A transputer farm accelerator for networked computing facilities

INMOS Technical Note 54

Andy Hamilton
Central Applications Group Bristol

September 1988
72-TCH-054-00

Contents

1 Introduction

This technical note describes the use of INMOS transputers as end-application accelerators to a larger computing resource. As well as describing a specific implementation, some general ideas and arguments are discussed.

1.1 A modern trend

In contrast to the prominence of centralized computing facilities traditionally associated with large companies and institutions, nowadays the trend is towards desktop personal computers and networked diskless-node workstations.

This trend has been brought about by the decreasing cost and increasing performance of personal computers and networking options, which are being offered by a growing number of manufacturers.

By having desk-top processing power at one's finger tips, users have the response, flexibility and the control they want over the software they use. In addition, they are not as dependent on the loading and reliability (or otherwise) of a centralized machine and its communications network.

As an example of this, it is not at all uncommon to see a centralized cluster of VAX machines, spanned by networks, with various satellite MicroVAX's and DECnet-DOS personal computers. This network is frequently composed of several sub-nets, spanning the geographical distances between the sites of a company. A fairly typical company network is shown below in figure 1. In the figure, the geographical dispersion of the computers may extend across several buildings, towns, or countries.

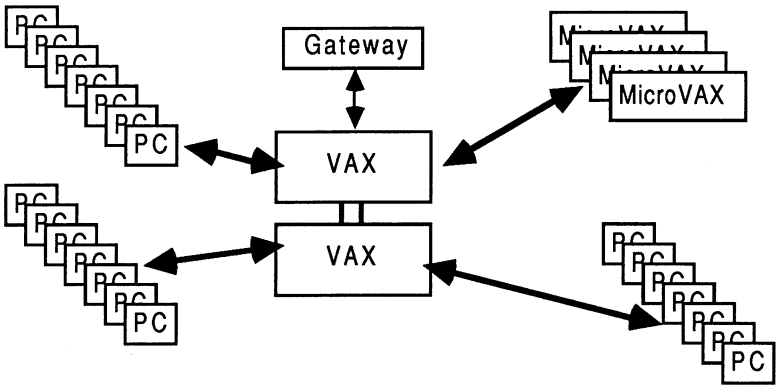

Figure 1: A typical computer network

1.2 Resolving the loading problem

Despite the proliferation of networked workstations and localized personal-computer processing power, the centralized resources are often overloaded with requests for large amounts of compute-power. Many tasks are just too large for the workstations and personal computers to handle in a reasonable timescale. It only takes a few such compilations and simulations to be concurrently executing on most machines to bring them to a virtual halt. How can this be avoided?

One approach would be to use an organization's existing infrastructure of networks and remote workstations to offload work from the centralized computers and MicroVAX's. This could be achieved by having, at any physical location on a connected network, a PC- or MicroVAX-hosted server, connected to a transputer farm capable of extensive number crunching. In the remainder of this document, the word 'farm' will be used to signify a collection of transputers, all executing the same application but on different data sets.

Such a system could be totally transparent to users of the services that most incapacitate the computing resources. They would invoke the application in exactly the normal way, except that the work would be performed remotely, by transputers, and the results would be returned shortly afterwards. Transputers can offer a previously impossible amount of compute-power in a small box.

A PC-hosted transputer server system can run existing applications, unmodified, and reduce loading from overworked machines, in a manner that is attractive because of it's flexible and infinite expandability. Further more, once the application has been ported to a transputer, it is independent of any of the other computer or communications equipment owned.

If you're still interested, read on ....

2 The systems involved

This chapter will discuss the hardware and software systems involved in the implementation to be described, with a view to placing the requirements and demands made of everything in some sort of perspective. The discussion focuses on specific systems, although the arguments are appropriate in a more general sense too.

The items that have to be discussed are the INMOS transputer, the transputer host, the centralized computing resource, and the communications

network. The arrangement is as shown below in figure 2.

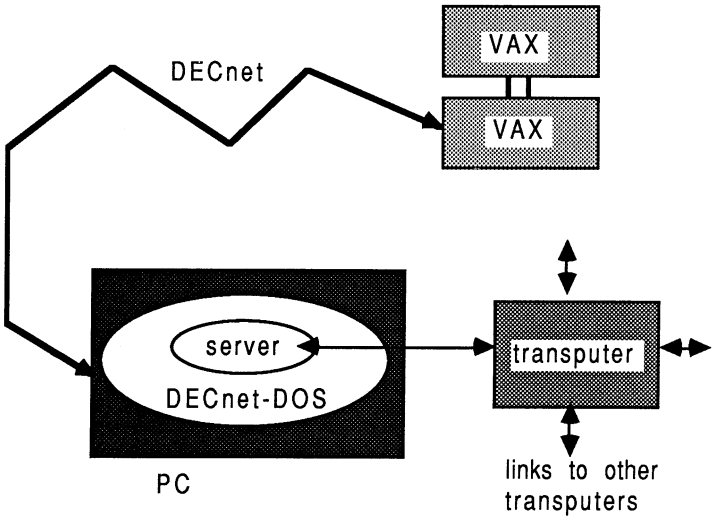

Figure 2: From the VAX to the transputer

2.1 The INMOS transputer

The INMOS transputer is a high performance micro-processor, offering a CPU, RAM, fast serial links, and various applications-specific facilities on a single chip of silicon. The IMS T800 transputer combines a 32-bit 10 MIPS integer CPU, a 1.5 MFlops 64-bit floating point processor (compliant to the ANSI/IEEE 754-1985 floating point arithmetic standard [1]), four 20 MHz serial links, and four kilobytes of fast single-cycle RAM (50ns access on the 20 MHz part). This technical note will make particular reference to the IMS T800, since the applications described make good use of the in-built floating point processor.

For a proper technical description of the INMOS transputer family, the reader is directed towards [2].

2.2 The transputer host

The transputer is normally employed as an addition to an existing computer, known as the host. In the context of this discussion, the host is a personal computer - an IBM PC or compatible. Through the host, the transputer application can receive the services of a file store, a screen, and keyboard.

The selection of the transputer host is important for two main reasons: firstly, it has to be able to accommodate and communicate with transputer

hardware; secondly, it has to be capable of appearing as a node on the network to the centralized computing resource. It should also possess some local file store such as a 20 Mbyte Winchester, although our implementation used a virtual disk on the VAX.

The IBM PC and most of its clones fulfil the main requirements.

2.3 The existing computing resource

This will typically consist of a centralized or distributed cluster of large mini-computers or main-frame computers, interconnected by one or more networks. The machines may all be of different type, manufacture, and specification.

In this technical note, the existing computing resources are represented by a pair of DEC VAX 11/785 mini computers, a number of DEC MicroVAX II's, and an Ethernet LAN (Local Area Network). The network involved is in fact more extensive than suggested here, but this document discusses the only relevant part of it.

2.4 The communications network

This section gives an overview of the network. In hardware terms, the relevant part of our network was an Ethernet network. An Ethernet network is a finite capacity shared-channel LAN. Sites on the network, called nodes, are connected by using vampire taps on a single coax cable.

In software terms, we used DECnet software to control the network activity.

DECnet Introduction

The term DECnet refers to a range of software that provides a network interface for Digital Equipment Corporation operating systems. A set of standards called the Digital Network Architecture defines the relationships between the various network components.

DECnet allows multiple computer systems to communicate and share resources within a network. Each computer system, called a node, is connected by some physical communications medium. Tasks that run on different nodes and exchange data are connected by logical links. Logical links are temporary software information paths established between two communicating tasks in a DECnet network.

DECnet-DOS is installed on a PC node in the network, on top of the existing

MS-DOS operating system. It is said to be a non-routing implementation of the Phase IV Digital Network Architecture.

DECnet concepts

A client task is the program that initiates a connect request with another task. The server task waits for and accepts/rejects the pending connect request. Client and server[1] tasks communicate through sockets. These tasks exchange data over logical links.

Sockets are the basic building blocks for DECnet-DOS task-to-task communication, and are created by tasks for sending and receiving data. They contain information about the status of the logical link connection.

Each system in a DECnet network has a unique node name and address. When initiating a connect request with a remote node; the node is identified by its name or address.

DECnet-DOS allows C and assembly-language programs to use sockets to perform DECnet functions. This allows a user application to communicate with another application running on a different node, using DECnet.

Refer to the DECnet-DOS programmer's reference manual [3] for further information.

2.5 How everything fits together

On local storage (or virtual disk) media, the PC will have copies of all the application code that may be required (suitably prepared for execution on the transputer). The relevant piece of application code is initially booted to the transputer network using a special PC file server / loader. The special server monitors DECnet instead of the PC's keyboard. This means that it can accept tasks over the network from the VAX automatically, and act as a completely un-attended autonomous system.

When the user wishes to send a task specification, the VAX software establishes a logical link by means of a connect request procedure. The logical link allows the exchange of data between the VAX and the PC server, because the PC has a unique node name and address on the network. The input data is forwarded to the transputer by the server, and the results are collected afterwards and returned to the originator of the task request.

Depending on the transputer hardware available in the system, several concurrent tasks can be underway at once. It is possible for all these tasks to

[1] The word server is used here in a different context from the rest of this document.

be different applications entirely. It's up to you, and is easily altered to suit the demands made on the transputer workers.

3 A specific implementation

This chapter describes a specific implementation of a DECnet hosted transputer server, undertaken at INMOS in Bristol. The system is extendible, transparent to the user, and involved little change to the application code. It is also simple to maintain from both hardware and a software point of view.

Firstly, an overview of the system is given, and then some design issues are discussed. Then, the PC, transputer, and VAX support detail is explored, followed by an outline of how the system was operated.

3.1 Overview

INMOS Corporation (Colorado) used a version of SPICE, written in FORTRAN, for intensive circuit simulation on a VAX. To relieve VAX CPU overhead, it was planned to offload SPICE jobs to a server which would simply return the results to the VAX. The system is referred to as Multi-SPICE.

The implementation consisted of three concurrently executing copies of SPICE each running on their own T800 transputer, shown in figure 3.

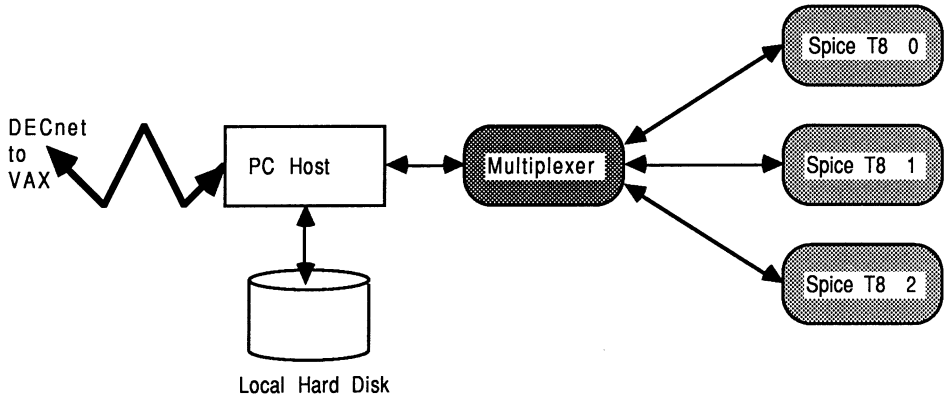

Figure 3: A four processor SPICE system

This level of parallelism granularity is at the job-level, in that there was to be no attempt to alter the code of SPICE itself to explicitly run parts of the application in parallel. Speedup can be achieved by means of adding

more processors, each running another completely independent version of the application code. The transputer network was hosted by a PC, from which it obtained local filing facilities. For this application, each SPICE worker packs about the same computation power as a VAX 11/785 with FPA support.

Each SPICE worker ran on a B405 TRAM module, offering an IMS T800-20 transputer with 8 Mbytes of RAM. For smaller simulations, the B404 TRAMs, which have 2 Mbytes of RAM with an IMS T800 transputer, were used. The connection to the IBM PC was by means of a single IMS B004 evaluation board with an IMS T414 transputer, which was used for the multiplexer.

3.2 System design notes

Requirements

The system had to be simple to operate and maintain, capable of operating with several transputers in a processor 'farm', capable of integrating additional applications, it had to be extensible, and it had to offer a useful service to the users. In the event of the application crashing, the system should endeavour to recover itself automatically. This had a bearing on the network topologies permitted and the distribution of support processes in the farm network. The interface to DECnet would be through the PC server, which would be modified to accommodate this requirement.

Some of these requirements are discussed below.

Overall system floorplan and development strategy

Each SPICE program runs on a separate T800 transputer, to obtain the maximum performance. In this and the following discussions, the term 'worker transputer' will be used to indicate a transputer which executes the target application.

Since the server would have to be modified anyway to communicate with DECnet, it was decided to delegate to the server the allocation of tasks to each worker transputer. A multiplexer was written in occam to ensure correct processor interleaving when communicating with the server.

Automated failure recovery and network topology implications

The occam multiplexer sits between all tasks on the worker transputers and the host server, as shown in figure 3. As such, it is aware of every inter-communication between any worker and the server. In addition to performing message interleaving between all the workers, it provides timeout facilities to identify if any worker has been 'silent' for greater than some specified time interval. This information can be used by the server to notify the VAX if any jobs fail to complete for some reason. This maintains the reliability and throughput of the system in the event of a partial failure, and allows a graceful degradation of the system. Individual jobs do not see any degradation.

The requirement for this capability arises from the fact that the SPICE program does occasionally crash while performing a simulation (not just on transputer systems).

By placing each SPICE on a transputer which is separate from the multiplexer transputer used by other worker transputers, it can be guaranteed that should any SPICE job crash a transputer, the multiplexer can detect this and take action to complete current jobs and re-boot the network automatically. The guarantee of automatic recovery from a crashed task arises from the fact that the multiplexer has a transputer to itself (which excludes parasitic shared-memory problems with rogue processes on other transputers), is written purely in occam and is significantly less complex than SPICE - it will not crash.

One consequence of the 'separate multiplexer per transputer' approach, as well as the cost, concerns the maximum number of SPICE workers that can be connected without compromising the recovery capability of the system. Because each transputer has four links, it can multiplex three inputs down to one output. So, up to three SPICE processors can be run with one multiplexer.

By cascading multiplexers, additional hardware links become available to accommodate more task processors. As an example, up to three 'worker multiplexers' (denoted Wkr Mux in figure 4) may be controlled from an 'intermediate multiplexer' (denoted Int Mux), and each worker multiplexer can of course accommodate three SPICE tasks. The distinction between the multiplexer types is as follows: a worker multiplexer connects directly to application workers and has the additional role of timeout monitoring to detect 'dead' workers; the intermediate multiplexer connects only to occam worker multiplexers and does not require to perform timeout detection.

This design of point-to-point communication allows for more T800s to be

controlled if necessary, and also serves to contain any 'failed' SPICE jobs, as discussed above. The multiplexers can be connected into quite complex configurations, providing that no implementation limits of the host, it's operating system, or the server are exceeded (for example, there may be a host operating system limit on the maximum number of files that can be open at once). Some possibilities are outlined in figure 4.

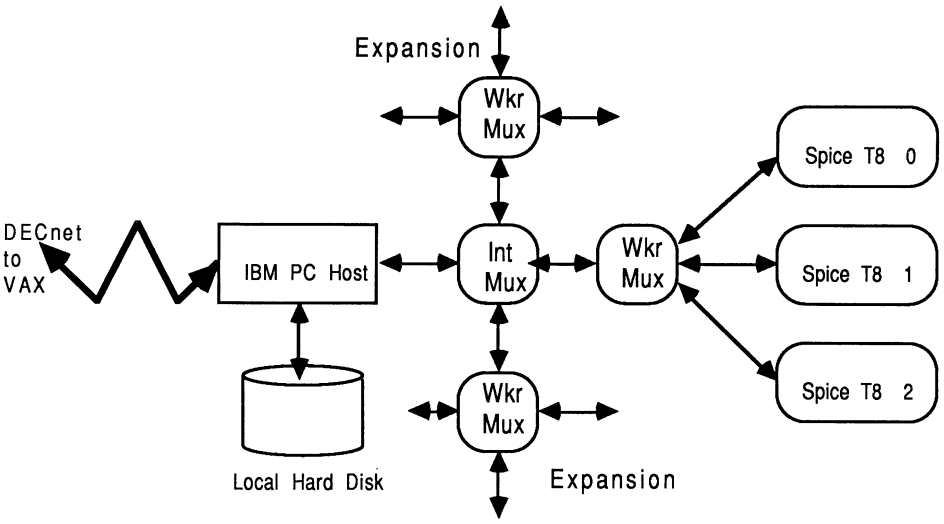

Figure 4: Multiplexer connectivity example

Had a pipeline-based topology been employed, this would have necessitated a multiplexer and SPICE to reside on each transputer. While this is straightforward to arrange, it is conceivable (although improbable) that should a SPICE task fail, this sharing of hardware by a rogue process and a healthy one might prevent the multiplexer from sending a failed signal to the host server, which would thereby prevent the network from automatically rebooting.

In most practical systems, however, this requirement for failure recovery can be satisfied to an almost equivalent level without the need for a transputer per multiplexer - this is useful in keeping the cost of the system realistic. The method we adopted is only one of several options we could have chosen.

3.3 PC support

The PC support is provided by modifications to the standard file server / loader program, called AFserver. These modifications allow the server to control bi-directional communications between the PC and the VAX, via DECnet.

An outline of the PC server

The PC server has two important functions to perform. Firstly, it must communicate with the transputer board, which will send requests for work. Secondly, it must communicate with DECnet and transceiver work requests and results between the PC file-store and the VAX.

The original PC server, called AFserver, is written in C. It consists of a small collection of functions which allow communication between the transputer system and the PC host. Normal server functions include file access and stream management etc. This communication is implemented using an INMOS link-adaptor, which interfaces a transputer link to the host PC's bus.

The small collection of routines provided by the server are grouped together in a flexible way, facilitating 'hooks' for adding additional commands into the body of the server. In different situations, where a lot of application code has to remain on the PC host, the source of the AFserver could be built into the PC-part of the application [4].

The transputer and the host conform to a master / slave relationship. The transputer is the master, implying that that all commands, which form part of the so-called 'AFserver protocol', are initiated from the transputer system. The function that decodes the command coming from the INMOS link-adaptor (connected to the transputer system) is called `read_link()`. This is outlined below:

```
void read_link ()
/* Read a message coming down the link. */
{
    if (read_integer (&command))
    {
        switch (command)
        {
            case TERMINATE_CMD:
                filer_close ();
                write_integer (F_OK);
                terminate_server (T_TERMINATED);
                break;
           .
           .
           .
            default:
                terminate_server (T_ILLEGAL_COMMAND);
        }
    };
}
```

It should be apparent how additional cases can be added to the AFserver protocol to accommodate the user's specific requirements, by providing further alternatives in the switch statement.

The design of the server, in connection with the protocols involved in managing interactions with the transputer on the one hand, and interactions with DECnet on the other hand, is now discussed.

Server extensions

It was decided to change the transputer's AFserver protocol as little as possible, and make the PC server contain all the DECnet accessing software (although this has the effect of making the PC server less general purpose). Only two additional commands were added to the AFserver's protocol. These were:

- FinishedTask.Cmd : Sent to the server from a SPICE worker transputer to identify a completed task and request more work. Parameters identify the current task completed, and the name of the new task to be started is returned.
- Failed.Cmd : Sent to the server by the occam multiplexer, following an extended period of inactivity by a SPICE worker. There are no parameters for this command.

The operation of the SPICE system, and its use of these commands is now outlined.

System operation

The FinishedTask.Cmd is sent to the server from an IMS T800 transputer running a SPICE task. When idle, each SPICE sends a FinishedTask.Cmd command with a null filename every second. If a SPICE has just completed a task, it sends the name of the completed task as a parameter to FinishedTask.Cmd. When the server receives a FinishedTask.Cmd, it checks the filename parameter. If the filename was a valid one, the server copies the output file to the VAX. Then, regardless of the filename parameter, the server polls DECnet (on the listening socket) to see if any new connect requests have been received. If so, the connect request is accepted, the logical link is attached to the data socket, and transfer can occur. The new file name is read, the file is copied onto local disk, and the local file name is sent to the available SPICE worker.

The server initially allocates a socket on which to listen for incoming connect requests (for task number 242 - see the implementation notes in the next section). It maintains a local list of file names, their corresponding VAX destinations, and their data socket numbers. This is because SPICE tasks can originate from any of several VAX's on the network, from different users, and from different directories, so the local filename is insufficient information to allow the results to be returned to the sender.

The Failed.Cmd, is sent to the server when an occam multiplexer believes that a SPICE simulation has failed. After receiving this command, the server does not accept any new connect requests. The server doesn't know which SPICE task has failed until the other tasks finish or fail, because it does not maintain a correspondence list of transputers executing specific tasks. When all current tasks complete, or fail, the server reboots the transputer network and sends messages down the remaining logical links to explain the failure to the VAX users. The whole network has to be rebooted because of the use of the standard Subsystem ports on the evaluation boards - it is not easy to reboot individual transputers in the network.

The server only polls DECnet when there is a transputer available to do work. The main reason for doing this was to improve the file server response for the transputer system, since polling DECnet is time consuming and only needs to be done when a SPICE processor is waiting for a task filename.

Implementation of the new server commands

This section discusses in some detail the C extensions written for the AF-server.

The `read_link()` function is shown here with the two additional hooks to implement the extra AFserver protocol tags, called `FINISHEDTASK_CMD` and `FAILED_CMD`:

```
void read_link ()
/* Read a message coming down the link. */
{
    if (read_integer (&command))
    {
        switch (command)
        {
            case TERMINATE_CMD:
                filer_close ();
                write_integer (F_OK);
                terminate_server (T_TERMINATED);
                break;
           .
```

```
            .
            .
            /* new AFserver protocol commands for SPICE farm */
            case FINISHEDTASK_CMD:
                finished_task ();
                break;
            case FAILED_CMD:
                failed ();
                break;
            default:
                terminate_server (T_ILLEGAL_COMMAND);
        }
    };
}
```

Consider the `FINISHEDTASK_CMD` first.

Implementing the FINISHEDTASK_CMD command

The `FINISHEDTASK_CMD` is sent by a SPICE application when it is available to do work. It is processed by the set of functions shown in figure 5. This figure also shows the hierarchy of these functions.

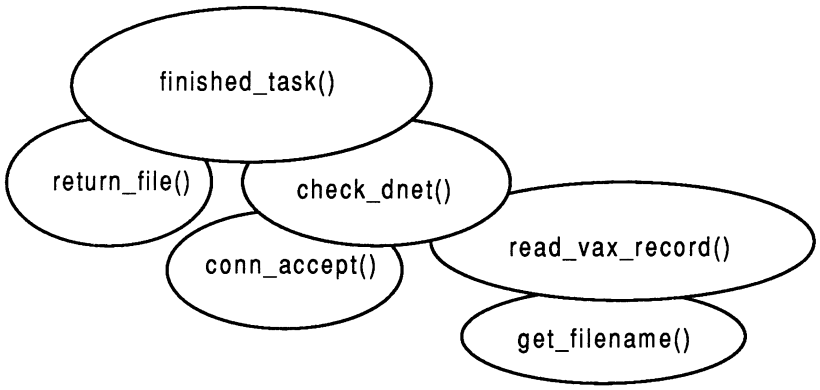

Figure 5: The FINISHEDTASK_CMD function hierarchy

When the server receives the `FINISHEDTASK_CMD`, it knows that there could be an output file ready to send to the VAX. If so, then it sends the file to the VAX using the `return_file()` function. Next, it polls DECnet to see if there are any connection requests. This is outlined in the `finished_task()` function

```
void finished_task()
    /* Just received FINISHEDTASK_CMD from worker multiplexer, */
    /* so send output file back to the VAX and check for a new */
```

```
    /* input file to be processed by the transputer farm.      */
{
    int block_size;
    char buffer [RECORD_LENGTH + 1];

    if (read_record (&block_size, buffer))
    {
        buffer [block_size] = '\0';
        /* buffer is the name of the finished task */
        if (block_size > 0)
            return_file(buffer, block_size);
        if (aborted)
        {
            write_record(0, "");
            write_integer(OPERATIONFAILED_ERR);
        }
        else
        {
            strcpy(buffer, check_dnet());
            /* about to write filename info to transputer */
            write_record(strlen(buffer), buffer);
            write_integer(F_OK);
        }
    }
}
```

Notice the use of the aborted flag in the above function, which is set as part of the `FAILED_CMD` handler. If the aborted flag is set, then the network will shortly reboot so no further polling of DECnet is entertained.

If the system is still allowed to poll DECnet, then it does so using the `check_dnet()` function

```
char *check_dnet()
    /* Check DECNet for incoming connect requests, */
    /* returns pointer to a filename or NULL       */
{
    struct timeval
    {
        long tv_sec;  /* seconds */
        long tv_usec; /* and microseconds */
    } tim;
    unsigned long read;
    int nfds;
    int i;
    int ready_bits;

    tim.tv_sec  = 0;
```

```
    tim.tv_uses = 25; /* check for activity for 25 microsec */
    read = 1<<sock_no;
    nfds = sock_no + 1;

    if (keyb_input())
        net_err(NULL, 0);
            /* check active sockets for input */
    if (select(nfds, &read, 0, 0, &tim) > 0)
        conn_accept();
            /* conn request received, wait for fnam */
    read = in_use_mask;
    nfds = MAX_SLAVES + 3;
    ready_bits= select(nfds, &read, 0, 0, &tim);
    if (ready_bits > 0)
        return( read_vax_record( read));

    if (ready_bits < 0)
        net_err("Cannot select data sockets:\n",errno);

    return( NULL );
}
```

The `check_dnet()` function polls DECnet and listens for connect requests. If it receives a connect request, it uses low level DECnet socket interfacing commands in `conn_accept` to establish a data logical link over a new data socket. The routine uses a number of global variables, most of which are concerned with managing the available/used DECnet sockets. `check_dnet()` uses `read_vax_record()` to read one of the sockets specified, display the record on the console, and return a pointer to valid local filename. This is done using `get_filename()` which performs NFT (Network File Transfer) commands to copy the input file from the VAX to the PC.

Implementing the FAILED_CMD command

The `FAILED_CMD` is received by the server from an occam multiplexer, rather than from a SPICE application - how could a SPICE application know it had failed if it was, itself, out of control? It is used to set a global flag in the server called aborted to prevent any further polling of DECnet for new tasks.

```
void failed()
    /* Just received a FAILED CMD from worker multiplexer,   */
    /* so set aborted flag to prevent further DECnet testing */
{
    write_integer(F_OK);
    aborted = TRUE;
    active_task_count-- ;
}
```

Initializing and closing down

There is, of course, a lot of additional code required to initialize DECnet and to close things down in an orderly manner.

For example, the following initialization sequence uses DECnet-DOS socket interface calls, and is called as part of the booting sequence:

```
void decnet_init()
    /* Initialise the DECNet side of things,               */
    /*  - return the socket number of the listening socket */
{
    slaves_init();
    if (system("NCP SET KNOWN LINKS STATE OFF"))
        net_err("NCP Call failed:\n",errno);
    printf("\n\t Inmos PC Server, Version 2.0\n\n");

    /* open a DECnet socket */
    if ((sock_no = socket( AF_DECnet, SOCK_SEQPACKET, 0 )) < 0)
        net_err("Socket allocation failed:\n",errno);
    /* bind an object num to the socket */
    bzero( &socket_char, sizeof( socket_char));
    socket_char.sdn_family = AF_DECnet;
    socket_char.adn_objnum = 242;
    /* 242 is DECnet server task number */

    if (bind( sock_no, &socket_char, sizeof( socket_char)) < 0)
        net_err("Bind to socket failed:\n",errno);
    /* listen for connect requests */
    if (listen( sock_no, backlog) < 0)
        net_err("Listen failed:\n",errno);
    return;
}
```

The following function is used to close all active sockets before the transputer system is rebooted after `FAILED_CMD`.

```
void remove_socks()
    /* Shut down all sockets still active, and */
    /* tell Vax user that his job has failed   */
{
    int i;
    char msg[MAX_BUY_SIZE];
    char command buff[MAX_BUY_SIZE];

    for (i=1; i<MAX_SOCK; i++)
        if (file_names[i].filename != NULL)
        {
```

```
            strcpy(msg,"Abnormal Completion for file : ");
            strcat(msg,file_names[i].file_name );
            strcat(msg,"\n");
            swrite(i,msg,strlen(msg));
            sclose(i);
            strcpy( command_buff, "NFT COPY ");
            strcat( command_buff, file names[ i ].file_name);
            strcat( command_buff, ".OUT " );
            strcat( command_buff, file names[ i ].full_spec);
            strcat( command_buff, ".OUT >>NFT.LOG" );
            system( command_buff );
        }
}
```

Hopefully, the above functions give some appreciation of the work involved in this part of the project - about 30k (source size) of specially-written C was required to interface to the DECnet-DOS software at the PC end. This amounts to more than half of the source size of the original AFserver. [3] gives useful examples and guidance for doing this type of work.

Once the appropriate network communications software exists for the environment, the stages to incorporate it into the AFserver software are trivial.

3.4 Transputer support

In arranging for an application to be incorporated into an 'autonomous worker' environment, there are two options concerning the amount of transputer support required. These are directly related to the mechanism of how new tasks are allocated to workers within the farm.

- New work tasks can be explicitly requested in the non-occam application code itself, and their dispensement can be controlled from the server. This results in minimal transputer-resident support software, because it removes the need for a transputer-resident farm controller (task allocator). A simple server-protocol multiplex is sufficient to interleave work requests to the server, and the server is extended to cope with additional protocols to handle work assignment in the farm and DECnet interfacing.
- A set of farm controller processes, written in occam, can be used to receive work tasks (from the DECnet server) and allocate them amongst available worker processors. This approach is the more general-purpose of the two, because it is completely host-independent, and it obviates the need to modify the application code or the host server-transputer

protocols. The application does not need to know it is in an autonomous working environment, or that it may be one of several running on the same transputer network.

In both cases, a process on a transputer is responsible for requesting more work from the server. In the first instance, the application itself directly asks the server for work. In the second case, an available worker application asks the transputer-resident farm controller for work, and the farm controller then asks the server. In both cases, once the server receives a request for work, it would check DECnet for any pending requests.

In our implementation, the first of these two options was selected. The application was slightly modified to 'ask for work' from the server, and the server assumes the responsibility of dispensing tasks.

Modifications to the application

Like SPICE itself, the modifications were written in FORTRAN, and placed around the 'root' part of the application. The modifications simply concerned the requesting for work (using the extended AFserver protocol defined earlier), and the establishment of data input and output file names for each simulation to run. It is an obvious requirement that since each SPICE worker is served from the same file store, the local file names being processed concurrently must not clash with any others. This is handled by the server.

The message-passing routines provided by the run-time libraries supplied with all the scientific-language compilers are used to communicate with the PC server, using the newly defined protocol. These protocols handle the requests for new work, and have already been explained at the host server end. Here, at the application end, the standard FORTRAN message passing routines called `CHANINMESSAGE()`, `CHANOUTMESSAGE()`, `CHANOUTBYTE()`, and `CHANOUTWORD()` are used [6], as shown overleaf in our implementation for the SPICE farm.

```
C     ************************************************************
C     * Communication with DECnet file Server                    *
C     ************************************************************

      SUBROUTINE ReadInteger( N )
        INTEGER Tag
        CALL CHANINMESSAGE( 1, Tag, 1 )
        CALL CHANINMESSAGE( 1, N, 4 )
        RETURN
      END
```

```
      SUBROUTINE WriteInteger( N )
        INTEGER Int32Value
        PARAMETER (Int32Value = 4)
        CALL CHANOUTBYTE( Int32Value, 1 )
        CALL CHANOUTWORD( N, 1 )
        RETURN
      END

      SUBROUTINE ReadRecord( Len, Record )
        INTEGER NilRecordvalue, Record32Value
        INTEGER Tag
        PARAMETER (NilRecordValue = 8, Record32Value = 12)
        DATA Tag / 0 /
        CALL CHANINMESSAGE( 1, Tag, 1 )
        IF (Tag.EQ.NilRecordValue) THEN
          Len = 0
        ELSE
          CALL CHANINMESSAGE( 1, Len, 4 )
          IF (Len.GT.0) CALL CHANINMESSAGE( 1, Record, Len )
        END IF
        RETURN
      END

      SUBROUTINE WriteRecord( Len, Record )
        INTEGER NilRecordValue, Record32Value
        PARAMETER (NilRecordValue = 8, Record32Value = 12)
        IF (Len.EQ.0) THEN
          CALL CHANOUTBYTE( NilRecordValue, 1 )
        ELSE
          CALL CHANOUTBYTE( Record32Value, 1 )
          CALL CHANOUTWORD( Len, 1 )
          CALL CHANOUTMESSAGE( 1, Record, Len )
        END IF
        RETURN
      END

      SUBROUTINE FinishedTask( SizeOldTaskName, OldTaskName,
     1                         SizeNewTaskName, NewTaskName,
     2                         Result )
        INTEGER FinishedTaskCmd
        PARAMETER (FinishedTaskCmd = 127 )
        CALL WriteInteger( FinishedTaskCmd )
        CALL WriteRecord( SizeOldTaskName, OldTaskName )
        CALL ReadRecord( SizeNewTaskName, NewTaskName )
        CALL ReadInteger( Result )
        RETURN
      END
```

The subroutine `FinishedTask()` makes use of one of the additional tags

to the AFserver protocol, called FinishedTaskCmd. It is used in the main top-level part of the application as follows:

```
      PROGRAM SPICE
        IMPLICIT NONE

        INTEGER MaxFileNameSize
        INTEGER SizeOldTaskName, SizeNewTaskName
        INTEGER OneSecond
        INTEGER Result
        PARAMETER (OneSecond = 120000, MaxFileNameSize = 20)
        CHARACTER*(MaxPileNameSize) OldTaskName, NewTaskName
        CHARACTER*(MaxFileNamesize) SpiceIn, SpiceOut

5       CALL Delay( Onesecond )


        CALL FinishedTask( 0, OldTaskName,
     1                     SizeNewTaskName, NewTaskName,
     2                     Result )
20      IF (SizeNewTaskName.EQ.0) GO TO 5

        SpiceIn  = NewTaskName( 1: SizeNewTaskName ) // '.in'
        SpiceOut = NewTaskName( 1: SizeNewTaskName ) // '.out'
        OPEN (UNIT=5,FILE= SpiceIn, STATUS='OLD')
        OPEN (UNIT=6,FILE= SpiceOut,STATUS='NEW')

        CALL SpicRoot()

        CLOSE (UNIT=5)
        CLOSE (UNIT=6)
        SizeOldTaskName = SizeNewTaskName
        OldTaskName     = NewTaskName
        CALL FinishedTask( SizeOldTaskName, OldTaskName,
     1                     SizeNewTaskName, NewTaskName,
     2                     Result )
        GO TO 20
       END
```

The line `CALL SpicRoot()` is the new call to the main SPICE application. Since the application has been delegated the responsibility of requesting more work, it has been made into a non-terminating work request loop. This means that once MultiSPICE is running, it will accept work continuously. Within this non terminating work request loop is a small delay, which is used to prevent an available worker from pestering the server continuously in cases where there is no new work, but where other workers may be busy. In [4], a general-purpose farming technique, which does not involve modifications to the application, is presented.

The occam harness

A SPICE worker is encapsulated by a small amount of occam, known as the harness. The harness is required to ensure that the FORTRAN application receives access to the server for filing, screen, and keyboard facilities. For a description of the occam language, developed to express and exploit the parallelism offered by the INMOS transputer, the reader is directed towards [5]. For information concerning the content and creation of an occam harness for non-occam programs, please refer to [6].

Due to MultiSPICE being an autonomous computing engine, none of the software is written to self-terminate. In this situation, either the application code or the harness must never terminate. Due to an earlier decision to modify the application to request work, the application itself was made non-terminating. This allowed the standard occam harness, as supplied with the scientific-language compilers and the D705A occam-2 toolset, to be used for each SPICE worker. All this work is still relevant in the context of the D705B occam 2 toolset. To stop MultiSPICE requires deliberate and specific user interaction on the host PC. Refer to section 3.6 for operation details.

The occam multiplexers

Running several SPICE jobs concurrently requires that their accesses to the host PC be dynamically inter leaved. This is most easily done in occam by having a single transputer that talks down the single channel to the AFserver on the PC.

A worker multiplexer sits between the IMS T800 transputers running the application code, and the server on the PC. It provides message interleaving and time-out services for each application transputer. The timeout is determined empirically. The multiplexer should sit on its own transputer, so as to preserve the crash-recovery capability of the MultiSPICE system. Its general structure is illustrated below:

```
WHILE TRUE
  BYTE Tag :
  SEQ
    ALT
      (NOT WorkerFail[0]) & FromWorker0 ? Tag
        ActOnMesage( FromWorker0, ToWorker0, 0, Tag )
      (NOT WorkerFail[0]) &
       Clock ? AFTER LastInput[0] PLUS InactiveDelay
        ActOnTimeOut( 0 )
```

```
(NOT WorkerFail[1]) & FromWorker1 ? Tag
  ActOnMesage( FromWorker1, ToWorker1, 1, Tag )
(NOT WorkerFail[1]) &
 Clock ? AFTER LastInput[1] PLUS InactiveDelay
  ActOnTimeOut( 1 )

(NOT WorkerFail[2]) & FromWorker2 ? Tag
  ActOnMesage( FromWorker2, ToWorker2, 2, Tag )
(NOT WorkerFail[2]) &
 Clock ? AFTER LastInput[2] PLUS InactiveDelay
  ActOnTimeOut( 2 )
```

This code fragment has the effect of allowing the first SPICE worker requiring access to the server, exclusive use of the server for a single AFserver protocol transaction. It also allows for inactivity timeout monitoring on any worker, and prevents workers previously identified as 'dead' from further servicing by the system.

Not shown here is the use of occam's PRI ALTS, which can be used to ensure fairness of servicing worker requests for all the participating workers.

The `ActOnTimeOut()` procedure is responsible for the origination of the Failed.Cmd. The `ActOnMesage()` procedure uses the InputOrFail.t and OutputOrFail.t communications procedures [7], allowing controlled recovery from failure of transputer link input / output.

The parameter InactiveDelay is designed to trap a crashed simulation. The nature of SPICE is such that it has a high computation to communication ratio. This means that relatively long periods of time can elapse between communication bursts and any observable link activity to the host server. Occam allows simple handling of timeout issues in comparison to other high- and low-level languages. Occam has constructs to allow the reading of the transputer's timers, and to cause delays until certain periods have elapsed. The transputer has two timers, one accessible during high priority execution, and the other accessible during low priority execution. In high priority execution, the timer tick once every micro-second. In low priority, the timer ticks once every 64 micro-seconds.

The InactiveDelay is set to correspond to a time interval of around an hour or so. It's value is specified in the occam configuration description for the system, and passed in to each processor as parameter. This allows its value to be changed easily without recompiling anything. The timeout period is calculated to be larger than the longest time taken by the largest simulation intended to run on MultiSPICE.

The simple design of the multiplexer software means that while a Finished-Task.Cmd request from a SPICE processor is being serviced, the server

blocks any of the other transputers from receiving or sending data to / from the PC. While this avoids routing overheads within the SPICE array (because there is no need to pass source and destination information with the message and explicitly route it within the farm), it also means that SPICE applications can remain unserviced for several minutes while file transfers over the Ethernet are taking place. This is not as severe a problem as it might seem, because only one device can have access to the file store at once anyway.

The multiplexer software can be compiled for execution on the 16-bit T212/T222 transputers, or for the 32-bit T414/T425 transputers - there is no real need to use a T800 here.

The afore-mentioned intermediate multiplexer connect only to occam worker multiplexers and does not require to perform timeout detection. This is because the worker multiplexers, by virtue of their design, will always be capable of identifying inactivity problems with any of their applications. Therefore the intermediate multiplexer need only have the capability to through-route timeout messages to the server.

3.5 VAX support

The VAX is the main central computing resource in this system. The PC has a unique node name and address on DECnet. When the user wishes to send a task specification, the VAX establishes a logical link which allows the exchange of data between the VAX and the PC server.

Two DCL[2] command files on the VAX were written to arrange for the SPICE input file on the VAX to be sent to the PC node on DECnet. This input file is then sent to an available transputer by the server.

The first command file, called SPICE.COM, receives from the user the name of the SPICE input deck to send to the PC server, the destination node, and the password. It then spawns a subprocess to do the actual data exchanges. Our implementation of SPICE.COM is shown below.

```
$ ! Command file used to talk to IBM PC SPICE Server
$ ! P1 is node name of target PC
$ ! Prompts for users password and list of filenames (1 per line)
$ ! A subprocess is spawned for each filename given.
$ ! All subprocess messages are displayed on the screen.

$ set NoOn
$ on control_c then goto cleanup
```

[2]DCL - Digital Command Language

```
$ IF P1 .EQS. "" THEN Inquire P1 "Node"
$ Set ter/noecho
$ Inquire Password
$ Set ter/echo
$Loop:
$ inquire record "Filename"
$ if record .EQS. "" then goto loop
$ File_spec = F$Parse(record)
$ File_spec = File spec - " "
$ Node = F$Logical("SYS$NODE")
$ Colon = F$Locate(":",Node)
$ Node = F$Extract(0,Colon,Node)'
$ Node = Node - " "
$ User = F$Getjpi("","USERNAME")
$ Space = F$Locate(" ",User)
$ User = F$Extract(0,Space,User)
$ spawn /nowait -
@subproc 'P3' 'Node' 'User' 'Password' 'File_Spec'
$ goto loop
$
$cleanup:
$ Del/sym Password
$Exit
$
```

The subprocess spawned by SPICE.COM opens a logical link to the PC (a DECnet node) and specifies the 'task number' to run at the PC. The task number specified is 242. Since the PC can only run one task at a time (the server), it has to be running this task before the VAX attempts to talk to it.

Here is the code for our implementation of the SUBPROC.COM DCL file

```
$ !Called by SPICE.COM to communicate with PC Server
$
$ set NoOn
$ open/read/write link 'P1'::"242="
$ write link "''P2'""''P3' ''P4'"":: ''P5'"
$ read link record
$ write sys$output "''f$getjpi("","PRCNAM")'"
$ write sys$output record
$ close link
```

The SUBPROC.COM file operates at a much lower-level than the SPICE.COM file that spawns it. It uses non-transparent DECnet commands. The choice of a task number of 242 was completely arbitrary, but mainly because the first and ultimate choice, 42, was already in use by DEC. The selected identifier 242 must correspond to the task number in the special PC server,

otherwise communication between the PC server and the VAX will not be possible.

3.6 Operating the system

MultiSPICE receives jobs over DECnet from the VAX. It is important to run the PC server to boot the transputer network before any attempt at the VAX is made to send work; otherwise the system is likely to fail to establish communication before a VAX timeout takes effect.

Running MultiSPICE at the PC end

On the PC, the special server was called SERVER.EXE. It understands the same command-line parameters as the INMOS AFserver, so to boot the transputer system with the SPICE workers and multiplexer file, called spicfarm.bt, the following MS-DOS command could be used:

```
server -:b spicfarm.bt
```

It may be necessary to perform some one-off set-up commands concerning the PC and DECnet, for example, ensuring that there are sufficient 'file links'[3] available over the network.

MultiSPICE will now load, execute, and wait for incoming jobs. Our version was tested and operated with up to three SPICE Applications executing concurrently. The system as it stands will not accept job assignments from MS-DOS; only requests from DECnet are recognised.

Running MultiSPICE at the VAX end

The VAX is responsible for sending jobs to MultiSPICE, and retrieving the results. To run the VAX DCL command file, called SPICE, one could invoke the command by typing `@SPICE`.

The command file prompts the user for the name of the node on DECnet that MultiSPICE can be found, a password to allow access to the file on the VAX, and the filename of the job to be simulated. Everything else happens in the background without the user being aware of anything exciting happening, until a message is displayed on the terminal screen describing the final status of the job. The job is either successful or aborted.

[3]This term is used in the DECnet context, rather than a transputer links context.

As files are being transferred, the support software at both the VAX and the PC ends issues messages to indicate the current activities. For example, the VAX shows the task completion status message, and the PC shows file transfer messages. These are also written into a log file called NFT.LOG for post-mortem debugging, by redirecting the output of the DECnet-DOS NFT command.

4 Other considerations

4.1 Implementation guidelines

Tools required

To reproduce any of the work described in this document, the occam-2 toolset is required to create and manipulate the transputer components. A transputer compiler for the non-occam application will also be required. To modify the AFserver, which is supplied in C source form on the PC, a (Microsoft) C compiler for the host computer is needed. This will allow the compiled server module to be linked with the C library supplied with DECnet-DOS.

Suitable applications

INMOS provide scientific language compilers for C, Pascal, and FORTRAN. The INMOS development systems allow applications written in mixtures of these languages, including occam to be easily executed on a transputer system. The range of INMOS' scientific language compilers is growing constantly - please refer to [8] for current product availability.

The applications should preferably be batch-like in nature, i.e. they take an input file, perform some compute intensive operations, and produce output files, without user interaction or screen access.

Especially appropriate are applications in which the ratio of 'computation to communication' is high. This means that the overheads in sending the input data to the transputer, and receiving the results back over the network, are low in comparison to the amount of computation that is to be performed on the said input data. Typical applications that fall into the category include simulation packages (chemical, thermal, dynamic, electrical), technical modelling packages, compilers, and text formatting packages (e.g. TX, troff, PostScript processors etc).

The more interactive an application, the less suitable it is for the type of

implementation which is described here. This is due to the latency and overheads in transporting the interactive commands and replies, between the user at one end of the network, and the transputer server at the other end of the network. Network latency is concerned with the delay before processing starts, between the user invoking the command on his / her terminal, and the transputer worker starting on the job. It consists of the time taken to get the input data sent from the VAX to the transputer host. There is an additional small delay due to the time taken by the transputer to read the input work task from the local file store. The response times normally associated with real-time interaction may be unacceptable given these overheads.

At times of heavy loading, the network latency will increase and the transfer rates will correspondingly drop. This will depend on the nature of the network. In non-deterministic Local Area Network's (LAN's) like Ethernet, one can observe almost order of magnitude fluctuations in response time, depending on the instantaneous system loading. So remember, use only applications with a high ratio of compute-time to communicate-time.

Implementation strategy

To implement a remote transputer server of the form described in this document, the first stage is to get a single un-modified version of the application running on a transputer board. This may involve making small changes to the application in order to get it through the scientific language compilers that INMOS provide. This is not because of any particular deficiency in the INMOS scientific-language compilers, but rather because many applications tend to make use of non-standard language-extensions provided on their native environment compilers.

The result of this is software that can be used to obtain performance measurements of the application running on a transputer. [4] discusses some application porting issues.

The next stage would be to replicate the application over a number of transputer workers, using techniques described previously or in [4]. Alternatively, one may wish to have only a single transputer worker in the system.

Next, modify the server to communicate with the network, DECnet in this case. The DECnet-DOS programmer's reference manual [3] is invaluable here, giving examples of how to establish two-way communication between any two nodes on the network. If you are using a different network, there will be a corresponding technical reference manual. Test the server with a small stub of occam or C (for example), on the transputer, to be certain that something on the transputer network can request and receive the correct

information over the network.

Finally, combine the modified server with the real application, and everything should operate correctly. If this is successful, then you can go live!

Timescales

Timescales for a project such as this are difficult to estimate. Many factors are involved. For example, depending on the application's use of non-standard language features, there may be effort required to reimplement these parts of the application in a standard manner before the INMOS scientific-language compilers will accept the source input.

The time and effort to make a suitable server will depend on the available documentation describing the interfacing and protocols between the PC server and your network. In the case of DECnet, the examples given in this technical note should be of use.

4.2 Multiple task farms

If a farm is created which has several different tasks running, each on their own transputer, then it would be necessary for each job request to be accompanied by some means of identifying the task. In such situations, a need for additional hardware to allow individual transputers to be reset/loaded could be identified.

If the code for that task is not currently loaded onto the farm, then it must be fetched from local file store and loaded into the appropriate transputer. A discussion of how to organize such a system, and how it might be implemented, is given in [4].

4.3 Receiving work from DOS rather than DECnet

Work done in February 1988 by the INMOS Central Applications Group produced a version of MultiSPICE which accepts jobs in a batch fashion from MS-DOS, and can run an arbitrary number of concurrent SPICES by using a pipeline of tasks run on B404 TRAMs on a B008 motherboard [8]. (B404's have an IMS T800 and 2 Mbytes of dynamic RAM on a size 2 TRAM module). The system was more general purpose than the one described in this technical note, for two main reasons. Firstly, the SPICE application was not modified. Secondly, the task allocation was controlled from processes executing on the transputer array rather than the on the host, which makes the

software more portable and more applicable in different host relationships. [4] discusses this approach in detail.

4.4 Network monitoring software

Our network had a traffic-monitoring system program which could detect periods of inactivity on 'open connections', and log-off users or applications that had not corresponded with the VAX for a certain time.

One of the main reasons for choosing SPICE as a candidate for a dedicated remote application server is it's good ratio of computation to communication. SPICE in particular produces all its output in one go at the end of each simulation, and hence will tend to communicate over DECnet in bursts, separated by (possibly) extended periods of inactivity, rather than in any continuous fashion.

This temporal distribution of activity (as far as the VAX is concerned) caused a few headaches initially, resulting in the transputer workers getting logged off before the results were produced. Once the network monitor had been instructed correctly, there were no more unintentional detachments.

4.5 Other transputer hosts

More recently, a number of manufacturers have produced a range of transputer-based boards for use with a DEC MicroVAX II. If your network includes MicroVAX's, this might be a preferable route to follow compared to using a PC host for the transputer board(s). Factors of cost, available driving software, and network performance would have to examine - as well as that of technical challenge!

The scope of this work would also be appropriate in connection with Sun's NFS[4] environment.

4.6 Is it worth it? - Weighing up the pros and cons of using transputers

In an effort to establish the suitability of a hosted transputer server to reduce the loading on existing computer facilities, the following discussion may be relevant.

To implement the system described requires a node on the network to host

[4]NFS is a network file system developed by Sun Microsystems to allow machines of different types to share files.

a transputer motherboard. Such a node could take the form of a personal computer, with an Ethernet card. The personal computer can accommodate several transputer modules, all of which can be performing the same or different tasks. As a rough guide, a single B404 transputer module [8], consisting of a 32-bit IMS T800 transputer [2] and 2 Mbytes of dynamic RAM, can give the same performance as a VAX 11/785 with floating point accelerator hardware, when executing non-modified non-occam code [9].

If the application is written in a largely standard dialect of a supported-language, then it is unlikely that there will be major problems in 'porting' the application onto a single transputer, in a timescale that could be measured in days. Several such applications can then be run together in a farm. The best results will come from applications that are highly compute-intensive, perform only limited file access and perform no user-interaction.

The personal computer host can also be used as a normal PC at times when it is not used by the network users. The effort of running your software on a transputer means that the software can now execute on any machine hosting a transputer product; given a suitable server. This opens up useful portability opportunities that were previously not feasible. The unparalleled inter-connectivity of the INMOS transputer means that once the application is running on a single transputer, one can explore the possibilities for further performance increases by re-structuring parts of the software and by using additional hardware. Not least of course is the impressive performance one would obtain even when using only one transputer.

Everything reduces down to the question 'How much does CPU time cost me on my existing computing facilities?'. This cost has to include factors for equipment maintenance contracts, spares, upgrades etc. What could that time have been better spent on doing? Just think ... with a modest amount of effort, all the benefits discussed previously become a reality. So, 'is it worth it?'. Yes.

5 Summary and conclusions

The transputer farm accelerator described in this document has proved to be a powerful, reliable and cost effective dedicated application accelerator. For example, in general, a single transputer is more than one and a half times faster than the Sun with 68881 (especially on larger jobs), and on-par with VAX 11/785 (with FPA) performance [9].

Another advantage of this system is that the node being used to host the transputer need not always be used as a server in this way. It can still be used as a normal networked / standalone PC, running normal PC software,

and indeed, running non-networked transputer software on a demand basis.

As well as the cross-host application portability that the transputer offers, there is also great flexibility and extensibility at the system design level. More and more power and capability can be added easily at any stage, by interconnecting different combinations of transputers and memory. By using the INMOS module motherboards and TRAM modules, one can 'pick and match' the appropriate performance and memory requirements for each stage of a progressive and well defined upgrade path.

In addition, this work could be extended to cover different network communication software such as Suns' NFS, by altering only the code that ran on the host – the transputer part wouldn't even require re-compilation.

It's this modular and flexible extensibility that promotes the transputer as a candidate for dedicated 'grow as you do' transparent job-level application acceleration. This technical note has discussed only one scenario to which transputers can be applied that of using a network to offload work from over-used resources. There are many other scenarios possible; limited only by imagination.

So modular, so flexible, so powerful. So buy some.

5.1 Acknowledgements

Glenn Hill, Vic Dewhust, and Olive Dyson, all of INMOS Bristol, have performed the bulk of the work described in this technical note.

References

[1] Transputer Instruction Set - A compiler writer's guide, INMOS Limited, Prentice Hall.

[2] Transputer Reference Manual, INMOS Limited, Prentice Hall.

[3] DECnet-DOS V1.1 Programmer's Reference Manual, 'Example Socket Interface Calling Sequence', Section 4.4.1, pp 4-6.

[4] Issues in Application Porting and Farming, Technical Note 53, INMOS Limited.

[5] Occam 2 Reference Manual, INMOS Limited, Prentice Hall.

[6] Using the occam toolsets with non-occam applications, Technical Note 55, INMOS Limited.

[7] Extraordinary use of transputer links, Technical Note 1, INMOS Limited.

[8] IMMOS Spectrum, (contains a brief description of INMOS products), INMOS Limited.

[9] Porting SPICE to the INMOS IMS T800 transputer, Technical Note 52, INMOS Limited.