PORTING LINEAR ALGEBRA SUBROUTINES FROM TRANSPUTERS TO CLUSTERS OF WORKSTATIONS[*]

RUDNEI DIAS DA CUNHA [†] AND TIM HOPKINS [‡]

Abstract. We report on our experiences in porting a number of linear algebra subroutines, written in `occam2`, from a transputer environment to a cluster of workstations using Fortran77 and the PVM message-passing system.

1. Introduction. Linear algebra subroutines (LAS) are the building blocks of many large scientific and engineering software packages. Indeed most applications spend a large proportion of their time executing one or more of those subroutines. We have already shown in [3] and [2] that highly efficient parallel implementations of these LAS can also provide the building blocks for the efficient parallel solution of systems of linear equations using iterative methods on transputer (T800) networks. In this paper we report on porting these implementations, written in `occam2`, to a cluster of workstations.

The software environment used on the workstations consists of Fortran77 and the Parallel Virtual Machine (PVM) message-passing system [1], version 2.4.1 (any references made to PVM in this paper refer to this version only). PVM is a highly-portable, public-domain package that provides the means of exchanging data between two processes executing on a (possibly) heterogeneous network. It provides a dæmon which, when executed on each of the machines involved in the computation, is responsible for starting up the processes. The application software, coded in C or Fortran77, communicates with other processes by calling the supplied PVM library routines. Different parallel programming models can be used, e. g. master-slave, single-program-multiple-data (SPMD), etc. , and the application may also use mixed models. Several machines are supported, ranging from Unix workstations to the Intel iPSC/860 and including vector supercomputers. PVM is being supported by several vendors, including IBM, Convex, HP and Meiko Scientific. The actual hardware used was a network of Sun Sparc2 workstations interconnected by Ethernet.

A brief overview of the paper is as follows. We describe in §2 the porting of the LAS implementations from an `occam2`/transputer environment to a Fortran77/PVM environment. In §3 we provide some experimental results obtained using the Fortran77/PVM implementations and compare those with the `occam2`/transputer version; we also provide a scalability analysis of both versions. Finally we offer some conclusions in §5.

2. Porting the subroutines. We consider in detail the conversion of two LAS routines, for computing inner-products and matrix-vector products; these are the routines where computation is distributed across the processors and involve communica-

[*]THIS IS A PREPRINT OF A PAPER SUBMITTED TO THE *WORLD TRANSPUTER CONGRESS*. SINCE CHANGES MAY BE MADE BEFORE PUBLICATION WE THEREFORE REQUEST THAT IT WILL NOT BE CITED OR REPRODUCED WITHOUT THE PERMISSION OF THE AUTHORS.

[†]CENTRO DE PROCESSAMENTO DE DADOS, UNIVERSIDADE FEDERAL DO RIO GRANDE
DO SUL, BRASIL AND COMPUTING LABORATORY, UNIVERSITY OF KENT AT CANTERBURY,
U.K.

[‡]COMPUTING LABORATORY, UNIVERSITY OF KENT AT CANTERBURY, U.K.

tion between them. For the other LAS needed, for example, saxpys, vector scalings, Givens' rotations, we used the Fortran77 codes provided in the BLAS Level 1 library [4].

The first important issue to be considered is that of topology. The `occam2` implementation of the LAS exploited the four links available on the T800 and a grid of transputers was used (a torus interconnection was not needed for our applications). On the cluster of workstations, however, we do not have the flexibility of choosing a particular topology that suits an application. With the Ethernet connection available, we are constrained to use a *logically connected* linear array of workstations. This change in topology dictated the major modifications which needed to be made to the LAS. Note that while nothing prevents us having a logically connected grid of workstations, this would increase communication times, since each workstation has only one connection to the network.

Before commencing the port, we carried out a number of experiments with PVM to assess its capabilities. We found that there is a significant overhead in using the usual send/receive routines, where the message is routed to the destination process through the dæmons. However, PVM offers faster versions of these routines (called `fvsnd`/`fvrcv`), which implement a point-to-point exchange of data between two processes without using the dæmons, and it is these routines that were used in our implementations.

The porting process consisted of two distinct phases: modification of the algorithms, due to the difference in topology, and translation from `occam2` to Fortran77 of the portions of the computational code.

The main difference in the coding is obviously in the communication part. The elegance of `occam2` with its built-in statements for the send/receive operations does not exist in Fortran77/PVM. The exchange of data between two processes using PVM involves

- initialisation of the PVM buffer area by the sending process,
- storing the data into the buffer,
- sending/receiving the data and
- extraction of the data from the buffer by the receiving process.

These operations are performed by calling the appropriate PVM routines.

The following example clarifies the modifications. Suppose a vector `u` of `n` double-precision elements is to be sent from processor 0 to processor 1. In Table 2.1 we show both the `occam2` and Fortran77 codes for the above situation.

Sending	Receiving
`out!  VEC64; n::[u FROM 0 TO n]`	`in?  VEC64; n::[u FROM 0 TO n]`

`call finitsend()`
`call fputndfloat(n,u,info)`
`msgtype=100`
`to=1`
`call fvsnd(msgtype,to,pname,info)`
	`msgtype=100`
	`from=0`
	`call fvrcv(msgtype,from,pname,info)`
	`call fgetndfloat(n,u,info)`

TABLE 2.1
`occam2` *and Fortran77/PVM example codes for data exchange.*

There are some similarities between the `occam2` and Fortran77 codes above. The messages are tagged, either by `occam2`'s `PROTOCOL` tag (`VEC64`) or the `msgtype` variable in PVM. Information about who the sending or receiving processes are may be hidden in `occam2` (if, for instance, the channels `in` and `out` are `PLACE`d on physical links which establish the connection between processors 0 and 1), while in PVM we must specify the name of both the receiver and the sender processes and their process numbers (the variables `pname`, `to` and `from`). The `info` variable in PVM returns information about the completion status of the routine being called. The message length is implicit in the `in? occam2` statement; however in PVM it is necessary to call another routine (`rcvinfo`) to obtain this information. Of course in some applications this may not be necessary, as in our case, where each process knows the message lengths explicitly.

We now give details of the inner-product and matrix-vector product routines that were ported to Fortran77/PVM.

2.1. Inner-product. The parallel inner-product computation has three distinct phases: a *local computation* on a subset of the vector elements, producing a partial value, an *accumulation* of these partial values, and finally a *broadcast* of the inner-product value to all processors.

In the `occam2`/transputer version we used a modification, for grids of processors, of the recursive-doubling algorithm, for both the accumulation and broadcast phases of the operation (for details see [3]). On a square grid of $p = \sqrt{p} \times \sqrt{p}$ processors, this modified algorithm has a total communication step count of $2\lfloor\sqrt{p}/2\rfloor$ for each of these phases. For $p < 64$ this never requires more steps than the recursive-doubling algorithm which uses $\lceil\log_2 p\rceil$.

Due to the topology used in the Fortran77/PVM implementation, we used the standard recursive-doubling algorithm for the accumulation and broadcast phases. PVM offers a broadcast operation which works in recursive-doubling fashion; unfortunately this operation is not available in this form for the socket-based send/receive operations (in fact, it is implemented as $p-1$ send operations). For efficiency reasons, we thus had to implement the accumulation and broadcast operations using recursive-doubling. Figure 2.1 shows the pattern of accumulation for both implementations. For the broadcast phase similar patterns occur.

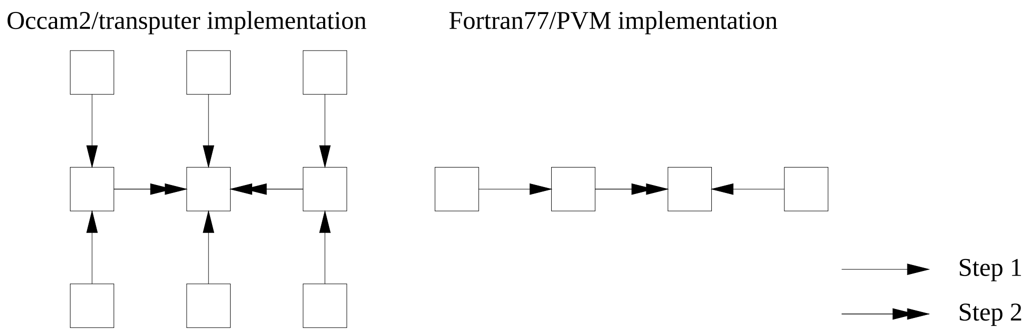
Occam2/transputer implementation
Fortran77/PVM implementation
Step 1
Step 2

FIG. 2.1. *Exchange patterns for accumulation phase of inner-product.*

2.2. Matrix-vector product. In the application of interest to us, we needed to solve systems of linear equations where the coefficient matrix is derived from the five-point finite-difference approximation of a partial differential equation (PDE).

The PDE is approximated by placing a square grid consisting of $l+1$ rows and columns over the region of interest giving l^2 internal grid points. At each point, the approximate value of the PDE is dependent on, at most, four neighbouring points in the vertical and horizontal directions.

This approximation provides us with a natural parallel implementation on a grid of processors, as used in the `occam2`/transputer implementation, where the exchange of data in the vertical and horizontal directions matches the interaction between the elements of the approximation grid. We may distribute the data among the processors using *geometric partitioning by blocks*: each processor holds $\lceil l/\sqrt{p}\rceil \times \lceil l/\sqrt{p}\rceil$ grid points and exchanges $\lceil l/\sqrt{p}\rceil$ grid points with 2, 3 or 4 nearest neighbours, depending on its position relative to the boundary of the grid.

In the Fortran77/PVM version, we can use another form of geometric partitioning, better suited to the linear array of workstations. We partition the data in vertical *panels*, where each process will hold $l \times \lceil l/p\rceil$ grid points. The processes then need to exchange l grid points with either 1 or 2 processes on either side. Figure 3.1 shows the data partitioning and exchange patterns for both versions, the black squares representing internal grid points and the white squares and rectangles indicating transputers and workstations, respectively.

3. Results. In this section we present some experimental results showing the scalability of the implementations. The `occam2` version was tested on a Meiko Computing Surface, using square grids of up to 25 T800-20 transputers each with 4Mbyte of memory. The tests with the Fortran77/PVM implementation were made on a network of 8 Sun Sparc2 workstations each with 32Mbyte of memory and a 64kbyte cache memory. The workstations were connected via an Ethernet. We stress the fact that the tests were made while other applications were using both the workstations and the network. The arithmetic operations were executed in double-precision in both versions.

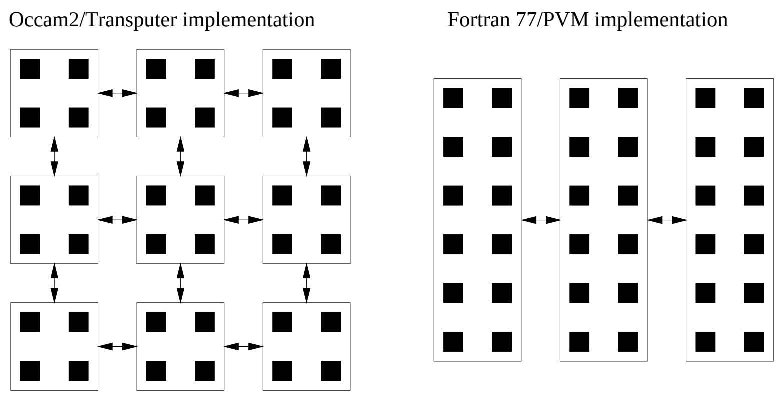
Occam2/Transputer implementation
Fortran 77/PVM implementation

FIG. 3.1. *Data partitioning and exchange patterns for matrix-vector product.*

The tests consisted of running the implementations for different values of n and p. The results showed that the Fortran77/PVM implementation was at least twice as fast as the `occam2`/transputer version. However, the scalability of the latter was higher, as shown in the speed-up graphs (Figure 3.2). The speed-ups are computed for each version with respect to a sequential implementation of the operations, executed on a single T800 or Sun Sparc2.

With respect to inner-products, we note that the `occam2`/transputer version achieves good speed-ups for smaller values of n than the Fortran77/PVM version and the same is true for the matrix-vector products. We note that the Fortran77/PVM implementations produce a superlinear effect for large n; we have traced this to the increased amount of cache memory available on the workstations.

The smaller speed-ups achieved by the Fortran77/PVM version are due to higher latency times in the Ethernet network compared to the transputer links. The transfer time is dependent on the communications hardware and on the software processes which are responsible for sending and receiving the data, and making it available to the application. For example, transferring 100 double-precision words over the transputer links takes 11ms in our implementation giving a rate of 5.58Mbit/s, while on the Ethernet network it requires 15ms (4.31Mbit/s).

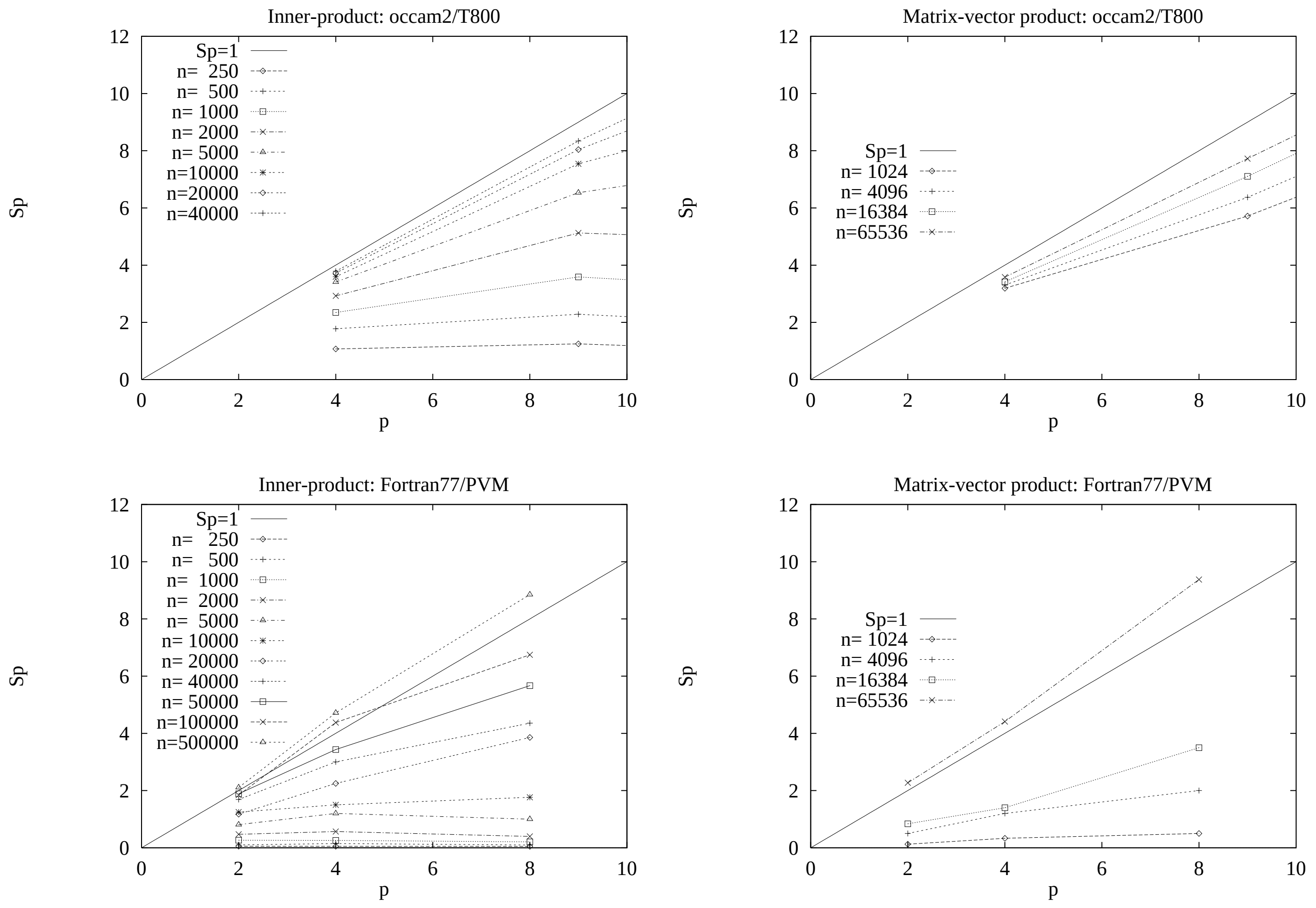

FIG. 3.2. *Speed-ups for the LAS implementations.*

4. Using the LAS as building blocks. We consider as an example one of the most widely used iterative method for solving systems of linear equations, the Preconditioned Conjugate-Gradients (PCG). Its implementation requires three LAS; saxpys, inner-products (and vector 2-norms) and matrix-vector products. We have shown in [3] and [2] that we may obtain an efficient parallel implementation of PCG (and other iterative methods) by using parallel implementations of the LAS, and that the overall efficiency of the method is closely linked to the efficiency of the LAS implementations.

In Figure 5.1 we show the speed-ups of both `occam2`/transputer and Fortran77/PVM implementations of PCG. This particular implementation of PCG uses polynomial preconditioning, where the preconditioning matrix is expressed as a sequence of saxpys and matrix-vector products, implemented using parallel versions of the LAS. Note that the scalability of both versions is similar to that of the underlying LAS operations. The Fortran77/PVM version no longer exhibits the superlinear effect shown in Figure 3.2 for the matrix-vector product. We attribute this to the increased communication costs imposed by the need to perform two inner-products and one vector 2-norm per iteration.

5. Conclusion. In this paper we have shown that we may successfully port linear algebra software developed in `occam2` on a network of transputers to Fortran77/PVM on a cluster of workstations. The use of `occam2` on the transputer allowed us to take advantage of the excellent development environments to produce efficient parallel implementations.

Although differences in the topology used, dictated by the availability of communication resources in the hardware, caused modifications to the algorithms used to implement those subroutines, the translation of the code was reasonably straightforward. The computational part of the code posed no problems whilst the interprocess communication code required slightly more work due to the different ways in which the exchange of messages are treated between `occam2` and PVM.

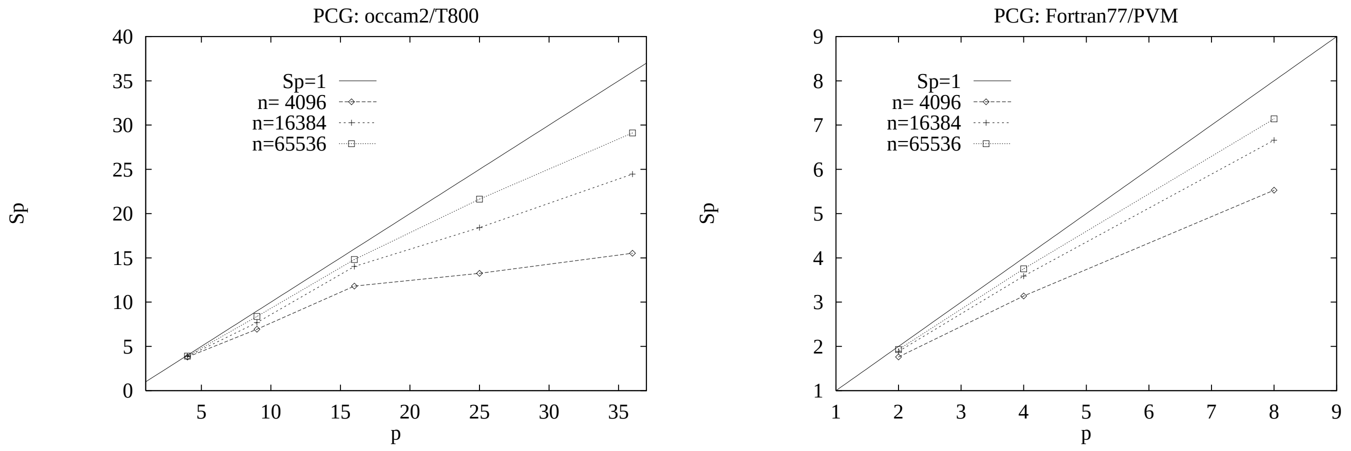

FIG. 5.1. *Speed-ups for the PCG implementations.*

The experimental results showed that for the hardware/software used in the tests, the Fortran77/PVM implementations preserved much of the parallel efficiency of their `occam2` counterparts. Although the Fortran77/PVM version did execute slightly faster due to the more powerful processor, its scalability was not as good as the `occam2`/transputer implementation.

We believe that with the current developments in hardware and software, better results both in terms of execution time and scalability can be achieved in the two environments. The use of the T9000 will certainly improve the execution times of the `occam2` version, while clusters of workstations, interconnected through FDDI or other technologies, together with improved message-passing systems, will provide a better scalability for the Fortran77/PVM implementation. In the future, we intend to extend our results to cover the T9000-based machines and the use of version 3.0 of PVM, which is expected to reduce the communication overheads.

REFERENCES

[1] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam, *A user's guide to PVM – Parallel Virtual Machine*, Research Report ORNL/TM-11826, Oak Ridge National Laboratory, 1992.

[2] R. da Cunha and T. Hopkins, *The Parallel Solution of Partial Differential Equations on Transputer Networks*, Transputing for Numerical and Neural Network Applications, IOS Press, Amsterdam, 1992, pp. 96–109. Also as Internal Report No. 17-92, Computing Laboratory, University of Kent at Canterbury, U.K.

[3] ———, *The Parallel Solution of Systems of Linear Equations using Iterative Methods on Transputer Networks*, Transputing for Numerical and Neural Network Applications, IOS Press, Amsterdam, 1992, pp. 1–13. Also as Internal Report No. 16-92, Computing Laboratory, University of Kent at Canterbury, U.K.

[4] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson, *An extended set of FORTRAN Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, 14 (1988), pp. 1–17.