DL/SCI/TM81T

# technical memorandum

Daresbury Laboratory

DL/SCI/TM81T

## NUMERICAL ALGORITHM LIBRARIES FOR MULTICOMPUTERS

by

R.J. ALLAN, SERC Daresbury Laboratory



JULY 1991

G91/293

Science and Engineering Research Council

DARESBURY LABORATORY

Daresbury, Warrington WA4 4AD

ISSN 0144-5677

Science and Engineering Research Council,
Daresbury Laboratory.

# Numerical Algorithm Libraries for multicomputers

R.J. Allan

Advanced Research Computing Group,
Daresbury Laboratory,
Warrington WA4 4AD

# Numerical Algorithm Libraries for Multicomputers.

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

Abstract

These notes were written to document experiments in the design of a parallel algorithm library for numerical applications in FORTRAN-77. The target computer is any one of the current multiple instruction multiple data (MIMD) distributed-memory multicomputers although the philosophy is also applicable to shared-memory computers. Versions of the software are also available for such UNIX-based multiprocessors and also networked UNIX hosts. I have used Intel iPSC/2, Intel iPSC/860, Meiko Computing Surface with i860 CPUs and Transputers, and other Transputer-based hardware, and also Convex and Alliant mini-supercomputers and workstations. Extensive reference will be made to a previous review [1] on programming of similar parallel machines in FORTRAN-77.

The communications paradigm is that of Fortnet [2,3], which is representative of the general scheme of parallel communications on multicomputers. Other public-domain and commercially available harnesses differ only in detail. Extensions of this scheme are illustrated which embody algorithms for global communication and exchange of data between subsets of processes. This is compared with global communications procedures available in other organised libraries of software such as the SUPRENUM project grid library and some commercial software which has ideas similar to our own.

Finally details of subroutine calling sequences and useage of a number of existing parallel numerical libraries is reviewed. Examples are also given, taken from working programs, which illustrate programming styles. These might serve as an introduction to parallel algorithm design.

Contents

## I Introduction

In this introduction I wish to discuss the motives behind my work on parallel algorithms and describe a number of ways in which such algorithms can be designed. Some solutions have been hand-optimised for specific algorithms. Whilst these will be illustrated in section IV they are not of primary interest, what IS is the more general type of parallel strategy which underlies a number of algorithm libraries which are now emerging. These libraries are fundamentally different one from another and something may be learnt from each of them. I shall describe

i) Intel globals -- Intel global communications routines [6]

ii) Suprenum grid communications -- Suprenum communications for grid topology [24]

iii) Argonne/GMD communications macros -- Argonne National Laboratory and Gesellschaft fur Mathematik und Datenverarbeitung general routing calls [27]

iv) FPS T-series topology descriptor routines [28]

v) Fortnet v3.0 (trace) -- Fortnet routing harness layers [2-4]

vi) Express -- Parasoft's message passing harness which includes some routines for data decomposition [49, 50]

vii) Eiscube and Lincube -- Intel global matrix algorithms and level-2 BLAS [55]

viii) Liverpool parallel library -- NA Software Limited's global algorithms and work for the Supernode project and NAG Ltd. [17, 25, 26]

ix) Topexpress Parallel library [29]

x) Fortnet library -- Fortnet algorithm library [0]

Other libraries and communications primitives have been written, and I mean no disrespect by omitting them, but I don't have sufficient information to do justice to a full description. Some examples which I have omitted more for reasons of space are work on SPLIB by Dave Snelling [9] and the parallel library project of MIMD Systems [47, 48].

Several strategies have been adopted to write parallel algorithm libraries. They should rely on some underlying global communications rather than ad hoc solutions for each routine. The reason for this is twofold; firstly there is a duplication of effort in writing communications into each algorithm where the job can be done once, secondly there is the question of data placement. Moving data between processors is currently expensive, although one hopes that in future machines it will appear to be of like speed to direct local memory access rather than an order of magnitude slower. Optimisation of performance currently involves packing data transfer into a single message with the use of an 'intermediate' buffer.

Other cacheing techniques could be consaidered. This is a messy business and had better be put into a subroutine rather than left explicit.

Some comments on assessing the performance of algorithms were given in [1] and [14], we will however consider performance to be largely irrelevant for the present discussion and concentrate on optimising 'programmability'. Programmability arises through hiding the communications routing altogether inside subroutines which carry out certain actions on globally distributed data. Assumptions must be made in doing this, but first what are the essential actions?

i) distribute data across global memory (scatter) from local
ii) collect data from global memory to local (gather)
iii) move data in global memory
iv) transform data in global memory

Item (iv) is the global algorithm which is the goal of our work. Items (i) to (iii) may be steps towards it. Other operations are identical to those on sequential vector computers such as local gather, local scatter, local copy and local vector algorithms and might even be implemented in hardware as on the Intel or Suprenum. I would like to consider data motion in a MIMD computer to be expressible in the same way as vector operations in a vector computer (such as those available in the VecLib library [6, 42]).

A decision has to be made which is: will data be moved to processors before the operation of each numerical routine or: will data be acted upon by the routines no matter where it is in the computer?

The first of these solutions has been widely used already, and is indeed the basis for a programming model with an array of processors attached to a host computer. The host executes a sequential program which from time to time will call library routines. The action of these routines is to farm data onto the attached processors and then get them to perform a parallel operation, the result being brought back to the host. This model, in a slightly more sophisticated form is typified by the Liverpool software [17].

Diametrically opposed to it is the model which I have used for my own experiments, and which is also used by Intel Scientific Computers, Parasoft and SUPRENUM. A number of global routines are provided which must be called simultaneously from all the nodes (called loosely synchronous). In the Intel library, routines operate on a set of locally defined vectors.

In my work presented here a a crucial distinction is made between a local variable (memory) space to each process and a global space. Normal FORTRAN-77 variables refer to local memory and no message-passing, cache or bus conflicts can arise from using them. A number of symbolic variables refer to data stored globally in the

computer's memory, the programmer doesn't know where. Data may be put into these variables, moved between them and operated upon by algorithms, but the user need have no knowledge of how the communication is carried out. This is a matter for the layers of global communication and routing software which is described.

The Suprenum project also identified the need for global operations using a number of pre-defined grid topogies [24]. A large library was provided for defining and using grid operations.

The aim of both these approaches is to allow several possibilitie to the programmer:

i) routing may be optimised, eventually in hardware

ii) global communications may be optimised, possibly in hardware

iii) distributed-memory algorithms may map on to shared-memory architectures or vice versa

iv) Communications can be tested once and for all and proved to be deadlock free

v) The programmer is free to concentrate on applications rather than difficult problems of data handling

vi) most programs spend a lot of time generating expensive data 'in situ' and true parallel programming is then essential

vii) portability can be assured by modifying only communications layers

viii) Analysis and performance tools may be built in to well-defined communications layers

ix) With a well-defined set of global routines accessible from the node program an automatic paralleliser can be written

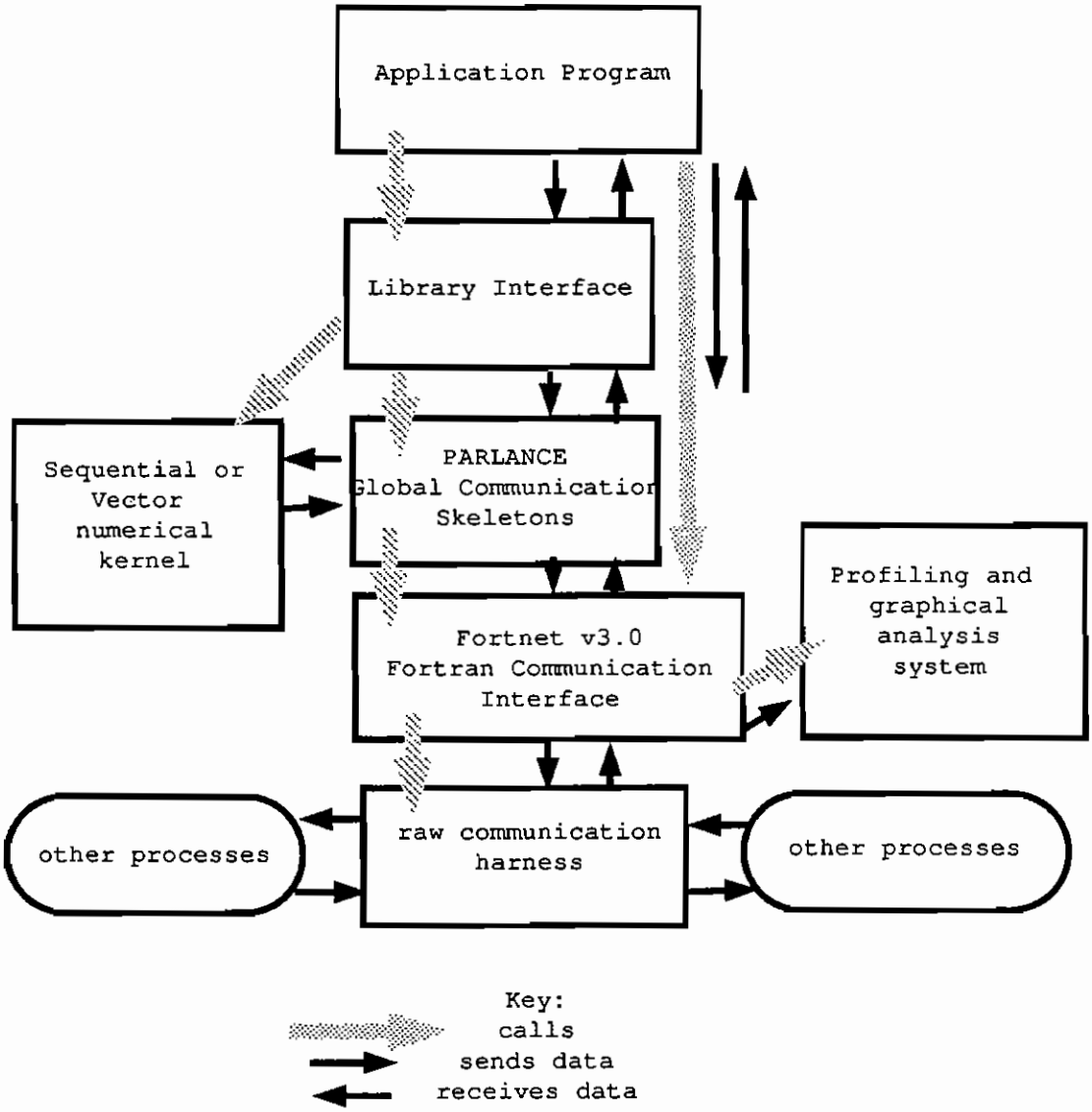
A feature of my own work is that in using the higher-level routines true parallel programs are written which execute concurrently on all the nodes. A routine for a global operation must be called simultaneously by all of them (unless you the user know otherwise after reading this document). Communications are then done internally and actions are performed on the true data which may be accessed by the calling program via a symbolic variable.

A summary of the layers of software involved in building a parallel library is shown in figure 1.

Single-node vector processing capability and justification of library strategy.

A number of the multicomputers mentioned here, the SUPRENUM, Intel iPSC/2 Intel iPSC/860 and also the old FPS T-series, derived their very high potential performance from vector pipeline units on each node. This is likely to continue and it is unlikely that current technology will increase the data rate far beyond the computation rate, or produce a very large-scale parallel array

Figure 1. Software Layers used in building a Library.



(exceptions being SIMD machines). In fact the opposite is happening following the introduction of the Intel i860 processor.

It is therefore essential to be able to include vector processing in the kernel of library routines. This can only be achieved if a vector of operands can be transmitted to the nodes before the operation is performed; one reason for separating the communication from the computation step. Vector operations are most efficient for long vectors, as is communication, which indicates the coarse-grained nature of the machine i.e. for a distributed vector there should be MANY more elements than nodes. This also implicates a large amount of memory per node > 4Mbyte. Except for special applications it would be useless to build vector multicomputers with more than 128 processors if a 128 element vector is the best length. This is relevant to problems of order 10000. The corollary is that long communication paths do not occur and a strategy of processor to processor rather than element to element communications is better. This will be adopted in the following discussion.

The possibility of Automatic Parallelisation.

A number of automatic vectorisors exist to convert sequential FORTRAN programs to ones in which operations are expressed in terms of calls to a library (like VecLib) rather than as nested do loops. This technique is now well understood. Once we have defined a set of global routines of a type similar to the sequential ones, and assumed that the same program is to run on all processors, and that communications are internal to the routines, the same principles of vectorisation may be applied to parallelisation.

This has now been done by the Suprenum project in their SUPERB (SUprenum ParallelisER Bonn) software [22], which produces code mapping directly onto their grid-topology communications library driven by the placement of data onto the processors. The ASPAR tool from ParaSoft performs a similar operation for the Express primitives [54].

In order that such parallelisers produce efficient code one further input is required to be given explicitly. That is the particular way a vector or matrix is distributed, or if it should be globally present on all nodes. The principle is completely general and any program can be parallelised, its efficiency depending only on the availability of underlying global routines for the application attempted.

Finally, I would not advocate immediate standardisation on any particular communication scheme or type of algorithm. It is still too early in the development of parallel computing and many avenues have yet to be explored which may be hindered by acceptance of a premature standard. The user must therefore bear this in mind. I have aimed at flexibility in my own work and might thus try to keep

abreast of what is a rapidly developing field.

II The development of Fortnet v3.0 (trace), and analogy to other portable communication harnesses.

The original concept of Fortnet and its v2.1 incarnation have been dealt with in refs. [2-4]. A few new additional features are worth mentioning, and a summary can be given of the current situation.

Fortnet is now available for use with the following lower-level routing software and compilers

i) Meiko Computing Surface occam-2 libraries, C and FORTRAN-77 [6]

ii) Intel iPSC C and FORTRAN-77 [4, 5]

iii) 3L Parallel FORTRAN-77 and C [16, 18, 19]

iv) UNIX 4.2BSD sockets (e.g Convex and SUN operating systems) using C and FORTRAN-77 compilers [43]

v) Meiko CStools on T800 and i860 nodes [7]

vi) the ipcv3 networked harness for UNIX nodes [51]

vii) TINY transputer harness [12]

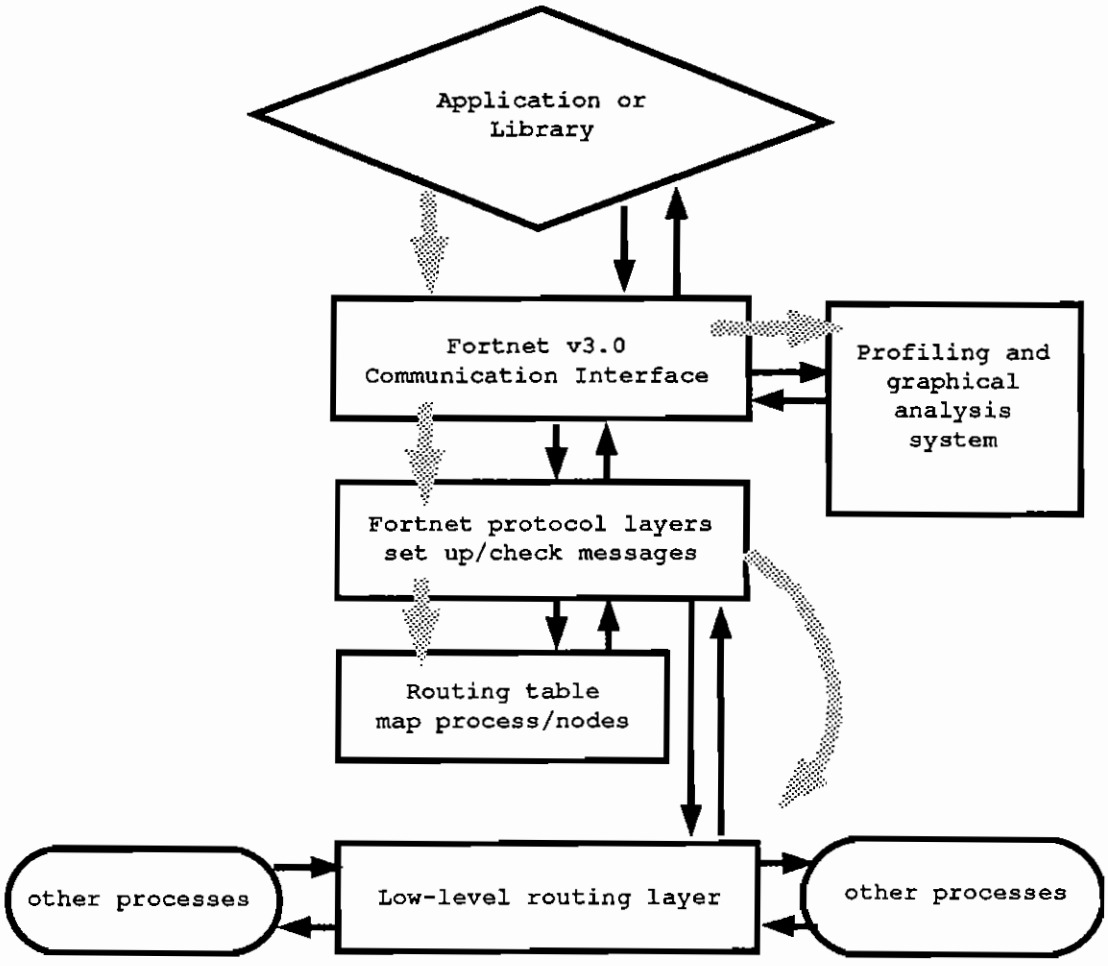
viii) Alliant fx/8 or fx/2800 shared-memory [56]

Fortnet is a multi-layered system of subroutines which roughly follow figure 2. Each layer is largely independent of the exact functions of the previous one providing calling conventions are adhered to. Thus each layer can be independently optimised or tailored to suit a wide variety of parallel computers. This can be done as new machines come onto the market. The top layer is always the application code. The structure of Fortnet is described as follows.

1) Initial development of Fortnet centred around the need to supply a convenient means to use FORTRAN on the Meiko computing surface for writing concurrent programs. This was not available from Meiko Ltd. when I started in 1987. The first stage of Fortnet was therefore a communication harness to pass messages (data) between the transputers in a controlled fashion. Similar work was done elsewhere in the UK, e.g. Southampton University with a harness called ECCL [52], and Edinburgh University with a harness called TINY [12]. Fortnet also performs some tasks like accessing the front-end file-store, printing diagnostic messages to the screen, and bookkeeping. This layer of code was written in the new concurrent language occam-2 which was designed for the transputer, and was partly the result of work by Sebastian Zurek who visited Daresbury during the summer of 1987. Fortnet could in principle be implemented on any type of transputer array.

2) The second stage of development is a layer of subroutines which may be called by the parallel FORTRAN-77 program as an interface to the occam. These incorporate a protocol to verify the

Figure 2. Software Layers in Communication Harness.



correct transmission of messages and warn the user of any problems. These problems are often of the sort that occur during early code debugging which would just cause hangup if no error-checking mechanism were present. A novel handshaking and blocking paradigm is used for this checking which differs from other systems. The routine calls are however superficially similar to those on hypercube machines such as the Intel iPSC. This was Fortnet v1.0

3) Further development of the v1.0 layer and incorporation of full FORTRAN i/o on all nodes of the Meiko occam version was done by Lydia Heck of Durham University. A full port to the 3L Parallel FORTRAN language [16, 18] and also to the TINY communications harness [12] was done by R.K.Cooper of Queen's University, Belfast. The i/o capability of the server process was also increased, and a study was made of intermediate buffering of messages and access to low-level channels to improve performance. The interface is now similar in style to other software mostly developed in the USA, e.g. the PICL harness at Oak Ridge [57], PARMACS and ipcv3 at Argonne [27, 51], and Express at Caltech [49, 50]. This was Fortnet v2.1.

4) Work started on an implementation to use UNIX sockets for a TCP network of workstations or other systems requiring UNIX-style inter process control. Somewhat earlier work had however been made in this direction at Argonne National Lab. USA so when R.J.Harrison visited Daresbury in the summer of 1990 it was decided to combine the two harnesses. Fortnet therefore now calls the ipcv3 harness to use sockets and UNIX shared-memory procedures, but retains its own higher-level functionality [59].

Completion of stages (3) and (4) yielded Fortnet v2.2 which is installed on a number of machines in the UK including:

Durham Physics Meiko M10 (CSTools and Occam)
Leeds Computing Science, M10 and InSun system
Lancaster Computer Science Campus M60
Birkbeck College Physics M60
Sheffield Transputer Centre M10 and M40
Liverpool Transputer Centre M40
Bath SWURCC Central Computing Services M60
Rutherford Appleton Lab. M10
Daresbury Lab. M10 (Occam)
Daresbury Lab. M60 (CSTools)
Daresbury Lab. Intel iPSC/2 (Intel version)
Daresbury Lab. PC-based system (3L version)
Daresbury Lab. Convex C-220 (UNIX version)
Daresbury Lab. SUN Sparcstation1 (UNIX version)
Daresbury Lab. Stardent P1500 (UNIX version)
Daresbury Lab. Silicon Graphics workstations (UNIX version)
National Physical Lab. M40
Queen's University Belfast Parallel Computer Centre Parsys (3L version)

Belfast Applied Maths M40
Northern Ireland Transputer Centre M10
Belfast Aeronautical Eng. PC-based system (3L version)
UMIST Alliant fx/2800
UMIST Stardent P1500 (UNIX version)

The Fortnet harness v2.1 is available from Computer Physics Communications and the Transputer Initiative Program Library at Liverpool University, and is included in the third-party software catalog of Meiko Scientific Limited (the Ensemble programme) and 3L Limited.

5) A demonstration interface has been written to the graphical post-mortem display package ParaGraph from the Oak Ridge National Lab. which was originally designed to replay execution of a program using the PICL harness [57, 58]. It depicts dynamic execution of an actual parallel program on the screen of a workstation running X-11 windows in pseudo-real time allowing one to identify hot spots, bottlenecks and errors and to effectively compare different algorithms. Some work is still needed to tidy up this interface and resolve some remaining questions about exactly how the display should look. Fortnet version 3.0 (trace) is however already available for use in certain implementations. More details of this are given below and an example is shown in figure 4.

A parallel profiler has also been written which shows the activity of each processor in terms of the communications functions and sequential code which it assumes to be cpu active. An example of output is shown in figure 5.

Implementation of stage (5) of Fortnet on the Intel hypercube allows us to benefit from these tools also in developing programs on the iPSC. Such an exercise was necessary because we do not have access to the internal workings of the Intel message-passing subroutines. Furthermore Fortnet now provides a common environment and devlopment platform on both the Meiko, Intel and UNIX-based computers allowing direct porting of applications.

6) Development of a generic set of global-memory operations required at the application level has started. This is a library of subroutines which handle synchronisation and communications to, for example, distribute or recall data in a known way over a known set of processors allowing results to be calculated in parallel and then be globally accessed. A number of frequent operations, such as generic vector-vector or matrix-matrix operations can then be programmed where elements of the vectors or matrices are implicitly distributed. This is the subject of section III of this report.

Further work is needed to investigate optimisation strategies which will fully extract the parallelism inherent in these global operations. Those involving a pair of elements will work well if the available processors are logically divided into sets of independent

Figure 3. Schedule/Trace display on a SUN workstation.

pairs for instance, and the same for any k-fold covering to implement an operation involving k distinct data elements. This is pursued in section III.

7) The highest layer of the environment would directly call these global routines to do numerical tasks. Standard library calls have already been implemented in this way for vectors and matrices. This is the subject of section V of this report.

Summary of Fortnet calls

The basic synchronisation and routing available in the Fortnet library is as follows:

inode=nodeid() -- get Fortnet id of current process, 0 for server, 1 for master and 2 to nnode+1 for slaves. This is the integer parameter used in the following routines

nnode=numslave() -- find number of slave processors in array

check(m) -- check to see if process m is waiting for data in order to synchronise communication. This together with subroutine wait constitutes the blocking mechanism.

wait(n) -- wait until process n checks, or acknowledge ready to receive data

xsend(m,nbytes,buffer) -- send nbytes to target process m from buffer. Strong typing is enforced if X takes any of the values: A - character, S - single precision, I - integer, D - double precision, C - complex, Z - complex double precision, L - logical. If the character X is omitted no strong typing is enforced. This is only important when passing messages between different processor types if it is necessary to change the internal number representation.

xreceve(n,nbytes,buffer) -- receive nbytes from source process n into buffer

xrecany(iproc,nbytes,buffer) -- receive nbytes into buffer regardless of which processor they came from, the source processor id is stored in iproc

xchange(n, m, nbytes, buffer1, buffer2) -- an efficient implementation to exchange nbytes of data between processes n and m.

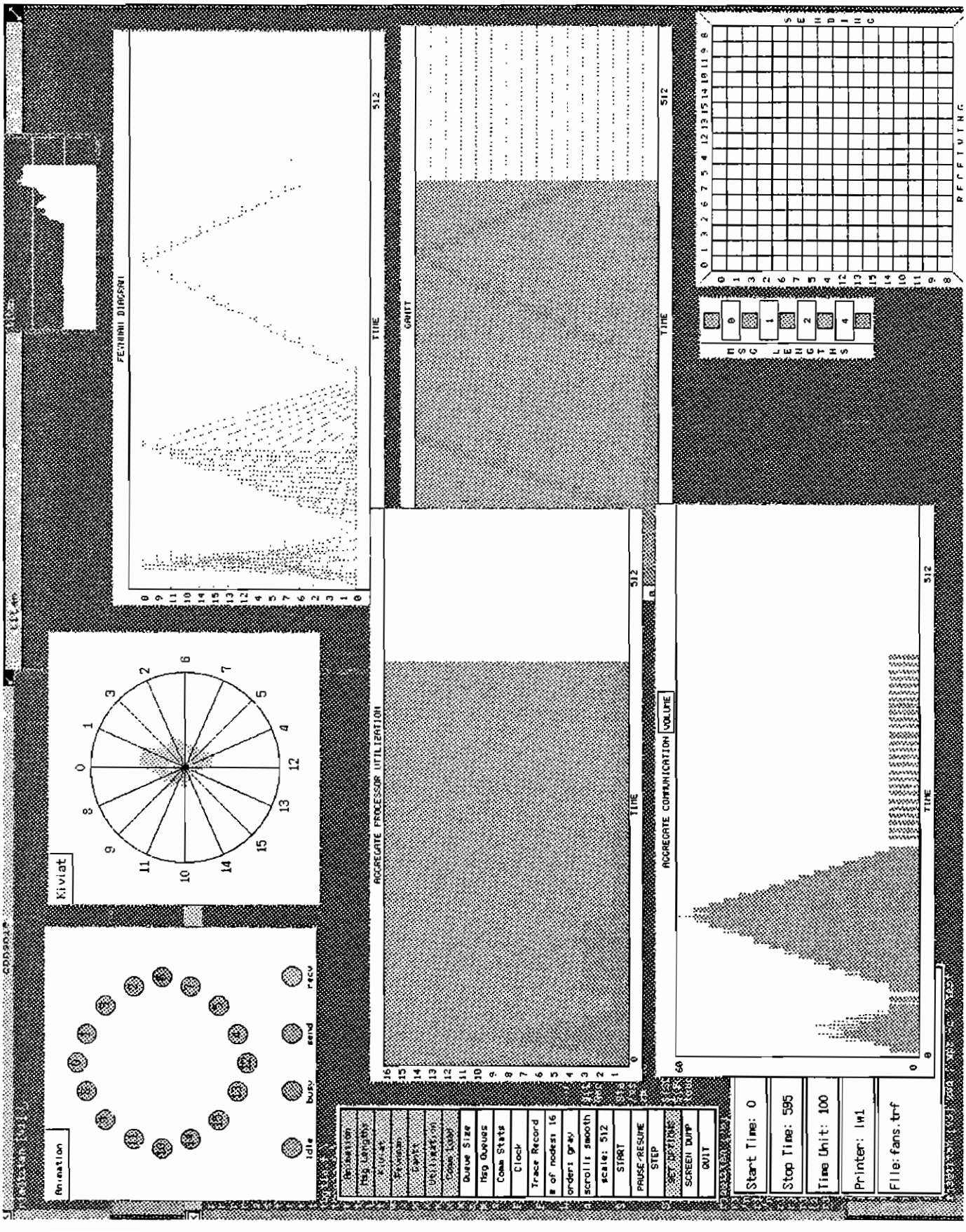
read(lu,nchar,buffer) -- read data from globally accessible file lu via the driver process

write(lu,nchar,buffer) -- write data to globally accessible file lu via the driver process. A number of other i/o functions have been written [19]

brlist(proclist, nproc) -- enter nroc processors in the integer list proclist into a database for broadcasting operations

brall(mode) -- where mode can be 'ON', 'OFF', 'TOGGLE', 'RESET' controls the behaviour of routines which require a broadcast step. This enables the user to specify that only partial local results of operations are required

Figure 4. Some of the display windows availble in ParaGraph.



brcast(root, nbytes, buffer) -- broadcast nbytes from buffer on node root to nproc processes whose ids are contained in the proclist, does a receive operation if inode not root. Loosely synchronous

sync(root) -- global barrier to synchronise processors starting from root, targets are specified in the proclist. Loosely synchronous

debug(mode) -- where mode is any of 'ON', 'OFF' or 'TOGGLE' sets the mode of event tracing for the calling node. This trace information is used by the profiler and graphical analysis tools.

lockon(lockno) -- lock a critical section of code and ensure that it is executed in numerical node order for all nodes appearing in the proclist. Each process waits for all those with a lower number to run and is blocked until they call lockoff(). Locks can be nested with different lock numbers. Error messages are printed if the nesting is not done correctly. This may be useful for handling i/o or other shared resources

lockoff(lockno) -- unlock a ciritical section of code spanned by a lock of the given number unblocking processes in proclist with higher numbers

The operation of these routines is described in references [2-4]. It is significant that they will hardly be used in the following pages but are incorporated inside higher-level calls.

In the early days Fortnet gave us a way to write parallel programs in FORTRAN for the Meiko Computing Surface. It has now become a basis for further developments and provides a standard portable platform for writing parallel code.

Definition of logical processes in Fortnet. Graphical analysis and profiling tools.

In order to take over and make use of, on local-memory MIMD computers, software developed for shared-memory ones (such as CRAY, Alliant, Convex etc.) it is necessary to define a 'logical process'.

A logical process is a piece of sequential code which runs on a processor and has communications at either end, unless it is the beginning or termination of the executing thread. A number of logical processes placed together end to end on one processor form what I shall call a 'job'. Jobs are linked together by complex data dependency paths in a topology which is representative of the overall 'task' or program.

In a shared-memory computer a logical process (or process) might, for instance, be a subroutine which is scheduled to run on a processor when data in its argument list is ready. In a local-memory machine a process is usually a shorter piece of code and data must be physically transferred to the processor on which it is to run.

The scheduling mechanism is however identical and does not occur until all the data dependency is satisfied (it is blocked until then). We will refer to a job as being a sequential collection of processes separated by communications to other jobs. A number of jobs running in parallel is a task.

Job scheduling can be done automatically under control of a main program, as has been done in the Argonne Schedule package [21]. In a similar way the complete task lends itself to graphical display and profiling in this form; details of the sequential chunks are less interesting than the passage of data between them. We have therefore used the Argonne Schedule Trace package [20] to display execution of parallel programs running with Schedule at the job level, as shown in figure 3, and using ParaGraph to show the Fortnet communications between these jobs as shown in figure 4.

Using this scheme it is possible to profile the length of time spent in doing data transfer, computation or waiting on each processor. There is a Fortnet parallel profiler which does this. This technique of active profiling is in general the only way to test real performance of a parallel algorithm, although valuable theoretical work has been done by the Liverpool group to predict the performance of some [see 1].

Figure 5. Typical output from the Fortnet profiler

```
% profile
 ***** Fortnet ***** Pretrace Facility, nnode = 4rx
 .....processor    1 started at   10.375
 .....processor    2 started at   11.875
 .....processor    3 started at   12.031
FORTRAN STOP
 .....node    1 finished
FORTRAN STOP
 .....node    2 finished
FORTRAN STOP
 .....node    3 finished
 .....sortp0 all finished

 ***** Fortnet Profiler *****

 .....Fortnet timings in seconds
 proc, receve, send, wait, check, sequential
0,          0,          0,          0,          0,          0
1,      142.5,      124.5,          0,          0,     2840.3
2,     144.38,      142.5,          0,          0,     432.38
3,      124.5,     144.38,          0,          0,     2123.5

 .....number of calls
```

```
 proc, receve, send, wait, check, sequential
0,  0,  0, 0, 0,  0
1, 12, 12, 0, 0, 24
2, 12, 12, 0, 0, 24
3, 12, 12, 0, 0, 24

 .....diagnostics
 processor 0 terminated
 processor 1 terminated  doing sequential code
 processor 2 terminated
 processor 3 terminated
%
```

## III Global Communications.

In this section I attempt to outline some software which is able to carry out global data movement. Mostly this takes the form of simple extensions of broadcast constructs with the contents of locally defined vectors of numbers being sent to other nodes and a function performed which may for instance add or contatenate them elementwise with existing data. There are however also some more powerful routines.

### III.1 Intel Globals

The iPSC communication library contains a limited number of bitwise or elementwise global operations between vectors defined locally on the nodes. Each of these is not a component of one globally defined vector, but is defined locally and they may be of differing lengths. For instance calling gdsum(x,n,work) on all nodes does

gdsum(x,n,work) -- (sum(x(n,i)),n=1,ni,i=1,p) where x(1:ni) is a partial vector on node i, and there are p nodes. Input arguments are all locally defined, and the work vector is required to hold values transmitted from other nodes.

Possible operations listed below, are 'and', 'max', 'min', 'or', 'prod', 'sum' and 'xor'. An external function may be placed in a skeleton call for a user-defined operation similar to these. A concatenation operation for vectors is provided. A sync operation is provided and a broadcast operation with the possibility to select nodes.

gcol(x,xlen,y,ylen,ncut) -- global concatenation, places xlen bytes of x from all nodes and processes with the same pid into y on every node in increasing order. A broadcast operation. ncut is the number of bytes returned in y. xlen may have different values on different nodes.

gopf(x,xlen,work,f) -- global operation of a user-defined associative and commutative function f. The external function f must have two parameters, the input value and an array for contribution from other nodes. Same behaviour as gdsum.

gsendx(type, x,xlen,nodenums,nlen) -- broadcast a vector x of length xlen bytes to a list of nlen nodes in nodenums() as messages of type 'type'

gland(lx,n,work) -- integer bitwise or logical global and operation for vectors of n elements. Work is a workspace of length >= n. N may have different values on different nodes.

giand(ix,n,work) -- same as above for integer operands

gshigh(x,n,work) -- global MAX operation on real vector x of

length n elements.

gdhigh(...) -- same as above for double precision operands
gihigh(...) -- same as above for integer operands

gslow(x,n,work) -- global MIN operation for real numbers
gdlow(...) -- same as above for double precision operands
gilow(...) -- same as above for integer operands
glor(x,n,work) -- global logical OR operation
gior(...) -- same as above for integer operands
gsprod(x,n,work) -- global multiplication of real numbers
gdprod(...) -- same as above for double precision operands
giprod(...) -- same as above for integer operands
gssum(x,n,work) -- global sum for real operands
gdsum(...) -- same as above for double precision operands
gisum(...) -- same as above for integer operands
glxor(x,n,work) -- global bitwise exclusive OR
gixor(...) -- same as above for integer operands
gsync() -- global synchronise all nodes

III.2 Argonne/GMD Communications Macros and ipcv3

I have included this section because of its use in portability and its potential extensions to global communication schemes.

The Argonne/GMD macros have been written to help port programs over a wide range of shared and local-memory MIMD computers. They consist of instructions in capital letters used as normal FORTRAN or C statements or variables with a possible list of arguments. They must be precompiled with a macro expander (such as m4 in UNIX environments) prior to compilation to produce source code specific to the computer on which the program is to run.

The available macros in the original suite are [27]

ENVHOST -- environment variables for host program. Must be placed before any executable statement of all routines using the macros

ENVNODE -- same for node programs

INITHOST -- used to initialise the host environment. Must appear once before any macro calls

INITNODE -- same as above for each node program

REMOTE_CREATE(process_file,procid_vector) -- loads the node programs from a configuration file process_file, and returns a vector of integer process ids in procid_vector which may be used to communicate with them.

TORUS(nx,ny,nz,slave,process_file) -- macro to create the above mentioned configuration file in a machine-independent way for 1d, 2d and 3d tori. Slave is the full name of the executable image file. It is planned to add more topology macros later. This has some resemblance to the FPS software also described in this document

MYPROC -- a variable which is the self process id

HOSTID -- a variable which is the host pid

SEND(target_id,buffer,length,type) --send a contiguous memory section to another process. Returns control to program when message has left sending process. Asynchronous.

RECV(buffer,buffer_length,length,sender,type,condition) -- type is a message tag and condition is a selection macro as given below. This macro receives a message asynchronously

'condition'=MATCH_ID(select_sender) -- select only if id matches

'condition'=MATCH_TYPE(select_type) -- select only if message type matches

'condition'=MATCH_ID_AND_TYPE((select_sender,select_type) -- select only if both type and sender match

'condition'=MATCH_ANY -- select any message in the 'mailbox', a mailbox is a fifo stack of message buffers for reception by a unique process.

SENDR(target_id,buffer,length,type) -- synchronous send

RECVR(buffer,buffer_length,length,sender,type,condition) -- synchronous receive

PROBE(flag,condition) -- check mailbox for existence of the specified message and return flag=1 if it is there, otherwise flag=0

WAIT_MESSAGE -- block the user process until a message enters the mailbox.

BARRIER -- global synchronise all node processes, blocking until they have all reached this macro

CLEAN_UP -- should be put at the end of the host program to kill and clean up processes on the nodes. This may be followed by a further REMOTE_CREATE macro.

ERROR -- variable referring to termination of previously invoked macro: 0 if normal: -ve if warning: +ve if severe error. It is the user's responsibility to test this condition, although an error message is sent to the stdout stream.

This provisional set of macros presents an extension of the Intel iPSC programming environment with the additional feature that resulting programs may be ported to other machines. The Intel NX/2 node operating system has been modifed to provide this extra functionality by the Gesellschaft fur Mathematische Datenverarbeitung in Germany and the Katholieke Universiteit Leuven in Belgium [27]. Its strength would be further increased by a set of more powerful global macros using topologies preplaced in the configuration file, such as the full SUPRENUM library.

A subset of these macros has been studied in intensive computational science applications by R.J.Harrison at Argonne, and has been re-written in a simple but robust form as a library called ipcv3 [51]. We have used this simpler library as a basis for porting

our own software in a UNIX ipc environment.

### III.3 SUPRENUM Grid Communications

In the application software of the SUPRENUM project (the German national supercomputer project) parallel grid-oriented algorithms are being written especially for multigrid CFD codes. Interest from this area is now widespread as expectation of parallel performance has increased. A central communication library greatly facilitates the data decomposition. Portability of the resulting applications is also achieved by means of a single communication library optimised for various machines (especially the SUPRENUM and Intel). Versions are available which use the message-passing primitives of either Intel, Suprenum or the portable Argonne/GMD macros (see above).

For a particular application a rectangular domain can be subdivided into smaller rectangles with some overlap, and each subdomain is assigned to a different process. This leads to a logical process grid. Mapping of this virtual process grid onto real processors is done by calling library routines which are optimised to the underlying architecture. A further assumption is that all nodes execute the same code (I have also assumed this in Fortnet, and it is also done in Express).

The library provides routines to set up the logical process mapping, carry out global operations, and do exchange of grid functions across two- and three-dimensional array boundaries. In iterative parallel algorithms this exchange across inner boundaries (i.e. data belonging to different processors on each side) is a common task. Data in an overlap region is stored, and is updated during the exchange operation to reduce the amount of communication necessary in the other library routines at the slight expense of additional computation and memory requirement for the duplicated points. Other multigrid operations involve increasing or decreasing the number of points considered in certain areas of the parameter space, with a need to re-partition them across processors.

The implementation is good, with buffering of all messages between two processors into a single message (intermediate buffer) to reduce overheads when possible.

The following routines are in place in the Intel version of the Suprenum grid communications library [24]. Some common blocks are used for data as described in the extensive documentation.

agglm2.f(nxnew, nynew, inew, jnew, array, n, nstart, iworkl, iworkx, idl, idx, jdl, jdx, il, ix, jl, jx, re, lenre, idimre, in, lenin, idimin, iorder, repeat, error) -- agglomeration routine for two-dimensional grid processes. On changing the scale of the grid not all processes are needed, this reduces the number of executing processes and scales the variables.

agglm3.f(nxnew, nynew, nznew, inew, jnew, knew, array, n, nstart, iworkl, iworkx, idl, idx, jdl, jdx, kdl, kdx, il, ix, jl, jx, kl, kx, re, lenre, idimre, in, lenin, idimin, repeat, error) -- agglomeration routine for three-dimensional process grids

cmlmsg.f(string) -- allows a message to be written to a protocol file

cmlver.f -- print out the version date of the communication library

crgr2d.f(nx, ny, slave, period, re, lenre, in, lenin, error) -- creates a grid process of two dimensions, an nx* ny logical node process whose name is contained in the character variable slave. Links the process information in both 2D grids and binary trees and

sends the information to each grid process on its neighbouring proceses. This routine must be called from the host

crgr3d.f(nx, ny, nz, slave, period, re, lenre, in, lenin, error) -- creates an nx* ny* nz three-dimensional grid process whose path name is slave. Links and sends information as above

gloph.f(re, lenre, in, lenin, tag, error) -- receive results of global operations over all node processes

glops.f(re, reop, lenre, in, inop, lenin, back, host, tag, error) -- perform global operations on real vector re and integer vector in over a tree structure. Operations available are '+', 'max' etc. Broadcasting of the results is optional, but routine gloph must be called if it is done

grid2d.f(nx, ny, i, j, re, lenre, in, lenin, error) -- node counterpart of crgr2d, must be called at the start of the node program execution. Receives information and passes it to the calling program

grid3d.f(nx, ny, nz, i, j, k, re, lenre, in, lenin, error) -- receive information as above, node counterpart of crgrd3d

gupdt2.f(array, n, nstart, idl, idx, jdl, jdx, il, ix, jl, jx, color, width, order, tag, error) -- sends the values of the grid functions at points of the overlap zones of inner processes to the neighbours and receives the corresponding messages. This is a combination of the funcionality of rupdt2 and supdt2

gupdt3.f(array, n, nstart, idl, idx, jdl, jdx, kdl, kdx, il, ix, jl, jx, kl, kx, color, width, order, tag, error) -- sends the valules of grid functions at points in the overlap zones on inner process interfaces to neighbour processes and receives the corresponding messages. As above for the 3D processes

hstart.f(error) -- initialise the process environment of the host

hstend.f -- clean up the process environment at the end of execution

intrpt.f(string) - allow a user to stop the distributed application

recia.f(intarr, idl, idx, jdl, jdx, kdl, kdx, imin, imax, jmin,

jmax, kmin, kmax, tag, error) -- receive a subarray of integer array intarr.

recra.f(array, idl, idx, jdl, jdx, kdl, kdx, imin, imax, jmin, jmax, kmin, kmax, tag, error) -- receive a subarray of real array array.

recvwh.f(ci, type, buf, len, cnt, node, pid, error) -- simulates the communication routine recvw in the host process

rupdt2.f(array, n, nstart, idl, idx, jdl, jdx, il, ix, jl, jx, color, width, dest, tag, error) -- receives the values of the grid functions at points in the overlap zones on inner 2D process interfaces from neighbouring processes

rupdt3.f(array, n, nstart, idl, idx, jdl, jdx, kdl, kdx, il, ix, jl, jx, kl, kx, color, width, dest, tag, error) -- as above for 3D process

rupsg2.f(array, n, nstart, idl, idx, jdl, jdx, il, ix, jl, jx, color, width, dest, tag, error) -- exchange-receive routine as above but for staggered grids

sendia.f(intarr, idl, idx, jdl, jdx, kdl, kdx, imin, imax, jmin, jmax, kmin, kmax, idest, jdest, kdest, tag, error) -- sends a subarry of the integer array intarr to another process

sendra.f(intarr, idl, idx, jdl, jdx, kdl, kdx, imin, imax, jmin, jmax, kmin, kmax, idest, jdest, kdest, tag, error) -- as above for a real array

supdt2.f(array, n, nstart, idl, idx, jdl, jdx, il, ix, jl, jx, color, width, dest, tag, error) -- sending routine to match rupdt2 above

supdt3.f(array, n, nstart, idl, idx, jdl, jdx, kdl, kdx, il, ix, jl, jx, kl, kx, color, width, dest, tag, error) -- sending routine to match rupdt3 above

supsg2.f(array, n, nstart, idl, idx, jdl, jdx, il, ix, jl, jx, color, width, dest, tag, error) -- sending routine to match rupsg2 above

swtdim.f(lostdm, error) -- switches from a 3D to a 2D environment. A further call restores the original 3D environment

syncre.f(isend, jsend, ksend, tag, error) -- used to synchronise two processes in a two-dimensional process grid.

syncse.f(isend, jsend, ksend, tag, error) -- as above for a 3D grid

testtag.f(tag) -- tests whether a message with the identifier tag is waiting in the mailbox of the current process

### III.4 FPS T-series

The FPS T series was not discussed in the previous document [1] because it is no longer relevant as a FORTRAN development platform. It is however useful to review the experience of global communications on this machine. FPS were probably the first company

to recognise the importance of this higher-level use of processor arrays.

The FPS T series was a hypercube of vector processors with transputers controlling data flow through multiplexed links. 64-bit arithmetic could be performed at peak vector speed of 12 Mflops, in part enabled by a novel use of dual-ported video RAM memory, 1 Mbyte per node. A complete node occupied a single board and each group of eight nodes, a 3-dimensional cube, was coordinated by a system node transputer with attached Winchester disk drive. Connections between nodes were over 4-way multiplexed transputer links.

A nice feature of the software which was being developed by FPS before the model was withdrawn was the capability to construct 'topology descriptors'. These allowed communication between nodes in a way specified by the topological symmetry (ring, mesh, torus etc.). More than one symmetry could be used simultaneously, and the mechanism for controlling it, described below, looked rather like file handling.

The operational environment of the T series was ULTRIX with a micro-Vax front end. A subset of UNIX enabled the cube to be allocated and programs compiled and loaded onto it. The C00 release of the Penguin Software (now Pentasoft) F77 compiler was rather primitive and did not have full F77 i/o facilities on the nodes. To communicate with the terminal the writexxx_h routines are used which passed data through a file server, where xxx stands for int, str, r64, nl etc. This is a feature shared with an earlier version of Fortnet (v2.0), which communicated with files via a server in the host processor. Later software is also able to provide full i/o. A host file server is also a crucial feature of Express.

To communicate between nodes on the T series, a topology must be chosen for the interconnection, and declared by calling one of the topology routines. The available topologies are: ring, torus or hypercube, and are declared by calling one of:

```
call config_hyp(idim, itd)
call config_torus_1d(isize, itd)
call config_torus_2d(isize1, isize2, itd)
```

This is discussed below.

T series utilities allow access to, and control, the T-series processor configuration from an executive node program (i.e. running on the front end). Executive node communication functions allow a vector node to communicate with the front end. They are described in section 5.2 of the manual [28]. System node disk communication allowed a vector node to access the system disk. Vector node communications functions allowed a node to communicate with another node. A list is not provided as it is irrelevant to the present discussion.

Vector node communication was used to balance communication and

computation and was accomplished through the following three steps:

Declare a topology in which communication occurs. This creates and initialises a topology descriptor on each processor. The descriptor contains information about the type of topology, the number of dimensions, the number of processors in each dimension, the given processor's position within each dimension, and the links between processors. You can declare any number of concurrent topologies, and every processor need not be included in every topology. Remember that declaring a topology does not physically change the machine, it just sets up a file which can map your code on to a subset of the processors in a particular way, and selects the existing physical links which are required to do this.

Open a topology for one or more types of structured communication. This must be done before communication can occur, rather like opening a file to read or write data. When you open a link file using open_l you must specify which topology it should belong to, the maximum number of asynchronous i/o operations which can execute concurrently on the link file, and the communications modes to use. Any number of link files can be opened for a particular topology, that just describes the processor mapping within which communication on the link file will occur.

Perform communication. Once a link file has been opened the vector (v-) node communications functions allow various patterns of structured communication within the topology that was associated with the file. The functions are valid only if they correspond to one of the declared access modes. A function call initiates communication asynchronous with the main process. If the next statement initiates another it will also execute, until the maximum number of processes declared in the link file is attained. Communication can in this way be overlapped with computation. If a following process depends on the result of communication a wait_l function call should be issued to ensure that the communication has indeed terminated. Note that if two communications require the same set of transputer links they will execute sequentially, whereas if the links are distinct they may execute in parallel (optimisation?).

ata_1do(cfh, idim, csend_buf, crcv_buf, icount) -- ordered all-to-all in 1D of torus

ata_1du(cfh, idim, csend_buf, crcv_buf, icount) -- unordered all-to-all

ato_1do(cfh, idim, idtest, csend_buf, crcv_buf, icount) -- ordered all-to-one

close_l(ifd) -- close v-node communication link

config_hyp(idim, itd) -- create a hypercube topology descriptor

config_torus(isize, itd) -- create a 1D torus

config_torus(isize1 isize2, itd) -- create a 2D torus

config_free(ctd) -- destroy a topology descriptor

config_dim(itd) -- get the number of dimensions in a topology descriptor

config_size(itd, idim) -- get the number of processors in a given dimension

config_pos(ctd, idim) -- get ordinal position in a given dimension

config_name(itd, cprefix, cname) -- build a unique file name

open_l(ctd, ipend, imode) -- open a vector node communications link

ota_1d(ifh, idim, iorigin, csend_buf, crcv_buf, icount) -- one-to-all communications in 1D of torus

oto_x(ifh, idim, ifrom, ito, csend_buf, crcv_buf, icount) -- one-to-one communications across a dimension

rotat_1d(ifh, idim, idistr, csend_buf, crcv_buf, icount) -- rotate data in 1D of torus

swap_x(ifh, idim, iproc1, iproc2, csend_buf, crcv_buf, icount) -- process pair data swap across a dimension

wait_l(ifd) -- wait for completion of pending v-node communications link activity

The information above related to the the FPS T-20 installed at D.L. at the end of 1987. I have no information on any later developments of the software.

### III.5 Linda and Kernel Linda.

The original Linda was developed by David Gelernter at Yale University. Kernel Linda was a trademark of Cogent Research Inc. and was integrated into the QIX operating system of the XTM workstation resulting from the work of Wim Leler [30-34]. Both are communication libraries for standard sequential languages which give an interface looking like a virtual shared memory computer, but with an explicit distinction between access to shared constructs and local variables. I have chosen to briefly outline the philosophy here because of its resemblance to the Fortnet system, and its potential use in gobal algorithm design.

Linda defines an abstract machine with a globally accessible memory for storage of both passive data and active processes with environment variables [34]. Lists of data and keywords against which it is matched are 'tuples', stored in 'tuple space' (TS). A tuple might for instance be ('house', 3, 4.6) which consists of a character string, an integer and a real number. This might be put into TS using the routine 'out' from any process

out('house', 3, 4.6)

Another process may search tuple space for a matching 'template' and either remove its entry using in or just read it using rd:

in('house', ?i, 4.6)

```
rd('house', 3, ?r)
```

In the first of these we have asked to match the string and real number and put the corresponding integer into variable i. Similarly in the second the integer and string are matched and the real value is read into r. The tuples can contain any number of items. In and rd block execution until data exists in TS, alternatives are inp and rdp which do not block, but return 1 if data is present or 0 if nothing is found. A fourth routine puts an active tuple into TS

```
eval('task',i,routine(i))
```

It is evaluated later by any processor requesting a work packet, initiated when its value is accessed with

```
in('task',i,result)
```

In Kernel Linda [34] only a single key is used, which might conveniently be a subset of TS called a dictionary. Variables are grouped into dictionaries like UNIX files into directories, and dictionaries of dictionaries are possible. Calls are for instance

```
k_out(dict, key, val)
k_in(dict, key, var)
```

Many other routines exist for process creation and management and for accessing system servers. It is easily possible to use TS in a similar way to FORTRAN common blocks and to define variables in global memory.

A symbol in Fortnet (see below) may correspond to a dictionary in Linda, and the index of the symbol to a keyword. Thus there is a correspondance between:

k_in(dict, key, var) and fetch(inode, 1, 1, dict, key, 1) or var=fetcha(dict, key)

k_out(dict, key, val) and put(inode, val, 1, 1, dict, key, 1) or call puta(val, dict, key)

Fortnet, like Linda, uses symbol tables present on each processor to look up data. There is however no hashing or subtables yet, and keys are instead a global offset of the requested symbol.

There are other differences. Kernel Linda is implemented at the operating system level, and memory in a remote processor is searched for the value of var without the user's intervention. In Fortnet the calls must be made in loosely synchronous manner on both source and target processors (or for safety's sake all processors in the network). This is however more transparent in the vector routines which would normally be called on all processors in any case to make use of the parallelism of vector operations.

### III.6 Express message passing and decomposition tools

Express [49] and the tools which use it are commercial products of ParaSoft Corp. USA, resulting from research work done at CalTech starting with the Crystalline Operating System [50]. It is a harness which is now widespread, ported to a wide range of machines and which implements a few generic global communication strategies. One tool is F90 which converts FORTRAN-90 source (with array and vector constructs) into parallel FORTRAN-77 source with Express calls. Another is ASPAR an automatic paralleliser which takes conventional FORTRAN or C programs and puts in Express calls for more common loop based parallelism, such as task farming, with interactive help where required (e.g. data dependency). There is also a debugger, NDBTOOL which is an extension of the original one for the NCube, and other profiling tools, VTOOL, CTOOL, ETOOL and XTOOL. One specific example of an application written using Express is DIME (Distributed Irregular Mesh Environment) by Roy Williams of CalTech [53] for use in CFD and Finite Element calculations.

Express itself is currently available for the following range of computers:

most Transputer-based systems including: INMOS B004 and B008 PC-AT boards, B011 abd B014 SUN boards, Microway, Definicon and CSA Tranputer boards for PC-AT, Levco Macintosh boards, Topologix SUN boards and Meiko systems

NCUBE

Intel iPSC

SYMULT

SUN-3, SUN-4, SUN-386i

some shared-memory machines

As well as decomposition tools and communications Express contains primitives for graphical applications. Thes are quite low level but again use the parallel philosophy in common with the message-passing harness, and are intended to be portable with interfaces to PC graphics or X11 windows for instance. We will not discuss here either the graphics or the basic message-passing layer. The original documentation is easily obtainable if details are required.

The global operations in Express are of two kinds, firstly data exchange and decomposition tools and secondly i/o control. They have many aspects similar to our own Fortnet harness, but work with locally-defined vector data in a way similar to the Intel global operations. There is however a more powerful set of decomposition tools to help form these local vectors from the global data structure underlying the problem. For i/o there are several modes suitable for most applications which allow existing programs to be run intact on the nodes, Every processor also has access to the operating-system capabilities of the host processor attached to the network. This is handled by a host process called Cubix which is

similar to the Fortnet Server process or the Inmos AFServer. The Express system, like the Intel communications and the Argonne software, requires typed messages. Some of the relevant routines and commands are now listed, only the ones used in global operations are described in detail.

User Commands:

acctool -- analyse accounting data
cnftool -- configure transputer systems
ctool -- analyse communication profile data
cubix -- download and execute Cubix programs, i/o server
etool -- analyse event profile data and 'toggles'
excustom -- modify Express system parameters
exdump -- retreive data from RAM files
exinit -- reboot and reload Express kernel
exreset -- reset transputer system
exstat -- display node useage information
ndb -- source level debugger
xtool -- analyse execution profile data

System initialisation:

kxinit() -- start up Express and initialise xpress common block. This routine must be called before any others on both host and nodes. The common block has its various integer variables set as follows:

common/xpress/nocare,norder,nonode,ihost,ialnod,ialprc

These variables contain special values used as parameters to the communication librarytelling it to ignore selection criteria, send to host, or send to all other nodes.

kxpara(ienv) -- determine run-time configuration. This yields the values ienv(1), processor number of the calling nodes; ienv(2), number of processors in group; ienv(3) processor group index; ienv(4), process identifier. The information provided by this call is used with the decomposition procedures to ensure reconfigurable applications.

Processor allocation and control:

kxclos() -- deallocate processor group
kxload() -- load program into all nodes
kxmain() -- start execution of main program
kxopen() -- allocate froup of processors
kxpaus() -- arrange for program to be loaded 'stopped'
kxpid() -- translate UNIX pid to Express process id
kxploa() -- load a program into a single node

kxshar() -- share a processor group between multiple host programs

kxstar() -- start execution of a node program

Basic communication system:

kxexct() -- define meaning of 'read/write' wildcards

kxinct() -- define meaning of 'read/write' wildcards

kxread() -- read a message

kxtest() -- test an incoming message, non-blocking

kxvrea() -- read a vector message

kxvwri() -- send a vector message

kxwrit() -- send a message

Global communication system

kxbrod() -- interprocessor broadcast

kxchan(ibuf, ilen, isrc, itype, obuf, olen, odest, otype) -- Synchronous data exchange between source node isrc and destination odest.

kxcomb(buffer, func, size, items, nnodes, nodel, type) -- Apply user-supplied integer function func(v1, v2, size) to distributed data. The routine broadcasts local values of buffer() to all other processors, they receive the values and perform the operation v1=func(v1,v2) where v1 is a local sub-vector of buffer of length size, and v2 the new subvector of length size received in a message of type type. The function returns an integer error code. This is repeated items times so buffer must be of total length size*items elements long. size may thought of as a vector stride. nodel is a list of nodes which will take part in the operation.

kxconc(mybuf, mybyte, resbuf, ressz, sizes, nnodes, nodel, type) -- concatenate data from nodes, effectively bringing the distributed data from local vectors mybuf into a single large vector resbuf, which is broadcast to all nodes in nodel.

kxsync() -- synchronise all node processors

kxvcha(ibuf, isize, ioff, iitems, isrc, itype, obuf, osize, ooff, oitems, odest, otype) -- synchronous vector exchange, similar to kxchan

Asynchronous communication system:

kxhand() -- install asynchronous message handler

kxrecv() -- read a message, non-blocking

kxsend() -- send a message, non-blocking

Hardware dependent communication system:

This provides access to the underlying hardware channels for the most efficient form of communication, it was originally designed to allow optimal use of the 3L Parallel FORTRAN language (see also [19]).

kxchon() -- re-enable Express processing on a channel
kxchof() -- disable Express processing on a channel
kxchrd() -- ready bytes from disabled channel
kxchwt() -- write bytes to disabled channel

Decomposition tools:

These so-called kxgrid utilities perform automatic decomposition of user domains onto the underlying machine topology. A problem specified as a cartesian grid in N dimensions is mapped onto the topology, and routines are availeble to provide the information for the communication calls to work in the user's topology, i.e. to define the mappings between the global data structure and local vectors of data. Note that none of these functions actually do any communication, they merely provide parameters for the user to program his own communicatio algorithms. The parameters may however also be put into the i/o system calls which share data from a file between the nodes. This is perhaps the most powerful feature of Express and is illustrated below.

kxgdbc(perbc) -- define boundary conditions on user domain. BY default they are periodic with each cartesian coordinate wrapping around at the two extreme ends of its axis. perbc is an N dimensional array of values which, if zero, suppress the periodicity for that dimension

kxgdco(procno, coord) -- determine position in user domain of a particular processor. Given processor number procno, the array coord contains the position in the cartesian coordinate space of this processor with the given split

kxgdin(grddim, nprocs) -- initialise decomposition system, this performs an elementary mapping and is called before the other routines. grddim is the number of dimensions N, and nprocs is the number of processors to be used.

kxgdno(procno, dir, dist) -- determine communication parameters from user domain. Given a processor procno (usually the calling processor ienv(1)), this returns a value which is the number of the processor dist away in the dirth dimension of the cartesian space. This can then be used as a parameter to the elementary communication functions.

kxgdpr(coord) -- map user domain to processor number, returns the number of the processor which is used for the cartesian coordinates coord

kxgdsi(procno, global, size, start) -- distribute data among processors. procno is usally the calling processor (ienv(1)), global

is an array containing the size in each dimension of the global data, size is again an array which contains the size in each dimension of the local part of the decomposed data structure belonging to procno. start is the global offset which corresponds to zero offset in the local data.

kxgdsp(nodes, grddim, nsplit) -- distribute processors on user domain. This divides up the nodes processors evenly between the grddim dimensions. The number in each direction is returned in the array nsplit(). This is called after kxgrid.

i/o:

This is probably the most powerful feature of Express, and also features in its graphical interface in an obvious way. Data is considered to be one- or two-dimensional and can be automatically split up over the nodes under control of the Cubix or Plotix host program. A similar function is performed by the Fortnet server process.

kabort(status) -- immediately terminate node program and print status on the standard output device

kmulti(lu) -- switch file i/o to 'multi'

ksingl(lu) -- switch file i/o to 'single'

isasyn(lu) -- inquire file i/o mode

ismult(lu) -- inquire file i/o mode

kmread(lu, buf, length, order) -- read independent data into nodes

kmrd2d(lu, buf, totcol, totrow, itemsz, col0, col1, row0, row1, skip) -- read two-dimensional data set into nodes

kmwrit(lu, buf, length, order) -- write independent data from node

kmwt2d(lu, buf, totcol, totrow, itemsz, col0, col1, row0, row1, skip) -- write two-dimensional data set from nodes

kcbxsy(flag) -- assign overall synchronous/asynchronous mode

An example of using kmrd2d follows:

```
...
integer ienv(4),gsize(2),lsize(2),offset(2)
c number of points in x and y directions
data nx,ny/...,.../
call kxinit()
call kxpara(ienv)

call kxgdsp(ienv(2),2,ndim)
istat=kxgdin(2,ndim)
gsize(1)=nx
gsize(2)=ny
call kxgdsi(ienv(1),gsize,lsize,offset)
```

```
c read in the data overlapping the edges of each domain by a strip
c one item wide
c global starting and finishing x offsets of part required
     nxstart=offset(1)-1
     nxend=offset(1)+lsize(1)
c global starting and finishing y offsets of part required
     nystart=offset(2)-1
     nyend=offset(2)+lsize(2)
c size of data columns
     nxdim=lsize(1)+2
c perform reading from unit lu into local storage data by all
c processors. This is coordinated by the host program and only the
c part of the data actually required is sent in the form of
c intermediate vecotr buffers
     istat=kmrd2d(lu, data, nx, ny, 4, nxstart, nxend, nystart,
    &             nyend, nxdim)
     ...
```

Multitasking:

A library of functions to handle process threads is provided.

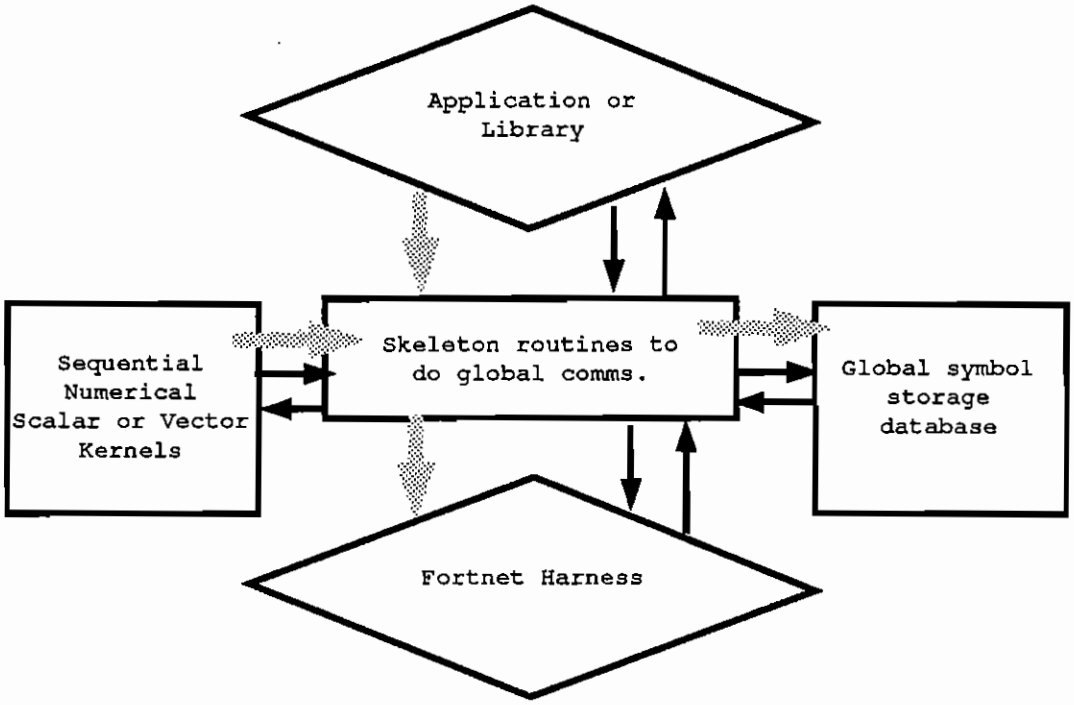
kexec() -- overlay a node program with another
kxhand() -- install asynchronous message handler
kxsema() -- allocate and initialise a semaphore
kxsemf() -- deallocate a semaphore structure
kxsems() -- exit a critical section and signal any waiting proceses
kxsemw() -- attempt to enter a critical section, waiting if necessary
kxslee() -- suspend process for an indicated time

### III.7 Fortnet 'skeleton' library. PARLANCE.

It is now necessary to describe the Fortnet global communication library. This differs significantly from other developments and depends fundamentally on firstly a set of globally defined symbolic variables which reference real data somewhere in the multicomputer's memory, and secondly, on a set of routines which control parallel overlapped communications and allow movement of data between these variables. This is summarised in figure 6.

All the symbolic variables in fact reference contiguous global vectors of numbers. Any other form of matrix can be mapped on to these by calculating the indices (routines are provided to do it) but, as will be seen below, that is not often necessary. Symbols are entered into a distributed database which contains details of actual data storage, or may in fact just be references to pointers in

Figure 6. Structure of PARLANCE library.



shared memory. Routines are provided to find which elements belong to a given process, the next element on that process from a given one, or which process has a given element. Other routines assign space, put and fetch real data from the symbolic storage, perform gather, scatter and more complex operations. This is done by interrogating and manipulating the database.

Global communications routines -- Canonical definitions

Most of the global communication routines involve either pairs of processors or higher orders (k-fold) sets. They must be called in a loosely synchronous manner on all processors which may be involved in communication. This precludes any use of global routines inside critical sections of code guarded by Fortnet locks or blocked for any other communication. Error messages are printed if this rule is violated.

assign('a',nbytes) -- assign nbytes of local memory for storage of data in symbolic variable 'a' (up to eight characters) on the processor which calls it. Enter 'a' into the symbol table database for this processor. This is a monadic routine and may be called any number of times by each processor to produce 'degenerate' table entries. In this way matrices may be stored by columns or blocks for instance since contiguous storage follows the first entry for each procesor, then the second, etc.

swaptable() -- global exchange of symbol tables between all processors. This need never be called by the user.

put(iproc,x,offx,stepx,'a',offa,stepa,n) -- This is how actual data is placed in the global storage, it copies n elements of data from real variable x defined locally on processor iproc into globally defined memory referenced by symbol 'a'. Offsets for start of data and strides are given. offx is the starting offset of the local vector, but offa is a global offset. Communications are internal and 'a' is as defined in previous assignment calls. A distributed scatter operation.

fetch(iproc,'a',offa,stepa,x,offx,stepx,n) -- reverse of above. A distributed gather operation.

gather0('a',offa,stepa,'b',offb,stepb,n) -- on the processor on which this is called, takes all available elements of 'a' starting from the given offset and with the given stride and attempts to store them in given elements of 'b' if such are in local memory area. This is a monadic routine.

scatter1(i,j,'a',offa,stepa,n) -- finds all available elements of 'a' as described which are present on processor i and attempts to send them as a single message to processor j. The data is preceeded by protocol as follows:

protocol index

```
     number of index bytes

     index bytes
     number of data bytes
     data
```

The protocol index is

```
     0 -- no data available, abort
     1 -- data as requested
     2 -- partial data, single offset and stride description
     3 -- partial data, separate offset sent for each element
```

This is a dyadic operation.

gather1(i,j,'b',offb,stepb,n) -- receives a message from processor i on processor j and attempts to store the data into 'b' in local memory if possible as described. A dyadic operation.

gather1, scatter1 and gather0 are not usually called by the user as they are too 'low level'. They do however form the basis of higher routines and were described for that reason. They are very powerful routines and quite novel in their behaviour. They implicitly carry out masking so that only elements which can actually be stored in local memory on j are sent. The message contains the 'intermediate' or 'compressed' vector bij.

gather2(j,'a',offa,stepa,'b',offb,stepb,n) -- collects data referenced by symbol 'a' from all processors, including processor j, and attempts to store it in 'b' on processor j. It builds up the sum of compressed vectors bj = sum bij, bj is the jth 'segment' of b. May be expressed as follows in terms of other routines.

```
     subroutine gather2(j,'a',offa,stepa,'b',offb,stepb,n)
     ...
     do i=1,nproc
     if(i.eq.inode.and.i.ne.j)
    1   call scatter1(i,j,'a',offa,stepa,n)
     if(inode.eq.j.and.i.ne.j))
    1   call gather1(i,j,'b',offb,stepb,n)
     end do
     if(inode.eq.j)
    1   call gather0('a',offa,stepa,'b',offb,stepb,n)
     end
```

This already begins to illustrate the power of the system. It is possible to build quite sophisticated routines from these more simple ones.

Other important subroutines are provided to determine which elements of the symbolic vectors are on which processors:

nextelement(iproc,'a',offa,exist) -- determines the next element on iproc of symbol 'a' following offa and changes offa to point to it. If it exists exist=.true. otherwise exist=.false. and no action

is taken.

deassign ('a') -- removes the symbol 'a' from the database entry of this processor only if it was the last entry, otherwise a warning message is printed on stdout. This is useful for creating temporary variables in a subroutine using assign. It will have to be called several times for degenerate entries. Note that the data is not lost, and may be picked up by a different assignment. This may be useful as a trick to change the format in which global data is distributed, but I have not tried it.

Covering Scheme

An important requirement of global communications is that they should all work in parallel. This question has not been addressed in the above routines since they all target a single processor so no overlap is possible. More complicated routines will require communications between constituants of all pairs of processors (e.g. in a matrix transpose), or of all sets of k processors in the array of nproc processors. It is possible to optimise the parallelism in this by 'covering' the machine's nproc processors with as many independant k-fold connected subsets as possible, and allow communications within each to be concurrent with the others, and then with another covering with different connections and so on until all possibilities have been exhausted. As mentioned in the introduction, this is a suitable strategy for coarse-grained machines. The solution is from permutation theory:

The number of different ways of covering nproc processors with k of them = ...

For pairs I have written a function

```
ncover=dyadic(nproc)
```

and for triads

```
ncover=triadic(nproc)
```

Higher schemes are possible. Of course if nproc is large, then ncover is VERY large illustrating a fundamental difficulty of global communications schemes - the inflation in data movement.

To use the above covering schemes a 'connection matrix' can be accessed.

```
common/cover/connect(nproc)
```

the nth entry of this indicates that processor n should talk to processor connect(n) unless connect(n)=-1. Degenerate entries are not allowed (they are 'redundant').

The matrix is filled by calling

covering(i,k,nproc) -- for the ith k-fold covering of nproc

processors.

These routines are written in C and are monadic.

I hasten to add that this is just one possibility for a general scheme. Hand optimisation will be used wherever possible, and other schemes could be tried. Hand optimisation in all cases would be time-consuming and I wish to avoid difficulties of that kind.

In addition to overlapping communications some systems allow the possibility to do 'asynchronous' communications which overlap computational activity. This introduces difficulties of data integrity if one is not careful to check that variables are not used until they have been updated, or are not changed before they have been transmitted. I have chosed to omit these issues from the present discussion.

Topology Routines for Optimal Storage.

As with other software the efficiency of any paritcular parallel algorithm will be governed by the actual data placement. The optimal placement is likely to vary from routine to routine and, whilst we aim to have routines which will give numerically correct results for any distribution of data, it may be better to re-arrange the input if possible. This re-arrangement is in fact performed on entry to some of the more intensive routines.

storebycol(...) -- arrange the storage of matrix 'b' to cover the nproc processors with interleaved columns of length lencol, a band of width widcol is put on each processor. Carry out the redistribution of data from matrix 'a'. The original storage is left in situ.

storebyblock(...) -- similar to the above, but distributing blocks of length lenblk and width widblk on each processor. These are interleaved and the communication is done to distribute the actual data.

I note that some very similar ideas on global data structures and their redistribution in linear algebra routines have gone into thinking on the GENESYS project of MIMD Systems [47, 48]. The GENESYS parallel library is currently written with Helios in mind as the operating and communication system for a transputer array attached to a VAX host [59]. Different levels of access to the array are possible ranging from 'opaque' in which the host program automatically calls parallel library routines on the array, to 'transparnt' with explicit control over message passing.

In the GENESYS software some emphasis is placed on Distribution Indicators (DIs) which can be carried around with the global data sets. They tell the system the type of storage used, e.g. column,

block etc. and obviate a great deal of database searching since the distributions are then pre-defined for special cases (e.g. full columns of a 2D matrix of order m on N processors). This could be used to improve the efficiency of the Fortnet system, but has not yet been investigated in detail.

Examples

The following example is the kernel of a matrix multiplication a=b*c where all three matrices may be distributed across the whole machine. It shows a way of using the dyadic covering with masks when not all processors are needed. This can be understood by reference to the matrix times vector multiply example which will be described in more detail in section IV and gather2 above.

In simple terms the algorithm is as follows, using FORTRAN-90 notation

```
do i=1,n
v1(1:n)=b(i,1:n)
do j=1,n
v2(1:n)=c(1:n,j)
a(j,i)=ddot(v1,1,1,v2,1,1,n)
end do
end do
```

Note that it separates moving the data into two vectors from the actual vector multiplication. The ddot routine is clever enough to execute the following code:

```
do i=1,n
do j=1,n
temp=ddot('b',i,n,'c',(j-1)*n+1,1,n)
call puta(temp,'a',(j-1)*n+i)
end do
end do
```

and will carry out communications internally for each vector multiply. However there will be no overlapping so this would be less efficient.

One solution to that problem is as follows, it is still not ideal as it collects vectors v1 and v2 repeatedly before the ddot operation instead of leaving v1 until all values of v2 have been used.

```
subroutine matmul(a,b,c,n)
character*8 a,b,c
real*8 temp
logical exist(nproc),exist1(nproc)
dimension in(nproc),jn(nproc)
common/cover/connect(nproc,2)
```

```
      data exist,exist1,in,jn/nproc*.false.,nproc*.false.,
     1                        nproc*0,nproc*0/
      call assign('v1',n*8)
      call assign('v2',n*8)
      kount=0
10    continue
      m=dyadic(nproc)
      do i=1,m
      do j=1,nproc
      exist1(j)=exist1(j).or.exist(j)
      exist(j)=.false.
      end do
      call covering(i,2,nproc)

c flow control loop
      do k=1,nproc
      k2=connect(k)
      if(k2.ne.-1)then
      if(.not.exist(k))call nextelement(k,a,jn(k),exist(k))
      if(.not.exist(k2))call nextelement(k2,a,jn(k2),exist(k2))
      in(k)=mod(jn(k),n)
      in(k2)=mod(jn(k2),n)
      if(k.eq.inode)then
      if(exist(k2))then
      call scatter1(k,k2,b,in(k2),n,n)
      call scatter1(k,k2,c,jn(k2)-in(k2)+1,1,n)
      end if
      if(exist(k))then
      call gather1(k2,k,'v1',1,n)
      call gather1(k2,k,'v2',1,n)
      end if
      else if(inode.eq.k2)then
      if(exist(k2))then
      call gather1(k,k2,'v1',1,n)
      call gather1(k,k2,'v2',1,n)
      end if
      if(exist(k))then
      call scatter1(k2,k,b,in(k),n,n)
      call scatter1(k2,k,c,jn(k)-in(k)+1,1,n)
      end if
      end if
      end if
      end do
      end do
      do i=1,nproc
      if(exist1(i))kount=kount+1
      end do
c monadic part completely parallel
```

```
      if(exist1(inode))then
      call gather0(b,in(inode),n,'v1',1,1,n)
      call gather0(c,jn(inode)-in(inode)+1,1,'v2',1,1,n)
      call brall('OFF')
      temp=ddot(1,'v1',1,1,'v2',1,1,n)
      call puta(temp,a,jn(inode))
      call brall('RESTORE')
      end if
      if(kount.lt.n*n)goto 10
c release virtual storage space
      call deassign('v2')
      call deassign('v1')
      end
```

The above code is somewhat complex and detailed study of it should perhaps be resumed after reading section IV. Almost all the code is doing communications, which proves that we must produce more powerful routines to simplify this. Normally applications would be written at a higher level in which no reference is made to a processor's identity. A second example, which is a matrix transpose, will demonstrate a more difficult line of development.

A good matrix transpose program is difficult to write, partly because the sequential version is so simple, partly because it ought to involve no computation so that the overhead in doing parallel communications and computing offsets are disasterously highlighted. This appears to be a case where a hand-coded solution with knowledge of the distribution of elements is needed. Let us however examine the more general problem in a series of steps. Firstly the sequential version

```
      do i=1,n
      do j=i+1,n
      temp=a(i,j)
      a(i,j)=a(j,i)
      a(j,i)=temp
      end do
      end do
```

Next a vector version

```
      do i=1,n-1
      k=i+1
      v1(k:n)=a(i,k:n)
      v2(k:n)=a(k:n,i)
      a(i,k:n)=v2(k:n)
      a(k:n,i)=v1(k:n)
      end do
```

This can trivially be expressed in a simple parallel form using

our Fortnet routines, but there is no overlapping communications!

```
      if(inode.eq.1)then
      call assign('v1',n*8)
      call assign('v2',n*8)
      end if
      do i=1,n-1
      k=i+1
      call dcopy(n-k+1,'v1',1,1,'a',(k-1)*n+i,1)
      call dcopy(n-k+1,'v2',1,1,'a',(i-1)*n+k,1)
      call dcopy(n-k+1,'a',(k-1)*n+i,1,'v1',1,1)
      call dcopy(n-k+1,'a',(i-1)*n+k,1,'v2',1,1)
      end do
      if(inode.eq.1)
      call deassign('v2')
      call deassign('v1')
      end if
      end
```

Whilst the above program should execute and carry out all the communications required implicitly, it would be hideously slow. One truly parallel matrix transpose can be done by employing a trick. That is to create an identical copy of the storage for symbol 'a' and call it 'b'. The elements are on the same processors as those of 'a' so that masking in the gather1 and scatter1 routines will work properly. Now we can use dyadic communications to put the transpose of a into b. Then do one big monadic gather at the end to move it back and delete b!

```
      call assign('b',n*n*8/nproc)
      m=dyadic(nproc)
      do i=1,m
      call covering(i,2,nproc)
c flow control loop
      do k=1,nproc
      k2=connect(k)
      if(inode.eq.k)then
      do l=1,n-1
      p=l+1
      call scatter1(k,k2,'a',(p-1)*n+l,n,n-p+1)
      call scatter1(k,k2,'a',(l-1)*n+p,1,n-p+1)
      call gather1(k2,k,'b',(l-1)*n+p,1,n-p+1)
      call gather1(k2,k,'b',(p-1)*n+l,n,n-p+1)
      end do
      else if(inode.eq.k2)then
      do l=1,n-1
      p=l+1
      call gather1(k,k2,'b',(l-1)*n+p,1,n-p+1)
      call gather1(k,k2,'b',(p-1)*n+l,n,n-p+1)
```

```
      call scatter1(k2,k,'a',(p-1)*n+1,n,n-p+1)
      call scatter1(k2,k,'a',(1-1)*n+p,1,n-p+1)
      end do
      end if
      end do
      end do
c remaining monadic part
      do 1=1,n-1
      p=1+1
      call gather0('a',(p-1)*n+1,n,'b',(1-1)*n+p,1,n-p+1)
      call gather0('a',(1-1)*n+p,1,'b',(p-1)*n+1,n,n-p+1)
      end do
      call gather0('b',1,1,'a',1,1,n*n)
      call deassign('b')
      end
```

This involves a factor 4 times more data movement than is actually required; a factor 2 times too much between nodes!

Finally here is a truly parallel version that manipulates the matrix in situ and involves a minimum of data movement. It does require quite a lot of index computation which, however, is done concurrently on the processor pairs.

```
      m=dyadic(nproc)
      call brall('OFF')
      do i=1,m
      call covering(i,2,nproc)
c flow control loop
      do k=1,nproc
      k2=connect(k)
c let processors k and k2 transpose any elements they can, like in
c the sequential version
      do 1=1,n-1
      do p=1+1,n
      if(inode.eq.k.or.inode.eq.k2)then
check for element a(1,p) on k
      isk=(p-1)*n+1
      ism1=isk-1
      call nextelement(k,ism1,'a',exist)
      if(ism1.eq.isk)then
check for element a(p,1) on k2
      isk2=(1-1)*n+p
      ism1=isk2-1
      call nextelement(k2,ism1,'a',exist)
      if(ism1.eq.isk2)then
c found it, swap elements
      if(inode.eq.k)then
      temp=fetcha('a',isk2)
```

```
      temp2=fetcha('a',isk)
      call puta(temp2,'a',isk2)
      call puta(temp,'a',isk)
      else(if.inode.eq.k2)then
      temp=fetcha('a',isk2)
      call puta(temp,'a',isk2)
      end if
      end if
      end if
check for element a(l,p) on k2
      isk=(p-1)*n+l
      ism1=isk-1
      call nextelement(k2,ism1,'a',exist)
      if(ism1.eq.isk)then
check for element a(p,l) on k
      isk2=(l-1)*n+p
      ism1=isk2-1
      call nextelement(k,ism1,'a',exist)
      if(ism1.eq.isk2)then
c found it, swap elements
      if(inode.eq.k)then
      temp=fetcha('a',isk)
      temp2=fetcha('a',isk2)
      call puta(temp2,'a',isk)
      call puta(temp,'a',isk2)

      else(if.inode.eq.k2)then
      temp=fetcha('a',isk)
      call puta(temp,'a',isk)
      end if
      end if
      end if
      end if
      end do
      end do
      end do
      end do
c remaining monadic part
      do l=1,n
      do p=l+1,n
      isk=(p-1)*n+l
      ism1=isk-1
      call nextelement(inode,ism1,'a',exist)
      if(ism1.eq.isk)then
      isk2=(l-1)*n+p
      ism1=isk2-1
      call nextelement(inode,ism1,'a',exist)
      if(ism1.eq.isk2)then
```

```
c both elements found
      temp=fetcha('a',isk)
      temp2=fetcha('a',isk2)
      call puta(temp,'a',isk2)
      call puta(temp2,'a',isk)
      end if
      end if
      end do
      end do
      call rball('RESTORE')
      end
```

Which of these various formulations will prove to be the most efficient depends on several issues. Firstly the size of the matrix dictates whether it must be handled in situ or not, secondly the speed of data transfer to other nodes compared to the individual processor speed will govern the choice of an algorithm which does redundant data movement or one which has expensive index calculations, loops and communications overheads for each matrix element. These issues cannot currently be decided due to the rapid evolution of hardware. It would be interesting to use the Fortnet profiler to investigate the differences between the different codes on current machines.

Remaining global communication and skeleton routines

One subset of the remaining routines are used to provide 'skeleton' forms of commonly-used communication patterns and call an external function which performs some transformation on one or more of the arguments. An example is the scalar-vector primitive which might be an elementwise addition, multiplication, fill, or more complex function like cosine. One skeleton routine global1a suffices to express all these forms and calls externals add, mult, fill, and intrinsic dcos to perform the correct action. The external function may be user supplied, and might be quite complicated since it could access data in common areas of the calling program. There is similarity here to routines provided by Intel and ParaSoft.

The real power of my approach to building parallel libraries is in the use of these 'skeleton' routines. The reason for this is

i) We accept that communications times will always be slow, or that the growth in data movement will kill off very 'fine-grained' programming

ii) current machines are 'medium grained' or 'coarse grained'

iii) programmability requires hiding the communication from the programmer so that he can think in his conventional model.

The outcome of this appraisal is that a set of 'communications

skeletons' is required for dyadic, triadic etc. operations into which very complicated operators may be substituted. Some of these skeletons are described below, and will be used to build specific vector routines in the following sections. I note that the use of skeleton routines is not original. Delves and Brown have already discussed the 'inside-out' nature of parallel routines, and circumvented the problem by the development of the Liverpool library sleeper and restriction of the user's program to the host computer (also facilitates multi-user programming).

Another further set of routines in PARLANCE obtain information from the system's symbol tables and may be used particularly to enquire about elements in a square matrix representation of the storage associated with a particular symbol. Possible questions are on what processor is element a(i,j) located, or what are the offsets and stride to describe the portion of the data belonging the a which is stored on the current processor?

List of routines in the present version:

subroutine allocate() -- initialise Fortnet global system. This must be done before calling any of the other routines.

subroutine extracta(A) -- extracts information about the symbol a from the tables and stores it in a common block:
common/extra/aproc(nproc),alocal(nproc),aglobal(nproc),lastex
where aproc(i) is the processor on which the ith entry resides (it might occur more than once if the storage is fragmented), alocal(i) is the offset from the beginning of the core storage are for the start of this piece of contiguous memory, aglobal(i) is the global offset of the start of this piece of virtual storage (in other words where it is in the complete vector a). Lastex just returns with the character string a to reduce redundant index calculations. The

number of elements of a stored on processor aproc(i) is clearly aglobal(i+1)-aglobal(i).

subroutine extractb(B) -- same as above but uses a common area
common/extrb/bproc,blocal,bglobal,lbstex

subroutine extractc(C) -- same as above but uses a common area
common/extrc/cproc,clocal,cglobal,lcstex

subroutine global1(A,iastart,iastride,in,fun) -- global skeleton for a=fun(a)

subroutine global2(C, iCstart, iCstride, A, iastart, iastride, in, fun) -- global skeleton for c=fun(a)

subroutine global3(a, iastart, iastride, b, ibstart, ibstride, in, fun) -- global skeleton for c=fun(a,b)

subroutine global1a(A,iastart,iastride,in,fun,alpha) - global skeleton for a=fun(alpha) where alpha is a real scalar

subroutine global2a(C,iCstart,iCstride,A,iastart,iastride, in,fun,alpha) -- global skeleton for c=fun(alpha,a)

subroutine global3a(c, icstart, icstride, a, iastart, iastride, b, ibstart, ibstride, in, fun, alpha) -- global skeleton for c=fun(alpha,a,b)

Various assorted routines are provided to expediate coding of higher-level algorithms as follows:

subroutine usea(a,idim,i,j,n) -- extracts information about element a(i,j) and stores it in the nth entry of tables in common common/liquorice/procs(nproc),offs(nproc),seg(nproc) where procs(n) is the processor which holds the element, offs(n) is the offset from the start of its local core memory and seg(n) is something else. This routine uses common extra for intermediate indices.

subroutine useb(a,idim,i,j,n) -- same as above but uses common extrb.

subroutine usec(a,idim,i,j,n) -- same as above but uses common extrc.

onea(ia,iacore,iaproc) -- for a symbol which has previously been extracted into common/extra/ this monadic routine finds the local core memory offset iacore and the id of the processor iaproc containing element with global offset ia.

oneb(ib,ibcore,ibproc) -- same as above for extrb

onec(ic,iccore,icproc) -- same as above for extrc

puta(temp,'a',ia) -- monadic routine, assuming the value of temp is globally present it puts into the local memory of whichever processor contains a(ia), uses common/extra/

putb(temp,'b',ib) -- same as above using common /extrb/

putc(temp,'c',ic) -- same as above using extrc

temp=fetcha('a',ia) -- fetches the value a(ia) and does a broadcast to all other processors, uses common/extra/

temp=fetchb('b',ib) -- same as above using extrb

temp=fetchc('c',ic) -- same as above using extrc

elementa(iproc,a,ifirst,jfirst,ilast,jlast,iapos,iadim) -- enquire to find what elements of a square matrix of dimension iadim are on processor iproc. Uses common extra and iapos points to the entry in the tables of this common block.

elementb(iproc,a,ifirst,jfirst,ilast,jlast,iapos,iadim) -- same as above but uses common extrb.

elementc(iproc,a,ifirst,jfirst,ilast,jlast,iapos,iadim) -- same as above but uses common extrc.

brall(directive) -- routine switches broadcasting on or off as described in section II above. It is useful in some algorithms to be able to use functions such as ddot in a monadic fashion without the requirement to broadcast results to all processors. Result is available only on the calling processors, and only if they contain data in the range of symbol indices. Possible values of the character string directive are 'ON', 'OFF', 'TOGGLE' and 'RESET'.

i=index2d(ndin,j,k) -- find the offest of element (i,j) from

the start of a 2-dimensional array of storage of column length ndim. A monadic routine.

brlist(proclist, nproc) -- installs the vector proclist into the database for subsequent broadcast operations, eraseing what was there already, but sets the entry for inode equal to -1 (a null node). This avoids a processor sending to itself.

IV General Programming Styles.

Most of this discussion has been placed in references [1-4] and we need say very little in addition. Some examples can however be given of a new programming style which, given the current memory limitations and slow i/o of multicomputers, may prove useful. It also illustrates simple calls to the global routines defined in the last section. I refer to this style as 'power sharing'.

Suppose a program requires a particularly large amount of data, and that the common 4 or 8Mbyte local-memory limit is insufficient. It may then be convenient to allow the processors to act in pairs, one simply storing some of the data and passing it to the second which performs some, but not all, of the computation. This is almost trivial using the routines described above. Other attempts to do this for particular applications have been made and the system need not be limited to pairs.

This scenario is the basis for a multiple memory server, however it would be rather inconvenient, but possible, to do the same using server protocols for messages requesting data transfer. Is it better to do this, or to store data on disk in a virtual-memory arrangement, and fully use the compute power of the extra nodes? The answer will depend on hardware available, at present i/o is slow. In any case it will be a useful exercise.

Suppose the matrix a is divided between processors iproc, which will be called the 'primary' processor, and iproc+1, which will be called the 'secondary'. This division is performed with the following assignment:

```
      character*8 a
      data a/'        '/
c just do code on one pair of procs, could do it on all pairs
      if(inode.eq.iproc.or.inode.eq.iproc+1)then
      a(1:1)='a'
c this gives a unique name to the symbol for this pair of procs
c and it is the same name for both procs in the pair
      write(a(2:3),'(i2)')iproc
      nproc=2
c a has n elements of 8 bytes each
      call assign(a,n*8/nproc)
c this assigns half of a to each processor
      end if
```

The local-memory data may be computed in situ by the processors executing concurrently, or in some other way. We now illustrate an operation which requires the data to be used on the primary processor, e.g. a matrix x vector multiply which yields a second

vector.

```
c v2=a * v1
      ncol=sqrt(n)
c assign vectors on iproc
      if(inode.eq.iproc)then
      call assign('v1',ncol)
      call assign('v2',ncol)
      call assign('temp',ncol)
c now compute contents of 'v1'
      ...
      end if
c now do multiplication
c first collect row of a into vector 'temp'
      do irow=1,ncol
      k=irow-1
      if(inode.eq.iproc+1)then
      call scatter1(inode,iproc,a,k*ncol+1,ncol,ncol)
      else if(inode.eq.iproc)then
      call gather1(iproc+1,inode,'temp'1,1,ncol)
c remaining monadic part
      call gather0(a,k*ncol+1,ncol,'temp'1,1,ncol)
      end if
c now do vector dot product on iproc
      call brall('OFF')
      x2=ddot('temp',1,1,'v1',1,1,ncol)
      call puta(x2,'v2',irow)
      call brall('RESET')
      end do
      ...
```

Notice that the code is executed by both processors and only in the gather/scatter part do we need to distinguish. We might alternatively replace this by the following calls on both processors (see section III.7):

```
      do irow=1,ncol
      k=irow-1
      call gather2(iproc,a,k*ncol+1,ncol,'temp'1,1,ncol)
      call brall('OFF')
      x2=ddot('temp',1,1,'v1',1,1,ncol)
      call puta(x2,'v2',irow)
      call brall('RESET')
      end do
```

This is much more consise and illustrates how I wish to simplify the programmer's job so that he may concentrate on more

important things. Furthermore the above code fragment would have worked equally well if the virtual matrix a were distributed over more than two processors.

Of course the ddot routine is more powerful and the above code might trivially be replaced by

```
call brall('OFF')
do irow=1,ncol
k=irow-1
temp=ddot(a,k*ncol+1,ncol,'v1',1,1,ncol)
call puta(x2,'v2',irow)
end do
call brall('RESET')
```

The original coding might however represent its internal working (but does not since optimisation was done).

## V Parallel Numerical Libraries

We will now dive into a description of parallel software which has been used for numerical algorithms for linear algebra. Omitted from this description is the GENESYS project of MIMD Systems, which was aimed particularly at sparse matrix decomposition and solution. No doubt there are also other linear algebra or BLAS libraries which have been adapted for distributed execution and of which I am unaware.

### V.1 Intel Eiscube and Lincube

A parallel version of the Eispack library [55,35,36] 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.

The following routines are used:

subroutine tridib(N,EPS1,D,E,E2,LB,UB,M11,M,W,IND,IERR,RV4,RV5) -- Bisection method of Barth, Martin and Wilkinson fo find eigenvalues of a tridiagonal symmetric matrix within specified boundary indices. Called by psytqr

subroutine psytre(A,LDA,N,M,P,ID,D,E,Z,WORK) -- Parallel SYmmetric Tridiagonal REduction. Reduces a real symmetric matrix to tridiagonal form

subroutine psytqr(D,E,SIGMA,N,X,MM,WORK,JOB,INFO) -- Parallel SYmmetric Tridiagonal QR. Computes the eigenvalues and optionally the eigenvectors of a real-symmetric tridiagonal matrix.

subroutine pflip(ID,N,M,P,AR,AC,W) -- transpose a matrix stored by rows into one stored by columns

The Lincube library developed from Linpack [7] for the Intel hypercube analyses and solves systems of linear algebraic equations involving dense matrices. Data is again distributed columnwise as in Eiscube.

The following routines are used:

subroutine pgesl(A,LDA,N,M,P,ID,IPVT,B,WORK,MSG) -- "Cornell" algorithm for parallel linear equation solver

subroutine pgemul (A,lda,n,m,p,id,x,y) -- computes y=A*x where

matrix A is distributed over p nodes, x is on node 0 and y is on node 0

subroutine pgefa(A,LDA,N,M,P,ID,IPVT,BUF) -- Parallel GEneral matrix FActor. Parallel version of dgefa from Linpack. LU factorisation by Gaussian elimination

There are also Intel implementations of parallel code to perform two and three-dimensional fast Fourier transforms. These call highly optimised assembler-coded one-dimensional routines for the primitive transform on the iPSC/2 vx and i860 nodes, but still require a transpose of one plane in the 3D space which is the most costly part of the algorithm.

### V.2 Topexpress Parallel Library

The former parallel processing division of Topexpress Ltd. [29] (Cambridge) wrote a number of libraries for T800 transputers. These were sequential and vector routines for single-transputers, and concurrent routines. The latter are called from a sequential host program and each call initiates a load of the relevant code into the transputer array via an executive. The library is available for Meiko FORTRAN, C and occam and is partly written in T800 assembler code, so is not portable to other systems. Routines provided include: Fast Fourier transforms, general matrix routines, eigenvalue/eigenvector routines, symmetric matrix routines, sparse matrix routines, iterative equation solvers and sorting routines. The routines can be run on a system with any number of transputers, and each routine has the same interconnect. Some typical routines are (I don't have a complete set of documentation):

GAUSSP -- solve a set of linear equations with partial pivoting for a single right hand side

CHLSLP -- solve a symmetric positive definite matrix equation with a single right hand side

STURMP -- calculate all eigenvalues/eigenvectors of a symmetric matrix by Sturn sequences

R2DFTP -- two-dimensional real to complex fast Fourier transform

ISORTP -- integer sort

### V.3 Liverpool, NAG (Supernode) and NA Software Limited

A large amount of systematic work has been done by the Liverpool group to investigate parallel algorithms and incorporate them into a library interface. Results of this work are available as reports to the original EEC Esprit P1085 project [26] and SERC EMR (ExtraMural Research) project [25].

Parallel codes for transputers are currently written in occam

or a language such as Fortran with an occam harness providing the communications. All occam channels must be placed explicitly, and they must correspond in a one-to-one way with transputer links. This has lead to the provision of pre-written harnesses with holes for the insertion of user code as the basis of library facilities. The drawbacks of this are:

i) only limited facilities are provided unless underlying information is available about data storage across the whole global memory

ii) It is difficult to adapt old Fortran codes for the new machines

The Liverpool group has however the following aims:

i) provide an environment similar to a sequential library for calling routines from a single host program

ii) allow more than one routine to be called with the same data placement

iii) routines should be pre-compiled and be

iv) calleable from occam, Fortran, C or Pascal

Furthermore a study was made of:

i) techniques of writing parallel algorithms for transputer arrays

ii) design an error mechanism similar to NAG

iii) writing a small number of finished routines

The work aimed at efficiency, and made a study of different arrays and the theoretical way the ratio of communications/cpu time changes the performance of an algorithm [see 1].

A basic premise of the work, mentioned in section I, is that the user of the library must have a single serial Fortran program from which the library modules may be called. This imposes the constraint that at the beginning of a routine data must be 'fanned out' from the host to the nodes. Some alternative methods have been devised for data that is already in place.

A discussion of the efficiency of algorithms related to the average path length in a given topology indicated that a two-dimensional grid was best for transputers (with 4 links). The older routines which were developed had however a daisy chain, grid, or tree topology depending on the routine. This is of course not a problem with routing hardware.

Delves and Brown [25] give a discussion of what they call the 'inside out' structure of parallel libraries which I refer to as a 'skeleton' structure. They concluded that for a single user program running on a host transputer and calling a library on an attached array, this structure could not be maintained. They believe that a single host calling program per parallel application is an

essential methodology for multi-user systems. The aim of the project was therefore to provide a paradigm that was identical to that found in a serial environment and could ultimately support multiple users.

The underlying library strategy is therefore as follows. A static harness is provided on the transputer array, referred to as a 'library sleeper'. This is able to load code, data or both onto the array from the master transputer, and execute the library routines. It provides a rudimentary overlay facility for multi-user systems.

The major impact of the Liverpool work has been in the theoretical evaluation of algorithms, and in the construction of an error mechanism (similar to NAG). Some final routines are fully documented and now marketed by NA Software Limited [17] (although I don't have a complete set of documentation). They are available for the following transputer arrays: Supernode, P1085, Meiko Computing Surface, Inmos Item, PC boards, Atari workstations, and are described as follows using occam syntax:

```
PAR
  ... other routines if desired
  ERROR.HARD(message,reply,errnum)
  Blas101(vector1,vectors,count,n,message,reply) -- evaluates
the inner product of two vectors. Error diagnostics are sent over
channel message, the reply channel can be used for the error
recovery handler. Chain topology.
PAR
  ERROR.HARD(message,reply,errnum)
  Blas201(matrix,vector,result,n,message,reply) -- evaluates a
matrix*vector multiplication. Chain topology
PAR
  ERROR.HARD(message,reply,errnum)
  Blas301(A,B,C,work,n,message,reply) -- Multiplies square
matrices A*B to obtain matrix C. Square array topology.
PAR
  ERROR.HARD(message,reply,errnum)
  LinAlg.Sing01(matrix,rhs,xvec,matrix.size,message,reply) --
Uses gaussian elimination with partial pivotting to solve Ax=b for
large dense systems with a single right hand side. Ring topology
PAR
  ERROR.HARD(message,reply,errnum)
  LinAlg.Sing02(amat,rhs,xvals,no.of.procs,message,reply) --
Solves Ax=b as above with multiple right hand sides handled in
parallel. Daisy-chain topology
PAR
  ERROR.HARD(message,reply,errnum)
  LinAlg.MM01(...) -- calculates the matrix produce c=ab. The
matrices need not be square.
PAR
  ERROR.HARD(message,reply,errnum)
```

```
      LinAlg.TriEg01(...) -- This routine solves TX=B for large
dense systems, where T is an upper triangular matrix. Parallelism is
achieved by placing columns of  the right hand side B  on each
processor and sending rows of the triangular matrix T in reverse
order.
    PAR
      ERROR.HARD(message,reply,errnum)
      LinAlg.Eig01(matrix,eval,evec,matrix.size,tolerance,
          max.loops,no.loops,message,reply) -- Evaluates a single
eigenvalue-eigenvector pair of the system Ax=kx using an iterative
inverse-power method given an estimate of the required eigenvalue.
Ring topology
    PAR
      ERROR.HARD(message,reply,errnum)
      LinAlg.Sparse01(nproc,xdir,ydir,overlap,rhs,x,pointer,
          max.loops,no.loops,tolerance,message,reply) -- Solves a
system of linear equations Ax=b using a sparse varient of Gaussian
elimination, where A is a block tri-diagonal, symmetric, positive-
definite matrix. Rows of A are distributed and a chain of
transputers is used.
    PAR
      ERROR.HARD(message,reply,errnum)
      LinAlg.BlockTriDiag(...) --Solves a system of linear
equations Ax=b, where A is a block tri-diagonal , symmetric,
positive definite matrix. Parallelism is achieved by 'tearing' the
region to compute the solution for the torn blocks in separate
processors
    PAR
      ERROR.HARD(message,reply,errnum)
      Linprog.Simplex(...) -- Finds the minimum of a linear
function subject to constraints using the Simplex method. Holds
blocks of constraint rows on each processor
    PAR
      ERROR.HARD(message,reply,errnum)
      poly.Solve(...) -- calculates approximate values for the
zeros of an nth degree polynomial. A number of zeros are calculated
on each transputer
    PAR
      ERROR.HARD(message,reply,errnum)
      FFT.Complex01(...) -- Performs an f=FFT on complex sample
data which is distributed among the processors. Each processor
performs its own FFT and passes it back one level  for combination
with the other data.
    PAR
      ERROR.HARD(message,reply,errnum)
      FFT.Complex02(...) -- Uses decimation in time and distributes
the data in a tree. Assumes a binary tree configuration of
```

transputers
PAR
ERROR.HARD(message,reply,errnum)
FFT.Multiple.Complex(...) -- Uses time decimation on complex radix.s input samples. Transforms are distributed among the transputers.
PAR
ERROR.HARD(message,reply,errnum)
Sort.Merge01(...) -- sorts a vector of input values into ascending or descending order. When all processors have sorted their subvectors a parallel mergs is performed.

Note that in occam the user's program may also execute in parallel with the library routines. A similar facility is planned for the Mk.2 FORTRAN library. The current library contains both single and double-precision versions of all routines.

V.4 Fortnet, PARLANCE library

PARLANCE stands for Parallel Library and Networked Computing Environment. Only the numerical library interface has been documented in this review. The network and task scheduler which is also contained in this software system are dealt with in anoth Technical Memorandum [59].

Syntax of calling sequences

The style of the PARLANCE global memory interface has been presented in section III.7 above. Arguments of the routines are always symbols, character*8 variables, which refer or point to globally accessible virtual storage as previously described or a scalar input quantity. Each of these symbols is followed by an integer offset referring to the starting point for computation from the beginning of the virtual storage, and an integer stride for vector computation, unless the quantity is a scalar. In my global routines the order of arguments is as follows:

processor pair source, target or target only or omitted
result
operands in order of rhs e.g. a=b*c
number of elements in vector
an external function
a scalar quantity (real number)

To remain compatible with the syntax of the VecLib and other libraries routines emulating these are slightly different. The first argument in most of the calling sequences is now usually the number of elements in the vectors for which computation is to be done. An

example of this was already used in section III.7 and IV.

```
temp=ddot(n,'v1',1,1,'v2,1,1)
```

which takes the dot product of vectors v1 and v2 and broadcasts the result to a variable temp. The actual data to which this calling sequence refers may be on any processors, and internal communications are done to access it, which should not be the concern of the application programmer. Broadcasting can be switched off using the routine brall, in which case the routine will yield only a partial result.

Many routines in this library, such as ddot, are intended to provide a parallel implementation of the VecLib library, for instance as described in the Intel or Convex documentation [4, 42] but with changes to the syntax of the argument list as noted above. This is a useful international standard and provision of it in parallel form should aid porting codes to multicomputers. Other libraries for level 2 and 3 BLAS and higher matrix operations are also being tackled.

The implemented routines are listed below along with their calling sequences.

Libraries in the PARLANCE system (for instance the Intel version)

fortnet.a -- The Fortnet v3.0 (trace) point to point communications library. Optimised versions of this are available for a wide variety of hardware as outlined in section II above. Software enabling graphical playback of parallel program execution and profiling of a task once it has completed (or hung) is available. This was also described in section II.

parlance.a -- underlying global-memory handling subroutines used by all the algorithm libraries. This is the code layer which provides the virtual machine interface. The global routines allocate an area of memory on each processor (common/core/... ) assign parts of that area to symbols in a table, and allow storage (distribution) of local data, and retrieval (gather) of global data. The way the data is distributed is known to these routines and global memory operations are then available to handle it. The operations are closely analogous to generic forms of the usual vector operations, and calls to VecLib routines and other libraries are mapped directly onto them.

The symbolic variables, for instance A, B, C, are assigned space in the local core memory and this forms contiguous storage distributed over the whole machine. Their allocations are found through a call to routines extract or use. These routines are meant to be optimised for each specific multicomputer, so that programs

calling them are machine independent. Definition of the routines was given in section III above.

veclib.a -- set of routines with the same names and calling sequences as the single-precision VecLib library routines. Mainly level-2 BLAS. These are mapped onto the above global routines so that they handle distributed data and can be made parallel. They use a skeleton philosophy to get the complete computational functionality from the basic communication modules.

The routines are machine-independent. Calls to the parlance library must have been used to set up the distributed data in the required place, thereafter the calls should be data-independant and internally consistent.

This library embraces the idea of 'skeleton' routines in parallel computing so that there is a common set of communications routines in parlance into which are put different numerical parts (the scalar*scalar functions) which could equally well be extremely complicated functions defined by the user. The global communications routines in PARLANCE are meant to be optimised for each different parallel computer. Current implementation is poor, with little intermediate buffering to transmit long messages. There is also no use of vector hardware. This will be improved in future implementations.

For a detailed description of each routine refer to the corresponding Intel or Convex documentation.

```
subroutine dfill(n,alpha,a,inca)
function dasum -- broadcsting of the result is optional
subroutine daxpy
subroutine dclip
function ddot -- broadcsting of the result is optional
subroutine dgathr
function idamax
function idamin
subroutine diclip
function idmax
function idmin
subroutine dneg
function dnrm2 -- broadcsting of the result is optional
subroutine dramp
subroutine dsadd
subroutine dscal
subroutine dscatr
subroutine dcopy(n,a,inca,b,incb)
subroutine dvatan(n,a,inca,b,incb)
subroutine dvcos(n,a,inca,b,incb)
subroutine dvexp(n,a,inca,b,incb)
subroutine dvlg10(n,a,inca,b,incb)
```

```
subroutine dvlog(n,a,inca,b,incb)
subroutine dvneg(n,a,inca,b,incb)
subroutine dvrecp(n,a,inca,b,incb)
subroutine dvsin(n,a,inca,b,incb)
subroutine dvsqrt(n,a,inca,b,incb)
subroutine dvabs(n,a,inca,b,incb)
subroutine dsadd(n,alpha,a,inca,b,incb)
subroutine dsdiv(n,alpha,a,inca,b,incb)
subroutine dsmul(n,alpha,a,inca,b,incb)
subroutine dssub(n,alpha,a,inca,b,incb)
subroutine dvadd(n,a,inca,b,incb,c,incc)
subroutine dvatan2(n,a,inca,b,incb,c,incc)
subroutine dvdiv(n,a,inca,b,incb,c,incc)
subroutine dvmax(n,a,inca,b,incb,c,incc)
subroutine dvmin(n,a,inca,b,incb,c,incc)
subroutine dvmul(n,a,inca,b,incb,c,incc)
subroutine dvpow(n,a,inca,b,incb,c,incc)
subroutine dvsub(n,a,inca,b,incb,c,incc)
function dsum -- broadcsting of the result is optional
subroutine dswap
subroutine dvabs
subroutine dsvmvt(n,alpha,a,inca,b,incb,c,incc)
subroutine dsvpvt(n,alpha,a,inca,b,incb,c,incc)
subroutine dsvtvm(n,alpha,a,inca,b,incb,c,incc)
subroutine dsvtvp(n,alpha,a,inca,b,incb,c,incc)
subroutine dsvvmt(n,alpha,a,inca,b,incb,c,incc)
subroutine dsvvp(n,alpha,a,inca,b,incb,c,incc)
subroutine dsvvtm(n,alpha,a,inca,b,incb,c,incc)
subroutine dsvvtp(n,alpha,a,inca,b,incb,c,incc)
```

The following monadic functions are used in the generic skeletons to provide full functionality, some are implemented as C code.

```
function aneg(b)
function arecp(b)
function adiv(a,b)
function amul(a,b)
function apow(a,b)
function asub(a,b)
function avmvt(alpha,a,b)
function avpvt(alpha,a,b)
function avtvm(alpha,a,b)
function avtvp(alpha,a,b)
function avvmt(alpha,a,b)
function avvpt(alpha,a,b)
function avvtm(alpha,a,b)
function avvtp(alpha,a,b)
```

MathAd.a -- This library implements some routines from the MathAdvantage collection [41]. Many of them are equivalent to ones in the VecLib library, and are implemented in the same way

```
subroutine vsmul(a,iastart,ia,alpha,c,icstart,ic,n)
subroutine vsin(a,iastart,ia,c,icstart,ic,n)
subroutine vfill(alpha,c,icstart,ic,n)
subroutine vgathr(x,ixstart,incx,y,iystart,iy,z,izstart,iz,n)
subroutine vlog(a,iastart,ia,c,icstart,ic,n)
subroutine vlog10(a,iastart,ia,c,icstart,ic,n)
subroutine vmov(a,iastart,ia,c,icstart,ic,n)
subroutine vmsa(a,iastart,ia,b,ibstart,ib,alpha,n)
subroutine vmul(a,iastart,ia,b,ibstart,ib,c,icstart,ic,n)
subroutine vneg(a,iastart,ia,c,icstart,ic,n)
subroutine vrecip(a,iastart,ia,c,icstart,ic,n)
subroutine vsadd(a,iastart,ia,alpha,c,icstart,ic,n)
subroutine vrvrs(c,icstart,icstride,in)
subroutine vsbsm(a, iastart, ia, b, ibstart, ib, alpha, d,
idstart, id, n)
subroutine vscatr(a,iastart,ia,b,ib,c,icstart,n)
subroutine vsdiv(a,iastart,ia,alpha,c,icstart,ic,n)
double precision function aindiv(a,b)
```

subroutine mtrans -- matrix transpose, see comments under the mtrans routine in the matrix library

blas3.a -- routines to do higher level BLAS operations. Not yet implemented

Cray.a -- Routines from the CRAY library [44]

```
double precision function cvmgp(a,iastart,b,ibstart,c,icstart)
double precision function cvmgz(a,iastart,b,ibstart,c,icstart)
```

Linpack.a -- Routines from Linpack [7]. Not yet implemented

Eispack.a -- Routines from Eispack [7]. Not yet implemented

Matrix.a -- This is a library of algorithms for real double-precision matrices which have been collected from various sources or adapted from existing sequential programs

matmul(a,iadim,b,ibdim,c,icdim,n) -- multiply square matrices a=b*c. Actual dimension of matrix in virtual storage is ixdim, actual order is n

mtrans(ndim,a,n) -- transpose matrix a. Actual dimension and order are ndim and n. Two versions of the matrix transpose routine exist as outlined in section IV above. This one transposes the

matrix in place. The other version is implemented in the MathAd library and yields the transposed elements in a second matrix. That is quicker but less efficient on memory.

matinv(ndim,a,det,n) -- invert matrix a and give its determinant. Actual dimension and order are ndim and n. This routine makes calls to the level-2 BLAS to swap elements of rows and columns of the matrix.

This library will also incorporate an interface the Intel routines for real symmetric matrices soon.

Suprenum.a -- Port of the SUPRENUM grid application library to the Fortnet and PARLANCE system. Not yet implemented.

signal.a -- some signal-processing routines in parallel, take single-precision complex data so far

cfft1d(...) -- fft of a vector of numbers, very inefficient
cfft2d(...) -- fft of a matrix of numbers, incorporates a single matrix transpose if matrix stored by columns
cfft3d(...) -- fft of a 3D array of numbers, incorporates a single matrix transpose if array stored by 2D planes

## Conclusions

The work presented above shows the evolution of global communications and distributed numerical algorithm libraries on a number of different computers. Clearly there is some trend, and perhaps in two directions. There is firstly the possibility to treat a processor array as being an attached compute engine which can be utilised by a parallel library called from a sequential host program. Secondly the array itself may be programmed, and it is rather this last issue which I hoped to address. A virtual-machine programming strategy is important as it removes a lot of the burden from the programmer. The mapping of processes and processors, and communications, should be contained within the subroutines rather than being explicit. I have used this approach, and it is also adopted by the SUPRENUM project, and is a fundamental part of the Linda [30-34] programming language extensions (lately developed for use in the QIX operating system by Cogent Research Inc. [45]), and also of dataflow and object-oriented languages such as Strand-88 [46].

If portability is to be addressed I would not contemplate a programming environment which does not address the issue of the virtual machine. It is no longer sufficient to sell boxes of processors with point to point communications software (occam, POSIX etc.) as this simply results in every application having its own restrictive harness layer which needs to be adapted to different hardware configurations.

A final area of great controversy is the mapping of shared-memory and distributed-memory software. It seems that there is currently no common ground, and that the two areas are diverging, with the American PFC (parallel FORTRAN committee) being assembled from shared-memory machine manufacturers, and European activity largely directed toward the distributed-memory ideal. Perhaps some compromise will be reached indeed if not in word with hybrid machines having a shared resource. The Cogent XTM Workstation already has a shared bus and controller processor which maintains much of the functionality of a conventional shared-memory resource in a distributed-memory environment, and this trend is continuing.

## Acknowledgements

Part of the practical work in developing Fortnet was carried out on a Meiko M10 Computing Surface which is on loan from the SERC/DTI Transputer Initiative loan pool. The work on Fortnet was done in collaboration with Dr. Lydia Heck of Durham University and Dr.Richard Cooper of Queen's University Belfast. My special thanks are due to them for many valuable discussions on parallel computing.

Work on the parallel algorithm library and PARLANCE system was carried out on the Intel iPSC/2 and iPSC/860 hypercubes and Alliant fx2808 machines in the Advanced Research Computing Group at Daresbury. Graphical work was also done on the Alliant which was bought by the Computational Fluid Dynamics Initiative and is now installed at UMIST, Manchester. Some of this work was carried out by Norman Clancy of Bristol Polytechnic, and has led to his obtaining an M.Sc. degree in parallel computing.

I also thank Drs. Karl Solchenbach and Reiner Vogelsang for helpful discussions and for enabling me to visit SUPRENUM GmbH in Bonn in December 1989, Prof. L.M.Delves and Drs. N.Brown and R.Wait for helpful discussion on libraries and in particular their numerical algorithms which are available from NA Software Ltd., Liverpool. Thomas K.Donaldson for his enthusiastic response and discussion of similar work going on at MIMD Systems, California. Dr. Dirk Roose who visited Daresbury in January 1991 shed light on many areas of common interest. Drs. Rolf Hempel of GMD Sankt Augustin and Patrick van Renterghem of Ghent gave helpful comments on an early manuscript. Finally I thank Dr. Richard Chamberlain of Intel Scientific Computers for advice on using the iPSC/2 and Adrian Lincoln of Scientific Computers Ltd. for introducing me to Linda.

Finally I thank my collegues at Daresbury for their support and stimulus to continue this work.

References

[0] this report

[1] R.J.Allan "FORTRAN-77 Programming of Parallel Computers" (1989) Daresbury Laboratory DL/SA/TM61T

[2] R.J.Allan, and L.Heck, "Parallel FORTRAN in scientific computing: a new occam harness called Fortnet" D.L. preprint SCI/P640T (1989)

R.J.Allan, and L.Heck, "Parallel FORTRAN in scientific computing: a new occam harness called Fortnet" in 'Transputer Applications 1' Proceedings of the International Conference on applications of transputers, Liverpool 23-25 August 1989 ed. L.M.Delves (IOS Press: 1990)

[3] R.J.Allan, L.Heck, and S.Zurek, "Parallel FORTRAN in scientific computing: a new occam harness called Fortnet" Computer Physics Comms. 59 (1989) 325-44

[4] L.Heck "Running FORTRAN programs in an occam environment" in 'supercomputational science' proceedings of the sumer school on computational science, september 18-29, 1989 Abingdon, eds. R.G.Evans and S.Wilson ISBN 0-306-43663-9 (Plenum Press, 1990)

[5] Intel Scientific Computers Limited "iPSC/2 and iPSC/860 User's Guide" order number 311532-006 (June 1990)

[6] Intel Scientific Computers Limited "iPSC/2 Fortran

programmer's reference manual" order number 311708-001 (March 1989)

[7] Meiko Scientific Ltd. "CSTools for SUNOS" two volumes, edition 83-009A00-02.01 (1990)

[8] Meiko Scientific Ltd. "Computing Surface Reference Manual" Bristol (March 1989)

[9] D.F.Snelling and Geerd-R.Hoffman "A comparative study of libraries for parallel processing" Parallel Computing 8 (1988) 255-66

[10] K.Solchenbach "Suprenum FORTRAN - an MIMD/SIMD language" Supercomputer 4(March 1989) 25-30

[11] U.Trottenberg "Suprenum - the concept" Supercomputer 4(March 1989) 5-12

W.Giloi "Suprenum - the system" Supercomputer 4(March 1989) 13-19

K.H.Werner, U.Brass and E.Thomas "The Suprenum User Interface" Supercomputer 4(March 1989) 20-24

[12] L.J.Clarke "TINY, Discussion and User Guide" Edinburgh Supercomputer Project ECSP-UG-9 (7/3/90)

L.J.Clarke Ph.D. Thesis University of Edinburgh (1990)

[13] R.J.Allan "FORTRAN-77 programming of parallel computers. I: Operating systems and environments" Parallelogram 19 (October 1989)

[14] R.J.Allan "FORTRAN-77 programming of parallel computers. II: Programming techniques and analysis tools" Parallelogram 20 (November 1989)

[15] R.J.Allan "FORTRAN-77 programming of parallel computers. III: Helios operating system and FORTRAN compiler" Parallelogram 20 (December 1989)

[16] R.J.Allan "FORTRAN-77 programming of parallel computers. IV: 3L Parallel FORTRAN" Parallelogram 20 (January 1990)

[17] NA Software Ltd. "The Liverpool Parallel Transputer Libraries" brochure NA Software Ltd., Transputer Division, Mersyside Innovation Centre, 131 Mount Pleasant, Liverpool, L3 5TF

[18] 3L Ltd., "Parallel Fortran Reference Manual" 3L Ltd., Peel House, Ladywell, Livingston, Edinburgh EH54 6AG

[19] R.K.Cooper and R.J.Allan "Fortnet (3L) v1.0: A message-passing system for transputer using 3L Parallel FORTRAN" Computer Phys. Comms. (1991) in preparation

[20] J.J.Dongarra and D.C.Sorensen "Schedule User's Guide" Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Ill. 60439

J.J.Dongarra and D.C.Sorensen "Schedule: tools for developing and analysing Fortran programs" Argonne National Lab. Technical Memorandum MCS-86-86 (1986)

[21] J.J.Dongarra and D.C.Sorensen "A Portable Environment for developing Parallel FORTRAN Programs" Parallel Computing 5 (1987) 175-86

[22] H.-J.Bast, M.Gerndt and C.-A.Thole "SUPERB - the Suprenum Paralleliser" Supercomputer 30 (1989) 51-7

U.Kremer, H.-J.Bast, M.Gerndt and H.P.Zima "Advanced tools and techniques for automatic parallelisation" Parallel Computing 7 (1988) 387-93

U.Kremer, H.-J.Bast and M.Gerndt "SUPERB: a tool for semi-automatic MIMD/SIMD parallelisation" Parallel Computing 6 (1988) 1-18

[23] B.Thomas and K.Peize "Suprenum comfort of parallel programming" Supercomputer 30 (1989) 31-43

[24] R.Hempel "The Suprenum communications subroutine library for grid-oriented problems" user manual GMD (Gesellschaft fur Mathematik und Datenverarbeitung; Sankt Augustin, 1989)

R.Hempel "The Suprenum communication subroutine library for grid oriented problems" Argonne National Lab. technical report ANL-87-23 (1987)

R.Hempel and A.Schuller "Experiments with multigrid algorithms using the Suprenum communication subroutine library" GMD Studie 141 (Sankt Augustin, 1988)

[25] L.M.Delves and N.G.Brown "SERC Extramural research contract. Numerical libraries for transputer arrays: project N2A 8R0775: Final report" Liverpool 5/3/88

[26] L.M.Delves and N.G.Brown "Esprit project P1085: Work package 17: Numerical libraries for transputer arrays: Interim report" Liverpool 16/11/87

[27] L.Bomans, D.Roose and R.Hempel "The Argonne/GMD macros in Fortran for portable parallel programming and their implementation on the Intel iPSC/2" Arbeitspapiere der GMD 406 (Gesellschaft fur Mathematik und Datenverarbeitung; Sankt Augustin, 1989) and Parallel Computing 15 (1990) 119-32

[28] FPS "T-series User's Guide"

[29] Topexpress library documentation

[30] N.Carriero and D.Gelernter "How to write parallel programs, a guide to the confused" Research Report YALEU/DCS/RR-628 (May 1988)

[31] A.Lincoln "Desk top parallel supercomputers - the future workstations" Scientific Computers Limited, Burgess Hill, W.Sussex, RH15 9LW

[32] D.Gelernter "Getting the job done" Byte (Nov. 1988) 301-7

[33] T.Merrow and N.Henson "System design for parallel computing" High Performance Systems (Jan. 1989) 36-44

[34] Wm. Leler "Linda meets UNIX" IEEE Computer Magazine 23 (Feb. 1990) 43-54

[35] J.J.Dongarra, J.J.Du Cruz, I.Duff and S.J.Hammarling "A set of level 3 basic linear algebra subproblems" ACM Trans. Math. Soft. (December 1989)

[36] J.J.Dongarra, J.J.Du Cruz, S.J.Hammarling and R.Hanson "An

extended set of FORTRAN basic linear algebra subproblems" ACM Trans. Math. Soft. 14 (1988) 1-32

[37] C.Lawson, R.Hanson, D.Kincaid and F.Krogh "Basic linear algebra subprograms for FORTRAN usage" ACM Trans. Math. Soft. 5 (1979) 308-25

[38] Linpack documentation

[39] B.T.Smith, J.M.Boyle, J.J.Dongarra, B.S.Garbow, Y.Ikebe, V.C.Klema and C.B.Moler "Matrix Eigenvalue Routines - EISPACK Guide" Springer Lecture Notes in Computer Science 6 (1976)

[40] B.S.Garbow, J.M.Boyle, J.J.Dongarra and C.B.Moler "Matrix Eigenvalue Routines - EISPACK Guide Extension" Springer Lecture Notes in Computer Science 51 (1977)

[41] Quantitative Technology Corporation "Math Advantage User Manual - FORTRAN version 2.0" QTC, Beaverton, Oregon (1986)

[42] "Convex VecLib User's Guide" 3rd edition, Convex Corporation (March 1988) Document No. 740-002330-202

[43] S.J.Leffler, R.S.Fabry and W.N.Joy "A 4.2BSD Interprocessor Communications Primer" Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkley, CA 94720

[44] CRAY Research Inc. "UNICOS autotasking user's guide" SN-2088 CFT77 3.1

CRAY Research Inc. CRAY X-MP user's manual

[45] R.J.Allan "The Cogent XTM Parallel Workstation" Parallelogram 22 (March 1990) see also refs. [30-34]

[46] I.Foster and S.Taylor "Strand - New concepts in parallel programming" Strand-88 (Prentice Hall, 1990) ISBN 0-13-859587-X

I.Foster and S.Taylor "Strand: a practical parallel programming tool" Argonne National Lab. preprint MCS-P80-0889 (1989)

[47] MIMD Systems Inc. "The Genesys Compute Server; Software Capabilities and Requirements Document" #10 Rev. #0 (1/1/89)

[48] Expert Systems Inc. "Final Report: SBIR Project NAS2-12968: Development of a System Library Facility for Parallel Computers"

[49] ParaSoft Inc. "Express User's Manual" (1989)

ParaSoft Corp. "An operating system for parallel computers" (1987)

[50] A.Kolawa and J.Flower "A 'packet' history of Message passing Systems" C3P report CalTech

[51] R.J.Harrison ipcv3 program documentation Argonne National Lab (1990)

[52] M.Surridge "ECCL a general communications harness and configuration language" in 'Applications of Transputers 2' proceedings of the second international conference on applications of transputers, 11-13 Jult 1990 ed. D.J.Pritchard and C.J.Scott ISBN 90 5199 035 9 (IOS Press: 1990)

[53] R.Williams "DIME: A Programming Environment for

Unstructured Triangular Meshes on a Distributed-memory Parallel Processor" C3P report 502, CalTech (1988)

[54] K.Ikudome "ASPAR: The Automatic Symbolic Paralleliser" presentation to the Fifth Distributed Memory Computing Conference, Charleston, South Carolina USA (April 1990)

K.Ikudome, G.C.Fox, A.Kolowa and J.W.Flower "An automatic and symbolic parallelization system for distributed memory parallel computers"

[55] Intel Scientific Computers Eiscube and Lincube documentation

[56] R.J.Allan and W.H.Purvis "The Alliant fx2800" Parallelogram 35 (March 1990) in press

[57] G.A.Geist, M.T.Heath, B.W.Peyton and P.H.Worley "PICL a Portable Instrumented Communication Library" Oak Ridge National Laboratory Technical Memorandum ORNL/TM-11130 (Oak Ridge, 1990)

[58] M.T.Heath "Visual Animation of Parallel Algorithms for Matrix Computations" preprint of IEEE ???? (1990) 1213-22

[59] R.J.Allan "Portable Message-passing Tools" Daresbury Laboratory Technical Memorandum DL/SCI/TM71E (November 1990)

[60] Distributed Software Ltd. "HELIOS-PC/s V1.1" (1989)

Distributed Software Ltd. "Meiko FORTRAN 77 Manual" (1989)

Distributed Software Ltd. "The Helios parallel programming tutorial" part number H09012 (1990)

Distributed Software Ltd. "The CDL Guide" part number H09014 (1990)

Index