Theor Chim Acta (1993) 84: 257–269

Theoretica Chimica Acta

Towards a portable environment for FORTRAN applications on parallel computers

R. J. Allan

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

Received October 1, 1991/Accepted January 14, 1992

Summary. This paper extends the communicating sequential process model of communications on multicomputers to a virtual machine interface with skeleton algorithms for global communication of data. The software is constructed using elements of object-oriented programming. Data is divided into two types of variables: those stored locally to each process and those accessible globally. Transfer of data between processes is implicit in the library structure, providing opportunities for optimisation.

The application-level interface currently resembles a vector linear algebra library. Generic communication skeletons are defined for building libraries specific to applications; they illustrate more general programming issues. An example from a wave-scattering algorithm is given.

Key words: Parallel FORTRAN programming – Communication harness – Numerical library – Application interface – Virtual shared memory – Molecular dynamics

1. Introduction

This paper aims to convey a picture of the current status of work done within the UK's Science and Engineering Research Council (S.E.R.C.) to develop a solution to parallel programming for portability of applications within the context of a rapidly evolving arena. Faced with the problem of proprietory approaches to software on every new hardware platform (typified by the appearance of the Occam language on the Inmos Transputer) and the need to preserve 15–20 years of development work in computational science we have defined and implemented a number of interface layers which permit source code written in a communicating sequential process (csp) style to be used on a wide range of computers. Two of the interface layers to which the user has access are described in this contribution. They are: Fortnet, a message passing harness, and PARLANCE (Parallel Library and Networked Computing Environment) a virtual shared memory interface currently specialised for linear algebra computations.

2. Fortnet v4.0 (trace)

The original concepts of Fortnet and its v2.1 incarnation have been dealt with in [1, 2]. A few new additional features are worth mentioning, and a summary can be given of the current situation and functional interface. Other message-passing harnesses have been developed in the USA along similar lines to ours: PICL from Oak Ridge National Laboratory [3], Express from CalTech (now marketed by ParaSoft Inc. [4]) and PARMACS and tcgmsg from Argonne National Laboratory [5, 6]. We have in fact used the tcgmsg toolset as a basis for our UNIX implementation of Fortnet [see contribution from R. J. Harrison in this volume].

Fortnet is now available for use with the following hardware and low-level routing software

(i) Meiko Computing Surface Occam-2 libraries
(ii) Meiko Computing Surface with CSTOOLS, both Transputer and i860 processors
(iii) Intel iPSC/2 and iPSC/860
(iv) Transputers running 3L Parallel FORTRAN-77
(v) UNIX Workstations
(vi) UNIX machines coupled by a TCP/IP Network
(vii) Alliant fx/80 and fx/2800 parallel computers
(ix) CRAY X/MP or Y/MP
(x) Silicon Graphics Parallel Workstation
(xi) Stardent Titan P1500 Parallel Workstation

Fortnet is a multi-layered system of subroutines accessible as a library. 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. A diagrammatic representation of the layered software approach is shown in Fig. 1. The structure of Fortnet is described as follows.

Initial development 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 we started in 1987. The first stage was therefore on Occam-2 communication harness to pass messages between the transputers in a controlled fashion. It also performs some tasks like accessing the front-end file-store, printing diagnostic messages to the screen, and bookkeeping. This layer has been re-written for whatever proprietory software is available on other systems.

The second stage of development is a layer of FORTRAN-77 subroutines which may be called from the application as a standard interface. These incorporate a protocol to verify the correct transmission of messages and warn the user of any problems. A novel handshaking and blocking paradigm is used for this checking which differs from other systems, but adheres to the csp model. The routine calls are however superficially similar to those on hypercube machines such as the Intel iPSC and NCube.

Fortnet is configured as a number of communicating processes. Process -32768 is a host program (a scheduler, using either the tcgmsg "parallel" tool or the Argonne National Laboratory Schedule software [7]) with a textual configuration file. Process 0 is the Fortnet file server which maintains information about the status of the current task. Process 1 is a master process,

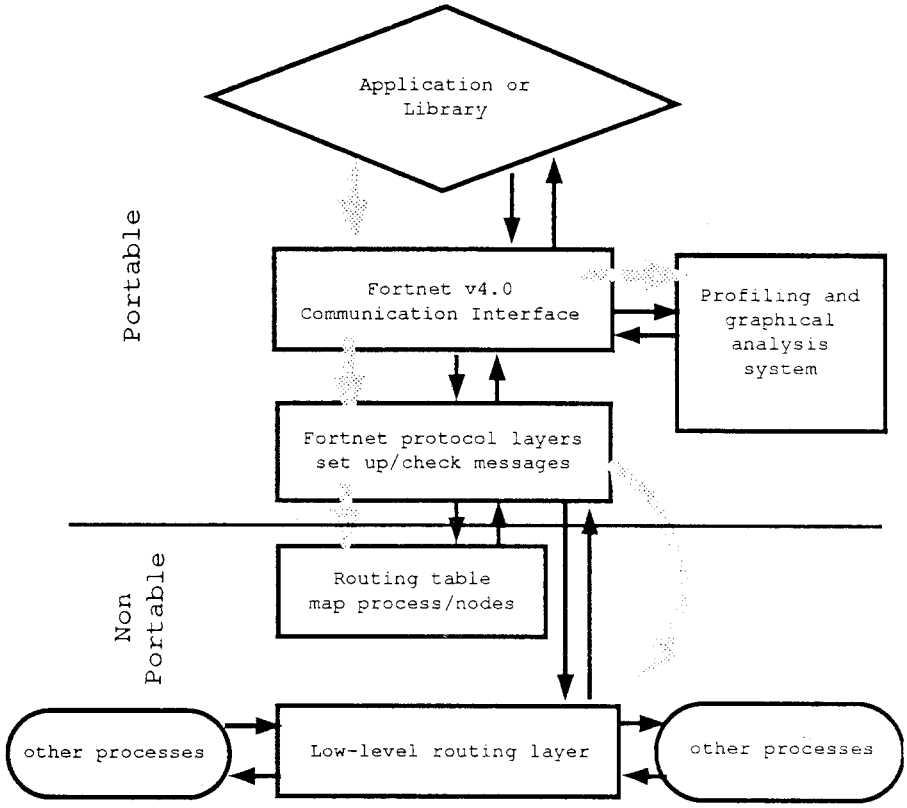

Fig. 1. Diagrammatic representation of software layers in the Fortnet Communication Harness. The lowest levels are machine dependent using proprietory software. The higher levels are portable and application libraries (such as PARLANCE) and profiling tools can then be provided as discussed in the text

part of the task, or it may be a copy of a slave process. Process 2 through numslave( ) + 1 are slave processes, which may either be invoked from the same executable or different ones. Processes 1 through numslave( ) + 1 comprise the task and they are referred to as jobs. Each job is a sequential stream of FORTRAN statements and subroutine calls, some of which send and receive data to other jobs.

A synopsis of the Fortnet library routines is now given, with extensions employed in the current work. These extensions were made to allow data transfer between heterogeneous processor types. The xdr (external data representation) routines standard to a number of UNIX systems are used. Strong typing of data is confirmed by using a Type Indicator (ti) which is a single character to denote the type of data handled:

A – character, I – integer, C – complex, S – single precision, D – double precision, Z – complex double precision, L – logical.

The message-passing harness interface library contains the following routines:

subroutine check(m) – send a handshake signal to process m and wait for a reply

subroutine wait(n) – wait for a handshake signal from process n and then reply

subroutine waitany(nproc) – wait for a handshake signal from process nproc and then reply returning the value of the source in nproc. This provides essential functionality for parallel programming similar in nature to the Occam ALT constructor.

subroutine send(n,mbytes,data,ti) – send nbytes from buffer data to process n. The Type Indicator ti was described above

subroutine receive(m,nbytes,data,ti) – receive nbytes of data of type ti into buffer data

subroutine exchange(n,m,nbytes,data1,data2,ti) – exchange nbytes of data of type ti between buffers data1 and data2 local to process n and m, respectively

subroutine brlist(proclist,nproc) – subsequent broadcast and syncronisation routines will involve job numbers from the nproc entries in proclist. An entry of −1 is a null process

subroutine brcast(nproc,root,nbytes,data,ti) – broadcast nbytes of data from root to all nproc entries of proclist. Sending to root is ignored.

subroutine brall(mode) – switch the subsequent mode of broadcast operations. Possible modes are 'ON', 'OFF', and 'RESET'.

subroutine sync(nproc,root) – synchronise processes which are the first nproc entries in proclist after sending a request to the Server.

Other routines give information and invoke instrumentation and diagnostics:

j = jobid( ) – return job id of calling process

nnode = numslave( ) – number of Fortnet slaves on which jobs can be scheduled as part of the current task

subroutine debug(mode) – switch the subsequent mode for instrumentation. Possible modes are 'ON', 'OFF', 'TOGGLE'. The trace file produced by this instrumentation can be fed into graphical analysis software such as ParaGraph [see 3].

subroutine setdbg(i) – invoke diagnostic trace from the underlying tcgmsg harness (if it is used)

subroutine cpu(time) – return real*8 cpu time used by the local job in seconds

subroutine walltime(time) – return real*8 elapsed seconds measured by the local processor clock

subroutine stop( ) – signal to the Server that this process is terminating. The ONLY legal statement to follow this is a FORTRAN STOP.

subroutine counter(k) – get the next value of a shared counter from the Server

Routines to do i/o via the shared file server running as job 0 will not be further discussed here. They are outside the current context but we note that shared i/o is one of a number of modes also provided by Express and the Intel iPSC concurrent file system.

We now describe an example of portable programming. It is a library of application-level algorithms to interface virtual shared memory.

3. PARLANCE (Parallel Library and Networked Computing Environment) – Distributed algorithm library strategy

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 application. 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. Programmability arises through hiding the communications routing altogether inside convenient packages 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).

The library approach has several advantages for application programming:

(i) routing may be optimised, eventually in hardware (e.g. KSR1)

(ii) global communications may be optimised, possibly in hardware

(iii) MIMD 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 a true parallel programming paradigm is essential

(vii) portability can be assured by modifying only communication layers

(viii) Analysis and performance tools may be built into well-defined communication layers

(ix) underlying vector hardware can be used

A feature of my work is that in using the application-level routines true parallel programs are written which execute concurrently on all the processing nodes. The algorithms provided in the library will work irrespective of the data placement. A routine for one of these global operations must be called simultaneously by all the jobs (they are loosely synchronous in the language of Express). Communications are then done internally in a general way and actions are performed on the true data which may be accessed by the calling program via a symbolic variable. Top level programming is therefore sequential and modular. Each module (skeleton) is a parallel operation implicitly doing message passing. The modules could also be optimised to run purely in shared memory.

3.1. PARLANCE skeleton library

The PARLANCE global communication skeleton library differs significantly from other developments and depends fundamentally on, firstly a set of global symbolic variables which reference real data somewhere in the multicomputer's memory, and secondly, on a set of routines which control completely parallel overlapped communications and allow movement of data between these variables.

All the symbolic variables are currently treated as contiguous vectors (although some of the routines access them as square matrices). Any other form of matrix can be mapped on to them by calculating the indices and, as will be seen below, that is not often necessary. Routines are provided to find which elements belong to a given job for instance. Other routines assign space, put and fetch real data from the symbolic storage, perform gather, scatter and more complex operations.

3.1.1. Access to symbols and tables in the library. The generic symbol operations and skeletons are not intended to be used by the application programmer, but are provided to build the libraries. They illustrate our use of ideas from object-oriented programming such as hierarchy, inheritance and re-useable code. A list of routines in the present version is:

subroutine allocate( ) – initialise PARLANCE global system.

This must be done before calling any of the other routines.

subroutine assign('a',nbytes,ti) – assign nbytes of local memory for storage of data in symbolic variable 'a' (up to eight characters) attached to the job which calls it. Enter 'a' into the symbol table for this job. This is a "monadic" routine (i.e. no communications). Ti is a type indicator as defined above.

subroutine deassign('a') – removes entry 'a' from the tables and performs garbage collection in the global memory if necessary.

subroutine swaptable( ) – global exchange of symbol tables between all jobs. This need never be called by the user.

subroutine put(iproc,x,offx,stepx,'a',offa,stepa,n,ti) – copy n elements of data from real variable x defined locally to job iproc into globally defined memory referenced by symbol 'a'. Offsets for start of data and strides are given. Communications are internal and 'a' is as defined in previous assignment calls. A distributed scatter operation.

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

subroutine gather0('a',offa,stepa,'b',offb,stepb,n) – on the job which called it, 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 the local memory area. This is a monadic routine.

subroutine scatter(i,j,'a',offa,stepa,n) – finds all available elements of 'a' as described which are belong to job i and attempts to send them as a single message to job j. The data is preceded 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 (i.e. pairs of jobs communicate).

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

gather1, scatter1 and gather0 form the basis of higher routines and were described for that reason. They are very powerful routines and quite novel in their behaviour. They also carry out masking so that only elements which can actually be stored in local memory on j are sent as a single buffered message.

subroutine gather2(j,'a',offa,stepa,'b',offb,stepb,n) – collects data referenced by symbol 'a' from all jobs, including job j, and attempts to store it in 'b' of job j. 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)
     &   call scatter1(i,j,'a',offa,stepa,n)
      if(inode.eq.j.and.i.ne.j)
     &   call gather1(i,j,'b',offb,stepb,n)
      end do
      if (inode.eq.j)
     &   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 simple ones.

Various assorted additional routines are provided, mainly to gain access to information about global storage of a particular symbol:

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

Aproc(i) is the job having the ith entry (it might occur more than once of the storage is fragmented). Alocal(i) is the offset from the beginning if the core storage for the start of this piece of contiguous memory. The core is currently implemented as a FORTRAN common block of predetermined maximum size, e.g.

common/core/dcore(big), . . .

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 job aproc(i) is clearly aglobal(i + 1) – aglobal(i). There are routines extractb and extractc defined in the same way for common areas extrb and extrc.

subroutine 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 job iaproc having element with global offset ia. There are similar routines oneb and onec.

subroutine puta(temp,'a',ia) – monadic routine, assuming the value of temp is globally present it puts into the local memory of whichever job contains a(ia), uses common/extra/. There are similar routines putb and putc.

temp = fetcha('a',ia) – fetches the value a(ia) and does a broadcast to all other jobs, uses common/extra/. This is affected by the broadcasting mode set by the call to routine brall (see above). There are similar routines fetchb and fetchc.

i = index2d(ndim,j,k) – find the offset of element (i,j) from the start of a 2-dimensional array of storage of column length ndim. A monadic routine.

3.1.2. Global communications skeletons in the library. The simpler vector and scalar code skeletons are listed below. A more complex one which is specialised to do a matrix multiply is illustrated in Sect. 3.3.

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

subroutine global2(c,icstart,icstride,a,iastrat,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)

3.2. Parallel communications and data placement

The efficiency of any algorithm is critically dependent on the speed of communications, and their scalability. One factor which influences scalability is data placement. We can easily see that if all communications are local (only to nearest neighbour processors on a network) then adding more processors will maintain the same ratio of communicate time to compute time if the size of problem is also scaled so that the same amount of computation is done per processor. A new body of theory and algorithms will need to be devised to permit this. This has been little considered in the literature and is the topic of future work. We so far have analysed only the communication skeletons themselves.

Fortnet is a purely synchronous system (or to be more accurate a blocked asynchronous system) which obeys the csp rules. We are however able to overlap communications within the skeleton operations as shown below. We devise algorithms which do not benefit from overlapping communication and computation as this would only yield unpredictable results in practice and complicate porting issues.

3.2.1. Covering scheme. An important requirement of global communications is that they should all work in parallel. The skeletons require communication between constituents of all pairs of jobs (e.g. in a matrix transpose), or generally of all sets of k jobs in the array of nproc jobs. It is possible to optimise the parallelism in this by "covering" the machine's nproc jobs with as many independent k-fold connected subsets as possible, and allow communications within each to be concurrent with the others. Another covering with different connections is then used and so on until all possibilities have been exhausted. The solution to this problem is from permutation theory.

The number of different ways of covering nproc jobs with k of them:

$$Mk(p) = p!/((k!)**N*N!*m!);\ N = p/k;\ m = (p - Nk)$$

the exclamation sign denotes a factorial.

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 job n should talk to job connect(n).

The matrix is filled by calling:

subroutine covering(i,k,nproc) – for the ith k-fold covering of nproc jobs.

These routines are written in C and are monadic. Note the no consideration is given to mapping onto the real processor network and that we assume complete virtual connectivity is provided at the highest speed by the hardware manufacturer. This will certainly be true in all future machines such as the Intel Delta and Sigma using a high speed grid backplane with wormhole routing chips and the Kendall Square and other machines with fast internal token ring networks.

3.2.2. Data placement. For good performance it is essential to have optimal placement of data. This is becasue in the current range of parallel machines fetching data from a distant job is more costly than from local memory.

We have so far established a number of ways to distribute data. The library algorithms, as well as handling an arbitrary distribution, therefore, contain specialisations for specific types of distribution, e.g. a matrix with one complete column per job. This is denoted by a Distribution Indicator (di) which forms part of the PARLANCE knowledge base and is attached to each symbol. A routine is provided to set up storage (calling subroutine assign repeatedly) or to modify existing storage; it also performs garbage collection if necessary.

storeby(type,a,ndiag,lencol,lenblock,widblock,nbytes,ti) – this routine will create storage for a particular type of matrix. Type can take the values 'COLUMN', 'BLOCK', 'NDIAGONAL', 'TRIANGLE'. Ti is the type indicator as above, and the other parameters detail the shape of storage required and the mapping to jobs.

3.3. Pedagogical example

The following example is the kernel of a matrix multiplication a = b*c where all three matrices may be distributed across the whole machine. The resulting routine matmul would be the one eventually accessed by the application programmer from a library. This example shows a way the library is constructed using the dyadic covering with masks when not all jobs are needed.

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)
call dot(a,(j − 1)*n + i,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. Our dot routine, which is also part of the distributed vector library, is clever enough to execute the following code:

```
do i = 1,n
do j = 1,n
call dot('a',(j − 1)*n + i,'b',i,n,'c',(j − 1)*n + 1,1,n)
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 dot operation instead of leaving v1 until all values of v2 have been used.

```
subroutine matmul(a,b,c,n)
character*8 a,b,c
logical exist(nproc),exist1(nproc)
dimension in(nproc),jn(nproc)
```

```
      common/cover/connect(nproc,2)
      data exist,exist1,in,jn/nproc*.false.,nproc*.false.,
     &                        nproc*0,nproc*0/
      call assign('v1',n*4,'S')
      call assign('v2',n*4,'S')
      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 j = 1,nproc
      k = connect(j,1)
      k2 = connect(j,2)
      if(.not.exist(k))call next.element(k,a,jn(k),exist(k))
      if(.not.exist(k2))call next.element(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) + 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)
      ene if
      end if
      end do
      end do
      do i = 1,nproc
      if(exist1(k))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 dot(a,jn(inode),1,'v1',1,1,'v2',1,1,n)
```

```
      end if
      if(kount.lt.n*n)goto 10
c release virtual storage space
      call deassign('v2')
      call deassign('v1')
      end
```

3.4. *Wave-scattering example*

The folllowing lines of code are taken from a wave-packet scattering program. The actual application is of particle scattering from surfaces and uses the method of Kosloff and Kosloff [see [8, 9] and also the contribution by David Chasman in this volume]. It repeatedly computes the second derivative of the wave-function using a Fast Fourier Transform method so that the time-dependent Schrödinger equation may be solved by Chebyschev propagation after a number of time steps. This is also typical of the standard molecular dynamics and also the Car–Parinello methods. The algorithm has been modified for use on parallel computers by computing the FFT only in the x,y plane (the surface plane for which the periodic conditions apply). Domain decomposition in the z direction is used and the FFT in that direction replaced by a finite-difference formula:

$$\nabla_Z^2 \Psi = [\Psi(z + Az) - 2\Psi(z) + \Psi(z - Az)]/Az^2$$

Only data transfer across the boundary to neighbouring domains is then necessary; which is highly efficient if domains are directly mapped onto processors. The example code has just one z point per processor for simplicity.

```
      ...
c number of points
      nxy = nx*ny
      nxyz = nxy*nz
c do x,y part using FFT in local memory
      call copy('dpsi',1,1,'psi',1,1,nxyz)
      call extracta('dpsi')
      do for all x,y planes
c get pointer to core (see explanations above)
      call onea(1 + nxyz*(inode - 1),iacore,iaproc)
      call tfft2d(dcore(iacore), . . . )
c do local phase-space factors and inverse transform
      ...
      end do
c do z part using virtual shared memory
c assign distributed temporary variable
      call assign('dpst',nxy*8,'D')
      call copy('dpst',1,1,'psi',1,1,nxyz)
      call ascal('dpst',1,1,-2.0,nxyz)
c loop over z
      do i = 0,nz - 1
      iz = i*nxy + 1
      izp1 = iz + nxy
      izm1 = iz - nxy
      if(izm1.gt.0)call add('dpst',iz,1,'psi',izm1,1,nxy)
```

```
      if(izp1.lt.nxyz)call add('dpstp,iz,1,'psi',izp1,1,nxy)
      end do
      call ascal('dpst',1,1,1.0/(dz*dz),nxyz)
      call add('dpsi',1,1,'dpst',1,1,nxyz)
c delete temporary variable and collect garbage
      call deassign('dpst')
      ...
```

Acknowledgements. The work on Fortnet was done in collaboration with Dr. Lydia Heck of Durham University. Dr. Richard Cooper of the Department of Aeronautical Engineering, Queen's University of Belfast, has helped in further developing the Fortnet harness. I thank Dr. Robert Harrison of Argonne National Laboratory for permitting me to use his tcgmsg toolset. Some work on the PARLANCE message-passing environment was carried out by Norman Clancy as part of his M.Sc. placement from Bristol Polytechnic at the end of 1990.

References

1. Allan RF, Heck EL (1990) in: Freeman TL, Phillips C (eds) Fortran harness for porting application codes to transputer arrays. Applications of Transputers I. Proc 1st Int Conf on Applications of Transputers, IOS Press, ISBN 90 5199 025
2. Allan RJ, Heck EL, Zurek S (1990) Computer Physics Communications 59:325
3. Geist GA, Heath MT, Peyton BW, Morley PH (1990) PICL A Portable Instrumented Communication Library. Oak Ridge Natl Lab Techn Memorandum ORNL/TM-11130
4. Kolawa A (1989) Express users' guide and reference manual. ParaSoft Inc.
5. Bomans L, Hempel R (1990) Parallel Computing 15:119
6. Harrison RJ (1990) Documentation of the tcgmsg toolset, and private communication. Argonne Natl Lab (see also another contribution in this volume)
7. Dongarra JJ, Sorenson DC (1987) Parallel Computing 5:175
8. Kosloff D, Kosloff R (1983) Comp Phys 52:35
9. Tal-Ezer H, Kosloff R (1984) J Chem Phys 81:3967