Reuse, Portability and Parallel Libraries

Lyndon J. Clarke, Robert A. Fletcher[1], Shari M. Trewin, R. Alasdair A. Bruce, A. Gordon Smith and Simon R. Chapple.

Edinburgh Parallel Computing Centre, The University of Edinburgh, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, United Kingdom.

Abstract

Parallel programs are typically written in an explicitly parallel fashion using either message passing or shared memory primitives. Message passing is attractive for performance and portability since shared memory machines can efficiently execute message passing programs, however message passing machines cannot in general effectively execute shared memory programs. In order to write a parallel program using message passing, the programmer is often obliged to develop a significant amount of code which manages distributed data and events and parallel input/output, and such code may have little or nothing to do with the application. However many parallel applications have common structural elements and much of this additional code can be encapsulated within a parallel library and reused in several programs. We discuss the requirements the library writer and user makes of the basic message passing interface and describe how we have addressed these requirements in our Common High-Level Interface for Message Passing (CHIMP) project. We also describe how these requirements are supported in the new standard Message Passing Interface (MPI). We then describe a selection of the parallel libraries which we have written in our Parallel Utility Library (PUL) project. These libraries encapsulate common approaches to parallel data and event management and parallel input/output.

Introduction

Parallel programs are typically written in an explicitly parallel fashion using either message passing or shared memory primitives for process coordination. Message passing is attractive for portability and performance since shared memory architectures can efficiently execute message passing programs whereas the reverse is not generally the case. In order to write parallel programs using message passing primitives the programmer is often obliged to develop a significant amount of software which manages distributed data and events and input/output of distributed data structures. Many parallel applications have similar structural components and much of the additional support software is potentially reusable. Such common

[1]Contact Email: bobf@epcc.edinburgh.ac.uk

software can be encapsulated in parallel utility libraries for reuse in many applications. The utilisation of parallel libraries reduces the additional effort required to exploit parallel systems to an understanding of the parallel concepts and relieves the programmer from the implementation of detailed message passing. Furthermore, encapsulation and reuse affords the opportunity to increase the time and effort involved in optimisations, since that cost can be spread over many applications. We describe a suite of libraries which we have written in the Parallel Utilities Library (PUL) project which support parallel input/output and parallel data and event management.

In order to realise the benefits of software reuse through utility libraries, such libraries must be portable across parallel computing platforms. Thus we require a uniform set of message passing primitives which support safe and efficient library implementation and utilisation. The message passing interfaces provided by vendors of parallel computing platforms differ, which has undermined the potential portability of message passing software. There have been a number of efforts to provide portable message passing libraries which have gone some way to solving this problem but have not addressed the requirements of library writers and users. We discuss these requirements and describe how we have provided support in our Common High-Level Interface for Message Passing (CHIMP) project. We also describe how these requirements are supported in the new standard Message Passing Interface (MPI).

Parallel Libraries and Message Passing

This section discusses facilities that parallel libraries require from the base message passing interface. The support for these requirements in two message passing interfaces is described: our Common High-level Interface to Message Passing (CHIMP) [14, 3]; and the new standard Message Passing Interface (MPI) [11].

Parallel Library Requirements

In order to provide generality, it is important to allow use of parallel libraries in conjunction with other communicating parallel libraries and with application message passing. The communications required by a library must be isolated from other communications in order to prevent interference. This can be achieved by assigning one or more different message *contexts* to each independently communicating agent within the same process. The defining property of a context is that a message cannot be received in a context other than that within which it was sent. Most existing message passing interfaces provide only a single global context. The message tag mechanism [17, 10, 2] alone is not sufficient as the user is entirely responsible for the interpretation of tag values [18].

Many useful parallel procedures involve collective communication within a group of processes. In general, users will apply these operations to arbitrary groups of processes, requiring an ability to form process groups in a general way. Most existing message passing interfaces provide collective operations that are applicable only to all processes in the system. This is unsatisfactory as it prevents uninterested processes from opting out of collective operations to perform other work. Process grouping can also be used to distinguish processes that have different functions. Some library implementations involve interaction with processes dedicated to providing certain services. An example is a client-server model where it is useful to group server processes separately from clients.

Portability of parallel libraries is enhanced by a logical process addressing scheme. For clarity, a library should be able to expose the structure of the algorithm or paradigm encoded within it in terms of the logical process names it uses (for example, a grid based algorithm addressing its constituent processes in a grid like fashion). Specification of the *virtual topology* of processes is also necessary if an implementation intends to map an algorithm onto hardware in an efficient manner.

CHIMP Parallel Library Support

In the CHIMP interface all communications are addressed through Service Access Points (SAPs). The CHIMP "send" primitives send a message through a SAP at the sender process to a SAP at the receiver process. The CHIMP receive primitives receive a message through a SAP at the receiver process from a SAP at a sender process. Messages may only be received through the SAP to which they were sent, which provides the required message context.

CHIMP service access points are configured into SAP groups such that each SAP is a member of exactly one group and each process may have SAPs in any number of such groups, although no process may have more than one SAP in the same group. A SAP group thus defines a group of processes, comprising the processes that own each constituent SAP.

Groups are identified by a textual name and manipulated by a group handle. Individual members of a group are identified by a membership identifier which is usually a number between zero and the number of members of the group. This provides a primitive virtual topology which is a simple one dimensional array.

Parallel libraries can create private communication contexts by creating private SAP groups. In the short example shown in Figure 1 a library initialisation call accepts a group handle which it duplicates, creating a new SAP group with a new context, and stores it in the library object for private use in further library calls. The initialisation is called by all processes in the identified process group. This call would also initialise any additional library state retained in the library object. The private message context and associated process group can be used in subsequent library calls without interference from other communications in progress. The example also shows how a group wide operation (in this case a cyclic shift of data across the group members) might be implemented.

EM [23] is a parallel utility library implemented using CHIMP message passing. It provides collective communication routines, applicable across processes

```
typedef struct {
int group;
...             /* library data */
} LIB_OBJECT;

LIB_OBJECT *lib_init(int group)
{
  LIB_OBJECT *lib = lib_objInit();

  lib->group = chp_join_duplication(group, NULL);
  return (lib)
}

void lib_shift(LIB_OBJECT *lib, void *in, void *out, int count)
{
  int nb;
  int size = chp_group_size(lib->group);
  int rank = chp_group_member(lib->group);

  nb = chp_recv_nb(lib->group, lib->group, (rank+size-1)%size, in, count);
  chp_send(lib->group, lib->group, (rank+1)%size, out, count);
  chp_test(&nb, 1, CHP_ALL, NULL);
}
```

Figure 1: Simple collective library example with CHIMP

in a CHIMP SAP group, to complement the facilities provided by CHIMP. The functionality of EM includes: a barrier operation; one-to-one, one-to-all and all-to-all message exchange such as concatenation and permutation; and global summation, minimum, maximum and logical operations. Although EM provides separate functions for the base data types, generic combination and selection services are also provided, permitting the user to define binary operations on arbitrary data types. EM demonstrates a trade-off between code reuse and performance in attempting to optimise communication patterns for particular target architectures. There are two alternative implementations, based on binary trees or parallel prefixes, which can be selected by the user, depending on the platform used. A more attractive solution would involve EM querying the platform in order to decide which implementation to adopt. This might involve lower level system software making machine parameters available to modules, or having modules execute test codes in order to determine these parameters at first hand. However, it is not clear how this querying could be achieved in a platform-independent manner or, indeed, what would constitute an adequate machine-independent set of parameters.

MPI Parallel Library Support

The support for parallel libraries in MPI is broadly similar to that in CHIMP. In MPI messages are addressed through a *communicator* object which can be viewed as a bundle of a message context and a process group, providing a safe communication space for library writer and user alike. The communicator may be

```
static int LIB_key = MPI_NULL_KEY;

typedef struct {
MPI_Comm comm;
...             /* library data */
} LIB_OBJECT;

void lib_shift(MPI_Comm comm, void *in, void *out, int count)
{
  LIB_OBJECT *lib;
  MPI_request req;
  int found, size, rank;

  if (LIB_key == MPI_NULL_KEY)
    MPI_Attr_get_key(&LIB_key, lib_cpObj, lib_rmObj);

  MPI_Attr_get_value(user, LIB_key, &lib, &found);
  if (! found) {
    lib = lib_initObj();
    MPI_Comm_dup(user, &(lib->comm));
    MPI_Attr_put_value(user, LIB_key, lib);
  }

  MPI_Comm_size(lib->comm, &size);
  MPI_Comm_rank(lib->comm, &rank);

  MPI_Irecv(in, count, type, (rank+size-1)%size, 0, lib->comm, &req);
  MPI_Send(out, count, type, (rank+1)%size, 0, lib->comm);
  MPI_Wait(&req, NULL);
}
```

Figure 2: Simple collective library example with MPI

implemented for example in terms of the port model of CHIMP or the extended tag model of Zipcode [20, 19].

The support for virtual topologies in MPI is more extensive than that in CHIMP and is broadly comparable to the process topologies support provided by PARMACS[2]. Cartesian and graph topologies are supported. These facilities enhance the basic process group and rank addressing methodology with structural information about the spatial relationship between component processes, and provide assistance with effective mapping of process group topologies onto hardware resources.

The communicator object also contains a user data cache which stores arbitrary data using a simple key mechanism. The communicator cache can be used to implement the virtual topology and collective communication layers of MPI which can be thought of as internal libraries. The communicator cache also makes MPI extensible since the library programmer can write MPI libraries with the same "look and feel" as the MPI internal libraries.

The code fragment in Figure 2 shows how the MPI interface can be used to implement the simple library example from the previous section, making use of the communicator cache mechanism. No explicit initialisation procedure is required and the communicator supplied by the caller is used to store the library state which is the reverse of the equivalent CHIMP example. In the first ever call, the library gets an attribute key value and specifies functions that the MPI caching mechanism should use to copy and to delete attributes. The attribute key is used to search the cache associated with the given communicator for an object holding library state. If there is no library object, the communication context defined by the communicator is duplicated to isolate library communications and the library object is allocated, initialised and inserted into the cache under the allocated key value. The secure communications context is used to perform the data shift in later calls with the same user level communicator.

Parallel I/O

There are two principal issues concerning I/O for MIMD machines. The first is the arbitration of access by a group of processes to a common, shared file. This is addressed by the GF (Global File) utility [4]. The second is the provision of scalable I/O performance through the distribution of a file over multiple discs. The PF (Parallel File) utility [6] adds this capability to a variation of the functionality of GF.

Multiple Processes and Global Files

GF provides structured and unstructured access by a group of processes to shared files. The I/O functionality is based on the `C stdio` library, providing formatted and unformatted I/O, and adding operations to read and write arbitrary sub-blocks of arrays.

The utility supports four Parallel File Access Modes (PFAMs) which determine the behaviour of I/O operations. The first two are *single*, where all processes in a group synchronously access the same data and *multi* where they synchronously access contiguous data. These modes are similar to their namesakes in the CUBIX operating system [16], although GF does not enforce an SPMD model. The other modes are *random* and *independent*, where processes may independently access arbitrary data using either a shared or local file pointer. Figure 3 summarises how the modes modify the semantics of the I/O operations.

Applications often require the ability to alter the PFAM whilst accessing an open file. A typical example is an application which reads common header information for each process at the beginning of the file (in *single* mode), and then reads specific data for each process from the remainder of the file (in *multi* or *random* mode).

The current implementation uses a client-server architecture, thus allowing GF to provide non-blocking reads and writes, enabling the application to control the overlap of computation with I/O.

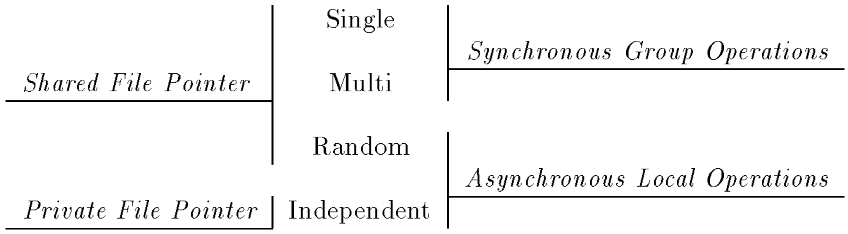

Figure 3: Comparison of GF parallel file access modes.

Multiple Discs and Distributed Files

I/O is one of the major bottlenecks that affects parallel codes. Parallel file systems represent a solution by multiplying the number of streams over which I/O is performed and thereby increasing the bandwidth. Such facilities are provided by the newest parallel computing systems (including Thinking Machines Corporation's CM-5, Meiko's Computing Surface 2 and Fujitsu's AP1000), but as is frequently the case, software support lags hardware capability.

It is the objective of the PF utility to provide a transparent, efficient and portable interface to parallel discs. The goal of transparency implies that the file should be presented as a single, logical unit to the processes accessing it. Thus, the I/O operations are unaware of the actual distribution of the parallel file. This ensures a simpler programming model, and a clean separation of the issues of functionality (access to the file) and performance-tuning (distribution of the file).

However, while PF provides the PFAMs supported by GF, we believe that for efficient parallel I/O, it is necessary to abandon the "Unix" byte stream model of GF. Instead, we allow the structure of application data to be embodied in the file access operations, so that the distribution pattern of the file may be matched to the access patterns of the application. To this end, PF uses an *I/O atom*, defined to be an addressable, user-defined record type, possibly of variable length. The name "atom" has been chosen because this is the finest unit of file distribution.

Files may be distributed by PF over discs in a variety of strategies, which are partly based on a taxonomy by Crockett [8]. A *Multi-Instance* distribution replicates the file on a set of discs as in RAID [13] Level 1. The *Partitioned-Interleaved* distribution stripes contiguous blocks of atoms across the discs, and the *Partitioned* and *Interleaved* distributions are special cases of this. *User Defined* distributions are also permitted by PF.

A single parallel file may have a number of concurrently-existing, equivalent instances with different distributions, from which the utility chooses the most appropriate on opening the file, or it may determine that the generation of a new distribution is required. In order to allow PF to determine what is an appropriate distribution, the user may provide the utility with "hints" about access patterns, such as the predominant PFAM. The "hints" mechanism also allows an appropriate caching strategy to be determined. The existence of a distributed, replicated cache is likely to be the source of a significant proportion of the performance gained [9].

Parallel Data Management

Many important classes of application, such as computational fluid dynamics (CFD) or seismic data processing, are expressed as regular local operations over a very large spatial data set. Parallelism offers a significant performance improvement to such applications, by splitting the data set between different processors and assigning processing according to the *owner computes* rule. Processes responsible for neighbouring portions of the data set must communicate to ensure that data elements on the boundaries of these portions are kept consistent. Applications amenable to this kind of parallelisation are commonly based on data structures such as rectilinear *grids*, or irregular *meshes*.

These paradigm specific requirements may be encapsulated within a library, such as the RD utility within PUL [5], which supports regular domain decomposition. This technique is appropriate for balanced grid-based problems, and involves dividing up the (multi-dimensional) array-based data set into a set of similarly sized blocks which are allocated to a logical grid of processes.

RD introduces the concept of an *operator stencil* to define the pattern of data access for updating the data elements. The operator stencil is used by RD to calculate the inter-process communications for internal boundary updates and may be changed over the lifetime of the grid.

Two interoperable levels of interface are provided. Under the *skeletal* level both the control and data-flow of the paradigm are managed automatically by RD, and the application need only provide a function to update an arbitrary region of the data set. This strategy owes much to Cole's "algorithmic skeletons" [7], and can allow very rapid parallelisation. With the *procedural* interface, boundary updates are still performed by the utility, but under the control of the application. Boundary updates may be non-blocking, allowing computation to be overlapped with the communication.

The class of applications which is probably the largest current consumer of supercomputing cycles is that of irregular mesh-based problems, such as finite element or volume codes, often used in CFD. These typically suffer from load imbalance [15], which becomes more severe when adaptive meshes (those which change shape as the computation proceeds) are implemented. Even meshes with a static structure may still be dynamically unbalanced.

The SM utility in PUL [24] supports two or three dimensional meshes. The utility includes procedures to ensure the consistency of data lying at process boundaries and also supports the migration of data elements between processes, in order to obtain optimal load balancing. Any dynamic load balancing technique must be cost effective, that is, the cost of making changes in processor responsibility and migrating data between processes must be greatly outweighed by the efficiency gained from the better load balance. Accordingly, the load balancing algorithms employed by SM use local decision-making and element migration. This approach is scalable, and although decisions are based on partial knowledge it has been shown to yield good results at low costs [1].

SM provides two load balancing functions in order to achieve cost-effectiveness under different conditions. The first migrates all mesh elements on a boundary, which is very inexpensive to execute, as it can use existing boundary update communications. The second migrates only a certain number of the "highest energy" elements, which is more costly to execute, but provides more accurate load balancing.

One alternative to the PUL approach of providing libraries explicitly supporting message-passing parallel programming paradigms is the approach taken in the High Performance Fortran (HPF) initiative [12]. In this case the user annotates almost serial source code with directives specifying how the data is to be distributed. The compiler generates a parallel executable using the owner computes rule, inserting parallel data management code as appropriate. This approach is excellent for the class of regular, static balanced problems tackled by RD. For unbalanced mesh problems, however, HPF is not powerful enough to provide the dynamic load balancing required to achieve efficient utilisation of a parallel machine. Run-time utility libraries such as PUL are flexible enough to tackle these difficult problems, while hiding the details of the parallel implementation from the application programmer.

Parallel Event Management

This approach is suitable for applications whose task can be split into two or more smaller, independent sub-tasks, the results of which are combined to solve the original problem. The sub-tasks may themselves be further subdivided, or solved directly. An advantage of over the data parallel approach is that unbalanced problems (which divide into tasks of varying degrees of difficulty) are easily load balanced.

This kind of *event parallelism* is typically found in ray tracing and tree search problems. Event parallel problems are often tackled by some form of *task farm*. In the classical task farm there is one source process, which generates tasks, a set of worker processes which transform tasks into results, and one sink process which collates results to form the final solution.

Within the PUL library, the TF utility [25] embodies a classical task farm. As with RD, two interoperable levels of interface as are provided. The higher, *skeletal* level encapsulates both the data and control flow of the application, where the user need only specify the source, worker and sink functions, and the *procedural* level provides facilities to distribute and collect tasks and results.

The basic task farming approach is inherently non-scalable, as task creation and result collation become bottlenecks with large numbers of workers. More powerful is the "divide-and-conquer" strategy, where the task is recursively broken down into a tree of sub-tasks, whose branches may be solved independently. Since the depth and breadth of the tree is unknown, the allocation of branches should be dynamic in order to balance the computational load, each process taking on work from a heavy branch when it has finished all local branches. A PUL utility to support this strategy would manage the computation tree, and be responsible for work migration.

Conclusion

In order to promote portable parallel software, there is a need for a common message passing interface. Portability need not be achieved at the expense of performance, if there is some form of communication of attributes between platform and messaging system. Building libraries on top of a messaging system requires it to have secure communication contexts (as in CHIMP, MPI and Zipcode), and is aided by allowing user data to be associated with communication contexts (as in MPI).

Effective use of I/O on MIMD architectures requires the co-ordination of multi-process access to a common file. The provision of scalable, but usable I/O performance necessitates providing a global view of a file distributed in an application-specific form over multiple discs. The PUL GF and PF utilities support these goals.

Many important parallel applications comprise regular, local operations on large spatial data sets, typically rectilinear grids or unstructured meshes. Effective parallelisation strategies for these may be encapsulated in libraries, which can accommodate irregular datasets and dynamic load-balancing more effectively than a compiler-based approach. The PUL RD and SM utilities encapsulate decompositions of rectilinear grids and unstructured meshes.

For applications whose task can be partitioned into independent sub-tasks, then an event-parallel paradigm can be suitable. This can take the form of a classical task farm (realised in the TF utility), or for amenable problems, a divide and conquer approach can be used.

The feasibility of employing a portable, library-based approach in large-scale applications is demonstrated by the use of CHIMP and PUL in industrial and academic applications such as global weather modelling, seismic image processing [22], network-management and a parallel genetic algorithm framework [21].

Acknowledgements

Lyndon J. Clarke acknowledges support for this work from the Science and Engineering Research Council. Shari M. Trewin acknowledges support from Software Engineering for Parallel Computers programme of the Information Systems Committee.

References

[1] A. Barah and A. Shiloh. A distributed load balancing policy for a multicomputer. *Software: Practice and Experience*, 15(9):901–913, 1985.

[2] Luc Bomans and Rolf Hempel. The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2. *Parallel Computing*, 15:119–132, 1990.

[3] R. Alasdair A. Bruce, James G. Mills, and A. Gordon Smith. Chimp version 2.0 interface. Technical Report EPCC-KTP-CHIMP-V2-IFACE 1.7, Edinburgh Parallel Computing Centre, University of Edinburgh, February 1994.

[4] Simon R Chapple. PUL-GF Prototype User Guide. Technical Report EPCC-KTP-PUL-GF-PROT-UG, Edinburgh Parallel Computing Centre, University of Edinburgh, 1992.

[5] Simon R Chapple. PUL-RD prototype user guide. Technical Report EPCC-KTP-PUL-RD-PROT-UG, Edinburgh Parallel Computing Centre, University of Edinburgh, 1992.

[6] Simon R. Chapple and Robert A. Fletcher. PUL-PF prototype functional specification. Technical Report EPCC-KTP-PUL-PF-PROT-FS, Edinburgh Parallel Computing Centre, University of Edinburgh, 1993.

[7] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[8] Thomas W. Crockett. File Concepts for Parallel I/O. ICASE Interim Report 7, Institute for Computer Applications in Science and Engineering, 1989.

[9] David Kotz and Carla Schlatter Ellis. Caching and Writeback Policies in Parallel File Systems. *Journal of Parallel and Distributed Computing*, 17, 1993.

[10] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated PVM framework supports heterogeneous network computing. *Computers in Physics*, 7(2):166–175, April 1993.

[11] Message Passing Interface Forum. Document for a standard message-passing interface. Technical Report CS-93-214, University of Tennessee, November 1993. Available on **netlib**.

[12] High Performance Fortran Forum. Draft High Performance Fortran Language Specification. Technical report, Center for Research on Parallel Computation, Rice University, Houston, TX, U.S.A., November 1992.

[13] Randy Katz and Garth Gibson. Case for redundant arrays of inexpensive disks. University of California at Berkely, 1987.

[14] James G. Mills, Lyndon J. Clarke, and Arthur S. Trew. Chimp concepts. Technical Report EPCC-KTP-CHIMP-CONC, Edinburgh Parallel Computing Centre, University of Edinburgh, April 1991.

[15] D. M. Nicol and J. H. Saltz. Dynamic remapping of parallel computations with varying resource demands. *IEEE Transactions on Computers*, 37(9):1073–1087, 1988.

[16] ParaSoft Corporation, 27415 Trabuco Circle, Mission Viejo, CA 92692. *Cubix*, 1988. Release 1.0.

[17] Paul Pierce. The NX/2 operating system. In *Third Conference on Hypercube Concurrent Computers and Applications*, 1988.

[18] A. Skjellum, N. Doss, and P. Bangalore. Writing libraries in MPI. Computer Science Department and NSF Engineering Research Center for Computational Field Simulation, Mississipi State University. Available on **netlib**, 1993.

[19] A. Skjellum and A. Leung. Zipcode: a portable multicomputer communications library atop the reactive kernel. In D. W. Walker and Q. F. Stout, editors, *Fifth Distributed Memory Concurrent Computing Conference*, pages 767–777. IEEE Press, 1990.

[20] A. Skjellum, S. Smith, C. Still, A. Leung, and M. Morari. The Zipcode message passing system. Technical report, Lawrence Livermore National Laboratory, September 1992.

[21] Patrick D. Surry and Nicholas J. Radcliffe. RPL2: A language and parallel framework for evolutionary computing. Technical Report EPCC-TR94-10, Edinburgh Parallel Computing Centre, 1994.

[22] C. Thornborrow, A. Wilson, and C. Faigle. Developing modular application builders to exploit MIMD parallel resources. In G. Neilson and D. Bergeron, editors, *Proceedings Visualization '93*, pages 134–141. IEEE Press, 1993.

[23] Shari Trewin. PUL-EM prototype user guide. Technical Report EPCC-KTP-PUL-EM-PROT-UG, Edinburgh Parallel Computing Centre, University of Edinburgh, 1992.

[24] Shari Trewin. PUL-SM prototype user guide. Technical Report EPCC-KTP-PUL-SM-PROT-FS, Edinburgh Parallel Computing Centre, University of Edinburgh, 1992.

[25] Shari Trewin. PUL-TF prototype user guide. Technical Report EPCC-KTP-PUL-TF-PROT-UG, Edinburgh Parallel Computing Centre, University of Edinburgh, 1992.