallelism" J. Reed, Harwell Laboratory

[II.1.3] L.M. Delves and N.G. Brown "SERC EMR, Numerical

Libraries for transputer arrays - final report" Liverpool University

(1988)

[II.1.4] "Ocmah Numerical Library Documentation" Liverpool

University (1988)


[III.2.1] "IPSC Concurrent Debugger" Intel manual, order no

310613-008B Intel scientific computers, Beaverton, Ore. (1987)

[III.2.2] W.M. Pan and V. Jackson "A Concurrent debugger DMCON

for IPSC/2 programs" Intel Scientific Computers, Beaverton, Ore.

(1987)

[IV.1.1] Liverpool ???

[IV.1.2] TRANSIM ???

[IV.1.3] Sorenson and Donapera "Schedule" Argonne National Lab


facility" IEEE Colloquium, IEEE Digest no 91 (1986)

[IV.1.5] T series FFS 860-0001-014A

[IV.1.6] S. Hawkinson "The FFS T Series, a parallel vector

supercomputer" FFS Inc., Beaverton, Ore. (1986)


FFS (UK) Ltd., Bracknell, Berkshire (1987)

[IV.2.1] IPSC/2 sales brochure, order number 10610-001 Intel

Scientific Corporation, Beaverton, Ore.

[IV.2.2] WCAtlas and CLSeitz " Multicomputers: Message-passing


[IV.2.3] P. Perlee "The NX/2 operating system" Intel Proc. 3rd

hypercube conference ACM (1988)

[IV.2.4] IPSC/2 "FORTRAN programmer's reference manual" Intel

Scientific Corporation, Beaverton, Ore.

[IV.2.5] Crystalline Operating System


Beaverton, Ore.


[IV.3.2] M. Surridge "A multi-transputer harness for 'farm'

parallelism using the FORTRANT programming language" Transputer

Support Centre, Southampton (1988)


migration aids for transputer systems" Dept. of Electronics and

Comp. Sci., Southampton (1988)

[IV.3.4] J. Reeve "A general communications harness for

transputer nets" Dept. of Electronics and Comp. Sci., Southampton

(1988)


R.2

R.3
### Appendix A. Timing of various FORTRAN calls on a variety of parallel computers.

<table>
<thead>
<tr>
<th></th>
<th>square root</th>
<th>exponential</th>
<th>logarithm</th>
<th>cosine</th>
<th>inverse cosine</th>
<th>sine</th>
<th>inverse sine</th>
<th>sum (v-v)**2</th>
<th>gaussian random numbers</th>
<th>vector swap</th>
<th>dot product</th>
<th>INT (v*scalar)+1</th>
<th>gather</th>
<th>scatter</th>
<th>max magnitude element</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Melko T6</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u-VAX/II FFS T20 scalar</td>
<td>0.84</td>
<td>0.26</td>
<td>0.84</td>
<td>0.48</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
</tr>
<tr>
<td>IPS/1 vector NCUBE</td>
<td>0.32</td>
<td>0.54</td>
<td>0.32</td>
<td>0.32</td>
<td>0.32</td>
<td>0.32</td>
<td>0.32</td>
<td>0.32</td>
<td>0.32</td>
<td>0.32</td>
<td>0.32</td>
<td>0.32</td>
<td>0.32</td>
<td>0.32</td>
<td>0.32</td>
</tr>
<tr>
<td><strong>Melko T4 MS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFS T20 vector</td>
<td>0.65</td>
<td>0.20</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
</tr>
<tr>
<td>IPS/2 SX</td>
<td>0.24</td>
<td>0.24</td>
<td>0.24</td>
<td>0.24</td>
<td>0.24</td>
<td>0.24</td>
<td>0.24</td>
<td>0.24</td>
<td>0.24</td>
<td>0.24</td>
<td>0.24</td>
<td>0.24</td>
<td>0.24</td>
<td>0.24</td>
<td>0.24</td>
</tr>
<tr>
<td><strong>Melko T8 MS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPS/1 vector</td>
<td>1.13</td>
<td>1.13</td>
<td>1.13</td>
<td>1.13</td>
<td>1.13</td>
<td>1.13</td>
<td>1.13</td>
<td>1.13</td>
<td>1.13</td>
<td>1.13</td>
<td>1.13</td>
<td>1.13</td>
<td>1.13</td>
<td>1.13</td>
<td>1.13</td>
</tr>
<tr>
<td><strong>ICHE T8</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPS/2 SX VX</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
</tr>
</tbody>
</table>

### RMS maths error

- 0.28111E-13
- 0.19629E-13

### timings for iteration length 50000

|                      |              |              |           |        |               |      |              |              |                       |             |             |                      |        |         |                     |
|----------------------|--------------|--------------|-----------|--------|---------------|------|--------------|--------------|------------------------|             |             |                      |        |         |                     |
| initialise one array | 0.51         | 0.32         | 0.24      | 0.24   | 0.24          | 0.24 | 0.24         | 0.24         | 0.24                   | 0.24       | 0.24       | 0.24                | 0.24   | 0.24   | 0.24               |
| initialise four array (1) | 0.20         | 0.17         | 0.20      | 0.20   | 0.20          | 0.20 | 0.20         | 0.20         | 0.20                   | 0.20       | 0.20       | 0.20                | 0.20   | 0.20   | 0.20               |
| initialise four array (2) | 0.30         | 0.045        | 2.60      | 2.60   | 2.60          | 2.60 | 2.60         | 2.60         | 2.60                   | 2.60       | 2.60       | 2.60                | 2.60   | 2.60   | 2.60               |
| IF test              | 0.28         | 1.55         | 0.36      | 0.36   | 0.36          | 0.36 | 0.36         | 0.36         | 0.36                   | 0.36       | 0.36       | 0.36                | 0.36   | 0.36   | 0.36               |
| V = V*V + V          | 0.2          | 0.614        | 0.23      | 0.23   | 0.23          | 0.23 | 0.23         | 0.23         | 0.23                   | 0.23       | 0.23       | 0.23                | 0.23   | 0.23   | 0.23               |
| 0.38                 | 2.27         | 6.20         | 0.53      | 0.53   | 0.53          | 0.53 | 0.53         | 0.53         | 0.53                   | 0.53       | 0.53       | 0.53                | 0.53   | 0.53   | 0.53               |
| call sub, 0 arguments | 0.74         | 0.68         | 0.68      | 0.68   | 0.68          | 0.68 | 0.68         | 0.68         | 0.68                   | 0.68       | 0.68       | 0.68                | 0.68   | 0.68   | 0.68               |
| call sub, 1 arguments | 0.72         | 1.13         | 0.90      | 0.90   | 0.90          | 0.90 | 0.90         | 0.90         | 0.90                   | 0.90       | 0.90       | 0.90                | 0.90   | 0.90   | 0.90               |
| call sub, 2 arguments | 0.80         | 0.92         | 0.92      | 0.92   | 0.92          | 0.92 | 0.92         | 0.92         | 0.92                   | 0.92       | 0.92       | 0.92                | 0.92   | 0.92   | 0.92               |
| call sub, 3 arguments | 0.88         | 1.03         | 1.00      | 1.00   | 1.00          | 1.00 | 1.00         | 1.00         | 1.00                   | 1.00       | 1.00       | 1.00                | 1.00   | 1.00   | 1.00               |
| call sub, 4 arguments | 1.00         | 1.13         | 1.10      | 1.10   | 1.10          | 1.10 | 1.10         | 1.10         | 1.10                   | 1.10       | 1.10       | 1.10                | 1.10   | 1.10   | 1.10               |
| scalar random numbers | 0.48         | 2.21         | 1.15      | 1.15   | 1.15          | 1.15 | 1.15         | 1.15         | 1.15                   | 1.15       | 1.15       | 1.15                | 1.15   | 1.15   | 1.15               |
| 1.76                 | 7.34         | 7.59         | 2.28      | 2.28   | 2.28          | 2.28 | 2.28         | 2.28         | 2.28                   | 2.28       | 2.28       | 2.28                | 2.28   | 2.28   | 2.28               |
| reciprocal           | 0.34         | 12.24        | 0.20      | 0.20   | 0.20          | 0.20 | 0.20         | 0.20         | 0.20                   | 0.20       | 0.20       | 0.20                | 0.20   | 0.20   | 0.20               |
| 0.23                 | 1.15         | 0.48         | 3.30      | 3.30   | 3.30          | 3.30 | 3.30         | 3.30         | 3.30                   | 3.30       | 3.30       | 3.30                | 3.30   | 3.30   | 3.30               |

A.1 A.2
1) The u-VAX clock is accurate to 0.01s, that of the Meiko and
time to 64ms and that of the IPSC/1 to 5ms. Thus the scaled
IPSC/1 times may be in error by as much as 0.3s

2) The IPSC/1 vector results were generated from the same
FORTRAN source as the scalar test using the VAST-2 vectoriser. FPS-T
series vector timings result from hard modifications using the
highest level routine available which would perform the task.

3) Both FPS-T series and Meiko mass store use 15MHz T414
transputers. The various Meiko times differ in transputer and memory
used. The mass-store (MS) board has an access time of 6 transputer
clocks, the compute boards have a 4 clock access.

4) NCUBE timing was produced at DL using an NCUBE-4 on loan
from Arrow Computers Swindon. An iteration count of 5500 was run
parallel on four MHz nodes. Times scaled to suit.

5) Meiko T800 times were generated on a compute board of the
Engineering Board Loan Pool M10 at DL, using FORTRAN code embedded
in the Fortnet harness. The compute boards have only 256k bytes
memory, so an iteration count of 2500 was used and results scaled by 20.

6) Intel IPSC/2 SX times were produced on the DL hypercube.

7) Appalling times on the FPS-T20 are due to poor mathematical
functions. This is because the intrinsic functions actually call the
C library in many cases and thereby introduce a large overhead.

8) If you want to generate big numbers use an FPS-T20 !

A listing of the program TIME, which was used to produce the
timing figures of this appendix, is given below. It was written by
R.J. Harrison, and should be useful on any 32-bit machine. Only the
scalar code is shown, vector code is produced either by hand coding
or by use of a vectorising compiler or precompiler such as VAST-2.

```
PROGRAM TEST
IMPLICIT REAL*8(A-H,O-Z)
PARAMETER(LEN=2550)
DIMENSION A(LEN),B(LEN),C(LEN),D(LEN),TA(LEN),
& T(30)
DATA T/30.000000/,
CALL cpu(START)
DO 10 I=1,LEN
   A(I)=0.000
10 CONTINUE
CALL cpu(FINISH)
T(J)=FINISH-START
C
CALL cpu(START)
DO 20 I=1,LEN
   A(I)=1.000
20 CONTINUE
CALL cpu(FINISH)
T(J)=FINISH-START
C
CALL cpu(START)
DO 30 I=1,LEN
   A(I)=0.000
30 CONTINUE
CALL cpu(FINISH)
T(J)=FINISH-START
C
CALL cpu(START)
DO 40 I=1,LEN
   IF(A(I).EQ.0.000) GOTO 40
   B(I)=1.000
40 CONTINUE
```

A3 A4
DO 90 I=1,LEN
  CALL SUB3(A(I),B(I),C(I))
90 CONTINUE
CALL cpu(FINISH)
  T(9)=FINISH-START
C
CALL cpu(START)
DO 100 I=1,LEN
  CALL SUB4(A(I),B(I),C(I),D(I))
100 CONTINUE
CALL cpu(FINISH)
  T(10)=FINISH-START
C
   ISEED=12345
CALL cpu(START)
DO 110 I=1,LEN
  CALL SRAND(ISEED,D(I))
110 CONTINUE
CALL cpu(FINISH)
  T(11)=FINISH-START
CALL SUB4(A,B,C,D)
C
CALL cpu(START)
DO 120 I=1,LEN
  A(I)=1.000/D(I)
120 CONTINUE
CALL cpu(FINISH)
  T(12)=FINISH-START
CALL SUB4(A,B,C,D)
C
CALL cpu(START)
DO 130 I=1,LEN
  B(I)=SORT(D(I))
130 CONTINUE
CALL cpu(FINISH)
  T(13)=FINISH-START
CALL SUB4(A,B,C,D)
C
CALL cpu(START)
DO 140 I=1,LEN
  C(I)=EXP(D(I))
140 CONTINUE
CALL cpu(FINISH)
  T(14)=FINISH-START
CALL SUB4(A,B,C,D)
C
CALL cpu(START)
DO 150 I=1,LEN
  A(I)=LOG(C(I))
150 CONTINUE
CALL cpu(FINISH)
  T(15)=FINISH-START
CALL SUB4(A,B,C,D)
C
CALL cpu(START)
DO 160 I=1,LEN
  C(I)=COS(A(I))
160 CONTINUE
CALL cpu(FINISH)
  T(16)=FINISH-START
C
CALL SUB4(A,B,C,D)

CALL cpu(START)
DO 170 I=1,LEN
  A(I)=ACOS(C(I))
170 CONTINUE
CALL cpu(FINISH)
  T(17)=FINISH-START
CALL SUB4(A,B,C,D)
C
CALL cpu(START)
DO 180 I=1,LEN
  C(I)=SIN(A(I))
180 CONTINUE
CALL cpu(FINISH)
  T(18)=FINISH-START
CALL SUB4(A,B,C,D)
C
CALL cpu(START)
DO 190 I=1,LEN
  A(I)=ACOS(C(I))
190 CONTINUE
CALL cpu(FINISH)
  T(19)=FINISH-START
CALL SUB4(A,B,C,D)
C
SUM=0.000
CALL cpu(START)
DO 200 I=1,LEN
  SUM=SUM+(A(I)-D(I))**2
200 CONTINUE
CALL cpu(FINISH)
  T(20)=FINISH-START
CALL SUB4(A,B,C,D)
SUM=SQR(SUM/FLOAT(LEN))
WRITE(6,'(6f16.10)') SUM
C
HALF=0.500
ONE=1.000
AMEAN=0.000
SDEV=1.000
CALL cpu(START)
DO 210 I=1,LEN
  CALL SRAND(ISEED,V1)
  CALL SRAND(ISEED,V2)
  V1 = -200(V1)
  V2 = -LOG(V2)
  X - V1 - ONE
  IF (V2.GE.HALF*X*X) GOTO 211
  CALL SRAND(ISEED,U)
  IF (U.GE.HALF) THEN
    C(I) = AMEAN - SDEV*V1
  ELSE
    C(I) = AMEAN + SDEV*V1
  ENDIF
210 CONTINUE
CALL cpu(FINISH)
  T(21)=FINISH-START
CALL SUB4(A,B,C,D)
C
CALL cpu(START)
DO 220 I=1,LEN
   TEMP=A(I)
   A(I)=B(I)
   B(I)=TEMP
220 CONTINUE
CALL cpu(FINISH)
T(22)=FINISH-START
CALL SUB4(A,B,C,D)
C
C SN=0.DO0
CALL cpu(START)
DO 230 J=1,LEN
   S=S + A(I)*C(J)
230 CONTINUE
CALL cpu(FINISH)
T(23)=FINISH-START
CALL SUB4(A,B,C,D)
CALL SUB1(S)
C
ZL=FLOAT(LEN)
CALL cpu(START)
DO 240 I=1,LEN
   IA(I)=INT(ZL*D(I)) + 1
240 CONTINUE
CALL cpu(FINISH)
T(24)=FINISH-START
CALL SUB1(IA)
CALL SUB4(A,B,C,D)
C
CALL cpu(START)
DO 250 I=1,LEN
   C(I)=A(IA(I))
250 CONTINUE
CALL cpu(FINISH)
T(25)=FINISH-START
CALL SUB4(A,B,C,D)
C
CALL cpu(START)
DO 260 I=1,LEN
   A(IA(I))=C(I)
260 CONTINUE
CALL cpu(FINISH)
T(26)=FINISH-START
CALL SUB4(A,B,C,D)
C
IZ=1
ZM=0.DO0
CALL cpu(START)
DO 270 I=1,LEN
   IF(ABS(ZM).GT.ABS(D(I))) THEN
      ZM=D(I)
      IZ=I
   END IF
270 CONTINUE
CALL cpu(FINISH)
T(27)=FINISH-START
CALL SUB2(ZM,IZ)
CALL SUB4(A,B,C,D)
C
WRITE(6,('"Timing for iteration length "',1x,1A))LEN
WRITE(6,('")'))
WRITE(6,('" Initialise one array "',1x,1A))T(1)
WRITE(6,('" Initialise four arrays (1) "',1x,1A))T(2)
WRITE(6,('" Initialise four arrays (2) "',1x,1A))T(3)
WRITE(6,('" If test "',1x,1A))T(4)
WRITE(6,('" V = V * V + V "',1x,1A))T(5)
WRITE(6,('" Call subroutine, 6 arguments "',1x,1A))T(6)
WRITE(6,('" Call subroutine, 1 arguments "',1x,1A))T(7)
WRITE(6,('" Call subroutine, 2 arguments "',1x,1A))T(8)
WRITE(6,('" Call subroutine, 3 arguments "',1x,1A))T(9)
WRITE(6,('" Call subroutine, 4 arguments "',1x,1A))T(10)
WRITE(6,('" Scalar random numbers "',1x,1A))T(11)
WRITE(6,('" Reciprocal "',1x,1A))T(12)
WRITE(6,('" Square root "',1x,1A))T(13)
WRITE(6,('" Exponential "',1x,1A))T(14)
WRITE(6,('" Logarithm "',1x,1A))T(15)
WRITE(6,('" Cosine "',1x,1A))T(16)
WRITE(6,('" Sine "',1x,1A))T(17)
WRITE(6,('" Inverse sine "',1x,1A))T(18)
WRITE(6,('" Sum (V-V)*2 "',1x,1A))T(19)
WRITE(6,('" Gaussian random no. s "',1x,1A))T(20)
WRITE(6,('" Vector swap "',1x,1A))T(21)
WRITE(6,('" Dot product "',1x,1A))T(22)
WRITE(6,('" INT(V-scale)+1 "',1x,1A))T(23)
WRITE(6,('" Gather "',1x,1A))T(24)
WRITE(6,('" Scatter "',1x,1A))T(25)
WRITE(6,('" Maximum magnitude element "',1x,1A))T(26)
END
C
SUBROUTINE SRAND(IS,C)
C
AM 16/10/81
C Should work on any 32 bit machine. Actual generator
C is only fair and not suitable for detailed work.
REAL*8 C.SCALE
INTEGER IS,IMULT,IMOD,IS1,IS2,IS82
5D-10 DATA IMULT/1807./IMOD/2147483647./SCALE/4.6566E28/
C IS = MOD(IS*16807,2**31-1).
IF(IS.EQ.0) IS = 1
ISX = MOD(IS,32768)
IS1 = (IS-IS2)/32768
IS2 = IS2 * IMULT
IS2 = MOD(IS2,32768)
IS1 = MOD(IS1*IMULT+(IS2-IS2)/32768,65536)
IS = MOD(IS1*32768-IS2,IMOD)
G = SCALE * FLOAT(IS)
END
SUBROUTINE SUBO
END
SUBROUTINE SUB1(X)
END
SUBROUTINE SUB2(X,Y)
END
SUBROUTINE SUB3(X,Y,Z)
END
Appendix B. CCTA Whetstone Benchmark

We have run the CCTA program foup12, which is the fully vectorisable double precision Whetstone benchmark, on a number of different single processors for comparison purposes. Results are given below. The parameters used in the runs were \( n_{v}=10, n_{v}n_{w}=16, n_{v}=1,2,4,8,16,32,65,96,128,129,192,256,257,384,512 \) for each of the processors.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Operation</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel 1PSC/2 SX</td>
<td>Total</td>
<td>1.4400</td>
<td>1.7732</td>
<td>1.7818</td>
<td>0.3679</td>
<td></td>
</tr>
<tr>
<td>IPSC/2 VX (f)</td>
<td>MFLOPS</td>
<td>435.359</td>
<td>0.1876</td>
<td>4.7201</td>
<td>6.9769</td>
<td>1.4980</td>
</tr>
<tr>
<td>Meiko T840</td>
<td></td>
<td>1899.710</td>
<td>0.9423</td>
<td>1.2424</td>
<td>1.2479</td>
<td>0.4463</td>
</tr>
<tr>
<td>Convex C2 -O1 (g)</td>
<td></td>
<td>7.5220</td>
<td>10.7062</td>
<td>10.8092</td>
<td>4.0253</td>
<td></td>
</tr>
<tr>
<td>Convex C2 -O2</td>
<td></td>
<td>35.440</td>
<td>3.6293</td>
<td>66.0030</td>
<td>71.7477</td>
<td>20.9120</td>
</tr>
</tbody>
</table>

- a) total time for run, in seconds
- b) MFLOPs vector length 1
- c) MFLOps vector length 64
- d) MFLOps vector length 512
- e) MFLOP performance in MFLOPS, vector length 512
- f) The Intel vector code was prepared using VAST2 and VecLib
- g) The following sequence of commands:

```
mv bench.f bench.v
vast2 -o bench.f bench.v
f77 -o -sx -vx bench.f
f77 -o bench.out bench.o -sx -vx -vec -node
getcube -t ivx
load bench.out
```

g) compilation option for the fc compiler is shown (May 1989)
Appendix C. Timing of FORTRAN communications and disk operations on parallel computers.

Disk and communications activity is hard to time, mainly due to interaction from system activity going on in parallel, either from other users or buffer activity, or other message passing in the system. I have tried to eliminate internal interaction with buffers and disk conflict by using the semaphores system mentioned in the text. Best times are given for an otherwise nearly empty machine.

Timing on the Melko is obtained using the Fortran v2.0 harness, and therefore will compare badly with similar processes on Occam-2. The 3L results should contain low overheads, as should the Melko CTools results.

Occam message passing is best for short messages, whereas the Intel is better for long messages, only one example of 80 kb messages is shown below. The former yields (15 + 1.2*nbytes) microseconds in Occam, whilst the latter is (2000 + 0.4*nbytes) microseconds in FORTRAN. Loading of an executable file from the SRM on the Intel is asymptotically (large file and large number of processors) 0.0065 secs per kbbyte per processor, this is more efficiently handled than the FORTRAN read.

Iteration length, dimension 20 10000. times in seconds
IPSC/2 sx Meiko Meiko Meiko Meiko
(Fortran) 3L 1 SUN CTools

(a) node initiated disk operations read or write len*8 bytes nloops times
node write to node 5.516
node read from node 5.187
node write to host 28.797 51.469 46.336
node read from host 28.563 45.437 35.434
host initiated disk operations read or write len*8 bytes nloops times
host write to host 3.410 57.882 45.653
host read from host 4.840 53.637 33.665
host write to node 0.340
host read from node 0.160
local memory operations copy b=c of len*8 bytes nloops*2 times
node memory copy 1.094 1.564 1.728
host memory copy 1.940 2.398 1.728
remote memory operations. send and receive a message packet of len*8 bytes nloops times
node 0 to node 1 1.688 10.185 9.360
node 0 to node 2 1.234 9.997 9.414
host to node 0 0.440 12.927 1.728
host to node 1 0.310 13.598 9.570

(a) I have used the CFSEMUL library to access the concurrent disk system on the Intel from the SRM host processor. This involves message passing and synchronisation. Times appear to be faster than for other means of access. This is probably because the optimised Intel c10 library calls, which use caching on cache nodes, are used to actually access the disk, and the overhead in message passing is minimal (about half of the round-trip times shown).

A fuller investigation of Intel communications, and its delinquency on message length, is given by L. Bouman and D. Roose. "Benchmarking the IPSC/2." Report TW114 (October 1988) Department of Computer Science, Katholische Universitat Leuven, Belgium. Asymptotic message rate is given by the manufacturers as 2.7 Mbyte/sec per DMA channel.
It is not clear why host-node operations are surprisingly fast.

(b) Timings were taken on the Meiko M10 at Daresbury with the Fortnet v2.1 harness. The host is the Fortnet Master process running on the mk021 board with a T414 integer transputer and 3-cycle access memory. The node processes run on the mk060 board with T800 transputers and twocycle access memory. Fortnet carries very large overheads in any kind of message passing as expected.

(c) Timings as above taken on the Meiko M60 at University of Bath. 11/7/89. The system is self hosted with MR060 boards having T800 processors running MMVCs and MelkOS.

(d) (e) times not yet available

The timings above were produced with the following program (Intel version shown).

```
program C T I M E
* timing of communications on parallel computers
* program tests host and node disk operations, synchronous
* communications, and memory copy operations
* test on 3/3/89
```

program ctime

```fortran
! version for Intel IPSC/2 with concurrent 1/o
! IMPLICIT REAL*(A-K,O-Z), INTEGER(*)
PARAMETER(LEN=10000,nloop=20)
DIMENSION A(LEN),B(LEN),C(LEN),D(LEN),IA(LEN),
& T(30)
character*2 ctype
caracter*8 cname
DATA T/'300.000D0'/
data ctype,cname,'/','ftime '/
open(2, file='ctime0.dat', stat='unknown')
c allocate cube and kill when this process finishes
call getcube(cname,ctype, '/', '0')
pid=0
call setpid(pid)
nnode=numnodes()
c load main elfongs packages
call load('ctime.out',-1,pid)
c load distributed file system handler driver on node 0 with pid=1
call load('server.out',nnode,1-1)
c write(6, '/)
c wait for node0 to start, assures that load has completed
call semain(100)
call cpu(START)
c binary write to host file ctime0.dat on unit 2
do 5 i=1,nloop
write(2,a)
5 continue
call cpu(START)
t(t)=finish-start
rewind(2)
call cpu(START)
do 10 i=1,nloop
```

C.1
read(2)b
continue
CALL cpu(finish)
t(2)=finish-start
copy operation
CALL cpu(start)
do 25 j=1,nloop*2
   do 25 i=1,len
      c(i)=b(i)
   25 continue
CALL cpu(finish)
t(5)=finish-start
c signal finished using SRM, this process can wait
CALL semout(1)
c wait for clearance to continue, SRM disk free
CALL semain(2)
c open cfs file defined in file server
CALL open(9)
CALL cpu(start)
c write to cfs file via file server
DO 15 i=1,nloop
   CALL write(3,a,len*8)
15 continue
CALL cpu(finish)
t(3)=finish-start
CALL rewind(3)
CALL cpu(start)
do 20 j=1,nloop
   CALL read(3,b,len*8)
20 continue
CALL cpu(finish)
t(4)=finish-start
c signal clearance to continue
CALL semain(3)
c send and receive message to node 0, timing their and back
CALL cpu(start)
do 35 j=1,nloop
   CALL csend(35,b,j*len,0,0)
   CALL crecv(35,b,j*len)
35 continue
CALL cpu(finish)
t(6)=finish-start
CALL cpu(start)
c same for node 1
do 36 i=1,nloop
   CALL csend(36,b,i*len,1,0)
   CALL crecv(36,c,i*len)
36 continue
CALL cpu(finish)
t(7)=finish-start
c write results at end so that no buffer processes impede
   c execution
WRITE(6,'(''Iteration length, dimension '',2i8)')
   1 nloop,len
   WRITE(6,1) ' host write to host ',t(1)
   1 FORMAT(1x,a,9.12,5)
   WRITE(6,1) ' host read from host ',t(2)
   WRITE(6,1) ' write to node from host ',t(3)
   WRITE(6,1) ' read from node on host ',t(4)
   WRITE(6,1) ' host to host memory copy ',t(5)
   WRITE(6,1) ' host to node 0 send and receive ',t(6)
   WRITE(6,1) ' host to node 1 send and receive ',t(7)
c wait for node to finish before shutting everything down
CALL semain(4)
end
c subroutine cpu(time)
real*8 time
integer stime,mclock
time=float(mclock())/1000.0
end
c subroutine semout(itype)
c node process to synchronise with host
common/talk/inode,nnode,host
CALL csend(itype,idum,0,0,0)
end
c subroutine semain(itype)
c node process to synchronise with host
common/talk/inode,nnode,host
CALL crecv(itype,idum,0)
end
********************************************************************
c C F S E M U L
c this library contains a set of routines used to emulate cio
c on the Intel IPSC/2 cfs from the front-end arm or remote host
c it is an example of their use look at the elkonos program
main0.f and server.f. In all cases the program server.f must run in
c parallel on one of the compute nodes of the system to receive fillin
c system requests from the host. Although this emulation is slow
c optimised routine calls have been used wherever possible
c r.a.
15/6/89
********************************************************************
c subroutine open(lu)
common/talk/inode,nnode,host
CALL csend(334,lu,4,nnode-1,1)
c handshaking
CALL crecv(334,du,0)
end
c subroutine rewind(lu)
common/talk/inode,nnode,host
CALL csend(331,lu,4,nnode-1,1)
c handshaking
CALL crecv(331,du,0)
end
c subroutine read(lu,buffer,nbytes)
dimension buffer(*)
common/talk/inode,nnode,host
iproc=nnode-1
CALL csend(333,lu,4,iproc-1)
CALL csend(333,nbytes,4,iproc-1)
CALL crecv(333,buffer,nbytes)
c no handshaking required
c subroutine write(1u, buffer, nbytes)
common/talk/inode, nnode, host
dimension buffer(*)
inproc=nnode-1
call csend(332, lu, 4, inproc, 1)
call csend(323, nbytes, 4, inproc, 1)
call csend(332, buffer, nbytes, inproc, 1)
c handshake
    call crecv(33r, dum, 0)
end c

r subroutine close(lu)
common/talk/inode, nnode, host
call csend(335, lu, 4, nnode-1)
call crecv(335, dum, 0)
end c

r subroutine copy(1dim, imax, jmax, ain, buffer)
dimension ain(1dim, *)
dimension buffer(*)
k=0
    do 5 j=1, jmax
      do 5 i=1, imax
        k=k+1
        buffer(k)=ain(i, j)
      end do
      k=0
  5  continue
end subroutine copy

r fill up buffer for read/write emulation routines for 4-byte variables
real*8 ain(1dim, *)
real*8 buffer(*)
k=0
    do 5 j=1, jmax
      do 5 i=1, imax
        k=k+1
        buffer(k)=ain(i, j)
      end do
      k=0
  5  continue
end subroutine copy

r fill up buffer for read/write emulation routines for 4-byte variables
dimension ain(1dim, *)
dimension buffer(*)
k=0
    do 5 j=1, jmax
      do 5 i=1, imax
        k=k+1
        ain(i, j)=buffer(k)
      end do
      k=0
  5  continue
end subroutine write

k=3
do 5 j=1, jmax
  do 5 i=1, imax
    k=k+1
    ain(i, j)=buffer(k)
  end do
  5  continue
end program ctime

r node program for Intel iPSC/2 with cfs
IMPLICIT REAL=(A-H,O-Z), INTEGER=(I-N)
PARAMETER(LEN=10000, nloop=20)
DIMENSION A(LEN), B(LEN), C(LEN), D(LEN), IA(LEN),
       T(30)
DATA T(30)*0.000/commun/talk/inode, nnode, host
inode=myinode()
host=myhost()
if(inode.eq.0)then
  open(2, file='cfs/2a-jain/ctime.dat', status='unknown')
  open(3, file='user/user/rja/tests/comms/ctime.dat',
       status='unknown')
c signal to host that node0 has started
    call semout(100)
c call cpu(start)
c write to cfs
    do 5 i=1, nloop
      write(2, a)
  5  continue
    call cpu(start)
t(1)=finish-start
  10  continue
    call cpu(start)
t(2)=finish-start
    call cpu(start)
c local node to node memory copy
    do 25 j=1, nloop
     do 25 i=1, imax
       c(i)=b(i)
  25  continue
    call cpu(start)
t(5)=finish-start
  c wait for clearance to use SRM
    call semain(i)
call cpu(start)
    do 15 i=1, nloop
      write(3, a)
  15  continue
    call cpu(start)
t(3)=finish-start
    call cpu(start)
    do 20 j=1, nloop
      read(3, b)
  20  continue
    call cpu(start)
c signal finished with SRM
call semout(2)
end if

in c test coms between nodes, sending message their and back again (as
my previous example in occam !)
call cpu(start)
do 30 i=1,nloop
   if(inode,eq.0) then
      call csend(30,b,8*len,1,0)
call crecv(30,c,8*len)
   else if(inode,eq.1) then
      call csend(30,b,8*len)
call crecv(30,c,8*len)
call csend(30,b,8*len,0,0)
   end if
end if
30 continue
call cpu(finish)
t(6)=finish-start
call cpu(start)
do 31 i=1,nloop
   if(inode,eq.0) then
      call csend(31,b,8*len,2,0)
call crecv(31,c,8*len)
   else if(inode,eq.1) then
      call csend(31,b,8*len)
call crecv(31,c,8*len)
call csend(31,b,8*len,0,0)
   end if
end if
31 continue
call cpu(finish)
t(7)=finish-start
c wait for signal to continue sending via host system process
call semain(3)
do 35 i=1,nloop
   if(inode,eq.0) then
      call crecv(35,c,8*len)
call csend(35,b,8*len,host,0)
   else if(inode,eq.1) then
      call csend(35,b,8*len)
call crecv(35,c,8*len,host,0)
   end if
end if
35 continue
if(inode,eq.0) then
   write(6,1)' node write to node ',t(1)
1 FORMAT(1X,A,12.5)
   write(6,1)' node read from node ',t(2)
write(6,1)' node read from host ',t(4)
write(6,1)' local node to node message copy ',t(5)
write(6,1)' node 0 to node 1 send and receive ',t(6)
write(6,1)' node 0 to node 1 send and receive ',t(7)
end if

c signal end of execution
call semout(4)
end c

subroutine semout(iotype)
c node process to synchronise with host
common/talk/inode,nnode,ihost
if(inode,eq.0)
der !c handshaking signal to make sure messages are received in correct or
call csend(iotype.dum.0.idhost.0)
goto 10

//
c 1   write(6,(" request to rewind a file "))
1 continue
   call crecv(iotype.lu.4)
   ipoint=laek(iu.0.0)
c  move file pointer
   call csend(iotype.dum.0.idhost.0)
goto 10

//
c 3   open
//
c 4   write(6,(" request to open file "))
4 continue
   call crecv(iotype.lu.4)
   lfile=laek(iu.0.0)
if(lfile.eq.3)then
   open(iu.0.0, file='cfs/rja/cftime0.dat', status='unknown',
        i  form='unformatted')
else
   write(6,("  error server  unknown file'"))
   end if
call csend(iotype.dum.0.idhost.0)
goto 10

//
c 5   close

//
c 5   write(6,(" request to close a file "))
5 continue
   call crecv(iotype.lu.4)
   close(iu.0.0)
call csend(iotype.dum.0.idhost.0)
goto 10

//
end
IV.2, IV.5, A, B

PSG see Intel
join III.1.1
kernel II, II.1, II.3, II.4,
III.1.2, IV.1, IV.5
libraries III.1.2
lightweight processes II.6, IV.7
LinCube library III.1.3
load I, III.1.1, III.1.2
load balance I, III.1.3
master III.1.1, IV.3
Melko II.5, IV.3
MelkOS IV.3
Messages I, II.2, II.3, II.4
MIMD II, III.1.1, III.1.3
MNWCS IV.3
NAG library I, III.1.3
NCUBE II.1, II.5, IV.4, A
MICHE II.1, II.2, II.5, III.1.3,
IV.5, A
NX/2 IV.2
Occam I, II.5, III.1.2, IV.3,
IV.5, C
OPS II.5, IV.3
PAR II.5
ParsiPal III.3.1
Parxys II.1, IV.6
PEACE II.3
Perihelion II.4
pid I, II.3, IV.5
pipelines III.1.1
ports II.7, IV.7
POSIX 1, II.1
PRE II.1, IV.5
procsem II, II.2, II.3, II.4
protocol II, II.3, IV.2
queue II
Schedule III.3, IV.3
sdb III.5
segment III.1.1, IV.3
semaphore II.7, III.4, IV.7
SEQ II.5
server II, II.1, II.3
shared memory, resources I, II.6, II.7, III.1.1
SIMD I
simulation III.3, IV.2
SISAL I
skelton III.1.1, III.1.2
slave III.1.1, IV.3
sockets III.1
sorting III.1.1, IV.3
spawn I, II.2, II.3
SPLIB II, II.3
SUN Cutils II.1, IV.3
Supernode IV.6
SUPRENUM II.5, III.1.2, IV.2
synchronous I, II. IV.3
systolic algorithm III.1.1
task II.4
tds II.5, IV.3

thread I, II.6, IV.7
Topology II.2
topology I, III.1.1, IV.1
trace II, IV.3, IV.2
TRAM IV.3
TRANSIM III.3.1
transputer II.5, IV.1, IV.3,
IV.6, IV.6, IV.7, A, B
Transtech see Miche
Trillium, Trollius II.1, II.2,
IV.5, see Miche, FPS
UNIX I, II, II.1, II.2, II.3,
II.4, IV.2, IV.4, IV.6
VAX IV.3, A
VERTEK II.1, IV.4
X-windows II.4
zL III.6, IV.3, IV.6, IV.7, B