con.

DL/SCI/TM61T

technical memorandum Daresbury Laborator

DL/SCI/TM61T

FORTRAN 77 PROGRAMMING OF PARALLEL COMPUTERS

by

R.J. ALLAN, SERC Daresbury Laboratory

LENDING

NOVEMBER, 1989

89/488

Science and Engineering Research Council
DARESBURY LABORATORY
Daresbury, Warrington WA4 4AD

ISSN 0144-5677

IMPORTANT

The SERC does not accept any responsibility for loss or damage arising from the use of information contained in any of its reports or in any communication about its tests or investigations.

FORTRAN77 Programming of Parallel Computers

R.J.Allan

Advanced Research Computing Group
Daresbury Laboratory
S.E.R.C.
Daresbury
Warrington, WA4 4AD
U.K.

29/6/89

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 algorithm 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
- Trollius (Trillium, Transtech Niche etc.)
- CE/RK (Cosmic Cube, Intel iPSC etc.)
- Helios (Perihelion, Atari etc.)
- Occam TDS (Inmos, Meiko, Transtech, Parsys etc.)
- Threads and Lightweight Processes (3L etc.)
- Semaphores 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)
- NCUBE
- Transtech NTP1000
- Parsys SN1000
- 3L Parallel FORTRAN

References

Appendices
- Timing intrinsic FORTRAN routines
- Whetstone Benchmark
- Communications Benchmark

I Introduction

Some of the material of this review was presented in the TCS Division seminar series on 3rd october 1988, and is intended in part as an introductory guide to programming the parallel computers at Daresbury Laboratory (DL) in the FORTRAN77 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 collegues. Many statements in the text are my personal views and are not considered to represent ARCG 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 F77 as opposed to C or the new parallel syntaxes (Occam [II.5.2], Ada [2], Concurrent Pascal [3], SISAL [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 F77 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 M interconnected sequential F77 programs) 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 [5]). Tasks may be doing 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 III.1. No consideration is given to the the debate over hybrid or distributed-memory machines, 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 correct 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, essential to a MIMD operation. While the argument that this causes load imbalance and therefore waste of resources is valid, it is no worse in practise than not fully utilising a vector processor in sequential code. Some redundancy of operation may indeed be an aid to concocting 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.

More general data transfer in messages 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 filing system is available) to reduce program complexity i.e. reduce calls to message passing routines. A file system must therefore be provided, possibly through UNIX servers or other processes operating concurrently and invisible to the programmer. All these features must be callable as F77 extensions. 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 special 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 similar in nature to those available on vector 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 software, and many strategies can be conceived and must be tested in practise. 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 existing on parallel machines is provided. 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 point knowledge of the hardware is required (leaving aside the fine-tuning issues). In a MIMD computer 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 buildup. The second approach is to signify an error to the sending process and block it until the message is required.

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-correcting scheme with handshaking protocol to account for both "lost" messages and to resolve blocking. Program errors are assumed to be the cause of one class of lost messages and are flagged immediately as such in a run time trace.

A description of parallel communications on the CRAY, IBM, ETA and FPS-T20 and in the SPLIB (standard parallel library) is given by David Snelling 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 [I.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 it difficult for pipes to be maintained 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 a buffer, 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 over a network. A client-server model is often used, and many UNIX facilities are available as servers, to which a process connects by a socket, albeit imbedded in another system library call. The three types of socket are, firstly a stream socket which provides bidirectional, reliable, sequenced, unduplicated data flow with no record boundaries, and is 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 whilst servers listen and accept connections before providing a facility.

Synopsis of UNIX ipc calls.

Establishing and using synchronous blocked sockets.

s=socket(domain, type, protocol) -- create a socket, in this call values of domain of 'AF_NET' 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

protocol=protobyname(tcp) -- may be used to get a protocol number.

bind(s, name, namelen) -- assign a name to a socket

connect(s, server, serverlen) -- rendezvous with process server using a streams socket

listen(s, n) -- wait for up to n connection requests from clients

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

write(s, buf, buflen) -- write to socket s with established connection to a streams socket

read(s, buf, buflen) -- read from socket s

send(s, buf, buflen, flags) -- send to socket s, like write

recv(s, buf, buflen, flags) -- receive from socket s, like read

shutdown(s, how) -- shut down connection prior to closing

close(s) -- discard a socket

sendto(s, buf, buflen, flags, to, tolen) -- send data to a datagram socket

recvfrom(s, buf, buflen, flags, from, fromlen) -- receive data from a datagram socket

select(nfds, readfds, writefds, execptfds, 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(...) --

getnetbyname(...) -- as above for networks

getnetbynumber(...) --

getnetent(...) --

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

getprotobynumber(...) --

getprotoent(...) --

getservbyname(...) -- for servers

getservbyport(...) --

getservent(...) --

Miscellaneous calls:

bcomp(s1, s2, n) -- compare byte-strings, 0 if same, <>0 otherwise

bcopy(s1, s2, n) -- copy n bytes from s1 to s2

bzero(base, n) -- fill n bytes from base with zeros

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

htons(val) -- convert short 16b from host to net byte order

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

ntohs(val) -- convert short 16b 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 processors 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 NiCHE PRE system which also uses UNIX with the Trillium ipc layers for underlying communications servers (see next section) and by Meiko 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 VERTEX. The Intel hypercube has implemented the NX/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 Trollius (lately called trillium)

Both trillium and trollius 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 hosted by a Gould UTX-32 and other connected UNIX-based machines on a network. This explains its use on the Transtech NT1000 platform developed by NiCHE Technology with its transputer array and SUN 3.0 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.5.

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 programmer. 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, by 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 so 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 ABAQ system. Perihelion also produce their own transputer cards, and Helios can be run on other systems such as the Transtech NTP1000, Parytek, Meiko 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 as an embedded system (c.f. section IV.7). 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 VII [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 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 assignes 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

v := e -- assign the result of evaluating expression e to variable v

c ! e -- output the result of evaluating expression e to channel c

c ? v -- input variable v 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 buffered in memory so that data is not lost as process 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 hardwire 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 so 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 a list of items, such as processes belonging to a 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 commonly used by C programmers, such as printf, putchar, getchar, lineptr, strcpy, sin, cos, etc.

Apart from this brief overview details of the Occam-2 programming language cannot be described here. Many books are now available which do that in easily readable form, some are listed in the references at the end of this document [2-5].

Occam Programming System and Folding Editor

The Occam Programming System OPS (MEiKO) and Transputer Development System TDS (Inmos) [6] are Occam environments which work through a folding editor. The environment (I shall refer to it generically as TDS) may be used to edit, compile, configure, extract, load and run programs on a transputer array. Editing, compilation and configuring are done on the host processor (or local host), and access is provided to the filing system on the front end or dedicated scsi disk (MeikOS). Files belonging to the program source and object code structure are automatically maintained by the editor.

The editor interface is based on a concept called "folding". The folding operations allow the text currently being entered to be given a hierarchical structure ("fold structure") which reflects the structure of the program under development. A section of code can be hidden from view of the current editor window by placing it in a fold which is then closed. Three dots "..." appear on the screen to remind one of its presence. Folds within folds are possible and, whilst the editor is good for on-line editing, listing of programs and reading listed programs is difficult - Inmos seem to have implicitly admitted this defect since they produce few listing tools.

One fold of the editor screen normally contains a set of Occam utilities. These comprise a compiler, a terminal emulator (so that your program can talk to the screen and keyboard) and wiring utilities to control the link switches between the transputers. An editor screen on the Meiko might therefore contain:

```
{{{F /home/rja/export/toplevel.top
{{{F /utilLib/util4Lib.lib
...F Occam Compiler T4 and T800
...F Terminal emulator

...F Global search and replace
...F Library compacter

...F Explicit configuration tool
...F Implicit configuration tool
...F Load wiring file
...F Test switch chips
...F Test switch chips: remote node

...F source/termemul.fex
}}}

... PROGRAM Program.tsr -- user's program
}}}
```

Utilities are accessed by moving the cur or 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 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 in use to 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 NCUBE, or indeed with transputer arrays like the Transtech NTP1000 and Meiko CStools system have more sophisticated operating systems.

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

II.6 Threads and lightweight processes

The ability to produce or spawn new concurrent processes has already been mentioned in the discussion of CE/RK and is also available in the Ncube 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 parallelisation 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 data. Ports are used by 3L and by Meiko, 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 useage 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.

Summary

Aspects of only a few operating systems have been mentioned for machines which were evaluated at Daresbury during 1988. One further example which is of some interest is the SUPRENUM node operating system PEACE (program execution and connection environment). SUPRENUM, which may later be combined with the Europeen ESPRIT P1085 Supernode [1], is a german project [II.1.2] which involves coupling together 256 subarrays of processors in clusters of 16 with a 200 Mbyte/s bus network arranged in a 16x16 grid. The operating idea is again to have a number of layers (ten), written in C, from i/o and message servers up to file servers and program loaders. A high-performance kernel is implemented (the nucleus) which provides for basic inter-process communication and cooperation, and all other operations are implemented as dedicated server proceses accessed by remote procedure calls. Another example is Snelling's personalised library SPLIB [II.1.1].

It is clear that there are now really two types of operating system for parallel computers: UNIX and Occam-TDS with of course some variations such as the 3L system. TDS is a beautiful concept as the full-screen folding editor facility provides easy-to-use menus and an excellent modular programming and filing system. It does not however at present provide sufficient separation between the user and the hardware, and will therefore be labelled "unfriendly" by anyone not enamoured with the subtlety of concurrent processing per se. UNIX on the other hand is a familiar, but old fashioned, environment that provides a sufficiently good line-mode interface to allow exploitation of large-scale parallel resources. Many screen window tools permit extended use of UNIX (see [II.4.2] for instance).

In none of the systems described here can loaded processes migrate between nodes. This is because of the requirement for the loader to establish simple bindings for communication which last the lifetime of the process. Future research may change this restriction.

A further broad distinction is between operating systems which retain "control" of the loaded tasks, and minimal embedded systems in which the task is self-sustaining and the operating system must be booted again when it terminates. The latter approach is used where memory is at a premium, or the process rarely terminates - such as in industrial applications like flight simulators or robot controllers.

It is impossible to predict what lies in the future. It seems however that UNIX is here to stay, more foreign language tools will be needed for occam to survive. It also seems that there is a tendency to make parallel languages and algorithms look sequential as will be seen in section III.1.3, by writing only host code for an attached array controlled by sophisticated subroutines. Its more fun at present to program the array itself, and after all someone has to provide the routines for the next decade of science. The more tools available to do this job the better. New tools to emulate shared-memory operations on distributed-memory machines will aid porting of programs.

Some scientists dream of unlimited resources and automatically parallelising compilers. These do not exist. If you want to use a parallel computer today you have to get your hands dirty; you can even build the machine yourself using transputers.

The next two sections will indicate how a program can be built from simple ideas and simple extensions to existing sequential FORTRAN.

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 FORTRAN8X 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 classed 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 problem, 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-type algorithms which might be of 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 be run many times with a stack of input data, different data being sent to each occurence 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 them 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 processors are doing the same task so it doesn't matter to which one the next batch of data is sent.

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 procede 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 (i)

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 FIFO buffer to be retrieved and displayed only during cycles (ii) and (iii) along with the other data to which it belongs.

host program

```
      SUBROUTINE DIFFXS
      ...
c workspace for header information
      character*132 head(50)
counter for number of headers per task
      dimension ihead(nloop)
      common/talk/inode,nnode,ihost
      khead = 1
      ihead(1)=1
      READ(5,...
      call crecv(-1,ns,4)
C LOOP OVER COLLISION PROCESSES
      ...
c start of sending loop, phase (i)
      iproc=0
      nsent=1
      nrecvd=1
 70   READ(5,...
      IF(finished)goto 30
c send data to idle processor
      call csend(1,nsent,4,iproc,0)
      ...
c internal write of header information to be later output with
c results
      write(head(khead),...
      khead=khead+1
 210  VMU=V*AMASS*1836.1
      ...
      if(wnorm.lt.0.99d0)goto 70
      nsent=nsent+1
      iproc=iproc+1
c starting point in vector head for first header of next task
      ihead(nsent)=khead
      if(iproc.ge.nnode)iproc=0
c receive cross sections back in correct order, phase (ii)
 86   iret=iprobe(nrecvd)
      if(iret.ne.1)then
      ...
      goto 70
      end if
```

```
      call crecv(nrecvd,bmpt2,mang*8)
      nrecvd=nrecvd+1
      call crecv(nrecvd,bmpt,mang*8)
      nrecvd=nrecvd+1
c print header information and then results from this proc
      k=nrecvd/2
      do 50 i=ihead(k),ihead(k+1)-1
      write(6,'(a132)')head(i)
 50   continue
      ...
      GOTO 86
 30   CONTINUE
c receive remaining results phase (iii)
      nsent=nsent-1
      if(nrecvd.le.nsent*2-1)then
      call crecv(nrecvd,bmpt2,mang*8)
      nrecvd=nrecvd+1
      call crecv(nrecvd,bmpt,mang*8)
      nrecvd=nrecvd+1
c print header information and then results
      k=nrecvd/2
      do 60 i=ihead(k),ihead(k+1)-1
      write(6,'(a132)')head(i)
 60   continue
      ...
      nsent=nsent+1
      goto 30
      end if
c shut down nodes
      do 31 i=0,nnode-1
      call csend(1,nsent,4,i,0)
      ...
 31   continue
      END
```

node program

```
      SUBROUTINE DIFFXS
      ...
      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 crecv(1,nsent,4)
      nretn=nsent*2-1
      call crecv(1,k0,4)
      call crecv(1,kf,4)
      if(k0.eq.0.and.kf.eq.0)return
      call crecv(1,mde,4)
      call crecv(1,weight,8)
 71   DO 25 I=NRO+1,NRO+NRO1
      nbytes=8*(2+ns+ns*ns)
      read(2)...
c compute intensive routine
      CALL EIKON(ZA,ZB,ZETI,ZETJ,RPHAS,V,VMU,MANG,ANG,MDE,JBP,BMP)
      ...
      IF(WTOT.LT.0.99D0)GOTO 95
      call csend(nretn,bmpt2,mang*8,ihost,0)
      nretn=nretn+1
      call csend(nretn,bmpt,mang*8,ihost,0)
      ...
 95   rewind(2)
      ...
      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 indeces and are able to execute synchronously with the host.

```
     host program

     ...
     knode=numnodes-1
     kount=3
     lword-4
     do 999 je=1,ng2
     ...
     do 777 jj=1,ng1
     ...
     do 666 ii=1,nan1
     ...
     do 5 j=1,ng
     kount=kount+1
     iproc=mod(kount,knode)+2
     call check(iproc)
     call receve(iproc,nan*lword,dsdo)
     ...
5    ...
666  ...
777  ...

999  ...
     end

     node program

     ...
     inode=mynode
     ...
c start of loops here
     ...
     if(inode.eq.mod(kount,knode)+2)then
     do 1 i=1,nan
     ...
1    continue
     call wait(1)
     call send(1,nan*lword,dsdo)
     end if
5    continue
666  continue
```

```
777  continue
999  continue
     end
```

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 a considerably more complicated operation. A number of tasks are joined one after the other, 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 algorithm we have met (and the simplest) which explicitly involves inter-node communications, albeit to nearest neighbours. Many extensions of this idea are possible, such as nearest-neighbour 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 object 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 neighbour 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 ameanable 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 N 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 load balance - we shall not do that here. The method consists of subdividing the data 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 result.

This might be achieved using fork and join procedures if such were available. I give an example, but untested, program 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**0 away, which receives it and merges it with its own data. Then send the data to the neighbour 2**i away and merge, and iterate from i=1,n-1. 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 that on task is put on a single processor, indeed it might be 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 ! Fortnet-like coding syntax
c node program of merge/sort routine, recursive binary tree
      dimension a(enormous),b(enormous)
      integer seg,seg0,segn
      common/talk/inode,npend(10),nw,nwait
      data ndim/power/
      lword=4
      nnode=2**ndim ! number of nodes
      call receve(0,lword,seg)
      call receve(0,seg*lword,a(1)) ! get share of data from host
      call sort(seg,a(1)) ! sort this data
c now merge first to left nearest neighbour, then to p-2**n;
c n=0,ndim
      do 5 i=0,ndim-1
      n=2**i
      segn=0
      if(inode.le.nnode-n)then
      call receve(inode+n,lword,segn)
      call receve(inode+n,segn*lword,a(seg+1))
      end if
      if(inode.gt.n)then
      call send(inode-n,lword,seg)
      call send(inode-n,seg*lword,a(1))
      end if
c merge/sort two vectors into b
      j=1; k=seg+1; l=0
 10   if(a(j).eq.a(k))then
      l=l+1; b(l)=a(j); j=j+1
      l=l+1; b(l)=a(k); k=k+1
      else if(a(j).lt.a(k))then
      l=l+1; b(l)=a(j); j=j+1
      else if(a(k).lt.a(j))then
      l=l+1; b(l)=a(k); k=k+1
      end if
      if(j.lt.seg.and.k.lt.seg+segn)goto 10
      if(j.lt.seg)then
      do 15 m=j,seg
      l=l+1
 15   b(l)=a(m)
      else if(k.lt.seg+segn)
      do 20 m=k,seg+segn
      l=l+1
 20   b(l)=a(m)
      end if
      seg=seg+segn
c all used, do next dimension
      do 25 i=1,seg
 25   a(i)=b(i)
 5    continue
c all merged, send from proc1 to host
      if(inode.eq.1)call send(0,seg*lword,a)
c this is real parallel programming isn't it!
      end
```

5) General algorithms.

Simple sequential processes were 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 vector processors many of these operations were carried out at once in SIMD fashion. All 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 discs 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 algorithm 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 array 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 the, now well known, fact that to achieve linear speedup on parallel machines it is necessary to scale up the problem rather than just subdividing it. A further discussion of such implicit shared memory emulation 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 following program that uses the entire computing resouce of an n-processor machine to transpose a large matrix, of which seg(i) elements are stored on each node i in contiguous fashion.

```
      subroutine mtrans(idim,a)
      parameter (nnode=nprocs)
      integer seg,procs,offs
      dimension a(*)
      common/proclist/procs(nnode),offs(nnode),seg(nnode)
      common/talk/inode,npend(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,j,i,2)
      if(inode.eq.procs(1).or.inode.eq.procs(2))then
      procs(3)=procs(1)
      call vcopy(1,a(offs(2)),1,b,1,2,3,equal)
      call vcopy(1,a(offs(1)),1,a(offs(2)),1,1,2,equal)
      call vcopy(1,b,1,a(offs(1)),1,3,1,equal)
      end if
      end do
      end do
      end

******************************************************************
*                     F O R T N E T   V 2.0                      *
* IOSUP parallel communicating subroutines                       *
******************************************************************
      subroutine use(idim,i,j,n)
c for a square array (idim,idim) return in procs(n) the number of
c processor on which element (i,j) is stored and its local offset
c in offs(n)
      parameter (nnode=nprocs)
      integer seg, seg0, procs, offs, offset
      common/proclist/procs(nnode),offs(nnode),seg(nnode)
      common/talk/inode,npend(10),nw,nwait
      offset=i+(j-1)*idim
      seg0=0
      do i=1,nnode
      if(offset.gt.seg0.and.offset.le.seg0+seg(i))goto 5
      seg0=seg0+seg(i)
      end do
 5    procs(n)=i
      offs(n)=offset-seg0
      end

      subroutine vcopy(nel,b,nb,a,na,npb,npa,func)
c modification of VecLib routine vcopy for distributed operation.
c sets a=func(b)
      parameter (nnode=nprocs)
      integer seg, seg0, procs, offs, offset
      common/proclist/procs(nnode),offs(nnode),seg(nnode)
      common/talk/inode,noend(10),nw,nwait
      lword=4
      if(procs(npb).eq.inode.and.procs(npa).eq.inode)then
      j=0
      do i=1,nel,na
      j=j+nb
      call func(a(i),b(j))
      end do
      else if(procs(npb).eq.inode)then
      do i=1,nel,na
      call receve(procs(npa),lword,temp)
      call func(a(i),temp)
      end do
      else if(procs(npa).eq.inode)then
      do i=1,nel,nb
      call send(procs(npb),lword,b(i))
      end do
      end if
      end

      subroutine equal(a,b)
      a=b
      end
```

The functionality of the above code looks sequential at first sight. Concurrency is however possible through the selective nature of the loop in mtrans. This is obviously true in the instance where a(j,i) and a(i,j) are both contained in the same segment, and vcopy will then just equate b=a(j,i) without any message passing. In the case where message passing is required the other processor must cooperate too, so both processors must execute the vector calls in order to synchronise. This does not dismiss the possibility that some other processors will be doing the same thing for a different selection of i and j.

Vcopy is an example of a "skeleton" routine since it can carry out any operation using the external function func. This might for example, in a more meaningful case, actually generate data to be put into the vectors depending on the node in use. It is a good way, I think the only one, to avoid wholesale data transfer by using elegant syntax.

III.1.2 Parallelism using FORTRAN 8X.

The definition of sequential FORTRAN 8X [5] explicitly includes array-handling syntax, through the provision of intrinsic functions and subroutines, and through whole-array equivalence and operators and generic functions which act element-wise [2]. This will allow importation of much current work on parallel array manipulation directly into the language if compilers are to be used which distribute compute-intensive tasks to attached processors. Some shared-memory machines such as the Alliant FX and CRAY X/MP already have this facility along with loop unrolling and microcoding. Such in-line whole-matrix operations will be of use in coding only programs on a host which has an attached processor array upon which the data is stored and operations are carried out. Our present discussion is however largely concerned with the details of how to program the array of MIMD processors. F8X does not offer a solution to this.

The German SUPRENUM project has adopted the syntax of F8X as a standard for their run time calls, and base their communication calls on modified READ and WRITE statements as in

```
     send(taskid=    ,tag=   )i/o list
     receive(taskid=    ,tag=   )i/o list
```

which implement F8X array notation. In this way SUPRENUM offers a 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

Eiscube

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 Housholder 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 analyses and solves systems of linear algebraic equations involving dense matrices. Data is again distributed columnwise as in Eiscube.

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] focusses 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 emphasised in that language with the need to write explicit harness code (see section IV.3 for instance). It seems rather to be deeply engrained 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 sleeper". 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 recognises 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 iPSC/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 processors 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
5) analyse and change variables in local store
6) analyse and change contents of messages
7) 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 Hoffmann [II.1.1]. They are as follows.

The "stampede" effect which occurs when evidence of an error in one process is destroyed by the continuing execution of other processes. It is hard to find out which process originally caused the error.

The "bystander" effect is seen 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 "Heisenberg" 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 objects pointed to in the source listing, such as variables, statements, labels, subprograms 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 inital loading as mentioned above. It furthermore fails to debug code which is loaded under program control, since the code starts executing immediately and no break points can be set.

The use made by NiCHE Technology (Transtech) and Meiko of the SUN dbxtool allows a complete parallel program to be compiled on the SUN and analysed, message passing and process contol being done by the trillium-like PRE 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 sdb on conventional systems.

III.3 Load Balancing and Graphical optimisation methods

Only a few hints on what is possible, and some examples of implemented 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 in the same node processor is yet 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 count 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 techique is the TRANSIM simulation tool and dynamic profiler package written in C and developed under the Alvey Project [2]. A similar tool is Schedule [3] 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 emphasises 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 1988. 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 intermediaries. 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(mcb) -- send the message described in the message control block (mcb)

ierr=GetMsg(mcb) -- receive a message into the mcb

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
node= RemHead(list) -- remove the head node of the list and

return it. or NULL if the list is empty
node=RemTail(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 synchronise 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) -- initialise the fields of the semaphore and seed 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
icount=TestSemaphore(semaphore)

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

port=EnableEvents(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(i))
ierr=SetEvent(handler) -- install the given event handler in the event chain
ierr=RemEvent(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, config, confsize)
ierr=EnableLink(link)
ierr=AllocLink(link)
ierr=FreeLink(link)
ierr=Reconfigure(linkconf)
ierr=LinkData(link, linkinfo)
iblock=AllocMem(size, pool)
ierr=FreeMem(iblock)
InitPool(pool)
FreePool(pool)
iblock=AllocFast(size, pool)
iresult=InPool(addr, pool)
Terminate()
Delay(microsecs)
```

```
itype=MachineType()
```

Other calls are available for system libraries and UNIX servers.

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 SUNs for internal users. I will assume that you are working from the Convex and that a shell has been opened in gemacs 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

```
rlogin ipsc
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 gemacs 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 communication 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
            f77 -o host host.f ftime.o -host

ftime.o:    ftime.c ! a small C timing routine
            cc -c ftime.c

node:       node.f
            f77 -o node -sx node.f -node

vnode:      vnode.f
            f77 -o vnode -sx vnode.f -single -lvxsxvec -lvx -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 -lsxvxvec 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 manage SRM resources which is why editing on the Convex is suggested and not using compute-intensive tasks.

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

```
getcube [-c <cubename>] -t 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 names 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, is 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

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

If the name 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>] [> <output file>] [< <input 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 iPSC/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/bsimlib.a
ln -s /priv1/wab/sim/bsim
```

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 library by typing

```
     fc -o host host.o bsimlib.a
```

and the simulator is started with the command

```
     bsim
```

After which a few lines of banner and a special prompt will be seen.

The command list of the simulator is quite short and the commands are different to those on the cube [5]. Nodes can be allocated (up to 16) and programs run in a similar way though, for instance

```
     getcube -t 4
     load node
     cubeman host
     start
```

The commands

```
     trace on
     status -a
```

will be found useful for debugging purposes. Trace gives a one-line message each time a system routine is called in the program, and gives information about message type, source and length. The DECON debugger cannot be used in simulation mode. Further information can be found in the manual [6].

The Vector utilities

Use of the vx vector-extension boards is greatly complicated by the fact that they do not share the memory of the sx boards. They must be thought of as separate nodes, and provision made for sending data to and receiving it from them, just as between any two scalar nodes. This can only be done however from the scalar node to which they belong by copying data from scalar variables to vector variables. There is a high overhead in doing this, which may be somewhat reduced through use of the system mscopy subroutine.

VAST2 will analyse and re-code a FORTRAN program assuming that all its variables are in vector memory. After some further modification by hand this can yield very efficient programs if less than 1 Mbyte of data is required. VAST2 may be used as a development tool, it can be applied to for instance just one critical subroutine which is to be placed in vector memory, the resulting VecLib calls being subject to further modification.

Partitioning of variables into vector and scalar space is done using the vxld utility. A block data program called for instance vxb.f can be written in which all vector variables are defined as follows

```
     block data vxb
     parameter (isize=big)
     common/vector/a(isize),b(isize,isize),c(isize),ivec,jvec
     end
```

This is then compiled and a link control file is produced using vxld which will place the variables of common vector in the rest of the program onto the vector board

```
     f77 -c -sx vxb.f
     vxld -single -o vxb vxb.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 -o vnode -sx vxb vnode.o -single -lvxsxvec -lvx -node
```

Concurrent file system

An intruduction to the Intel concurrent i/o facilities has been given by D.Moody [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 SRM to cfs file systems, and to inspect files. It can be inconvenient not 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
     nsh
     cd /cfs
     mkdir <id>/<directory>
     cd <id>/<directory>
     cp /usr/user/<id>/<directory>/<file> .
     ls -l
     [control-d]
```

It is referred to in the node program as /cfs/<id>/<directory>/<file> in standard UNIX fashion and all FORTRAN access modes are available. In addition some further optimised byte-oriented access modes are described in the reference manual [3] and below, these may not be mixed with normal FORTRAN calls to the same file as the resulting file has a different format.

The concurrent file system is very useful for storing executable images of node programs, this being the one occasion when the host can access the cfs in for instance

```
     call load(/cfs/<id>/<directory>/node, -1, 0)
```

Our host file server is loaded in parallel on the last node

```
     call load(/cfs/rja/eikonxs/fserver.out, nnode-1, 1)
```

and its message passing library, contained in /usr/user/rja/eikonxs/cfsemul.f is compiled and linked with the host program. Routines available are

```
     open(lu) -- open file on lu (defined in fserver.f)
     close(lu) -- close file on lu
     rewind(lu) -- position file pointer at beginning of file on lu
     read(lu,variable,nbytes) -- read nbytes from file on lu
     write(lu,variable,nbytes) -- write nbytes to file on lu
```

Although this emulation is slow it does contain the optimised Intel i/o routines to reduce actual disk access time.

List of FORTRAN routines for handling the cube

attachcube(cubename) -- attach a cube and make it the current cube

cprobe(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(ct, numslots, global) -- obtain information about allocated cubes, see [3]

flick() -- relinquish processor to another process

flushmsg(type, node, pid) -- flush specified message from system

getcube(cubename, cubetype, srmname, keep) -- allocate a cube

gray(j) -- return the binary-reflected Gray code for an integer

ginv(j) -- return the inverse of gray. These calls are used when devising the shortest actual path between nodes.

handler(type, ifunc) -- provide user-written exception handler

infocount() -- returns information about a pending or received message

infonode()

infopid()

infotype()

ctohd(vs, n)

htocd(vs, n)

ctohf(vs, n)

htocf(vs, n)

ctohl(vs, n)

htocl(vs, n)

ctohs(vs, n)

htocs(vs, n)

hrecv(type, buf, len, procedure) -- interrupt-driver receive with handler procedure

iprobe(type) -- determine whether a message of a selected type 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.

killcube(node, pid) -- terminate and clear out a process

killproc(node,pid) -- terminate a process

killsyslog() -- terminate a syslog process

led(lstate) -- turn the node board's green led on or off

load(filename, node, pid) -- load a node process

masktrap(mask)

mclock() -- return the time in milliseconds

msgcancel(id) -- cancel an operation with identification id returned by a call to irecv or isend

msgdone(id) -- determine whether a send or receive operation has completed

msgwait(id) -- wait for completion of a send or receive operation

ihost=myhost() -- obtain node id of the host machine

inode=mynode() -- obtain the node id of the calling process

ipid=mypid() -- obtain the process id of the calling process

newserver(cubename) -- start a new file server for the specified cube

ndim=nodedim() -- obtain dimension of hypercube

nnode=numnodes() -- obtain the number of nodes in the cube = 2**ndim

relcube(cubename) -- release a cube

setpid(pid) -- set process id of a host process, must be done before loading node programs

setsyslog(stdfd) -- start a syslog program

trapblock(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, cpid, 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=iodone(id) -- has an asynchronous read/write operation terminated ?

istat=iomode(lu) -- find mode of a cfs file

iowait(id) -- wait for termination of an asynchronous read/write operation

id=iread(lu, buf, nbytes) -- do asynchronous read from file lu

id=iwrite(lu, buf, nbytes) -- do asynchronous write to file lu

istat=iseof(lu) -- is file pointer at EOF ?

ipoint=lseek(lu,ioffset,iwhence) -- move file pointer in a file

ibytes=lsize(lu,ioffset,iwhence) -- size of file from given position

istat=restrictvol(lu,nvol,nvollist) -- restrict distribution of concurrent files to a particular disk volume

setiomode(lu,mode) -- set file pointer mode for multiple access of cfs file on lu

Example programs

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 process wants 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 iPSC/2 host program for parallel execution farming data from file
c rate2.go
      implicit real*8(a-h,o-z)
      logical list
      character*2 ctype
      open(2,file='rate2.go',status='OLD')
c jump over data lines in this file not required on the host
      read(2,'(f4.1)')dum,dum,dum,dum
      read(2,'(i3)')nstep
      read(2,'(l1)')list
      write(5,'('' ***** program rate2 ***** '')')
```

```
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',-1,0)
c start clock
      start=float(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.lt.1.0e-6)goto 10
c send data to inode
      call csend(2,temp,8,inode,ipid)
      goto 5
 10   finish=float(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 iPSC/2
C
C CALCULATES MAXWELL-BOLTZMANN AVERAGED RATE COEFFICIENTS BY
C  SPLINE  INTERPOLATION  OF CROSS SECTIONS  FOLLOWED  BE  ANALYTIC
C INTEGRATION OVER A SMALL INTERVAL WITH STRAIGHT-LINE FIT
C
      IMPLICIT REAL*8(A-H,O-Z)
      PARAMETER (INS=5)
      LOGICAL LIST
      DIMENSION E(50,ins),SIGMA(50,INS),CSIGMA(50,INS),AAA(50),BBB(50),
     1              CCC(50),SIGA(INS),SIGB(INS),sigas(ins),sigbs(ins),
     2                        rate(ins),thresh(ins),nest(ins)
      open(1,file='rate218.dat',status='OLD')
      open(5,file='rate2.go',status='OLD')
      READ(5,10)REDM
 10   FORMAT(F10.5)
      READ(5,10)CONV
      READ(1,'(2i3)')NE,NS
      READ(5,10)EMAX
      read(5,'(8f10.5)')(thresh(j),j=1,ns)
      DO I=1,NE
      READ(1,'(g12.5)')E(I,1),(SIGMA(I,J),J=1,NS)
      end do
      ...
C loop over temperatures, data is farmed out from the nost
C get host's id
      idhost=myhost()
C tell the host I am ready for data
```

```
 50   call csend(5,dum,0,idhost,0)
C receive temperature for this loop
      call crecv(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),ratet
 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
      itype=i+300
      if(iprobe(itype).ne.0)goto(1,2,3,4,5),i
300   continue
      call flick()  ← goto 10
1     ...
      goto 10
2     ...
      goto 10
3     ...
      goto 10
4     ...
      goto 10
5     ...
      goto 10
      end
```

IV.3 Meiko Computing Surface (installed at DL summer 1987, upgraded winter 1988)

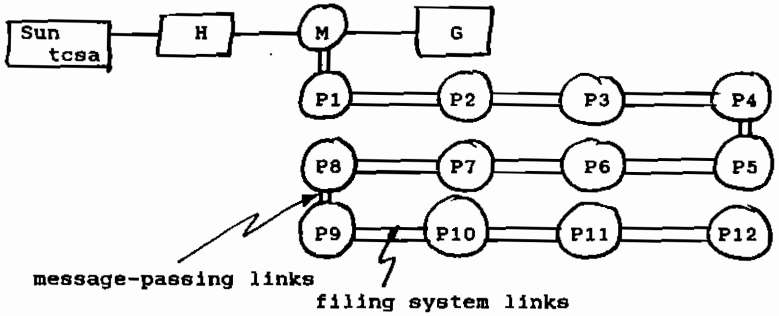
Acknowledgements: the original M10 computing surface is on loan from the SERC/DTI Transputer Initiative loan pool at RAL. DL is an Associated Support Centre to the Initiative, and undertakes to make programs developed on the machine nationally available through the Sheffield National Transputer Support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

Configuration of the current surface at DL, transputer types:

1x T414 mk014 3Mbyte host processor H
1x T414 mk015 128kbyte graphics processor G
1x T414 mk021 8Mbyte mass-store board M
4x T800 compute nodes (1 quad board mk060) 4Mbyte 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
300Mbyte scsi disk and 1/4" tape cartridge driven off the mk021
front end Sun tcsa 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 1.5 Mflops/sec (Inmos figures)

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



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 cmdtool window. The Meiko at Daresbury has also been provided with the MMVCS and MeikOS operating systems, however this requires the host, mass-store (to drive the disk) and one mk060 node to be dedicated to the system leaving only 11 nodes (mostly of low memory size) to the user. Alternative software tools, called CStools, are available from Meiko for the in-Sun hardware, and will provide a more familiar UNIX-like environment. These are breifly 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 [1]

ibli(ichan, ibuff, isize(1)) -- inputs integers from ichan,

iblo(ichan, ibuff, isize(1)) -- 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=itime() -- gets the current transputer clock time in 64us ticks.

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 one user. Another harness runs a cell-type application with the possibility to swap data at cell edges, useful in CFD or lattice-guage type work.

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

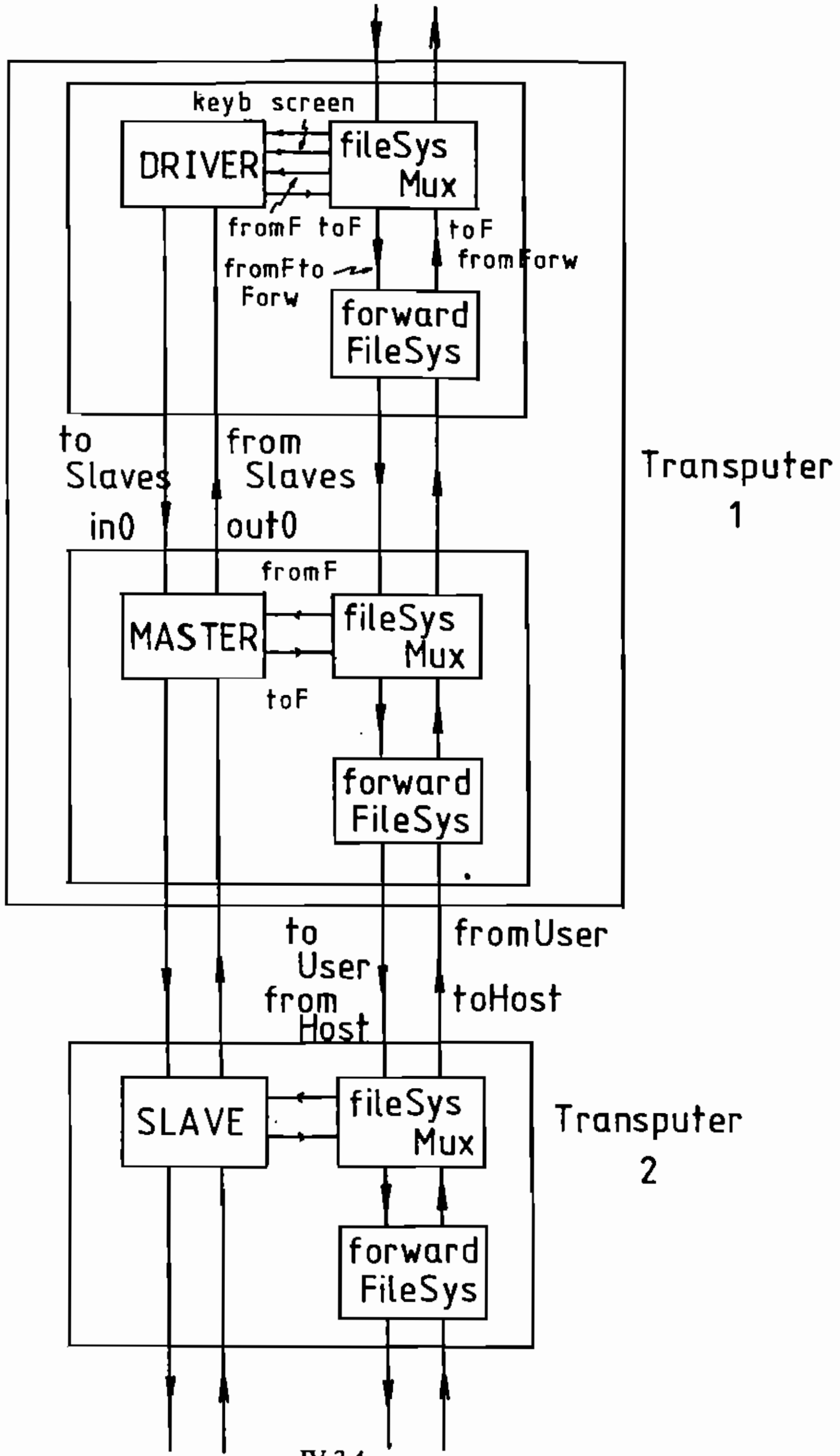
The harness used at DL and Durham University is called Fortnet v2.1. It was developed by Sebastian Zureck (Cambridge), Lydia Heck (Durham) and myself. A more detailed description and published version of the code is currently being prepared for publication [6]. A version could also be made available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

"une double postulation simultanée" Baudelaire

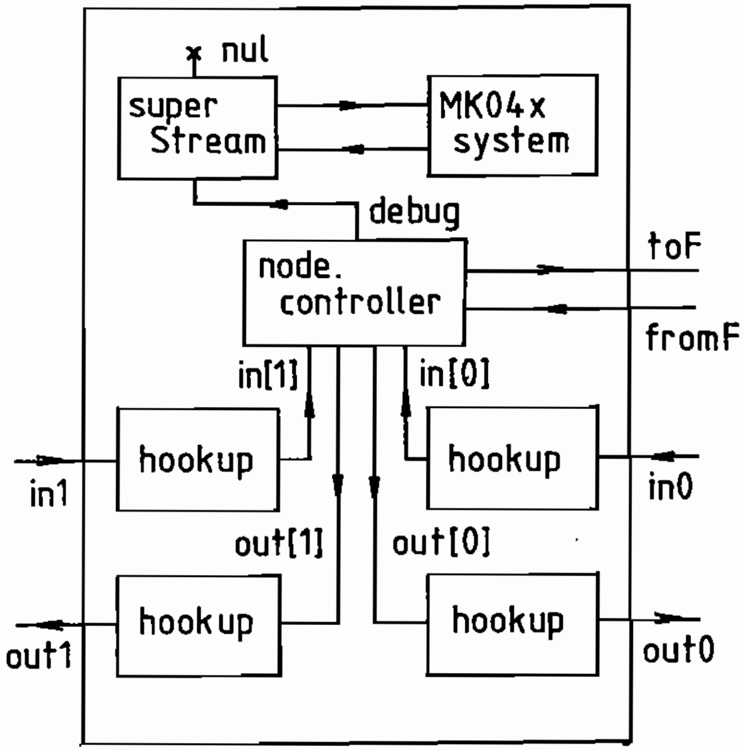
The original harness program at Daresbury was written entirely in occam-2 by Sebastian Zurek during the summer of 1987. The code was subsequently simplified and partly converted to FORTRAN by Bill Purvis of DL shortly after Seb left. Since then I have taken over the work of improving and maintaining 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 SEND and RECEVE operations and version 2.1, written by Lydia 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 programs is now implemented on all the nodes to reduce the disk storage overheads and complication of the harness (see Fig. 1).

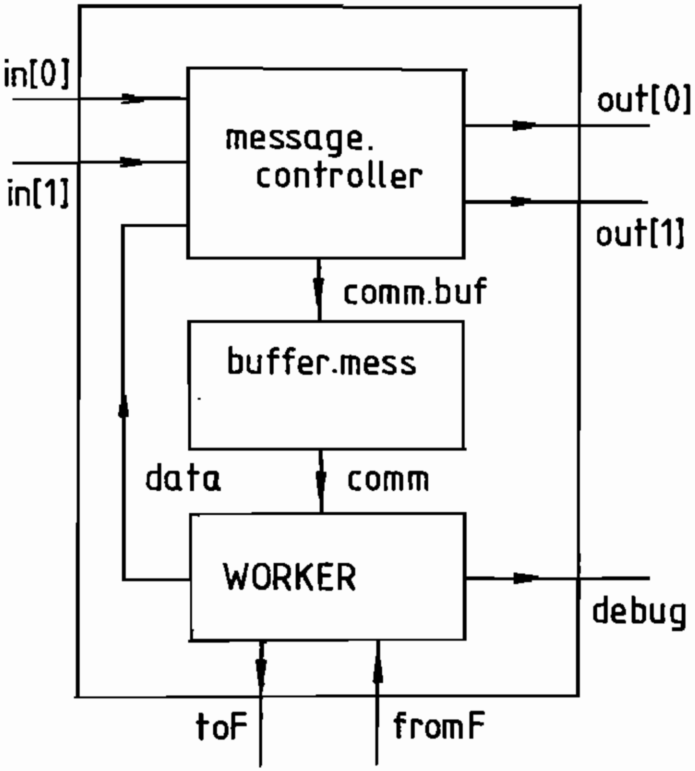
The slave processes (fig. 2) consist of a buffer process (fig. 4) to handle transmission of messages along the chain (fig. 1), and a worker process (fig. 3) which contains the FORTRAN. READ and WRITE and SEND and RECEVE communication is done using the protocol of fig. 5. The user's program is embedded in 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 tf77 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.


keyb
screen
DRIVER
fileSys Mux
fromF
toF
toF
fromForw
fromFto Forw
forward FileSys
to Slaves
from Slaves
in0
out0
Transputer 1
fromF
MASTER
fileSys Mux
toF
forward FileSys
to User
from Host
fromUser
toHost
SLAVE
fileSys Mux
forward FileSys
Transputer 2

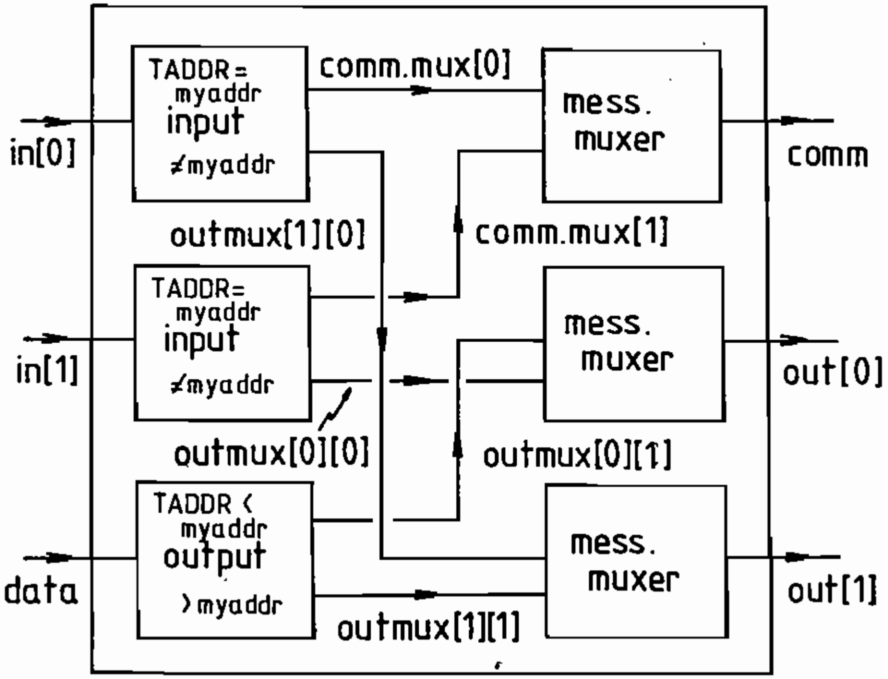
IV.3.4


nul
super Stream
MK04x system
debug
node. controller
toF
fromF
in[1]
in[0]
hookup
hookup
in1
in0
out[1]
out[0]
hookup
hookup
out1
out0

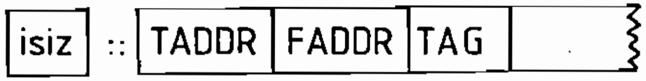
IV.3.5


in[0]
in[1]
message.
controller
out[0]
out[1]
comm.buf
buffer.mess
data
comm
WORKER
debug
toF
fromF

IV.3.6


in[0]
TADDR =
myaddr
input
≠myaddr
comm.mux[0]
mess.
muxer
comm
outmux[1][0]
comm.mux[1]
in[1]
TADDR=
myaddr
input
≠myaddr
mess.
muxer
out[0]
outmux[0][0]
outmux[0][1]
data
TADDR <
myaddr
output
> myaddr
mess.
muxer
out[1]
outmux[1][1]

IV.3.7



```
dimension isiz(1), ibuff(n)
isiz(1)=n
iblo(ichan,isiz,1)
iblo(ichan,ibuff,isiz(1))
```

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 shoudl correctly terminate a process.

```
CHARACTER*132 BUFFER
CALL READ(LU,NCHAR,BUFFER)
```

READ(BUFFER,<fmt>)<iolist> -- 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 transpuer 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 rate2 example in section II.2.

```
CHARACTER*132 BUFFER
WRITE(BUFFER,<fmt>)<iolist>
```

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 subroutine 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 RECEVS(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 occam buffer process via channels.

CALL RECEVE (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 occam 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 RECANY (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, changes are needed in the occam to improve it.

CALL BRCAST (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 IOSUP library.

INODE=MYNODE() -- Obtains the address of the current processor which can be used in program constructs of the form illustrated in section III.1.1

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 message are printed out during the execution of each of the IOSUB 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 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 TRAM.

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 mass 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/f77/fortran/hex/f77heado
PARAMETERS CHAN OF ANY fromf, tof, keyb, screen, user1, -
      user2, report
/home/rja/export/worker
<user's main program modified as above>
[<user's subroutines>]
/home/rja/export/iosub
```

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 opstrans.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/
utilLib    usr/meiko/occam/util/
complib    usr/meiko/occam/library/
t4sys_objs usr/meiko/f77/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 F77 compiler as follows

```
tf77 +f <file.f>
```

After successful compilation, code for each node is linked with

```
tlink worker t8 library
```

This has the effect of producing a library file worker.l18 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=mynode()
      if(inode.eq.n)then
      ...
      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 iosub files in the system directory.

Code for the mass store board is linked from the worker0.tpo file which references the 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>
      tlink worker0 library
      tf77 server.f
      tlink server library
```

Using OPS to attach the FORTRAN code to the occam harness

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

```
      sboot -- if on a Sun or in 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/toplevel.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 Fortnet harness is contained in the fold

```
      ...PROGRAM Fortnet v2.1
```

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

```
      .../bob/<sub directory>/worker.l18
      .../bob/<sub directory>/server.l14
      .../bob/<sub directory>/worker0.l14
```

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)
c routine to partition and distribute data for simple parallel
c sort version (1)
c 14/3/88             R.A.
      parameter (nproc=8,nmax=500)
      dimension x(nmax)
      dimension na(nmax),a(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 MK021 board
      call read(1,4,buffer)
      read(buffer,'(i4)')n
      call read(1,10,buffer)
      read(buffer,'(f10.5)')x(1)
c search through data for largest and smallest elements
      xmax=x(1)
      xmin=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=time-time0
      write(buffer,'(g12.5,'' seconds to read data'')')time
      call write(6,33,buffer)
c partition for p processors into segments of width (maxa-mina)/p
      width=(xmax-xmin)/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,nproc
 20   na(i)=1
      do 15 i=1,n
c first processor handled separately
      if(x(i).lt.xmin+width)then
      a(1,na(1))=x(i)
      na(1)=na(1)+1
      end if
      do 35 j=2,nproc-1
      xminp=xmin+width*float(j-1)
      xmaxm=xmin+width*float(j)
      if(x(i).ge.xminp.and.x(i).lt.xmaxm)then
c row j of a() will go to processor j+1
      a(j,na(j))=x(i)
      na(j)=na(j)+1
      end if
 35   continue
c last processor handled separately
      if(x(i).ge.xmax-width)then
      a(nproc,na(nproc))=x(i)
      na(nproc)=na(nproc)+1
      end if
 15   continue
      call cpu(time)
      time=time-time0
      write(buffer,'(g12.5,'' seconds to partition data'')')time
      call write(6,38,buffer)
c send data
      do 25 i=1,nproc
      ip1=i+1
      call check(ip1)
      nsend=na(i)-1
      call send(ip1,4,nsend)
      do 25 j=1,nsend
      call send(ip1,4,a(i,j))
 25   continue
      call cpu(time)
      time=time-time0
      write(buffer,'(g12.5,'' seconds to send data'')')time
      call write(6,33,buffer)
c receive ordered results, smallest number comes first
      do 30 i=1,nproc
      ip1=i+1
      call check(ip1)
      nrec=na(i)-1
      do 30 j=1,nrec
      call recevs(ip1,10,buffer)
      call write(2,10,buffer)
 30   continue
      call cpu(time)
      time=time-time0
      write(buffer,'(g12.5,'' seconds to receive data'')')time
      call write(6,36,buffer)
      end
```

worker program

```
      subroutine user
c slave coding of sort program version (1)
c new code 14/3/88 single file with fortran for all slaves on
compute boards.
      character*132 buffer
      dimension x(500)
      real*8 time,time0
      common/talk/inode,npend(10),nwait,nw
c      if(inode.gt.8)return
      call debug('OFF')
      nrec=1
      call cpu(time0)
      call wait(nrec)
      call receve(nrec,4,n)
      write(buffer,'('' slave '',i2,'' receiving '',i4,'' elements'')')
     1      inode,n
      call write(6,33,buffer)
      do 5 i=1,n
```

```
      call receve(nrec,4,x(i))
 5    continue
      call sort(x,n) ! any sequential sorting routine can be used
      call wait(nrec)
      do 10 i=1,n
      write(buffer,'(f10.5)')x(i)
      call send(nrec,10,buffer)
 10   continue
      call cpu(time)
      time=time-time0
      write(buffer,'('' time on slave '',i2,2x,g12.5,'' seconds'')')
     1      inode,time
      call write(6,39,buffer)
      end
```

Future versions of Fortnet will include the following additions:

i) dynamic loading of tasks from a names file, or just by reference to the executable. This permits a batch queue to be easily divised.

ii) a way to kill executing tasks will be implemented

iii) subroutine names will be made to resemble more those on the hypercube machines

iv) the end processor of the chain will be given a link to the first for more efficient two=way message routing. This will in general create problems of flow control which must be avoided.

v) a message queuing system will be implemented in addition to the current self-correcting protocol. This will enable the asynchronous communications calls and polling calls to be implemented similar to those on the hypercubes.

vi) a high-level library of global and shared-memory emulation routines will be provided

vii) 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 OPS system on the Meiko, or TDS on an Inmos system, and indicated use of one particular harness to run parallel FORTRAN code. It is significant that Meiko Scientific Ltd. have recently announced their CStools products, initially available 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 ALT 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 trillium. The designers of CStools have benefitted from experience of the latter product on the Niche system (section IV.4).

Further functionality is provided in the CStools environment (actually in the underlying Virtual Computing Surface VCS) to load processes 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

```
networkis ternary tree
```

which is the default, binary tree, or unary tree (a pipeline).

The *.tpo files used to create the executable images might be of the form

```
/INCLUDE /usr/meiko/fortran/hex/f77head
/INCLUDE /usr/meiko/fortran/hex/cs
master
```

which would be linked at run time with the command tlink <name> t8.

In addition to running a transputer network CStools can run a network of UNIX processes (on one or more SUNs), giving the obvious capability to debug with dbxtools. Again much has been learned from early attempts by Niche to do this. Interface to SUN host processes and graphics processes is clearly also possible

Fortran subroutines in CStools

idesc=cs_createPort(name) -- creates a port and returns a port descriptor, or -1 if unsuccessful

idesc=cs_findPort(name, iblock) -- searches for a port with given name, waits until one exists if iblock=1

ierr=cs_send(idesc, data, nbytes/4, iblock) -- attempts to send data to the port described by idesc, blocking as above

ierr=cs_recv(idesc, data, nbytes/4) -- attempts to receive data from idesc 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 Epsom.

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-correcting memory interface and 22 independent DMA links. The speed of a single chip is around 500 kflops or 2 mip in 32 bit, or 300 kflops in 64 bit. Up to 500 Mflops 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-performancs graphics and an open-system board are other options.

The NCUBE seven and 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 Nmacs, 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

sw=nwrite(buffer, length, proc, type, flag) -- send vector of length "length" to proc

sr=nread(buffer, length, proc, type, flag) -- receive vector of length "length" from proc

whoami(inode,proc_id, host, iorder_of_cube) -- call information about node and host

Other facilities and the ability to load and kill processes are rather similar to those on the Intel hypercube.

IV.5 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 say 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 Mbytes or less, or 8 with 32 kbyte memory. A fully populated board might therefore have 16x T800 + 32 Mbyte total memory. The system if 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). NiCHEs 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 types are the SUN 3.0 host and IMST800 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 dbxtool 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 farming 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 NSEND and NRECV and asynchronous NTRY_SEND 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

```
     #include <trillium/NET.h>
     #define ESTART 11 -- start event
     #define EEND   12 -- end event
```

The F77 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
ret=kinit(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=getnodetype() -- return the type of node
ipid=getpid()
get_route -- given destination, find datalink to use
m=ltot(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)
tsend -- transport layer message send
trecv -- transport layer message receive
n=ttol(m) -- 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
itime=ldtimer() -- read the transputer timer; clock ticks at 64 microsecond intervals
setpri -- set process priority on a transputer
i=tprint(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()

In the above list there appears to be no means to dynamically load, kill or spawn processes. In more recent documentation this feature has been added.

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 Meiko 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 Europeen ESPRIT 1085 project it has undoubtably 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 3Ls software for NiCHE (Transtech) and PC-based transputer arrays such as the Gemini system [2] and also the Meiko 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 channels and scheduling. This results in a very low overhead of around 5 kbytes per node. Furthermore, since there is no buffering of communications through occam harness multiplexors, inter-processor communication is efficient.

The 3L compiler accepts standard FORTRAN [3] and produces binary object code in Inmos 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 Meiko 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 configurer FCONFIG. 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 configurer CONFIG 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 t4f or t8f commands e.g.

```
t8f source
```

which looks for a file source.f77. This produces an object file source.bin as output. This is linked with the parallel run time library using either the t4flink or t8flink commands. Executable files source.b4 or source.b8 are produced. In MS-DOS the linkt linker can alternatively be used for several object files.

The program is run using

```
config sourcg.cfg source.app
```

```
     afserver -:b source.app
```

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 afserver ins=1 outs=1 ! declares tasks with the number of
! input and output ports and amount of workspace
     place afserver host
     place one root
     place two root ! placement on host and first transputer
     connect ? two[0] afserver[0]
     connect ? afserver[0] two[0]
     connect ? two[1] one[1]
     connect ? one[1] two[1] ! shows how the tasks are connected
! together by their ports
```

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 farms. Channels may be bound to ports of a task, as by the configurer, 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 configurer 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(ichan) -- return address of internal channel word ichan

F77_Chan_Init(iaddr) -- initialises an internal channel word whose address is iaddr for communication. All internal channel words must be initialised. Channels bound to ports should not be initialised.

F77_Chan_out_Byte(ibuff,out) -- send lowest byte from ibuff on channel out

F77_Chan_in_Byte(ibuff, 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

l=F77_Chan_in_Byte_t(ibuff, in, itimeout) -- try to read during next itimeout ticks and return value l=.false. if no byte found. Equivalent functions are available for other message types as above.

in=F77_Alt_Wait(inchan, 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_noWait(inchan, iaddr1, ..., iaddrn) -- as above but does not wait. Returns a value of 0 if nothing is ready

in=F77_Alt_Wait_Vec(inchan,invec) -- as above but the argument list is replaced by a vector of channels

in=F77_Alt_noWait_Vec(inchan,invec) -- as above but the argument list is replaced by a vector of channels

F77_Net_Send(nbytes, array, lcomplete) -- send processor farm message. Master to slaves or slaves to master only

F77_Net_Receive(nbytes, array, lcomplete) -- receive a processor farm message

l=F77_Timer_after(timer1, timer2) -- is .true. if timer1>timer2, otherwise .false.

F77_Timer_delay(iticks) -- causes current thread to wait for at least iticks

itime=F77_Times_now() -- return current value of timer

F77_Timer_wait(itime) -- wait until the value of the priority time is at least itime

nport=F77_Chan_out_Ports() -- returns number of output ports

nport=F77_Chan_in_Ports() -- 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_Thread_start(subroutine, iwsarray, nwsbytes, F77_Thread_urgent, nargs, arg1, ..., argn) -- start a named thread from a subroutine. F77_Thread_noturgent may also be used as an argument

lstatus=F77_Thread_create(subroutine, nwsbytes, nargs, arg1, ..., argn) -- attempts to create a thread using subroutine of the same priority as the current thread and taking nwsbytes from the heap as workspace. This workspace is never returned, and F77_Thread_start should be used alternatively.

F77_Thread_stop -- stops the current thread

i=F77_Thread_priority() -- returns the priority of the current thread

F77_Thread_use_RTL() -- ensure that no other threads use the run time library, waits if it is already in use. This is again beacuse FORTRAN is not reentrant and library routines can be invoked only once at a time.

F77_Thread_free_RTL() -- release the RTL

F77_Thread_restart(ihandle) -- ihandle points to the workspace of the thread to be reastarted. It was obtained from F77_Chan_reset

F77_Thread_deschedule() -- causes the current thread to be descheduled and allows another thread to take over execution. Same purpose as the Intel flick() routine (see example in section IV.2).

F77_Sema_init(mysema, ivalue) -- initialise the semaphore variable to an empty queue

F77_Sema_signal(mysema) -- choose one of the thread waiting for mysema to be reactivated. The value of the semaphore increases by one only if no threads are waiting. Only threads executing at the same priority can synchronise with a semaphore, otherwise they must use messages on channels

F77_Sema_signaln(mysema, n) -- call above routine n times

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

F77_Sema_waitn(mysema, n) -- call above routine n times

iaddr=F77_alloc_Host_mem(nbytes) -- allocates a block of at least nbytes in the base memory of the host and returns its 32-bit address

F77_free_Host_mem(iaddr) -- 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 iaddr

F77_Block_from_Host(iaddr, outaddr, nbytes) -- reverse of above

F77_read_Segments(idosblock) -- reads processor segment registers on a PC

F77_Host_interrupt(intno, isegs, idosblock) -- loads the contents of DOS 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 Meiko Scientific Ltd., Richard Chamberlain of Intel Scientific Computing Ltd., and David Fincham 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 availbale from the Sheffield National Transputer Centre, is [R.1.1]. Other sources of useful information are the Occam Users' Group [R.1.2], the Edinburgh Supercomputer Centre Newsletter [R.1.3], the Transputer Initiative Mailshot [R.1.4] and ourselves [R.1.5].

[I.1.1] "The programming language FORTRAN" ANSI X3.9 (1978)

[I.1.2] "Reference manual for the Ada programming language" US DoD Report (1980)

[I.1.3] B.Hansen "The programming language Concurrent Pascal" IEEE Trans. Software Engineering 1 (1975)

[I.1.4] J.R.McGraw et al. "SISAL - streams and iteration in a single assignment language, reference manual" Lawence Livermore Natl. Lab. (1980)

[I.1.5] reference to DAP

[I.1.6] CSP

[I.1.7] "IEEE 1003.1 POSIX standard draft 13" IEEE Working Group Technical Committee on Operating Systems of the IEEE Computer Society (New York, 1988)

[II.1.1] D.F.Snelling and G.-R. Hoffmann "A comparative study of libraries for parallel processing" Parallel Computing 8 (1988) 255-66

[II.1.2] "Proceedings of the 2nd International SUPRENUM Colloquium, 30th September- 2nd October 1987, Bonn, FRG" Parallel Computing 7 (1988) 263-499

[II.2.1] A.A.Brown and G.D.Burns "Users' guide to the Trillium operating system (release 1.0)" Cornell Theory Center, New York (1987)

[II.3.1] W.C.Atlas and C.L.Seitz " Multicomputers: Message-passing concurrent computers" Computer August (1988) 9-24

[II.4.1] Perihelion Software Limited, 24 Brewmaster Buildings, Charlton Trading Estate, Shepton Mallet, Somerset, BA4 5QE.

[II.4.2] X-windows VII

[II.4.3] "Helios developer's notes" Perihelion Software Ltd. (1987)

[II.5.1] D.May EPL

[II.5.2] Inmos Limited "Occam Programming Reference Manual" Prentice-Hall (1988)

[II.5.3] Dick Pountain "A tutorial introduction to Occam programming" Blackwell Scientific Publications for Inmos Limited, Bristol (1986)

[II.5.4] G.Jones "Programming in Occam" Prentice-Hall

International

[II.5.5] "IMS T800 Architecture" Inmos Limited, Bristol, Technical report 72-TCH-006

[II.5.6] Inmos Limited "Transputer Development System" Prentice Hall (1988) ISBN 0-13-928995-X

[II.5.7] Edinburgh Harnesses, TINY etc.

[II.7.1] A.M.Lister "Fundamentals of Operating Systems" MacMillan Press (1979) ISBN 0-333-27287-0

[II.7.2] A.S.Tanenbaum "Operating Systems: Design and Implementation" Prentice Hall (1987) ISBN 0-13-637331-3

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

[III.1.2] "FORTRAN 8X features that assist in the exploitation of parallelism" J.Reid, Harwell Laboratory

[III.1.3] L.M.Delves and N.G.Brown "SERC EMR, Numerical Libraries for transputer arrays - final report" Liverpool University (1988)

[III.1.4] "Occam Numerical Library Documentation" Liverpool University (1988)

[III.1.5] TCS Annual Report (1989)

[III.1.6] TCS Annual Report (1989)

[III.2.1] "iPSC Concurrent Debugger" Intel manual, order no 310613-00B Intel scientific computers, Beaverton, Ore. (1987)

[III.2.2] W.M.Pan and V.Jackson "A concurrent debugger DECON for iPSC/2 programmers" Intel Scientific Computers, Beaverton, Ore. (1987)

[III.3.1] Liverpool ????

[III.3.2] TRANSIM ????

[III.3.3] Sorensen and Dongarra "Schedule" Argonne National Lab

[III.3.4] P.C.Capon et al. "ParSiFal: a parallel simulation facility" IEE Colloquium, IEE Digest no 91 (1986)

[IV.1.1] T series FPS 860-0001-014A

[IV.1.2] S.Hawkinson "The FPS T Series, a parallel vector supercomputer" FPS Inc., Beaverton, Ore. (1986)

[IV.1.3] D.A.Tanqueray "The Floating Point Systems T Series" FPS (UK) Ltd., Bracknell, Berks. (1987)

[IV.2.1] iPSC/2 sales brochure, order number 280110-001 intel Scientific Corporation, Beaverton, Ore.

[II.2.2] WCAtlas and CLSeitz " Multicomputers: Message-passing cuncurrent computers" Computer August (1988) 9-24

[IV.2.3] P.Pierce "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

[IV.2.6] D.Moody "Intel ins and outs" Parallelogram 15 (1989)

[IV.2.7] "Simulator reference manual" Intel Scientific Corp., Beaverton, Ore.

[IV.3.1] Meiko Computing Surface Fortran Manual, Meiko Scientific Ltd. Bristol (1987/1988/march 1989)

[IV.3.2] M.Surridge "A multi-transputer harness for 'farm' parallelism using the FORTRAN77 programming language" Transputer Support Centre, Southampton (1988)

[IV.3.3] A.J.G.Hey, J.S.Reeve and M.Surridge "Software migration aids for transpoter 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)

[IV.3.5] The Computing Surface Reference Manual, Meiko Scientific Ltd., Bristol (1987, march 1989)

[IV.3.6] R.J.Allan, L.Heck and S.Zureck "Parallel FORTRAN in scientific computing: a new occam harness called Fortnet" Comp. Phys. Comm. (1989) submitted

[IV.3.7] R.J.Allan and L.Heck "..." Liverpool International Conference on the application of Transputers, proceedings, ...

[IV.4.1] NCUBE ten sales brochure, NCUBE Corp, Beaverton, Ore. (1988)

[IV.5.1] NiCHE NTP1000 Technical Summary of ACP, NiCHE Technology, Bristol (1988). The NiCHE system is currently marketed by Transtech Devices, Bristol.

[IV.7.1] A.D.Culloch "Parallel programming toolkit for 3L-C, FORTRAN and Pascal" preprint, 3L, Livingston, (march 1988)

[IV.7.2] Gemini

[IV.7.3] "3L Parallel Fortran User Guide" 3L Ltd., (1988)

[R.1.1] A.J.G.Hey and M.R.Sleep "Transputer Bibliography" SERC/DTI Initiative on Engineering Applications of Transputers, contact Roger England National Transputer Support Centre, Sheffield Science Park, Arundel Street, Sheffield, S1 2NT (1988) or NTC @ uk.ac.shef.pa. A library of transputer software is also maintained and available upon request.

[R.1.2] G.Jones "Occam Users' Group Newsletter" Oxford. An e-mail network is also established, contact gj @ uk.ac.oxford.prg or derek @ uk.ac.bristol.compsci

[R.1.3] "Edinburgh Concurrent Supercomputer Newsletter" contact David Mercer, Edinburgh University Computing Service, The King's Buildings, Mayfield Road, Edingurgh EH9 3JZ or D.Mercer @ uk.ac.edinburgh

[R.1.4] contact M.R.Jane, Transputer Initiative, Informatics Department, Rutherford Appleton Laboratory, Bldg. R1, Didcot, Oxon., OX11 0QX

[R.1.5] contact R.J.Allan, Daresbury Laboratory, S.E.R.C., Daresbury, Warrington, WA4 4AD or RJA @ uk.ac.dl.dlgm or ARCG @ daresbury

```
     Appendix A. Timing of various FORTRAN calls on a variety of
parallel computers.

                         u-VAX/II       Meiko T4 MS    Meiko T8 MS
           Meiko T8      FPS T20 scalar FPS T20 vector iPSC/1 scalar
           iPSC/1 vector NCUBE          iPSC/2 SX      iPSC/2 SX VX
           NiCHE T8
RMS maths error
                0.28111E-13

                0.19629E-13
timings for iteration length 50000
initialise one array           0.51            0.32            0.24
                0.20           1.17            0.012           0.65
                0.03           0.54            0.20
                0.16
initialise four array (1)      2.20            0.78            0.70
                0.52           3.06            0.046           2.60
                0.05           0.98            0.70
                0.5
initialise four array (2)      2.20            0.40            0.35
                0.28           1.55                            1.28
                0.05           0.49            0.36
                0.24
IF test                        0.70            0.855           0.32
                0.24           1.50                            1.05
                0.08           0.614           0.23
                0.2
V = V*V + V                    2.27            6.20            0.53
                0.38           14.01           0.082           4.35
                0.05           1.25            0.47
                0.34
call sub, 0 arguments          0.74            0.68            0.68
                0.56           0.81                            1.05
                               0.69            0.16

                0.22
call sub, 1 arguments          0.74            0.80            0.80
                0.72           1.13                            2.93
                               0.90            0.19

                0.28
call sub, 2 arguments          0.81            0.94            0.92
                0.80           1.52                            4.63
                               0.99            0.23

                0.36
call sub, 3 arguments          0.87            1.03            1.00
                0.88           1.89                            6.33
                               1.06            0.26

                0.42
call sub, 4 arguments          0.97            1.13            1.10
                1.00           2.21                            8.08
                               1.15            0.27

                0.48
scalar random numbers          7.34            7.59            2.28
                2.40           15.3                            72.10
                               5.36            1.91

                1.76
reciprocal                     1.09            5.06            0.42
                0.34           12.24           0.20            3.30
                0.23           1.15            0.48
                0.30
```

```
square root                    4.52            66.78           0.91
                0.84           617.95          0.26            4.10
                0.26           1.07            1.13
                1.22
exponential                    6.88            70.73           4.60
                5.08           305.84          0.37            13.93
                0.33           3.49            1.46
                9.88
logarithm                      7.56            65.37           3.75
                4.02           256.76          0.30            9.50
                0.50           11.3            1.38
                16.16
cosine                         6.59            60.87           2.64
                3.00           520.82          0.22            15.4
                0.30           2.65            1.08
                15.14
inverse cosine                 11.52           166.58          5.94
                5.62           2317.78                         20.15
                               11.54           4.00

                51.26
sine                           6.74            63.26           2.79
                3.18           399.15          0.26            15.35
                0.33           2.59            1.11
                12.4
inverse sine                   10.67           162.77          5.91
                5.46           1571.32                         15.53
                               3.99            4.58

                49.34
sum ((v-v)**2)                 1.65            8.99            0.56
                0.46           21.81           0.14            4.48
                0.08           1.22            0.52
                1.02
gaussian random numbers        122.28          676.53          51.0
                54.42          2645.49                         752.58
                               146.2           30.28

                167.5
vector swap                    1.28            0.64            0.61
                0.46           2.58            0.29            2.73
                0.05           1.06            0.34
                0.38
dot product                    1.53            6.58            0.43
                0.36           14.69           0.012           4.3
                0.05           1.09            0.41
                0.34
INT (v*scalar)+1               1.25            4.86            0.50
                0.42           10.83                           3.2
                0.08           2.12            0.23
                0.34
gather                         2.79            0.39            0.36
                0.28           1.56                            1.88
                0.08           0.85            0.19
                0.26
scatter                        2.82            0.41            0.37
                0.28           1.56                            1.85
                0.05           0.85            0.20
                0.24
max magnitude element          0.87            3.03            0.90
                0.80           2.69            0.066           3.05
                               1.08            0.52
```

0.28

Notes.

1) The u-VAX clock is accurate to 0.01s, that of the Meiko and T series to 64us 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 hand 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 acess.

4) NCUBE timing was produced at DL using an NCUBE-4 on loan from Arrow Computers Swindon. An iteration count of 5000 was run parallel on four 8MHz nodes. Times scaled to suit.

5) Meiko T800 timings 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. Penguin software is responsible. The same problem occurs on the NICHE system since their compilor was obtained from Pensoft (as Penguin now call themselves)

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

A listing of the program TTIME, 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=2500)
      DIMENSION A(LEN),B(LEN),C(LEN),D(LEN),IA(LEN),
     &          T(30)
      DATA T/30*0.0D0/
      CALL cpu(START)
      DO 10 I=1,LEN
         A(I)=0.0D0
10    CONTINUE
      CALL cpu(FINISH)
      T(1)=FINISH-START
C
      CALL cpu(START)
      DO 20 I=1,LEN
```

```
         A(I)=0.0D0
         B(I)=1.0D0
         C(I)=2.0D0
         D(I)=3.0D0
20    CONTINUE
      CALL cpu(FINISH)
      T(2)=FINISH-START
      CALL SUB4(A,B,C,D)
C
      CALL cpu(START)
      DO 30 I=1,LEN,2
         A(I)=0.0D0
         B(I)=1.0D0
         C(I)=2.0D0
         D(I)=3.0D0
30    CONTINUE
      CALL cpu(FINISH)
      T(3)=FINISH-START
      CALL SUB4(A,B,C,D)
C
      CALL cpu(START)
      DO 40 I=1,LEN
         IF(A(I).EQ.0.0D0) GOTO 40
C This not executed
           B(I)=1.0D0
40    CONTINUE
      CALL cpu(FINISH)
      T(4)=FINISH-START
      CALL SUB4(A,B,C,D)
C
      CALL cpu(START)
      DO 50 I=1,LEN
         A(I)=B(I)*C(I)+D(I)
50    CONTINUE
      CALL cpu(FINISH)
      T(5)=FINISH-START
      CALL SUB4(A,B,C,D)
C
      CALL cpu(START)
      DO 60 I=1,LEN
         CALL SUB0
60    CONTINUE
      CALL cpu(FINISH)
      T(6)=FINISH-START
C
      CALL cpu(START)
      DO 70 I=1,LEN
         CALL SUB1(A(I))
70    CONTINUE
      CALL cpu(FINISH)
      T(7)=FINISH-START
C
      CALL cpu(START)
      DO 80 I=1,LEN
         CALL SUB2(A(I),B(I))
80    CONTINUE
      CALL cpu(FINISH)
      T(8)=FINISH-START
C
      CALL cpu(START)
      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)=1.0D0/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)=SQRT(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
```

```
      CALL SUB4(A,B,C,D)
C
      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)=ASIN(C(I))
190   CONTINUE
      CALL cpu(FINISH)
      T(19)=FINISH-START
      CALL SUB4(A,B,C,D)
C
      SUM=0.0D0
      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=SQRT(SUM/FLOAT(LEN))
      WRITE(6,'('' RMS error from math routines '',g12.5)')SUM
C
      HALF=0.5D0
      ONE=1.0D0
      AMEAN=0.0D0
      SDEV=1.0D0
      CALL cpu(START)
      DO 210 I=1,LEN
211       CALL SRAND(ISEED,V1)
          CALL SRAND(ISEED,V2)
          V1 = -LOG(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
      S=0.0D0
      CALL cpu(START)
      DO 230 I=1,LEN
          S=S + A(I)*C(I)
 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=0.0D0
      CALL cpu(START)
      DO 270 I=1,LEN
          IF(ABS(ZM).GT.ABS(D(I))) THEN
            ZM=D(I)
            IZ=I
          ENDIF
 270  CONTINUE
      CALL cpu(FINISH)
      T(27)=FINISH-START
      CALL SUB2(ZM,IZ)
      CALL SUB4(A,B,C,D)
C
```

```
      WRITE(6,'('' Timings for iteration length '',i8)')LEN
      WRITE(6,'(/)')
      WRITE(6,1) ' Initialise one array          ',T(1)
 1    FORMAT(1X,A,F8.3)
      WRITE(6,1) ' Initialise four arrays (1)    ',T(2)
      WRITE(6,1) ' Initialise four arrays (2)    ',T(3)
      WRITE(6,1) ' IF test                       ',T(4)
      WRITE(6,1) ' V = V*V + V                   ',T(5)
      WRITE(6,1) ' Call subroutine, 0 arguments ',T(6)
      WRITE(6,1) ' Call subroutine, 1 arguments ',T(7)
      WRITE(6,1) ' Call subroutine, 2 arguments ',T(8)
      WRITE(6,1) ' Call subroutine, 3 arguments ',T(9)
      WRITE(6,1) ' Call subroutine, 4 arguments ',T(10)
      WRITE(6,1) ' Scalar random numbers         ',T(11)
      WRITE(6,1) ' Reciprocal                    ',T(12)
      WRITE(6,1) ' Square root                   ',T(13)
      WRITE(6,1) ' Exponential                   ',T(14)
      WRITE(6,1) ' Logarithm                     ',T(15)
      WRITE(6,1) ' Cosine                        ',T(16)
      WRITE(6,1) ' Inverse cosine                ',T(17)
      WRITE(6,1) ' Sine                          ',T(18)
      WRITE(6,1) ' Inverse sine                  ',T(19)
      WRITE(6,1) ' Sum ( (V-V)**2 )              ',T(20)
      WRITE(6,1) ' Gaussian random no.s          ',T(21)
      WRITE(6,1) ' Vector swap                   ',T(22)
      WRITE(6,1) ' Dot product                   ',T(23)
      WRITE(6,1) ' INT(V*scale)+1                ',T(24)
      WRITE(6,1) ' Gather                        ',T(25)
      WRITE(6,1) ' Scatter                       ',T(26)
      WRITE(6,1) ' Maximum magnitude element     ',T(27)
      END
c
      SUBROUTINE SRAND(IS,C)
C RJH 16/10/87
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,ISS2
5D-10 DATA IMULT/16807/,IMOD/2147483647/,SCALE/4.6566128/
C IS = MOD(IS*16807,2**31-1).
      IF(IS.LE.0) IS = 1
      IS2 = MOD(IS,32768)
      IS1 = (IS-IS2)/32768
      ISS2 = IS2 * IMULT
      IS2 = MOD(ISS2,32768)
      IS1 = MOD(IS1*IMULT+(ISS2-IS2)/32768,65536)
      IS = MOD(IS1*32768+IS2,IMOD)
      C = SCALE * FLOAT(IS)
      END

      SUBROUTINE SUB0
      END

      SUBROUTINE SUB1(X)
      END

      SUBROUTINE SUB2(X,Y)
      END

      SUBROUTINE SUB3(X,Y,Z)
      END

      SUBROUTINE SUB4(X,Y,Z,Q)
      END

      subroutine cpu(time)
      real*8 time
C return current cpu clock time in seconds
      time=float(mtime())/1000.0
      end
```

Appendix B. CCTA Whetstone Benchmark

We have run the CCTA program fovp12, 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 i2=10, nvn=16, nv=1,2,4,8,16,32,65,96,128,129,192,256,257,384,512

	a	b	c	d	e
Intel iPSC/2 sx	1345.406	1.4400	1.7732	1.7818	0.3679
iPSC/2 vx (f)	435.359	0.1876	4.7201	6.9769	1.4980
Meiko T800	1899.710	0.9423	1.2424	1.2479	0.4463
Convex C2 -O1 (g)	220.063	7.5220	10.7062	10.8092	4.0253
Convex C2 -O2	35.440	3.6293	66.0030	71.7477	20.9120

a) total time for run, in seconds
b) MWhets vector length 1
c) Mwhets vector length 64
d) MWhets vector length 512
e) Mflop performance in loop n2, vector length 512
f) The Intel vector code was prepared using VAST2 and VecLib with the following sequence of commands:

```
mv bench.f bench.v
vast2 -o bench.f bench.v
f77 -c -sx -vx bench.f
f77 -o bench.out bench.o -sx -vx -vec -node
getcube -t 1vx
load bench.out
```

g) compilation option form 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 semaphore system mentioned in the text. Best times are given for an otherwise nearly empty machine.

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

Occam message passing is best for short messages, whereas the Intel is better for long messages, only one example of 80 kbyte messages is shown below. The former yields (15 + 1.2*nbytes) microsecs in Occam, whilst the latter is (2000 + 0·47*nbytes) microsecs 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 kbyte 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
		(Fortnet)		3L	SUN CStools
	(a)	(b)	(d)	(d)	(e)
node initiated disk operations read or write nlen*8 bytes nloop 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 nloop times					
host write to host	3.410	57.885	45.653		
host read from host	4.840	53.637	33.655		
host write to node	0.340				
host read from node	0.160				
local memory operations copy b=c of len*8 bytes nloop*2 times					
node memory copy	1.094	1.564	1.728		
host memory copy	1.940	2.598	1.728		
remote memory operations, send and receive a message packet of len*8 bytes nloop 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	9.364		
host to node 1	0.310	13.588	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 cio library calls, which use cacheing on the i/o 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 delendence on message length, is given by L.Bowmans 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-cyle access memory. The node processes run on the mk060 board with T800 transputers and two-cycle 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 MK060 boards having T800 processors running MMVCS and MeikOS.

(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
*       r.a.      3/7/89
**************************************************************
c     program ctime
c version for Intel iPSC/2 with concurrent i/o
      IMPLICIT REAL*8(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)
      character*2 ctype
      character*8 cname
      DATA T/30*0.0D0/
      data ctype,cname/'4 ','ctime   '/
      open(2,file='ctime0.dat',status='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 eikonxs package
      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)
      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(finish)
      t(1)=finish-start
      rewind(2)
      call cpu(start)
      do 10 i=1,nloop
```

```
          read(2)b
 10       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(3)
       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 i=1,nloop
          call read(3,b,len*8)
 20       continue
       call cpu(finish)
       t(4)=finish-start
c signal clearance to continue
       call semout(3)
c send and receive message to node 0, timing their and back
       call cpu(start)
       do 35 i=1,nloop
          call csend(35,b,8*len,0,0)
          call crecv(35,b,8*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,8*len,1,0)
          call crecv(36,c,8*len)
 36       continue
       call cpu(finish)
       t(7)=finish-start

c write out 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,g12.5)
       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,ihost
      call csend(itype,idum,0,0,0)
      end
c
      subroutine semain(itype)
c node process to synchronise with host
      common/talk/inode,nnode,ihost
      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 srm or remote host
c as an example of their use look at the eikonxs programs
c 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 filin
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,ihost
      call csend(334,lu,4,nnode-1,1)
c handshaking
      call crecv(334,dum,0)
      end
c
      subroutine rewind(lu)
      common/talk/inode,nnode,ihost
      call csend(331,lu,4,nnode-1,1)
c handshaking
      call crecv(331,dum,0)
      end
c
      subroutine read(lu,buffer,nbytes)
      dimension buffer(*)
      common/talk/inode,nnode,ihost
      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
```

```
      end
c
      subroutine write(lu,buffer,nbytes)
      common/talk/inode,nnode,ihost
      dimension buffer(*)
      iproc=nnode-1
      call csend(332,lu,4,iproc,1)
      call csend(332,nbytes,4,iproc,1)
      call csend(332,buffer,nbytes,iproc,1)
c handshakeing
      call crecv(332,dum,0)
      end
c
      subroutine close(lu)
      common/talk/inode,nnode,ihost
      call csend(335,lu,4,nnode-1,1)
      call crecv(335,dum,0)
      end
c
      subroutine copy4(idim,imax,jmax,ain,buffer)
c fill up buffer for read/write emulation routines for 4-byte variabl
      dimension ain(idim,*)
      dimension buffer(*)
      k=0
      do 5 j=1,jmax
         do 5 i=1,imax
            k=k+1
            buffer(k)=ain(i,j)
 5    continue
      end
c
      subroutine copy8(idim,imax,jmax,ain,buffer)
c fill up buffer for read/write emulation routines for 8-byte variabl
      real*8 ain(idim,*)
      real*8 buffer(*)
      k=0
      do 5 j=1,jmax
         do 5 i=1,imax
            k=k+1
            buffer(k)=ain(i,j)
 5    continue
      end
c
      subroutine un4(idim,imax,jmax,ain,buffer)
c fill up buffer for read/write emulation routines for 4-byte variabl
      dimension ain(idim,*)
      dimension buffer(*)
      k=0
      do 5 j=1,jmax
         do 5 i=1,imax
            k=k+1
            ain(i,j)=buffer(k)
 5    continue
      end
c
      subroutine un8(idim,imax,jmax,ain,buffer)
c fill up buffer for read/write emulation routines for 4-byte
c variables
      real*8 ain(idim,*)
      real*8 buffer(*)
```

```
      k=0
      do 5 j=1,jmax
         do 5 i=1,imax
            k=k+1
            ain(i,j)=buffer(k)
 5    continue
      end

c     program ctime
c node program for Intel iPSC/2 with cfs
      IMPLICIT REAL*8(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*0.0D0/
      common/talk/inode,nnode,ihost
      inode=mynode()
      ihost=myhost()
      if(inode.eq.0)then
      open(2,file='/cfs/rja/ctime.dat',status='unknown')
      open(3,file='/usr/user/rja/tests/comms/ctime.dat',
     1    status='unknown')
c signal to host that node0 has started
      call semout(100)
      call cpu(start)
c write to cfs
      do 5 i=1,nloop
         write(2)a
 5       continue
      call cpu(finish)
      t(1)=finish-start
      rewind(2)
      call cpu(start)
      do 10 i=1,nloop
         read(2)b
 10      continue
      call cpu(finish)
      t(2)=finish-start
      call cpu(start)
c local node to node memory copy
      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 wait for clearance to use SRM
      call semain(1)
      call cpu(start)
      do 15 i=1,nloop
         write(3)a
 15      continue
      call cpu(finish)
      t(3)=finish-start
      rewind(3)
      call cpu(start)
      do 20 i=1,nloop
         read(3)b
 20      continue
      call cpu(finish)
      t(4)=finish-start
c signal finished with SRM
      call semout(2)
      end if
in c test comms between nodes, sending message their and back again (as
c 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 crecv(30,c,8*len)
            call csend(30,b,8*len,0,0)
         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.2)then
            call crecv(31,c,8*len)
            call csend(31,b,8*len,0,0)
         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,ihost,0)
         else if(inode.eq.1)then
            call crecv(36,c,8*len)
            call csend(36,b,8*len,ihost,0)
         end if
 35      continue
      if(inode.eq.0)then
      write(6,1)' node write to node ',t(1)
 1        FORMAT(1X,A,g12.5)
      write(6,1)' node read from node ',t(2)
      write(6,1)' node write to host ' ,t(3)
      write(6,1)' node read from host ',t(4)
      write(6,1)' local node to node memory copy ',t(5)
      write(6,1)' node 0 to node 1 send and receive ',t(6)
      write(6,1)' node 0 to node 2 send and receive ',t(7)
      end if

c signal end of execution
      call semout(4)
      end
c
      subroutine semout(itype)
c node process to synchronise with host
      common/talk/inode,nnode,ihost
      if(inode.eq.0)
```

```
     1    call csend(itype,idum,0,ihost,0)
      end

      subroutine semain(itype)
c node process to synchronise with host
      common/talk/inode,nnode,ihost
      if(inode.eq.0)
     1    call crecv(itype,idum,0)
      end
c
      subroutine cpu(time)
      real*8 time
      integer stime,mclock
      time=float(mclock())/1000.0
      end

c     program server
* *********************************************************************
c server program to run on node 0 in parallel with other processes,
c emulates calls to distributed file system from the host !!!!!
c  r.a.      15/6/89
c
c protocol for server file handling calls
c itype= 1 -- rewind
c        2 -- write
c        3 -- read
c        4 -- open
c        5 -- close
c
* *********************************************************************
      common/workspace/buffer(20000)
      idhost=myhost()
      inode=mynode()
      write(6,'('' .....server program up and running on node'',
     1    i4)')inode
c try to clear host-to-cfs i/o when this process swapped in
 10   do 15 i=1,5
         itype=330+i
         if(iprobe(itype).eq.1)goto(1,2,3,4,5),i
 15   continue
c no pending requests
carry on computing and try again at next swap
      call flick()
      goto 10
c read
c 3    write(6,'('' request to read from a file '')')
 3    continue
      call crecv(itype,lu,4)
      call crecv(itype,nbytes,4)
c user iPSC/2 high=speed i/o routine
      call cread(lu,buffer,nbytes)
      call csend(itype,buffer,nbytes,idhost,0)
      goto 10
c write
c 2    write(6,'('' request to write to a file '')')
 2    continue
      call crecv(itype,lu,4)
      call crecv(itype,nbytes,4)
      call crecv(itype,buffer,nbytes)
      call cwrite(lu,buffer,nbytes)
```

```
der !c handshaking signal to make sure messages are received in correct or
           call csend(itype,dum,0,idhost,0)
           goto 10
     c rewind
     c 1    write(6,'('' request to rewind a file '')')
      1    continue
           call crecv(itype,lu,4)
     c move file pointer
           ipoint=lseek(lu,0,0)
           call csend(itype,dum,0,idhost,0)
           goto 10
     c open
     c 4    write(6,'('' request to open file '')')
      4    continue
           call crecv(itype,lu,4)
           if(lu.eq.3)then
           open(3,file='/cfs/rja/ctime0.dat',status='unknown',
          1    form='unformatted')
           else
           write(6,'('' **** error server **** unknown file'')')
           end if
           call csend(itype,dum,0,idhost,0)
           goto 10
     c close
     c 5    write(6,'('' request to close a file '')')
      5    continue
           call crecv(itype,lu,4)
           close(lu)
           call csend(itype,dum,0,idhost,0)
           goto 10
           end
```

Index (numbers refer to sections rather than pages)

Ada I
algorithms III.1.1
Alliant III.1.2
ALT II.5, IV.2, IV.3, IV.7
Alvey project III.3
Argonne library I
asynchronous I, II
Atari II.4
AXIS II.1, IV.4
binary tree III.1.1
blocking I, II, IV.3
bottleneck I, III.1.3
C I, II.5, IV.5
CE/RK II.3, IV.2
channel II.5
client II.1
coarse-grained III.1.1
Concurrent Pascal I
configuration I, II.5, IV.3, IV.7
conveyor belt III.1.1
Convex IV.2, B
Cosmic Cube II.3
CPC library I
CRAY II, III.1.2
CSP I
CStools II.1, III.2, IV.3, B, C
DAP I
dbxtool III.2
DECON III.1.3, IV.2
debugger I, III.1.3, IV.3
divide and conquer III.1.1
Eiscube library III.1.2
Emacs II.5, IV.4
embedded system II.3, IV.7
emulator IV.2
EPL II.5
errors II
Esprit project 1085 IV.6
ETA II
Ethernet II.1
event IV.5
farming III.1.1, IV.2
fine-grained III.1.1
folding editor II.5
fork III.1.1
Fortnet IV.3, C
FORTRAN 8X III.1.2
FPS II, II.2, IV.1, A
geometric III.1.1, IV.1
harness II.5, IV.3, IV.5, IV.7
Helios II.4
hybrid memory emulation II.6, III.1.1
hypercube IV.1, IV.2, IV.4, see Intel, Ncube, Cosmic Cube
IDRIS II.1, IV.6
IBM II
Inmos II.4, II.5, IV.3, IV.6, IV.7
Intel II.5, III.1.2, III.1.3,

IV.2, IV.5, A, B
iPSC 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
Meiko II.5, IV.3
MeikOS IV.3
messages I, II.2, II.3, II.4
MIMD I, II, III.1.1, III.1.3
MMVCS IV.3
NAG library I, III.1.3
NCUBE II.1, II.5, IV.4, A
NiCHE 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.6, C
OPS II.5, IV.3
PAR II.5
ParsiFal III.3.1
Parsys 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 I, II.1
PRE II.1, IV.5
process I, II.2, II.3, II.4
protocol II, II.3, IV.2
queue II
Schedule III.3, IV.3
sdb III.2
segment III.1.1, IV.3
semaphore II.7, III, 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
skeleton III.1.1, III.1.2
slave III.1.1, IV.3
sockets II.1
sorting III.1.1, IV.3
spawn I, II.2, II.3
SPLIB II, II.3
SUN CStools II.1, IV.3
Supernode IV.6
SUPRENUM II.3, 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
Topologix 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.5, IV.6, IV.7, A, B
Transtech see Niche
Trillium, Trollius II.1, II.2,
IV.5, see Niche, FPS
UNIX I, II, II.1, II.2, II.3,
II.4, IV.2, IV.4, IV.5
VAX IV.3, A
VERTEX II.1, IV.4
X-windows II.4
3L II.6, IV.3, IV.6, IV.7, B