technical memorandum Daresbury Laboratory

DL/SCI/TM71E

PORTABLE MESSAGE-PASSING TOOLS

by

R.J. ALLAN, SERC Daresbury Laboratory

NOVEMBER, 1990

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

ISSN 0144-5677

Portable Message-passing Tools for Shared-memory, Distributed-memory and Networked Computing Resources. An Introduction and Guide to the Work done at Daresbury Laboratory.

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

Abstract

The Daresbury Laboratory distributed message-passing harness consists of a combination and extension of several portable tools which are available through the public domain. These are firstly described as independent pieces of software in sections A to C, and then they are described as a single complete tool in the Daresbury implementation in section D. The software is in order of appearance; A. Fortnet and its numerical library interface developed by R.J.Allan, E.L.Heck and R.K.Cooper; B. ipcv3, the ANL networked package developed by R.J.Harrison from the PARMACS software and extended for use on multicomputers; C. the ANL Schedule package with graphical front and back end tools in its distributed-memory form developed by A.Beuglin of the AMOCO Oil Company. The graphical tools have been converted for use in xview by W.H.Purvis of D.L., and the original machine-dependent shared-memory code has been corrected and brought together in a standard interface library for applications which use a message-passing paradigm by the author.

Terminology

Multicomputer -- a number of independent computers, each with its own memory, in one cabinet and requiring message-passing for communications to achieved parallelism. Typical examples are the Intel and NCube hypercubes.

Multiprocessor -- A number of compute elements, possibly with their own cache memory, in one cabinet which share the system's main memory and bus. Parallelisation is achieved through operating system hooks to manipulate the shared memory, or by compiler directives. Typical examples are the Alliant fx2808 and Convex C220.

Networked resource -- a number of independent computers, which might include workstations as well as the above types, linked with a network such as ethernet. The network must be capable of supporting communications between operating system hooks such as TCP sockets to achieve parallelism. A typical example is an ethernet with a number of UNIX hosts running NFS and rpc protocols.

General Introduction

In the following pages we describe the useage of three portable message-passing tools for parallel computing resources which are essentially in the public domain (but the authors should be consulted and preferably included in publications). These tools evolved at approximately the same time, concurrently, in the U.K. and U.S.A. and were driven by the need to express parallel threads of execution in scientific applications on available high-performance hardware platforms. Ths software is known individually as

A) Fortnet (DL)
B) ipcv3 or the PARMACS (ANL)
C) Schedule (ANL and AMOCO Oil Company)

By means of justification for the adoption of such software, as opposed to any other available (Linda, Express or NCS for instance [22-24]), we cite the ease of availablility in the public domain, as compared to the need to obtain software licences, and the distribution across a large range of hardware. Only Express can claim the latter, although the language Strand_88 [25] is available on nearly as many machines but is not useable in current scientific programs without considerable modification. The work presented here combines the U.K. and U.S.A. experience on many machines, and also the sequential functionally driven and object oriented approaches to programming. A further important consideration is the development of the software considered within the scientific arena, enabling useful functionality to be inbuilt.

There is a lengthy and ongoing discussion about the validity of a message-passing philosophy as opposed to a shared-memory database-style philosophy when programming parallel computers. Clearly both are admissible, and portable tools may be devised as pros for each regime, but the former are natural for multicomputers with synchronisation problems, and the latter for multiprocessors with cache and bus contention as cons in the balance of consideration.

It seems that in scientific applications, which are presently aimed at the course-grained level of programming, the former approach is preferable since it makes any cause of delay in computation explicit, and therefore allows the user the possiblity to optimise his program regardless of any quirks of the operating system.

The object-oriented style has however been embodied in the Schedule software, which is available in a distributed memory form, and is also included in our collection of portable tools.

The Daresbury Laboratory Harness therefore combines Fortnet, ipcv3 and Schedule in one package available across the widest range of resources.

A. Fortnet v3.0 (trace).

R.J.Allan Advanced Research Computing Group, S.E.R.C., Daresbury Laboratory, Warrington, WA4 4AD
E.L.Heck Physics Department, Science Laboratories, South Road, University of Durham, Durham DL1 3LE
R.K.Cooper, Department of Aeronautical Engineering, Queen's University, Stranmillis Road, Belfast

I. Introduction

The original concept of Fortnet and its v2.1 incarnation have been dealt with in refs. [3, 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 low-level routing software and compilers

i) Meiko Computing Surface occam-2 libraries, C and FORTRAN-77 [7]

ii) Meiko Computing Surface with CS-tools, C and F77 compilers [18]

iii) Intel iPSC/2 and iPSC/860 C and FORTRAN-77 [5, 6]

iv) 3L Parallel FORTRAN-77 [8, 9, 17]

v) UNIX 4.2BSD sockets (e.g Convex and SUN operating systems) using C and FORTRAN-77 compilers [10]

vi) Alliant fx2800 Concentrix with C and Fortran [11]

Fortnet is a multi-layered system of subroutines. 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. The top layer is the application code. This 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 work started in 1987. The first stage of Fortnet was therefore a communication harness to pass messages (data) between the transputers in a controlled fashion. It also performs some tasks such as accessing the front-end file-store, printing diagnostic messages to the screen, and bookkeeping. This layer of code is written in the concurrent language occam-2 which was designed for the transputer, and was partly the result of work by Sebastian Zurek (TCM Group, Cambridge) who visited Daresbury during the summer of 1987; it is configured for a dual daisy chain of transputers with a return connection forming a ring for the data path (but not for the

file system calls). Fortnet could in principle be implemented on any type of transputer array. More recent work on harnesses for general topologies is now available [19, 20] and the Fortnet protocol is being implemented upon this (R.K.Cooper, priviate communication, 1990). Other work in the area is still ongoing [21]. On hardware platforms which offer an alternative interface to occam this routing layer is naturally abandoned in favour of the manufacturer's software.

2) The second stage of development is a layer of FORTRAN-77 subroutines which may be called by the parallel program as an interface to the occam or to provide a standard library to reference machine specific routing software. These incorporate a protocol to verify the 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 deadlock if no error-checking mechanism were present. A novel handshaking and blocking paradigm is used for this checking which differs from other systems. Fortnet is envisaged to be the simplest possible communication system with explicit synchronisation of processors and further free transmission of data with low overheads. The routine calls are however superficially similar to those on hypercube machines such as the Intel iPSC/2. Development of this layer and incorporation of the full FORTRAN i/o on all nodes of the Meiko version was done by Lydia Heck of Durham University.

Completion of stage (2) yielded Fortnet v2.2 which is installed on a number of machines in the UK.

Durham Physics Meiko M10
Leeds Computing Science, M10 and In-Sun 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
Daresbury Lab. Intel iPSC/2 (Intel version)
Daresbury Lab. Intel iPSC/860 (Intel version)
Daresbury Lab. PC-based system (3L version)
Daresbury Lab. Convex C-220 (UNIX version)
Daresbury Lab. SUN 3/260 (UNIX version)
National Physical Lab. M40
Belfast Applied Maths M40
Northern Ireland Transputer Centre, Belfast M10
Belfast Aeronautical Eng. PC-based system (3L and TINY versions)
Bristol Polytechnic Meiko

The Fortnet harness is available from Computer Physics Communications, and is included in the third-party software catalogue of Meiko Scientific Limited (the Ensemble programme) and 3L Limited.

3) A demonstration interface has been written to the graphical post-execution display package Schedule/Trace from the Argonne National Lab (described in section C here). This depicts dynamic execution of an actual parallel program on the screen of a SUN or X-window workstation in pseudo-real time allowing one to identify hot spots, bottlenecks and errors and to effectively compare different algorithms. An example is shown in the figure.

Some work is still needed to tidy up this interface and resolve some philosophical questions about exactly how the display should look. Fortnet version 3.0(trace) is however currently in use. More details of this are given below.

A parallel profiler has 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. See the example output below.

Implementation of stage (2) of Fortnet on the Intel hypercube and other hardware allows us to benefit from these tools also in developing portable programs. Such an exercise was necessary because we do not have access to the internal workings of the manufacturers message-passing subroutines. Furthermore Fortnet now provides a common environment and devlopment platform on both the Meiko, Intel and UNIX-based shared-memory computers allowing direct porting of applications.

4) Development of a generic set of global-memory operations 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 distributed.

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 pairs for instance, and the same for any k-fold covering to implement an operation involving k distinct data elements. This is the subject of a larger report and separate publications [12-13].

5) The highest layer of the environment is the application which would directly call these global routines to do numerical tasks. Standard library calls have already been implemented in this way for vectors and matrices enabling the interface to look like familiar mathematical library operations.

II. Summary of Fortnet calls

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

STOP() -- when first called by a node it initialises the Fortnet system and signals to the server that the node is active. If called a second time it tries to shut down that node.

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

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

SEND(m,nbytes,buffer) -- send nbytes to target processor m from buffer

RECEVE(n,nbytes,buffer) -- receive nbytes from source processor n into buffer

ASEND(m,nchar,string) -- send nchar to processor m from character string

ARECEVE(n,nchar,string) -- receive nchar from processor n into character string

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

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

inode=NODEID() -- get node id of current processor, 0 for server, 1 for master and 2 to nnode+1 for slaves. Note that the name of this routine has been changed in line with the ipcv3 routines to avoid conflict on the Intel hypercube. This reflects the logical position of the process, and inode is used as the argument to the message-passing routines; it may bear no relation to the physical processor placement

nnode=NUMSLAVES() -- find number of slave processes in array. Note that the name of this routine has been changed to avoid conflict on the Intel hypercube. The total number of processes is nnode + 2

BR$ALL(mode) -- allows the broadcasting mode to be controlled, mode can take the values 'ON', 'OFF' or 'RESET'

BRCAST(proclist,nproc,nbytes,buffer) -- broadcast nbytes from buffer to nproc processors whose ids are contained in the integer vector proclist

STATS(mode) -- collects and provides information about messages arriving or sent to each node. Mode can take the values 'ON', 'OFF', or 'PRINT'

The operation of these routines is described in references [3] and [4]. Some additional calls have been implemented in the 3L version to give more advanced access to shared files through the file server. ASEND and ARECEVE above are examples of routines with strong typing enforced (SEND and RECEVE can send any type, but there will be no data translation on the receiving machine so the two nodes must have the same data type). This convention corresponds to the naming of vector mathematical library routines. Its implementation is discussed in more detail in sections B and D.

ALLOCATE() -- initialises the Fortnet symbolic data structure system. This is used to distribute data across the machine which can then be accessed by the following subroutines and referenced by strings consisting of up to eight characters.

ASSIGN(symbol, nbytes) -- symbol is a character string up to eight letters long. This assigns nbytes of physical memory on the calling processor to be used as part of the global definition of symbol

PUT$A(temp,symbol,ioffset) -- given a value in local variable temp, put this value into the memory element referenced by symbol(ioffset)

temp=FETCH$A(symbol,ia) -- inverse of the above. Boadcasts the result if BR$ALL mode is 'ON'.

Note that the above four routines just form part of a larger set for distributed data handling and are used in the libraries associated with Fortnet [12]. Their description is included here since they have been used as part of the interface between Schedule

and Fortnet to be covered in sections C and D.

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 later developments and provides a standard portable platform for writing parallel code. It furthermore includes

diagnostic and graphical display tools to help program development as seen in the figure, these are now discussed.

III. Definition of logical processes in Fortnet. Data dependency and Scheduling. Graphical analysis and profiling tools.

In order to take over and make use of, on local-memory multicomputers, software developed for shared-memory computers 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, or hooks into the operating system, at either end. A number of logical processes placed together end to end on one processor form what I shall call a "sequential program". Programs are linked together by complex data dependency paths in a topology which is representative of the overall work to be done. We will come back to this definition in the section on the Schedule software (section C).

In a shared-memory computer a logical 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 conceptual scheduling mechanism is however identical and does not occur until all the data dependency is satisfied (it is blocked until then).

This scheduling can be done automatically under control of a main program, as in the Argonne Schedule package [14, 15]. In a similar way the job 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 analysis package to display execution of parallel programs running with the Fortnet communications routines as shown in the figure.

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

Typical output from the Fortnet profiler (time stamps were zero

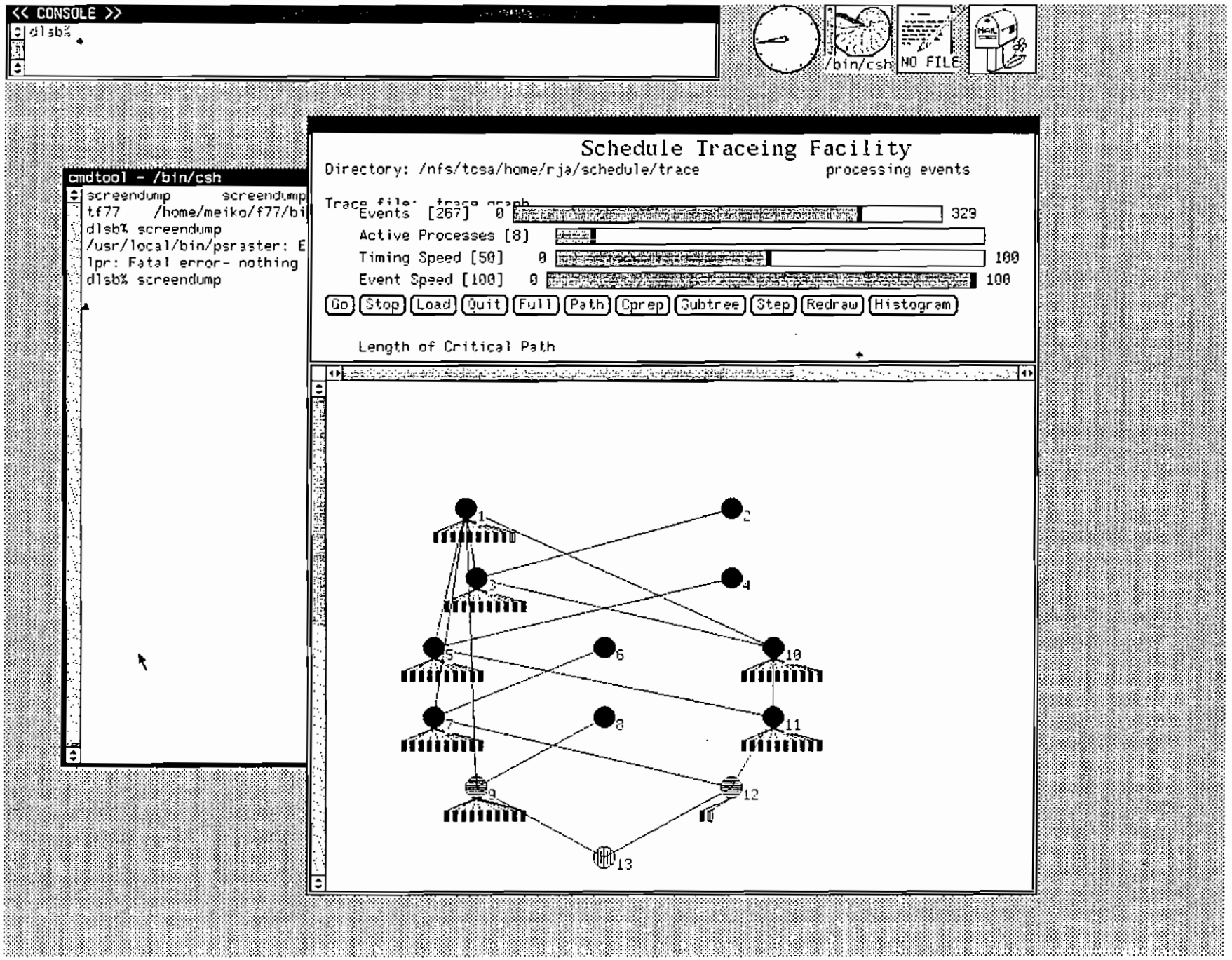
```
for this run).

 ***** Fortnet Profiler *****
 case 14 not understood
 case 14 not understood
 case 14 not understood
 case 14 not understood
 case 14 not understood

 .....Fortnet timings in seconds
 proc, receve, send, wait, check, sequential
0, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000
1, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000
2, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000
3, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000
4, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000
5, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000


 .....number of calls
 proc, receve, send, wait, check, sequential
0, 0, 0, 0, 0, 0
1, 1, 0, 0, 1, 2
2, 0, 1, 1, 0, 1
3, 0, 0, 0, 0, 0
4, 0, 0, 0, 0, 0
5, 0, 0, 0, 0, 0

 .....diagnostics
 processor 0 terminated
 processor 1 terminated
 processor 2 terminated  doing send
 processor 3 terminated
 processor 4 terminated  doing sequential code
 processor 5 terminated
```


<< CONSOLE >>
dlsb%
/bin/csh
NO FILE
cmdtool - /bin/csh
screendump screendump
tf77 /home/meiko/f77/bi
dlsb% screendump
/usr/local/bin/psraster: E
lpr: Fatal error- nothing
dlsb% screendump
Schedule Traceing Facility
Directory: /nfs/tcsa/home/rja/schedule/trace
processing events
Trace file: trace.graph
Events [267] 0 329
Active Processes [8]
Timing Speed [50] 0 100
Event Speed [100] 0 100
Go
Stop
Load
Quit
Full
Path
Cprep
Subtree
Step
Redraw
Histogram
Length of Critical Path
1
2
3
4
5
6
10
7
8
11
9
12
13

B. ipcv3 Harness for Networked Computing Resources, version 3.0 (9/27/90).

with apologies to: Robert J. Harrison
Bldg. 200, Theoretical Chemistry Group,
Argonne National Laboratory,
9700 S. Cass Avenue, Argonne, IL 60439.
tel: (708) 972-7197
E-mail: harrison@tcg.anl.gov, harrison@anlchm.bitnet

I. Introduction

These routines have been written with the objective of providing a robust, portable, high performance message-passing system for distributed-memory FORTRAN applications. The C interface is also portable but not as clean as FORTRAN. The syntax is nearly identical to that of the iPSC subroutines, but the functionality is restricted to improve efficiency and speed of implementation. On machines with vector hardware sustained interprocess communication rates of 6.0Mb/s have been observed. This toolkit (referred to as ipcv3) only strives to provide the minimal functionality needed for current applications. It is only a stopgap until some better model becomes widely (and cheaply) available. However, I believe that many (not all) chemistry and physics problems are readily and efficiently coded with this simple functionality, and that such effort will not be wasted when better tools are found.

Currently tested implementations are:
iPSC/i860 (stand alone)
Sun (Sun O/S 4.0*)
Alliant (Concentrix 5.*, and 2800/1.1.02)
Ardent (O/S 3.*)
Convex C220 (v7.0)
Cray (UNICOS 5.*)
* Encore (UMAX-4.3; for 4.2 replace u_short with ushort or vice versa)
* Sequent Balance (DYNIX V3.0.14)
(* only v2.1 of ipcv was checked)

Underway now:
NCube

The programming model and interface is directly modelled after the PARMACS (Parallel Macros) of the ANL ACRF [see 16]). Thanks are extended to Jim Patterson and Pete Bertoncini (and other members of ANL ACRF and CTD) for their help while I was working with PARMACS.

These macros (and many other packages like them) were fine for modest test programs, but unsuitable for large applications. In particular no allowance was made for the FORTRAN runtime environment required on many machines to perform FORTRAN I/O the system has therefore been re-written.

Communication is over TCP sockets, unless indentical processes are running on a machine with support for shared memory, which is then used, or on a distributed-memory multicomputer such as the Intel of NCube hypercubes. Applications can therefore be built to run over an entire network of machines with local communication running at memory or message-passing bandwidth (plus synchronisation overhead) and remote communication running at Ethernet speed (close to the maximum of 10 Mbits/s can be seen on quiet networks).

A configuration file specifies the placement of processes over the network. The message-passing program is invoked with the command 'parallel' which reads this configuration file and invokes the required processes using fork() or rsh. This has the benefits that the placement of processes are identical no matter from where on the network the command is given, and the 'spare' process running the command parallel is used to provide load a balancing service in a straightfoward fashion.

Hooks are in the lower level routines for conversion of data between machines with different byte-ordering or floating-point numerical representation. This is currently only implemented for FORTRAN integer and double precision (C long and double) data types but will be extended in the future.

II. User interface

The FORTRAN programming interface is quite straightforward. The C interface is identical to that of FORTRAN except that procedure names are defined through macros (so that the FORTRAN interface is portable) and all arguments are pointers. Thus the FORTRAN statement

```
call snd(type, buffer, lenbuf, node, sync)
```

is expressed as the C prototype

```
#include "sndrcv.h"
void SND_(long *type, char *buffer, long *lenbuf, long *node,
                                                  long *sync);
```

The only exceptions to this are that C calls the routine pbegin rather than PBEGINF_, and C calls Error instead of PARERR. Have a look in sndrcv.h and srftoc.h for more details on the C interface.

NB: Use of FORTRAN character variables in argument lists is NOT supported as some implementations pass them using two arguments, or

as a pointer to a structure.

NB: Strong typing of messages is enforced (it was not in the version 1.0). Thus the type on a send and receive must match or an error will result. The type is now an input-only parameter.

NB: All user detected errors requiring termination MUST result in a call to PARERR (see testf.f for an example). From 'C' call Error(char *message, long info).

NB: The value of message types must be in the range 1-32767. Bits higher than this are used (amongst other things) to indicate that data translation is requested (see point 3). Bits 0 and less may be used by the system.

1) On entry the first thing all processes must do is call PBEGINF. This connects all them together and initialises the environment. (C calls PBEGIN_()).

2) Immediately before exit all processes must call PEND to tidy up any shared resources and notify the load balancing server that it has completed. PEND does return but only to allow you to STOP or call the FORTRAN version of exit, so that the FORTRAN runtime environment can tidy up. Calling PBEGIN or PEND more than once per process is bound to produce some bizarre sort of screw up. Note that you must call PEND only afte ALL processes have completed (use handshaking messages to be sure) otherwise everything will be zapped.

3) Data translation is enabled by OR-ing your message type with

the appropriate choice of:

MSGDBL - for double precisoin floating point data
MSGINT - for FORTRAN integer (C long) data

These are defined in include files:
msgtypesf.h - for FORTRAN
msgtypesc.h - for C

Obviously all the data in the message must be of the same type. It should be simple to add extra types if required.

e.g. to send a double precision array X with translation if necessary simply code the matching calls

```
TYPE = IOR(TYPE, MSGDBL)
CALL SND(TYPE, X, LENX, NODE, SYNC)
--------
TYPE = IOR(TYPE, MSGDBL)
CALL RCV(TYPE, X, LENX, LEN, NODESEL, NODEFROM, SYNC)
```

4) Between the calls to PBEGINF and PEND any of the following may be used.

INTEGER FUNCTION NNODES() -- Returns no. of processes

INTEGER FUNCTION NODEID() -- Returns logical node no. of the current process (0,1,...,NNODES()-1)

SUBROUTINE LLOG() -- Opens separate logfiles in the current directory for each process. The files are named log.<MYNODE()>.

SUBROUTINE STATS() -- Print out summary of communication statistics for calling process.

INTEGER FUNCTION MTIME() -- Return wall time from an arbitrary origin in centi-seconds

```
INTEGER TYPE        [input]
BYTE BUF(LENBUF)    [input]
INTEGER LENBUF      [input]
INTEGER NODE        [input]
INTEGER SYNC        [input]
```

SUBROUTINE SND(TYPE, BUF, LENBUF, NODE, SYNC) -- Send a message of type TYPE to node NODE. Type is an arbitrary integer from 1 to 32767 which may be iored with a data translation mask if required. LENBUF is the length of the message in bytes. BUF may be any type other than CHARACTER. SYNC indicates synchronous (1) or asynchronous (0) communication. Note that only sycnhronous communication is supported in the UNIX environment.

```
INTEGER TYPE        [input]
BYTE BUF(LENBUF)    [output]
INTEGER LENBUF      [input]
INTEGER LENMES      [output]
INTEGER NODESEL     [input]
INTEGER NODEFROM    [output]
INTEGER SYNC        [input]
```

SUBROUTINE RCV(TYPE, BUF, LENBUF, LENMES, NODESEL, NODEFROM, SYNC) -- Receive a message of type TYPE from node NODESEL. LENBUF is the length of the receiving buffer in bytes. Type must attach a type in a corresponding snd call, and may not be a wildcard value. LENMES returns the length of the message received and NODEFROM returns the

node from which the message was received. If the NODESEL is specified as -1 then the next node to send to this process is chosen. The length of the buffer is checked and the type of the message must agree with that being received (there is only one channel between processes so messages are received in the order sent). BUF may be of any type other than CHARACTER. SYNC indicates synchronous (1) or asynchronous (0) communication. Note that only sycnhronous communication is supported in the UNIX environment.

```
      INTEGER TYPE        [input]
      BYTE BUF(LENBUF)    [input/output]
      INTEGER LENBUF      [input]
      INTEGER IFROM       [input]
```

SUBROUTINE BRDCSTN(TYPE, BUF, LENBUF, IFROM) -- Broadcast from process IFROM to all other processes a message of type TYPE and length LENBUF. All processes call this routine which uses a hypercube-like pattern to distribute the data in O(log p) time.

```
      INTEGER TYPE        [input]
```

SUBROUTINE SYNCH(TYPE) -- Synchronize all processes by exchanging zero length messages of type TYPE with process 0.

```
      INTEGER ONOFF       [input]
```

SUBROUTINE SETDBG(ONOFF) -- Switch debugging output on (ONOFF=1) or off (ONOFF=0). This output is useful to trace messages being passed and also to help debug the message passing software.

```
      INTEGER MPROC       [input]
```

INTEGER FUNCTION NXTVAL(MPROC) -- This call simulates a simple shared counter by communicating with a dedicated server process. It returns the next counter associated with a single active loop (0,1,2,...). MPROC is the number of processes actively requesting values. After the end of the loop each process calls NXTVAL(-MPROC) which implements a barrier. It is used as follows:

```
  ...
  next = nxtval(mproc)
  do 10 i = 0,big
     if (i .eq. next) then
      ... do work for iteration i
      next = nxtval(mproc)
     endif
10  continue
c call with negative mproc to indicate end of loop ... processes
c block here until mproc processes have registered completion
  junk = nxtval(-mproc)
...
  while ( (i = NXTVAL_(&mproc)) < big ) {
    ... do work for iteration i
  }
  mproc = -mproc;
  (void) NXTVAL_(&mproc);
 ...
```

Clearly the value from NXTVAL can be used to indicate that some

locally determined no. of iterations should be done as the overhead of NXTVAL may be large (approx 0.5s per call ... so each process should do about 50s of work per call for a 1% overhead).

SUBROUTINE PARERR() -- Call to request error termination, it tries to zap all the other processes and succeeds in causing havoc. C should call Error(char *message, long status).

```
      INTEGER NODE [INPUT]
```

SUBROUTINE WAITCOM(NODE) -- Wait for all asynchronous communication with node NODE to be completed. This syntax matches with common implementations of asychronous i/o to disc. This may be modified subsequently. Again note that all UNIX communication is synchronous.

III. Configuration file and server

The command 'parallel' is used to execute a program. It reads a configuration file (the PROCGRP file in the parlance of the PARMACS) to determine which process to run where. Currently it tries the following in order to determine the file name:

1) the first argument on the command line with .p appended
2) as 1) but also prepending $HOME/pdir/
3) the translation of the environmental variable PROCGRP
4) the file PROCGRP in the current directory

The command line arguments of parallel are currently propagated to all processes, though it is probably not advisable to rely on this (?). There is currently no widely accepted way to write or implement configuration files. Note that extra arguments are appended and that the file, if specified, is still there, so any arguments that you use must be flagged rather than just positional.

If you want a process to interactively read input then it must be running on the same machine as parallel. This is because remote processes are invoked with 'rsh -n', necessarily or the parallelism will cause problems.

The configuration file is read to the EOF marker. The character '#' (hash or pound sign) is used to indicate a comment which continues to the next EOL character. For each 'cluster' of processes the following whitespace separated fields should be present in order.

<userid> <hostname> <nslave> <executable> <workdir>

userid -- The username on the machine that will be executing the process.

hostname -- The hostname of the machine to execute this process. If it is the same machine on which parallel was invoked the

name must match the value returned by the command hostname. If a remote machine it must allow remote execution from this machine (see man page for rlogin).

nslave -- The total number of copies of this process to be executing on the specified machine. Only 'clusters' of processes specified in this fashion can use shared memory to communicate. If no shared memory is supported on machine <hostname> then only the value one (1) is valid.

executable -- Full path name on the host <hostname> of the image to execute. If <hostname> is the local machine then a local path will suffice.

workdir -- Full path name on the host <hostname> of the directory to work in. Processes execute a chdir() to this directory before returning from pbegin(). If specified as a '.' then remote processes will use the login directory on that machine and local processes (relative to where parallel was invoked) will use the current directory of parallel. e.g.

```
harrison boys 3 /home/harrison/c/ipc/testf.x /tmp # my sun 4
harrison dirac 3 /home/harrison/c/ipc/testf.x /tmp # sun4
harrison eyring 8 /usr5/harrison/c/ipc/testf.x /scratch # fx/8
```

The above configuration file would put processes 0-2 on boys (executing testf.x and using files in /tmp), 3-5 on dirac (executing testf.x and using file in /tmp) and 6-13 on eyring (executing testf.x and using files in /scratch). Processes on each machine use shared memory to communicate with each other or native message-passing protocol in multicomputers, sockets otherwise.

Note that the number of processes and where they are executed is the same no matter where the command parallel is invoked, as long as the configuration file is the same.

If programs are correctly set up they will function as expected when invoked with parallel no matter how many processes are specified in the configuration file.

IV. Installation

If your machine has been ported to already then simply edit the makefile to comment out unwanted machine options and uncomment the desired machine. Then:

a) Check testf.f for the definitions of LOG, MAXLEN, NBYTPI and NBYTPD. These should only need modifying if you want the Cray or Sequent versions of the test program. LOG is defined in two places.

b) Type 'make'. This makes the library and also FORTRAN and C test programs.

c) Edit the files test.p and testf.p to reflect your local network, working directory and the number of processes you desire.

d) Invoke the FORTRAN test program with the command 'parallel testf'.

e) Try the C test program with the command 'parallel test'. The C program interactively prompts for tests to run and stresses the system more effectively than the simple FORTRAN code.

To reduce some of the message passing overhead compile and link without -DTIMINGS (-DTIMINGS enables gathering of timing statisitcs everytime a snd/rcv is made).

If your machine has not been ported to, initially try compiling with the definitions of the machine closest to yours, but without the -DSHMEM flag (i.e. only support the socket IPC). This should work apart from possibly a few include file complaints which should be readily resolved. If you have the system V semaphores and shared memory interface include the flags -DSYSV and -DSHMEM. If you have some other routines providing this functionallity then you will have to code the routines in shmem.c, and sema.c (this again is quite easy).

If you look at the source it will be quite apparent that I am a FORTRAN programmer and know very little about UNIX (especially what distinguishes various breeds of UNIX). If you have any comments about the source, how to clean it up, and make it more portable then please contact me at the above address. Similarly reports of problems or bugs will be well received, though no support is guaranteed.

A sample makefile for an application is in Makefile.sampl.

The above comments are valid only for multiprocessors (shared memory). For multicomputers with distributed memory there is a server program which must be made to run on the host processor, that is the one which accepts the rsh calls. The nodes are then loaded with the instances of the cluster, and then run by communicating between themselves using the locally defined library and with the rest of the task by passing messages to the host which relays data to its socket connection. Thus code for multicomputers is necessarily very machine dependent.

V. Miscellaneous and Bugs

if a program crashes on machines with the system V shared memory and semaphores some of these resources may not be deallocated. If these are not tidied up the system can run out. The shell script 'ipcreset' (by Jim Patterson?) removes all such resources currently allocated by a user. Removing the resources for a running process will cause it to crash the next time it tries to access it. Try 'man' on ipcs, ipcrm for more details. On a Sequent ipcs, ipcrm are found in '/usr/att/bin', and on the Encore in '/etc'.

Many machines have the number of sockets fixed, either statically in the kernel, or capped by some number. While debugging a program that keeps crashing sockets may be left open. Most systems tidy these unused sockets up every few minutes (?). However when the system runs out of these resources it will wreak havoc with all networking and windowing operations and possibly crash the system (e.g. Ardent Titan, 2.2).

The script 'zapit' (bsd and sysv versions, courtesy of JP again?) kills all processes whose command contains a given string. This is useful for a crash or deadlock occurs which leaves junk processes lying around (rsh is prone to run away on some machines).

DYNIX on the Sequent Balance (at least as configured at ANL) is limited to 20 open file descriptors and 24 (?) semaphores per group. These limit the total number of processes to approx. 15 and the maximum no. of processes in a cluster to 8. Similarly on the Sequent Encore there is a maximum of 8 processes per cluster. I have successfully run with three such clusters (i.e. three lines in the configuration file). On the Alliant each process is limited to 32 semaphores and the system to a total of 128. This limits you to 10 processes in a cluster, and 4 clusters of this size on the machine. (If you talk nicely to Alliant's excellent customer support staff they will give you the names of the kernel variables that can be patched to up the limit to 128 semaphores per process and 512 system wide).

The message-passing routines want message lengths in bytes, which is machine dependent. Have a look in testf.f for one solution to this problem. Also see the routine pfname for how to create unique file names for each process.

For shells that support it STOP/CONT signals should work OK. Interrupts (SIGINT) are trapped and should cause everything to tidy up and die (with a few error messages). To kill a program the best way is just to use ^C on the parallel program or to send it an interrupt with kill -2 (usually).

C. SCHEDULE, Guide to use on Multicomputers.

R.J.Allan (Intel version) A.R.C.G., Daresbury Laboratory, S.E.R.C., Daresbury, Warrington, WA4 4AD e-mail: RJA@UK.AC.DL.DLGM

with apologies to: Adam Beuglin (NCube version) Amoco Production Company, Tulsa, Oklahoma, USA. 23/11/87, e-mail: adamb @ boulder.Colorado.EDU

I Introduction

SCHEDULE, a tool for developing portable parallel Fortran programs, was converted for use on the NCube hypercube by A. Beuglin and translated for use on the Intel iPSC by R.J.Allan. The original SCHEDULE was developed for shared-memory parallel processors; Sched3, the hypercube version, was developed around a master/slave model on a local-memory parallel processor. The implementation of Sched3 is discussed in section II along with the differences between Sched3 and standard SCHEDULE. Limitations of the current version of Sched3 are also discussed with suggestions for future improvements. For information on running schedule the original documentation should be consulted [14, 15]. The original shared memory version of schedule has been corrected by R.J.Allan and W.H.Purvis, and is now running on the following machines at Daresbury along with the new sched3 software:

i) Convex C220 (cxa v7.0)
ii) SUN 3.0 (available)
iii) SUN (v4.0 on tcsm)
iv) Alliant fx2808 (parallel version on dlal1)
v) Intel iPSC/2 ipsc (Sched3)

Differences in useage between the Daresbury iPSC version and the AMOCO NCube version of Sched3 revolve around the provision of a configuration file facility for the host program. By this means a description of the job may be supplied in a standard text file, with executable routine names, their dependency, and explicit variables which need to be passed between them. This has been done by using the method of symbolic variables, developed for use with distributed data structures within Fortnet. The variable names may be read from the configuration file by a standard program invoked from Schedule, and given memory space. This is dealt with in detail in section D, only the direct means of calling the system is described here.

In addition to this the post-execution display package sched.trace [15] is now running under sunview and was converted to xview by W.H.Purvis. An example of its use was shown in section A. This package is now accessible from Fortnet as described in sections A and D. It is available for:

i) SUN 4.0 on tcsm

The front-end build package is currently only working in the public-domain version on SUN 3.0 machines, and has been found to be very unreliable. Further work is required to release this.

The source and executable files are to be found in directories as follows:

/cxa/priv3/rja/schedule/schedule
/tcsm/home/rja/schedule/schedule
/tcsm/home/rja/schedule/trace
/tcsm/home/rja/schedule/open
/dlal1/user/rja/schedule/schedule
/cxa/priv3/rja/ipsc/schedule/schedule

II.1 Implementation on the NCube and Intel iPSC hypercubes.

In the shared-memory version of Schedule routines are saved in a queue by storing the address of a routine and its parameters. On the hypercube the routines will be loaded in the local memory of another processor so having a pointer to the address of the routine on the host would be of little value when starting the process on a node. In Sched3 the name of the file to be loaded on the node is therefore given instead of the address of the routine; it can then be loaded or called by a UNIX rsh command. The file to be loaded is created by compiling and linking the subroutine along with the program worker. Worker will act as the main program for the subroutine (called subroutine user) when it is loaded on the node. Worker receives the parameters from the host, calls the subroutine with the proper pointers into the parameter buffer and sends the parameters back to the host when the node subroutine returns. As a convention each parallel subroutine is in a different file which will be compiled and linked placing the executable in a file with a descriptive name. For instance the file 'dotprod.fn' might contain a parallel subroutine to compute a dot product. The routine would be compiled and linked to worker and the executable file would be called dotprod. When the process is put in the queue the string 'dotprod' is used to specify it.

Since there is no shared memory, the parameters are sent to the parallel subroutine via a message. This message is built by copying data from the pointers to the parameters stored in the queue. When the subroutine completes a message returns the values of the parameters and these values are stored back into the memory of the host. This strategy results in the parameters being passed by address if no other parallel routines access the memory while the parallel routine is executing. Non-determinacy may be introduced if routines are allowed to operate in parallel on the same data. It is the programmer's responsibility to control such non-determinacy by

building a proper dependency graph for parallel routines. To copy the parameters to a message buffer Sched3 needs the length of the parameters. The shared-memory version of Schedule doesn't need this information since the parameters are in shared memory and pointers can simply be passed to a parallel subroutine. Originally the code to save the pointers were written in C since Fortran cannot manipulate pointers. To allow pointers to be manipulated from FORTRAN three routines have been written in assembler for both the host and the node processors on the NCube (these have been re-written in C and two more added for portability at Daresbury).

p= ptoi(i) stores the address of integer i into the integer p

p= ptod(d) stores the address of double precision variable d into the integer p

i= star(p) the value pointed to by p is assigned to i

call lstar(p,i) loads the value of i to the location pointed to by p

p= pointer(symbol, ioffset) stores the address of the memory referenced by symbolic variable symbol and offset into the integer p.

The original Schedule code that decides when to run which subroutines has not been changed except the routines GTPRB and LIBOPN. In the shared-memory version each processor would call GTPRB to get the next routine to execute. In Sched3 there is a main loop executing on the host which calls GTPRB to get the next job. Previously GTPRB would block waiting for work. In Sched3 GTPRB returns a zero if there is no work to be had. LIBOPN previously started processes on every processor and the processors would then ask for work. In Sched3 the host will poll for work and only load processes on a node when a routine is ready to run. If there is a job to run it is loaded onto the next available node. Nodes are simply chosen by picking the lowest numbered node that is available. Perhaps a better strategy would be to chose nodes closest to the host first. It may also be advisable to avoid node zero since it may be busy as a communication gateway to the host. When a job is ready to run a node is selected and the filename containing the routine is loaded on the chosen node.

Processes may spawn other processes. The dependency information of a spawned process is the same as its parent. To spawn a process NXTAG must first be called to obtain a unique jobtag for the process to be created. NXTAG is implemented by sending a message from the node to the host requesting a new jobtag. The host services the request by calling NXTAG (the NXTAG routine from the original shared memory code) and sending the result back to the node in a message.

After a new jobtag has been obtained there is a call to SPAWN. This call will send the name of the routine and its jobtag to the host. The host loads the routine on the next available node and then notifies the parent where the child has been placed. The parameters

are then sent out from the parent to the child.

After a parent has completed its processing it must make a call to WAITCHILD(n) (the paramenter n is currently redundant). This is different to the original NCube version of Sched3 in which the subroutine was simply called WAIT. The name is changed to avoid a conflict with the Fortnet names. When a child completes it sends its parameters to its parent thus notifying the parent that it has completed. WAITCHILD will receive the returned parameters from all of the children that have been spawned and notify the host that the children have completed. This strategy assures that the host receives a child's completion before its parent's completion. Unfortunately this strategy does not assure that a process will complete after its grandchild. This stems from different message lag time between different nodes and the host. When a process calls WAITCHILD all processing in the parent is blocked until all of the children have completed. This is mainly a result of the lack of multitasking on the nodes in the NCube hypercube, and similarly on the Intel iPSC/860. The shared memory version of SCHEDULE will use the wait time to do more processing. The blocking wait makes it very easy to write programs which will deadlock on the hypercube. For instance if every node is loaded with a routine which spawns exactly one child and then waits for the child to complete. The child will never complete since there are no available processors. Since no child can complete no parent can complete resulting in deadlock.

II.2 Differences in shared-memory SCHEDULE and Sched3

The implementation of Sched3 on the NCube hypercube resulted in several changes to the syntax and semantics of a SCHEDULE program. The most apparent was the requirement that routines that are to be run in parallel must be placed in a separate file and linked to worker.fn. The NCube supports the loading of programs on the nodes; not simply routines. Using a standard main program allowed the running of routines on the nodes as complete programs. Worker is also needed to parse the messages containing the parameters to the parallel routine. Using a makefile greatly facilitates creating Sched3 programs and their related files. This is done through the use of file name extensions which have an extra 'n' for a node process. e.g. Host programs use the routines in sched.a while the node routines use those in sched.an. An example is given below.

A parallel routine must always have a fixed number of parameters. This is because Sched3 was written almost exclusively in FORTRAN, which does not allow variable numbers of parameters to subroutines, and even in C one cannot pass data from non-existant memory. The current implementation allows 4 parameters to the parallel routines. This can be easily be extended however. The original version of schedule had routines written in ansi C and a

variable number of parameters could be passed, you do however need to tell the system how many there are for it to work correctly on modern processors such as the Intel i860 in the Alliant. A new parameter 'N' is introduced to do this.

Commons are not really commons across the parallel routines. Shared memory is only simulated on the parameter lists to parallel routines and commons are only common to a routine and any routines it may call, not routines on other processors (or spawned routines).

Three of the SCHEDULE calls have slightly different arguments. A number indicating how many parameters the subroutine requires 'N' is essential in all routines and an array of the lengths of those parameters 'L' has been added to SPAWN and PUTQ which pass parameters to other processors. The external subroutine parameter to PUTQ has also changed to a character*25 variable containing the name of the executable file to be loaded on the remote processor. This file may be the original subroutine, renamed subroutine 'USER' and linked to a main program called 'worker.fn' (see example makefile). In addition, for us to be able to integrate the system fully to the message-passing harness, and to provide a configuration file driver, we needed to make all formal arguments in the host program character*8 symbols which refer to double precision data. This is illustrated as follows:

```
   N=4 (extra obligatory parameter to correct original code)
   CALL SCHED(nprocs, paralg, N, x(1),a(1),b(1),c(1))

becomes:
   character*8 x,a,b,c
   N = 4
   CALL SCHED(nprocs,paralg,N,x,1,a,1,b,1,c,1)
```

Note that no L parameter is not needed, the original documentation is misleading in this respect.

```
   external dotprod
   CALL PUTQ(jobtag, dotprod, N, x(100),a(1),b(1),c(1))

becomes:
   character*8 x,a,b,c
   character*25 DOTPROD
   DOTPROD = 'dotprod'
   N= 4
   L(1) = 200*8 (length in bytes of double precision data types)
   L(2) = 1*8
   L(3) = 1*8
   L(4) = 1*8
   CALL PUTQ(jobtag, DOTPROD,N,L,x,100,a,1,b,1,c,1)
```

similarly for routine SPAWN.

Note that only symbolic references to double precision type variables and arrays are currently able to be passed throught these routines, this will be relaxed in future releases of the software as strong typing is introduced throughout. A more comprehensive example program is shown at the end of this section. An example of their use is shown at the end of this section. The integer number following each symbolic variable is an offset equivalent to a one-dimensional vector index.

The WAITCHILD call mentioned above is required and blocking in Sched3 where it is neither in SCHEDULE.

LOCKON and LOCKOFF are not implemented on the nodes. They are implemented on the host but in a trivial manner. The host has no shared memory so no test and set is needed. Implementing some sort of locks for synchronising parallel processes may be advantageous.

II.3 Future Improvements

The syntactic differences in the calls to the Sched3 could be eliminated by building a preprocessor to convert SCHEDULE calls to Sched3 calls. The preprocessor would have to be able to deduce the size of variables and pad the parameter lists if necessary. A very aggressive preprocessor could also separate the parallel routines into the proper files for Sched3. A more useful WAITCHILD strategy should be developed to allow work to be done if children have not been completed. The current spawn scheme doesn't allow spawned children to spawn their own children. This mule syndrome should be removed. Given these considerations the Sched3 methodology is probably the most appropriate one on which to standardise given that it is applicable across a networked computing resource. Apart from the last one these improvements will probably not be considered at Daresbury. We will however increase the number of data types available in the calls to SCHED and PUTQ, in line with those available in the PARLANCE mathematical libraries (see references to Fortnet), and continue development of the configuration file interface and binding to the Fortnet message-passing harness to be described in section D.

II.4 Routines available in the user interface to Sched3

sched(nprocs,'paralg',n,a,ia,b,ib,c,ic,d,id) -- initialise system and start processing dependency tree described by routine paralg. This routine runs on the host. A graphical builder is available to produce skeleton code forming the paralg routine (see section III below).

parconfig(a,ia,b,ib,c,ic,d,id) -- standard dependency tree routine which reads data from a textual configuration file.

dep(jobtag, icango, nchks, mychkn) -- put dependency data into queue for the next process with id jobtag. This routine runs on the host.

putq(jobtag,'filename',n,l,a,ia,b,ib,c,ic,d,id) -- put the name of executable file filename and its parameters into queue associated with jobtag. This routine runs on the host.

nxtag(dummy,jobtag) -- find the next available jobtag in the queue. This routine runs on a node.

spawn(dummy,jobtag,'filename',n,l,a,b,c,d) -- put the name of executable file filename and its parameters into queue associated with jobtag. This routine runs on a node.

waitchild(n) -- wait for completion of all spawned children, a barrier. Parameter n currently has no meaning. This routine runs on a node.

III. The graphical Interface to Schedule

An additional facility of the Schedule software which is not described in the original documentation is the graphical builder. This allows a display of subroutines and their data paths to be constructed on a workstation screen. A main program corresponding to the display can then be produced automatically for inclusion in a task. This display program currently works only on sunview screens on SUN3.0 and SUN4.0.

To use the facility it is best to start with an existing file which contains a dependency tree and alter it. Some are available in the directory

/home/rja/schedule/trace

Type bld when in suntools to open a new window and start processing. Load and go on a file and redraw its contents. The use of mouse buttons is then as follows:

1) left mouse button is used to select menu options and to put or erase objects from the screen. Press the button on the screen to put a new (next numbered) object, or select erase and then the object to remove it.

2) the middle button is used to form the dependency tree. Place the mouse arrow over the parent routine and press the button. Holding it down move over the child and release it. The software will draw a line connecting the two and save the hierarchy.

3) Use the right mouse button to attach subroutine names to the objects. Select a subroutine from the list in the menu (alter the file contents of subs.list if necessary). Select the target object by moving to it and pressing the right button. Select the subroutine option with the left button. If you hold down the right button over a target which is assigned it will show the name in the menu window.

The objects are displayed in number order from left to right and top to bottom, unless the dependencies are otherwise. Redraw will move objects following the whim of the software, so do not be surprised at bizarre results.

Having completed the display select the dump button on the menu to save a FORTRAN equivalent to the standard output device (redirect output from bld to a file for example). This may also be bizarre and needs checking.

One further word of warning: this software is clearly not tested and often hangs up the workstation, or aborts with an arithmetic error. If it hangs up log onto another terminal and kill off the bld process!

IV. The graphical Trace facility

The graphical playback facility is started by typing
sched.trace

in a suntools session and opens any files with the extension .trace (the default one is graph.trace produced by Fortnet or Schedule in its running directory). The display is as in bld above, the software reads the dependency tree and puts up the objects and lines between them when load and go is selected with the left mouse button; this also starts the playback sequence in the original sunview version - rather irritating if it is over quickly. In the Xview version this has been amended. The objects change colour (grey hashing) to indicate activity and small boxes appear to indicate spawned processes (or send/receive calls in the Fortnet version). The contents of the .trace file are described in the original documentation, and the correponding Fortnet output in fortnet.trace is described in section D below along with the use of the

preprocessor to convert it.

- -

Example program written using the Sched3 system for the Intel iPSC/2 hypercube.

Makefile for the test program:

```
#########################################################################
#            makefile for SCHED3 version on Intel hypercube
#                           r.a. 26/7/89
#########################################################################

# host objects
SCHEDOBJS =     ftsubs.q putq.q work.q wrapup.q slen.q swrapup.q \
```



```
                sndtag.q sched.q gtprb.q

#node objects
SCHEDOBJSN = spawn.qn wait.qn slen.qn nxtag.qn

# generic compilations
.SUFFIXES: .f .fn .q .qn .a .an

.f.q:
        f77 -c $*.f
        cp $*.o $*.q
.fn.qn:
        cp $*.fn temp.f
        f77  -c -sx temp.f
        cp temp.o $*.qn

all:    sched sptest

# make libraries only
sched: sched.a sched.an symbols/iosup.a worker.qn ptr.a ptr.an

#host library
sched.a: $(SCHEDOBJS)
        ar rv sched.a $(SCHEDOBJS)

# node library
sched.an: $(SCHEDOBJSN)
        ar rv sched.an $(SCHEDOBJSN)

# host pointer library
ptr.a: ptoi.q star.q lstar.q
        ar rv ptr.a ptoi.q star.q lstar.q

#host symbol library
symbols/iosup.a:
        (cd symbols; make iosup.a)

# node pointer library
ptr.an: ptoi.qn star.qn lstar.qn
        ar r ptr.an ptoi.qn star.qn lstar.qn

# example link and run to user's code
sptest: sp spt sumsqrt


# host program example
spt: spt.q sched.a ptr.a symbols/iosup.a
        f77 -o spt spt.q sched.a symbols/iosup.a ptr.a -host
```

```
#node program example
sp: worker.qn sp.qn sched.an ptr.an
        f77 -o sp -sx sp.qn worker.qn sched.an ptr.an -node

sumsqrt: worker.qn sumsqrt.qn sched.an ptr.an
        f77 -o sumsqrt -sx sumsqrt.qn worker.qn sched.an ptr.an -node
        cp lstar.o lstar.qn
```

- -

The code for this example is as follows:

The host program:

```
c     program main
      include 'symbols/parameter.f'
      external paralg
      integer  nprocs
      double precision al,bl,cl
      character*8 a, b, c, dum
      data a,b,c,dum/'a       ','b       ','c       ',
     &         'dum     '/
      nprocs= 3
      print *,'using 3 procs'
c initialise symbolic system
      call allocate()
c assign memory to symbols
      call assign(a,8)
      call assign(b,8)
      call assign(c,8)
      call assign(dum,8)
c store real data into them
      call put$a(1.0,a,1)
      call put$a(65000.0,b,1)
c start sched3
      call sched(nprocs,paralg,3,a,1,b,1,c,1,dum,1)
c retrieve results
      cl=fetch$a(c,1)
      print *, 'the sum of the square roots from ',al,'to',bl,' is ',c
      end

      subroutine paralg(a,ioa,b,iob,c,ioc,dum,iodum)
      character*8 a,b,c,dum
      integer jobtag, icango, nchks, mychkn, n, l(3)
      character*25 subname
      jobtag= 1
```



```
      icango= 0
      nchks= 1
      mychkn= 1
      call dep(jobtag, icango, nchks, mychkn)
      n= 3
c l now contains number of bytes
      l(1)= 8
      l(2)= 8
      l(3)= 8
      subname= 'sp'

c start job queue for parallel execution on nodes
      call putq(jobtag,subname,n,l,a,ioa,b,iob,c,ioc,dum,iodum)
      end
```

- -

Node programs linked with worker.fn, sp.fn:

```
        subroutine user(a,b,c,dum)
        double precision a,b,c
        call sp(a,b,c,dum)
        end

        subroutine sp(a,b,c,dum)
        double precision a,b,c,b1,b2,c1,c2,dummy
        integer l(4),n, stat
        character*25 subname
        write(6,'('' in sp '',3g12.5)')a,b,c
        subname= 'sumsqrt'
        n= 4
c in bytes now
        l(1)= 8
        l(2)= 8
        l(3)= 8
        l(4)= 0
        b1= (b-a)/2.0
        b2= b1 + 1.0
c spawn two parallel jobs on unused nodes
        call nxtag(dummy, jobtag)
        call spawn(dummy,jobtag,subname, n, l, a,b1,c1,dummy)
        stat= 1
        call nxtag(dummy, jobtag)
        call spawn(dummy,jobtag,subname, n, l, b2,b,c2,dummy)
        stat= 2
c value of n not used in this call
        call waitchild(n)
        stat= 3
```

```
   c= c1 + c2
   end

sumsqrt.fn:

   subroutine user(a,b,c,dum)
   double precision a,b,c,dum
   write(6,'('' in sumsqrt '',3g12.5)')a,b,c
   c= 0.0
   ia=int(a)
   ib=int(b)
   do i=ia,ib
     c= sqrt(float(i)) + c
   end do
   end
```


D. Interface between the Tools; Execution and Location of Files on the Daresbury Network.

I. Fortnet and ipcv3.

The interface between Fortnet and ipcv3 is straightforward. The implementation at Daresbury now allows calls to both sets of routines in the library file fortnet.a, although one should be careful about mixing them. This library should be linked to an application which does message passing, and there is a copy on most of the Daresbury computers.

The reason for the proviso is because Fortnet needs to assign message types internally (as it does on all the strongly typed systems, such as the Intel, and also Express) and it does it in a cyclic manner reserving message types of (iproc+1)*1000+2 up to (iproc+2)*1000+2 for messages sent from node iproc to another node. Lower numbers will be used by the Schedule software and have therefore been avoided, likewise negative types are for system use on most machines. Fortnet has its own typing scheme as detailed in the original documentation. This has been somewhat extended for the wider useage described here, but a discussion is not relevant.

Some routines may be mixed, such as most of the informational routines. If using Fortnet you should not call the ipcv3 routines pbeginf() or pend() which are already included in the worker program and Fortnet routine stop(). The shared file service of Fortnet is available as normal through the server program, which must be placed as the first single entry in the ipcv3 configuration file. Note that it is distinct from the ipcv3 parallel server, and may be run on another machine in the network (this might pose problems if you want to do i/o however, always a critical issue !). A master program may or may not be used at the programmer's discretion, but its existence is not acknowledged in the call to nslaves which only counts from process 2 up to nnode-1. Thus the Fortnet call

n=numslaves() and the ipvc3 call

m=nnodes() yield numbers related by n=m-2

Implementation of Fortnet within ipcv3 allows portabilty to a wider range of machines than previously available, and also to networked resources. It also gives ipcv3 the ability to use the Schedul-trace analysis and Fortnet profiler. A further extension which evolved during the merging of the two systems was the means to transfer messages between machines with a different data representation using the standard xdr routines. Strong typing is thus enforced, partly as described in the ipcv3 section B, and through calling the Fortnet routines with prefix

_ -- type of any

```
a -- type of character
l -- type of logical
i -- type of int
d -- type of double precision (real*8)
r -- type of real*4
c -- type of complex*8
dc -- type of complex*16
```

Two modes of debugging an application are available within the combined system. One is the Fortnet debug('ON') option, which dumps data to the Trace file fortnet.graph for conversion to Trace format in trace.graph by the utility pretrace.c. The other is the setdbg(1) call from the ipcv3 harness which produces very detailed screen diagnostics of the internal ipcv3 and socket calls. Knowledge of the ipcv3 harness workings are needed to interpret this output!

The combined ipcv3 and Fortnet harness is contained in the following directories:

```
cxa /priv3/rja/hybrid/ipcv3
cxa /priv3/arcg1/?
tcsm /home/rja/hybrid/ipcv3
tcsm /home/arcg1/?
i860a /usr/user/rja/hybrid/ipcv3
dlal1 /user/rja/hybrid/ipcv3
dlal1 /user/arcg1/?
titan /user/rja/hybrid/ipcv3
titan /user/arcg1/?
iris2 /usr1/rja/hybrid/ipcv3
ipsc /usr/user/rja/hybrid/ipcv3
cray /?/mfg
```

The library file fortnet.a should be linked with the application which is then invoked by the parallel command through the configuration file <application>.p. Note that Fortnet also requires the application program on each node to be driven by a main program called 'worker', and this now extends to ipcv3 and also Schedule (which in the original document used a main program called 'dmain'). A sample makefile for the Intel hypercube is as follows:

```
all: server.out master.out slave.out
server.out:
        f77 -o server.out -i860 server.f fortnet.a -node
master.out:
        f77 -o master.out -i860 worker.o master.f fortnet.a -node
slave.out:
        f77 -o slave.out -i860 worker.o slave.f fortnet.a -node
```

The execution can be invoked from a configuration file called fortnet.p such as:


```
rja i860a 1 server.out .
rja i860a 1 master.out .
rja i860a 14 slave.out .
```

with the command

```
parallel fortnet
```

which grabs 16 processors and runs the task.

II. Fortnet and Trace.

Version 2.2 (trace) adds to the basic Fortnet system a sophisticated graphical post-execution trace and playback facility. This is in the form of an interface to the Schedule Trace package developed by Dongarra and Sorensen of Argonne National Lab [15]. A display on the SUN screen shows, in pseudo-real time or in event time, the state of execution and communication events in a real network of communicating processes. The data dependency tree for the processes is constructed from the run-time dump by a C pre-processor called pretrace.c. By this means parallel FORTRAN programs can be visually debugged and optimised since the cause of deadlocks and bottlenecks can be spotted.

To use the graphic display of sched trace files you need to do

```
sched.trace
```

and follow the instructions of section C.

The information in the trace files has the following meaning from SCHED.

```
0 - static node <id, # child, # parents, [parent ids]>
1 - start execution <id>
2 - stop execution <id> timing info
3 - create dyn node <owner id, id>
4 - start dyn node <owner id, id> timing info
5 - stop execution dyn node <owner id, id> timing info
```

To compile the directory on the SUN, /home/rja/schedule/trace, do

```
make sched.trace
```

The trace facility has been interfaced to the Fortnet v3.0 (trace) harness to produce graphical interpretation of a FORTRAN program running on various hardware platforms. We could for instance run the above application on the Intel, with debug mode 'ON' in the routines we want to see. When execution has finished a file fortnet.graph will be present. This is true of all Fortnet versions accessed through a library file called trace.a, trace.o or

fortnet.a. This graph file must be passed through the preprocessor pretrace.c by running

```
pretrace.out
```

in the directory, with no arguments, which creates the file trace.graph for input to the Trace program exactly as in the references to the original work. The graphical tool must be invoked on the a SUN or X-window workstation to see the result (we do not yet have X installed on the Intel).

Meaning of the information in the file fortnet.graph

```
6 - check called <taddr> <faddr> timing info
7 - end of check <taddr> <faddr> timing info
8 - wait called <taddr> <faddr> timing info
9 - end of wait <taddr> <faddr> timing info
10 - send called <taddr> <faddr> timing info
11 - end of send <taddr> <faddr> timing info
12 - receve called <taddr> <faddr> timing info
13 - end of receve <taddr> <faddr> timing info
14 - debug on 0 <inode> timing info
15 - debug off 0 <inode> timing info
16 - debug toggle 0 <inode> timing info
17 - stop active 0 <inode> timing info
18 - stop terminate 0 <inode> timing info
```

III. Interface of Schedule and the message-passing utilities.

The message-passing harness which is the combined result of Fortnet and ipcv3 may now be used in a task which has its major modules controlled by the Schedule software. We have enforced the rule that only jobs scheduled on the same level of data dependency may communicate using the message-passing primitives. It this is not done the coding becomes unnecessarily complicated. Any such deviation is construed to be a user error, and there is no protection.

The interface may be driven through a configuration file which is an extension of the ipcv3 and Schedule notation as shown below. The 'parallel' server program interprets this file as a stream of

jobs to be scheduled, in which the execution tree allows communication between branches at a similar height. Furthermore several tasks may be specified in the configuration file, to allow a rudimentary batch processor to be implemented. A certain amount of resource load balancing can be expected during the course of execution.

This full implementation is in somewhat early days and may not always work as expected. Any bugs should be reported and I will try

to sort them out!

III.1 Fortnet version of Schedule

The currently working version of Schedule on the Intel hypercube incorporates the Fortnet message-passing harness. This has been possible without great difficulty, but did require the addition of a message router table to translate schedule job tags (by which the individual pieces of the application are known to the user) into real processor addresses which can be handled by the Fortnet router. This is now present also in the generic ipcv3-based version, along with the xdr routine calls. There is a slight overhead in looking up entries in this table, and a slightly greater overhead in starting a new job from the Schedule queue, which necessarily involves updating the router table on the server process for forwarding to the jobs at communication time (only when a call to check or wait is made). If a job is not scheduled when a message is addressed to it, or messages are incorrectly sent, a warning is output to the standard output device. In futurre versions of the software we will try to make such calls invoke the scheduler for te required process.

Use of the system is no more complicated than previously calling the schedule routines but with the possible addition of message passing. The Fortnet library is used for this, with strong typing, and the Schedule job tags must be used as the logical address of each process. The server is still loaded on node zero by the Schedule system (it must have the name 'server.out') and acts as a shared device to support i/o and router table manipulation from a fixed point. This combined system offers the possibility to use object-oriented programming and also a message-passing style within one package, and is one of the only message-passing schemes to have dynamic configuration (another is Linda).

Consider reworking the Intel example given abovve for a simple tree structure in which master1 sets up data, a number of slave routines work on it, and master2 collects the results.

```
all: host.out server.out master1.out slave.out master2.out
host.out:
        f77 -o host.out host.f sched.a iosup.a -host
server.out:
        f77 -o server.out -i860 server.f fortnet.a sched.an -node
master.out:
        f77 -o master1.out -i860 worker.o master1.f fortnet.a
sched.an -node
slave.out:
        f77 -o master2.out -i860 worker.o master2.f fortnet.a
sched.an -node
slave.out:
        f77 -o slave.out -i860 worker.o slave.f fortnet.a
```

sched.an -node

Execution is invoked by starting the host program with
host.out

which loads the cube. The data dependency graph might be described in a configuration file as follows, and the host program (shown next) will try to read this file to set up the task if it is of the standard form:

```
program main
integer dum1, dum2, dum3, dum4, l(4)
data n/4/,l/4*0/
nprocs=4
call sched(nprocs,parconfig,n,l,dum1,dum2,dum3,dum4)
end
```

The routine parconfig reads the configuration file shown below and makes calls to DEP and PUTQ after allocating the Fortnet distributed data scheme, and assigns memory for each of the variables used as shown in section A.

A configuration file, modified from the ipcv3 form to include Schedule functionality (the second line is currently unused) might be as shown next. Note that at present all subroutine arguments are double precision variables. This can soon be relaxed since strong typing of variables is implemented. (c.f. section B and above on the xdr implementation)

```
TASK
nprocs, jdf, priority, submit time, run time requested
rja i860a 1 master1.out .
master1(a,b,c,d)
icango=0 nchks=4 mycheckn=(2,3,4,5)
rja i860a 4 slave.out .
slave(a)
icango=1 nchks=1 mycheckn=6
slave(b)
icango=1 nchks=1 mycheckn=6
slave(c)
icango=1 nchks=1 mycheckn=6
slave(d)
icango=1 nchks=1 mycheckn=6
rja i860a 1 master2.out .
master2(a,b,c,d)
icango=4 nchks=0
END TASK
```

IV. Networked version of Schedule with ipcv3 and Fortnet

The combined Fortnet and Schedule interface is not portable at present, but will be so in the near future. To achieve this it is necessary to include a higher level of message passing in the Schedule scheme, across a network within the ipcv3 harness, and then also the additional user interface as at present. Full addition of strong typing of data and xdr translation has begun. An extension of ipcv3 must be made which can rsh jobs at any time rather than just statically at the start of execution, and this must be driven by the present host program of Schedule.

V. Final comments

The portable software available at Daresbury Laboratory has been presented in the current implementation as of November 1990. It results from original work done in designing a message-passing harness and a library of high-level numerical operations (described separately) combined with concurrent progress at Argonne National Laboratory. Most of the work on the user interface to the combined Fortnet and Schedule system is being done by N.Clancy from Bristol Polytechnic as part of his M.Sc. placement project. It is hoped that in combining these software resources a standard interface may be established, through its widespread acceptance in physical and chemical science, which will permit relatively uninterrupted development of parallel applications on an abstract but low level.

If you require to use, or wish to know more about any of the software described in this document please contact

rja @ uk.ac.dl.cxa on JANET

References.

[3] R.J.Allan, E.L.Heck and S.Zureck, "Parallel FORTRAN in scientific computing: a new occam harness called Fortnet" Computer Physics Communications 59 (1990) 325-44

[4] R.J.Allan and E.L.Heck "Fortnet: A parallel FORTRAN harness for porting application codes to transputer arrays" in "Applications of Transputers 1" ed. T.L.Freeman and R.Wait Liverpool (23-25 August 1989)

[5] Intel Corporation "Intel iPSC/2 Programmer's Reference Manual" October 1989, order number: 311708-002

[6] Intel Corporation "Intel iPSC/2 and iPSC/860 User's guide" June 1990, order number: 311532-006

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

[8] 3L Ltd., "Parallel FORTRAN Reference Manual" 3L Ltd., Peel House, Ladywell, Livingston, Edinburgh EH54 6AG

[9] R.J.Allan "FORTRAN-77 Programming of Parallel Computers. IV: 3L Parallel FORTRAN" Parallelogram 22 (January 1990)

[10] S.J.Leffler, R.S.Fabry and W.N.Joy "A 4.2BSD Interprocess Communication Primer"

[11] Alliant Computer Systems Corporation "Concentrix: System Reference version 5.0.0" Alliant (Feb. 1990) order number: 301-02009-A

[12] R.J.Allan "Numerical Algorithm Libraries for Multicomputers" D.L. Technical Memorandum (1990) in preparation

[13] R.J.Allan "An Explicit Library Interface for Scientific programming of Parallel Computers" for submission to Parallel Computing (1990)

[14] J.Dongarra and D.Sorensen "A Portable Environment for Developing Parallel Fortran Programs", Parallel Computing, Volume 5, Numbers 1&2, July 1987, pp. 175-186 or available as a postscript document from netlib@argonne, or from rja@dl.dlgm

[15] J.Dongarra and D.Sorensen "Schedule Users' Guide" postscript document available from netlib@argonne, or from rja@dl.dlgm

[16] "Portable Programs for Parallel Processors", J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, R. Stevens. Holt, Rinehart and Winston, Inc., 1987, ISBN 0-03-014404-3

[17] R.K.Cooper and R.J.Allan "Fortnet (3L) v1.0. Implementation and Extensions of a message passing harness for transputers using 3L Parallel FORTRAN" for submission to Computer Physics Communications (1990)

[18] Meiko Scientific Ltd., "SUNOS CS-tools" Vols 1 and 2 order number: 83-009A00-02.01 Meiko (1990)

[19] L.J.Clarke Ph.D. Thesis, University of Edinburgh (1990)

[20] L.J.Clarke "TINY, Discussion and User's Guide" Edinburgh Concurrent Supercomputer Project, order number: ECSP-UG-9 University of Edinburgh (7th March, 1990)

[21] M.Surridge "ECCL: A general communications harness and configuration language" in "Applications of Transputers 2" proceedings of the second International Conference on the Application of Transputers, ed. D.J.Pritchard and C.J.Scott. Southampton (1990)

[22] N.Carriero and D.Gelernter "How to write parallel programs, a guide to the confused" University of Yale Research Report, order number: YALEU/DCS/RR-628 (May 1988)

[23] ParaSoft Corporation "Express 3.0 User's Guide" ParaSoft (1988, 1989, 1990)

[24] Apollo "NCS User Manual"

[25] I.Foster and S.Taylor "Strand - New concepts in parallel programming" Strand_88 (Prentice Hall, 1990) ISBN 0-13-859587-X