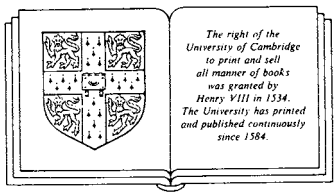
Distributed Ada: developments and experiences

Proceedings of the Distributed Ada '89 Symposium
University of Southampton, 11-12 December 1989

Edited by
JUDY M BISHOP

Department of Electronics and Computer Science
University of Southampton



CAMBRIDGE UNIVERSITY PRESS

Cambridge
New York Port Chester Melbourne Sydney

Published by the Press Syndicate of the University of Cambridge
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
40 West 20th Street, New York, NY 10011, USA
10 Stamford Road, Oakleigh, Melbourne 3166, Australia

First published 1990

Printed in Great Britain at the University Press, Cambridge

Library of Congress Cataloguing in Publication data available

British Library cataloguing in publication data available

ISBN 0 521 39251 9

Contents

Preface vii

Distributed Ada – the Issues 1
Judy M Bishop and Michael J Hasling

Distributed Ada – a Case Study 15
Richard A Volz, Padmanabhan Krishnan,and Ronald Theriault

Parallel Ada for Symmetrical Multiprocessors 58
V F Rich

The York Distributed Ada Project 67
A D Hutcheon and A J Wellings

From DIADEM to DRAGOON 105
Colin Atkinson and Andrea Di Maio

Honeywell Distributed Ada – Approach 137
Rakesh Jha and Greg Eisenhauer

Honeywell Distributed Ada – Implementation 158
Greg Eisenhauer and Rakesh Jha

Ada for Tightly Coupled Systems 177
Lawrence Collingbourne, Andrew Cholerton and Tim Bolderston

A Pragmatic Approach to Distributed Ada for Transputers 200
Brian Dobbing and Ian Caldwell

Distributed Ada on Shared Memory Multiprocessors 222
Robert Dewar, Susan Flynn, Edmond Schonberg and Norman Shulman

The MUMS Multiprocessor Ada Project 235
Anders Ardö and Lars Lundberg

A Portable Common Executable Environment for Ada 259
D Auty, A Burns, C W McKay, C Randall and P Rogers

Supporting Reliable Distributed Systems in Ada 9X 292
A B Gargaro, S J Goldsack, R A Volz and A J Wellings

Distributed Ada – an Introduction

JUDY M BISHOP and MICHAEL J HASLING

Department of Electronics and Computer Science,
The University, Southampton, England

ABSTRACT

Although Ada is now ten years old, there are still not firm guidelines as to how the distribution of an Ada program onto multiprocessors should be organised, specified and implemented. There is considerable effort being expended on identifying and solving problems associated with distributed Ada, and the first aim of this paper is to set out where the work is being done, and how far it has progressed to date. In addition to work of a general nature, there are now nearly ten completed distributed Ada implementations, and a second aim of the paper is to compare these briefly, using a method developed as part of the Stadium project at the University of Southampton. Much of Southampton's motivation for getting involved in distributed Ada has been the interest from the strong concurrent computing group, which has for several years taken a lead in parallel applications on transputers. The paper concludes with a classification of parallel programs and a description of how the trends in distributed Ada will affect users in the different groups.

1 COLLECTIVE WORK ON DISTRIBUTED ADA

The major forums where work on distributed Ada is progressing are Ada UK's International Real-Time Issues Workshop, the Ada 9X Project, SIGAda ARTEWG and AdaJUG CARTWG. Reports of these meetings appear regularly in Ada User (published quarterly by Ada UK) and Ada Letters (published bi-monthly by ACM SIGAda). The status of their activities is summarised here.

1.1 Real-Time Issues Workshop

The International Real-Time Issues Workshop has met three times since 1987, and is due to have its fourth meeting in July 1990. The Workshop is restricted to 35 participants, who are chosen on the basis of position papers. At the meeting in Nemacolin Woodlands outside Pittsburgh in July 1989, six subgroups were formed, covering:

Asynchronous Transfer of Control
Time and Delay Semantics
Communication
Compiler Support
Real-Time Ada Tasking Semantics
Virtual Nodes/Distribution.

The recommendations of the group are summarised in [Burns 1989] and most of the papers should appear in Ada Letters. One of the papers, by Burns and Wellings [1989] gives a very clear outline of the outstanding issues for real-time applications. These are divided into five areas – distribution, change management, mode changes, software reliability and hard real-time. For the issues facing distribution, a daunting diagram (Figure 1) lists some 14 problems of expression in Ada .

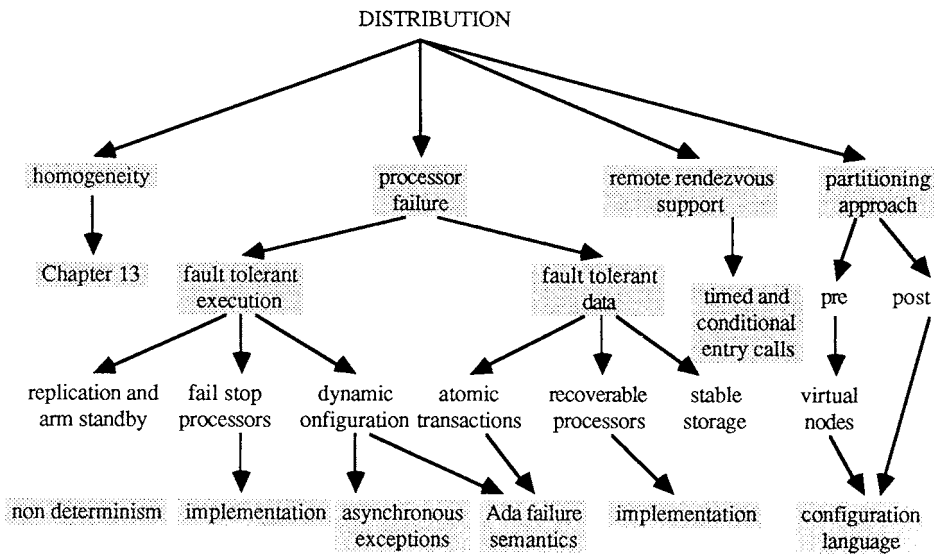


Figure 1 Distribution Issues
Shaded terms indicate problems, plain terms are potential solutions
(from Burns and Wellings [1989])

The majority of these are related to processor failure, and following the workshop, a small group took up the task of proposing language changes to facilitate the programming of fault-tolerant distributed real-time applications, with support for

partitioning and dynamic configuration/reconfiguration. The results of their deliberations are given in this issue [Gargaro *et al* 1990]. Partitioning has also received a good deal of attention from the same group, who produced a comparison of pre- and post approaches after the second Workshop [Hutcheon and Wellings 1988b].

One of the functions of the working group was to sift Ada Revision Requests, and in so doing they concluded that in a number of areas the required flexibility was best obtained by defining standard options and secondary standards, rather than by trying – and perhaps failing – to incorporate all the changes into the LRM. This topic will be taken up again in the fourth workshop, along with the need to develop codes of practice in areas not requiring specific language changes. This is a healthy change in emphasis, since up until now, the effort being devoted to revising the language has left little time for developing expertise in using it.

An important proposal of the working group was a variant of the select statement which provided neat termination features without the disadvantages of asynchronous exceptions, a topic that had been hotly debated in the past.

1.2 Ada 9X Project

The Ada 9X Project is in the Requirements phase, and as part of the development of a requirements document, a workshop was held at Destin, Florida in May 1989. The full proceedings of the workshop are available from the Ada 9X Project [1989], and a report of the Distributed Systems group (one of five) is in Taylor [1989]. Here, 13 requirements are listed, which will go forwards to the requirements team. These covered types of distributed architecture, partitioning, fault tolerance, inter-task communication, adaptive scheduling, memory management, identification of raised exceptions and threads of control, the meaning of time and multi-programming. Some of these were controversial, but there was general agreement that the language should not preclude the distribution of a single Ada across a distributed architecture. Partitioning of a single program was also supported, with specific mention of features to support pre-partitioning. In many cases, the group stated categorically that no feature should be added to the languages until it had been shown to be successful in several real time applications. The 9X project is therefore clearly going to depend very heavily on the continued research and experimentation of the community.

1.3 ARTEWG and CARTWG

SIGAda's ARTEWG (Ada Real-time Environments Working Group) has also been busy producing Ada Revision Requests, and a report of the recent meeting in Seattle in July can be found in Wellings [1989]. In the Distribution Task Force, a conceptual model of distribution has been developed, but there was concern as to

whether the model was at too high a level of abstraction. The sheer difficulty of the task ahead, a view echoed by the 9X forum [Barnes 1989], has led the group to concentrate on deriving entries in the Catalogue of Interfaces and Options (CIFO) maintained by AdaJUG's CARTWG (Common Ada Runtime Working Group). Once again, the importance of experience in individual approaches to distributed Ada was emphasised.

1.4 Summary

The groups described above serve the dual function of actually working on solutions to problems, and of directing and co-ordinating activities. It is clear that there is an enormous amount of work still to be done, and that the more experience in using as well as implementing distributed Ada systems that can be gained in advance of decisions on Ada 9X the better. There is a strong conservative tendency in the 9X movement, and proof of the efficacy of any proposals will be essential. To a considerable degree, this book sets out to provide that proof.

2. CLASSIFICATION OF EXISTING PROJECTS

Anyone wishing to embark on work with distributed Ada is faced with an array of choices. In no two projects are the parameters the same. The type of hardware, the requirements for fault tolerance and the constraints on compiler development are just some of the factors to be considered. In an earlier paper [Bishop and Hasling 1989] projects were classified according to ten factors – the updated chart is shown overleaf. We concluded that the projects differed in four important respects – input as a single or as multiple programs, the allowable units of partition, the type of communication between units and the presence of configuring information. Given these variables, it was possible to group projects in phases which balance software investment and functionality (Figure 2).

Phase	Input	Communication	Partitions	Configuring	Examples
0	Multiple	Explicit	Restricted	Explicit	Transputer Ada
1	Single	Explicit	Restricted	Explicit	York, DIADEM
2	Single	Implicit	Restricted	Explicit	Michigan, MUMS, NYU
3	Single	Implicit	Not Restricted	Explicit	Honeywell
4	Single	Implicit	Not restricted	Implicit	none yet

Figure 2 A multiphase classification of distributed Ada projects

Distributed Ada Experience Chart November 1989

Executes	Compiler	Memory	Unit	Replicable	Communication	Partitioning	Configuring	Example	Hardware
single	modified	shared (virtual distributed)	fragment = collection of named entities[1]	not yet	surrogate tasks supporting all operations	special language	special language	Honeywell Distributed Ada Project	UNIX network
single	modified	shared (virtual distributed)	every task	yes	full rendezvous on special hardware	n/a	run-time dynamic load balancing	MUMS Parallel Ada (Lund)	*n* dual NS32016s with a bus
single	modified	shared	every task	not yet	full rendezvous	n/a	automatic	NYU Ada/Ed	IBM RP3
multiple	off-shelf	distributed	virtual node = collection of library units[2]	no	client-server stubs and remote procedure calls	identifying root library unit[3]	graphical tool	York Distributed Ada (ASPECT)	M68010s with token ring network
multiple	off-shelf	distributed	virtual node = collection of library units[2]	yes	surrogate tasks and remote entry calls	special language	primitive[4]	DIADEM CEC Project	Sun work-stations with UNIX network
multiple	off-shelf	shared and distributed	library packages and library subprograms[5]	no	agents and remote procedure calls	pragmas	pragmas	Michigan University Robotics Laboratory	Two VAXes
multiple	off-shelf	distributed	programs	yes	channel i/o	n/a	via occam[6]	Alsys	Transputers

1 provided they are library units[2] or in the visible part of a library unit

2 i.e. a package declaration, subprogram declaration, generic declaration, generic instantiation or subprogram body

3 The library units forming a virtual node are assumed to be connected via with statements.

4 At present, the virtual nodes are configured "by hand" onto processors, but a tool is envisaged.

5 Originally intended to be "any object which can be created".

6 At present – work continues on a configuring language.

The projects looked at were:

- Alsys Transputer Ada – developed by Alsys Ltd (UK) [Dobbing and Caldwell 1990]
- DIADEM – CEC MAP Project undertaken by GEC Software Ltd, (UK) TXT (Italy) and Imperial College [Atkinson *et al* 1988] and now evolving into DRAGOON [Atkinson and Di Maio 1990]
- Honeywell Distributed Ada [Jha *et al* 1989, Jha and Eisenhauer 1990, Eisenhauer and Jha 1990]
- MUMS Multiprocessor Ada Project – University of Lund, Sweden [Ardö and Lundberg 1990]
- Michigan Distributed Ada Project – University of Michigan, Ann Arbor,and Texas A&M University [Volz *et al* 1990].,
- NYU Ada/Ed – New York University [Dewar *et al* 1990]
- York Distributed Ada – University of York [Hutcheon and Wellings 1990] originally the Alvey Aspect Project undertaken by Systems Designers, MARI, ICL and the Universities of York and Newcastle [Hutcheon and Wellings 1988a]

The idea of the multi-phase approach is that the progression from Phase 0 to Phase 4 represents increasing power to be bought by more sophisticated system software. The purpose of this paper is to spell out more exactly the features (and restrictions) offered by each level and the software needed to achieve them. Examples drawn from a variety of distributed applications are used to illustrate the ideas.

The philosophy behind the phased approach is to be able to distinguish between the management and technical problems more easily. Thus we would wish to be able to establish:

- **Language issues :** what can be done within the Ada mould, judiciously extended.
- **Implementation issues**: what does an implementor choose to invest software cost in.
- **Usage issues**: what does the user choose to use, given the relative efficiency and ease-of-use factors.

For example, it was noted at the Ada (UK) Conference in 1989 that many users felt they wanted to use simple message-passing between independent programs (Phase 0), in order to be sure of the efficiency of their systems with today's software. However, this does not stop an implementor investing in a more sophisticated implementation, which these users may migrate to later on.

2.1 Phase 0 – MERE

Phase 0 (Multiple Instructions, Explicit Communication, Restricted Partitions, Explicit Configuring) represents the ground level of distribution. The user is entirely aware of the hardware configuration and writes p separate programs for each of the p processors. The programs communicate by calling an independently supplied message-passing system (MPS) which may be defined at any level between that approaching the synchronous rendezvous [Dobbing and Caldwell 1990] and that resembling a run-time executive [Bamberger and van Scoy 1988].

A frequently stated disadvantage of this approach is the lack of type checking across programs, but it has been pointed out that almost complete type checking can still be obtained by deft use of the library [Dobbing and Caldwell 1990]. Common type definitions and packages can be kept in a single library, and these are then imported by each of the separate programs. It is also possible to replicate nodes in this Phase, simply by invoking the compiler several times on the same source file, provided changes are made to any static configuration information that appears therein.

The restrictions on the use of Ada are, of course, sweeping. The only communication between the code running on different processors is through the RIS. There are no procedure calls, entry calls or global variables shared between programs.

The advantages of this phase are all in the ease of getting it up: off the shelf compilers can be used and only the MPS need to be defined and implemented. This is about 6 months effort for a good Ada shop. The portability of the resulting programs will depend on the portability of the RIS and on the way in which the configuration information is presented.

2.2 Phase 1 – SERE

The move from Phase 0 to 1 is a big one in that the input is assumed to be a single Ada program. The consequence of this is that the program has to be **partitioned** in some way into n nodes to run on the p processors, where $n \geq p$. Such a partition is now usually referred to as a **virtual node** in that it defines an indivisible unit which can run on a separate processor [Hutcheon and Wellings 1988b]. The stipulation that n may be less than p allows for the possibility of several such virtual nodes co-existing on a single processor. In the simplest case, $p = 1$ and all the virtual nodes run on the same processor. In other words, the virtual nodes define the finest grain of partition that the programmer wishes to allow. If there are not sufficient processors to support this granularity, then grouping can occur. The idea is that such grouping would be done at the

configuration stage, thus maintaining a distinction between the logical and physical views of partitioning.

The issues in Phase 1 are:

- **Q1** What constitutes an allowable partition and can they be replicated?
- **Q2** What communication is permitted between partitions?
- **Q3** How is the partitioning and configuring communicated to the Ada system software?
- **Q4** How is the program translated to run on several processors?

There has been much discussion on the topic of allowable partitions. Placing no restriction puts a system into Phase 3. In Phases 1 or 2, it would seem that the choice boils down to collections of library units or task types. Because of the Ada philosophy that tasks are not library units, the two approaches are seemingly mutually incompatible. The need for task types arises from their replicability and use of the rendezvous for communication. In some systems which use collections of library units, such as DIADEM, replicability of a sort can be achieved, but not easily on a large scale, as would be required by a numerical grid problem.

With both options, there exist problems. In some of our examples, it was necessary to place parts of a matrix on each processor, not for remote data access, but mirroring a SIMD mode of operation, where each processor operates on its portion of a grid and communicates boundary values with its neighbours. With virtual nodes, such placement is difficult to achieve without convoluted programming.

It does seem as though partitioning restrictions are caused by the need to work with off-the-shelf compilers and to equate partitioning information with Ada constructs. Witness the statement in a comparison on ASPECT and DIADEM: "In applying the virtual node idea to Ada, it is necessary to associate some language construct(s) with a virtual node." [Burns and Wellings 1989]. If one allows partitioning information to be spread throughout the source or to exist as instructions in a parallel source file, then the restrictions can be formed on a what-can-be-implemented basis, rather that from a what-can-be-expressed standpoint. In essence, one can design for Phase 3 by defining a partitioning and configuration language (PCL) (Q3) and **then** decided that under Q4 above, we are not going to alter the compiler and so need to constrain what the PCL will allow for Phase 1.

2.2 Phase 2 – SIRE

According to the table in Figure 2, the move to Phase 2 (Single program, Implicit communication, Restricted partitions, Explicit configuring) is made when there is

no longer a visible message passing system that the programmer must use. Two of the projects that fall into this phase (MUMS and NYU) have adopted tasks as their units of partition and used the ordinary Ada tasking and variable access mechanisms for communication. As Dewar [1990] puts it "A multi-tasking program in Ada ... maps smoothly to a multiprocessor architecture". Two other projects described in this volume – the parallel Ada for the Multimax developed by Encore Computer [Rich 1990], and the avionics multiprocessor system developed by SD-Scicon [Collingbourne *et al* 1990] – have also adopted this view. Both MUMS and NYU have been able to adapt the compiler to suit their particular distributed architectures, and both are geared towards a shared memory model. The NYU project has uncovered some particularly difficult aspects of shared variables in Ada and proposals to address these are contained in Dewar *et al* [1990].

Michigan Ada on the other hand [Volz *et al* 1990], has gone for the virtual nodes approach described under Phase 1 and uses a more sophisticated pre-processor to detect inter-node communication and convert a single Ada program into the Phase 0 model of multiple programs communicating via an MPS.

2.3 Phase 3 – SINE

The philosophy of Phase 3 (Single Instruction, Implicit communication, No restrictions on partitions, Explicit configuring) is simple: take an existing Ada program and couple it with a description of how it should be distributed – the partitioning and configuring information. Then either put it through a preprocessor to reduce it to a Phase 0 or 1 form, and put the resulting pieces through a compiler **or** put it through a clever compiler to produce multiple load modules directly. The increase in functionality over Phase 2 is in the allowable units of partition: in essence it will be any named object The Honeywell Distributed Ada project is so far the only one that has gone this far.

In Phase 3, the problem reduces to the definition of the PCL – Partitioning and Configuration Language. A good example of such a language is Honeywell's APPL language [Jha and Eisenhauer 1990] in which Ada-like packages called fragments are defined as collections of named objects in the associated Ada program. The deficiencies of APPL which we have noticed are:

1. Parts of named objects cannot be selected e.g. a row of a matrix of a "slice" of a for loop
2. There is no load-time connection between APPL and Ada to enable configuring to take place on the basis of actual Ada variable values e.g. place task array subscript n on processor n.
3. There is no replication facility, and certainly no way of determining the number of processors at load time (on which replication may be based).

The last point is important. In a typical example, consider an odd-even sorter which is implemented with one process per element to be sorted. Two constants, n and p, define the number of elements and the number of processors respectively. If n is a very large number such as 5 000 and p is a reasonable number such as 4, then it is clear that the configuration will be based on a slice of n/p sorters to each processor. If n and p are to vary at run-time, or even at load-time, as they may well, then generating code for each processor is tricky, because the number of processes is not known by the compiler, nor is the size of the workspace.

The need for configuration languages is not confined to phase 3: all the earlier phases need some way of communicating to the binder or loader the placement of partitions on processes. In the transputer compiler discussed in Dobbing and Caldwell [1990], extensive use has to be made of low-level software written in occam, and the user has to be aware of channels, harnesses and a great deal else. Work is proceeding on a means of raising the configuration to the Ada level, and there is a strong feeling that configuration languages may well be a candidate for secondary standards.

2.4 Phase 4 (SINI)

Phase 4 (Single program, Implicit comunication, No restrictions on partitions and Implicit configuring) may well be beyond the scope of Ada programming. Nevertheless, there is work going on in other language groups (notably Fortran) on the automatic transformation of programs to distributed targets, and the Ada community should keep its ears to the ground on this one!

3. ADA FOR PARALLEL SYSTEMS

3.1 Parallel paradigms

While the main effort in distributed Ada is centered on systems composed of **different** "heavyweight" processes [Atkinson and Di Maio 1990] the focus for parallel systems is on **replicated** "lightweight" processes. Applications on transputer arrays, for example, will usually follow computational models where there are many copies of a given process, and where communication between neighbours is a regular and frequent occurrence. A classification of the broad classes of paradigms in parallel programming [Pritchard 1989] is:

- Processor Farms. A farmer processor distributes independent packets of work to a set of worker processes, which send back results.
- Geometric Arrays. The data of a geometric structure is distributed uniformly across the processor array, and acted upon by identical copies of the same process, which interact with their neighbours in regular ways.

- Algorithmic Pipes. Individual parts of a program are distributed onto a vector of processors, with each passing results onto the next.

In addition, combinations of these groups can often be found in a single program.

3.2 Transputer arrays

With parallel machines such as transputer arrays, significant speedup can be achieved with deft programming in occam. The question is: can Ada take its place in this arena as well as in the real-time embedded systems market? It is our strong belief that Ada needs transputer arrays in much the same way as transputer arrays need Ada! Ada is a large language and benefits from the power of the transputer, and the transputer, with its excellent distributed capabilities, can adapt very easily to the traditional Ada market, including defence systems.

3.3 The Stadium project

In order to substantiate this belief we have embarked on a study for the upgrade of the present Phase 0 transputer Ada compiler to Phase 2 and possibly 3 (the Stadium Project – Southampton Transputer Ada Implementation Using Multiphases). The first part of the study involves collecting suitable Ada programs representing the three paradigms above, and writing them in the form required by each Phase. From this we hope to gain a better understanding of the needs of this class of problems, derive a suitable PCL (which will certainly include replicability of nodes) and design a Phase 2 or 3 preprocessor. Ultimately, we should consider the needs of Phase 4 – after all, this where the "dusty deck" Fortran types usually start! Some preliminary results of the study follow.

3.4 Examples

The programs that we have written (or adapted from the occam) within each classification are as follows:

Farm – Mandelbrot set
Array – Laplace's equation, Odd-even sorter
Algorithmic – Garage service station simulation

The Laplace program is a good example of a parallel Ada program. The conversion of the program to each of Phases 0, 1 and 2 has proceeded fairly straightforwardly, except for the problem of initialisation: until a process is initialised, it does not know where to expect its initialisation from! The Mandelbrot and Odd-even sorter programs are in the final stages of conversion and are following similar patterns. Although there is a lot of work to be done to convert a program into several communicating programs (Phase 0), much of this is methodical and there is good reason to believe that it can be automated.

The fuel service station program, currently being adapted from a Phase 4 version, is not a performance critical program and much of the time spent is in delay statements, which represent a period of time passing while a transaction is taking place. However, the program is very dynamic in nature, with customer tasks being created and terminated continuously. If these tasks are to be distributed over a number of processors, then it will be necessary to add a large amount of extra code to enable the dynamic creation of a customer task on another processor, and message routing between processors when it is not known where a customer is placed. This program will exercise the PCL as well, in that many alternative configurations of the different tasks (pumps, cashiers, customers) can be tried out.

It is interesting that none of our farm or array examples uses the Ada task communication model in any way other than for a simple task to task rendezvous. Such communication is simple to model using a channel based communication package, as provided by Alsys Ada. However the fuel service station which uses multiple tasks calling single entries needs to be considerably restructured in order to mainatin the semantics of the original program.

CONCLUSIONS

There is still a considerable amount of work to be done on defining what we want for distributed Ada, and how to achieve it. This paper set out to provide a framework for classifying projects according to the facilities they provide and the software investment they require. On top of this, there is the provision for fault tolerance to consider, and perhaps a similar chart should be devised. In looking at the projects, it was clear that the area of scientific parallel programming had been neglected, to the extent that decisions were made that pre-empted workable solutions for this class of problems. In extending the Phase 0 transputer Ada compiler, the Stadium project hopes to ensure that this group is properly catered for.

ACKNOWLEDGEMENT

We acknowledge the financial assistance of Alsys (UK) in this work.

REFERENCES

Ada 9X Project Requirements Workshop Report, Office of the Under Secretary of Defense for Acquisition, Washington DC 20301, June 1989.

Ardö A and Lundberg L, The MUMS multiprocessor Ada project, **Distributed Ada –**

Developments and Experiences, 243–266, Cambridge University Press 1990.

Atkinson C, Moreton T and Natali A, **Ada for distributed systems**, Cambridge University Press 1988.

Atkinson C and Di Maio A, From DIADEM to DRAGOON, **Distributed Ada – Developments and Experiences**, 109–140, Cambridge University Press 1990.

Bamberger J and van Scoy R, Returning control to the user, *Ada User* **9**, Supplement, 29–34, 1988.

Barnes J, Ada (X Project Forum Report, *Ada User*, **10** (3) 132–135, 1989.

Bishop J M and Hasling M J, Distributed Ada projects: what have we learnt, *Ada User* **10** Supplement, 70–75, 1989.

Burns A, Third International Real-Time Ada Workshop, Conference Report, *Ada User* **10** (3) 141–142, 1989.

Burns A and Wellings A J, Real-time Ada: outstanding problem areas, *Ada User* **10** (3) 143–152, July 1989.

Collingbourne L, Cholerton A and Bolderston T, Ada for tightly coupled systems, **Distributed Ada – Developments and Experiences**, 183–206, Cambridge University Press 1990.

Dewar R, Flynn S, Schonberg E and Shulman N Distributed Ada on shared memory multiprocessors, **Distributed Ada – Developments and Experiences**, 229–242, Cambridge University Press 1990.

Dobbing B and Caldwell I, A pragmatic approach to distributed Ada for Transputers, **Distributed Ada – Developments and Experiences**, 207–228, Cambridge University Press 1990.

Eisenhauer G and Jha R, Honeywell Distributed Ada – Implementation, **Distributed Ada – Developments and Experiences**, 183–206, Cambridge University Press 1990.

Gargaro A B, Goldsack S J, Volz R A and Wellings A J, Supporting Reliable and Distributed Systems in Ada 9X, **Distributed Ada – Developments and Experiences**, 267–300, Cambridge University Press 1990.

Hutcheon A D and Wellings A J, Distributed embedded computer systems in Ada – an approach and experience, Proceedings of the IFIP/IFAC working conference on hardware and software for real-time process control, Warszawa, Poland, 40–50, 1988a.

Hutcheon A D and Wellings A J, The virtual node approach to designing distributed Ada programs, *Ada User* **9** Supplement, 35–42, 1988b.

Hutcheon A D and Wellings A J, The York Distributed Ada Project, **Distributed Ada – Developments and Experiences**, 71–108, Cambridge University Press 1990.

Jha R, Eisenhauer G, Kamrad J M and Cornhill D, An implementation supporting distributed execution of partitioned Ada programs, Ada Letters **IX** (1) 147–

160, January 1989.

Jha R and Eisenhauer G, Honeywell Distributed Ada – Approach, **Distributed Ada – Developments and Experiences**, 163–182, Cambridge University Press 1990.

Pritchard D J, Performance analysis and measurment on transputer arrays, Proc. of Seminar on Software for Parallel Computers, Unicom, 1989.

Rich V R, Parallel Ada for symmetrical multiprocessors, **Distributed Ada – Developments and Experiences**, 61–70, Cambridge University Press 1990.

Taylor, B, Distributed Systems in Ada 9X, Ada User **10** (3) 127–131, July 1989.

Volz R A, Krishnan P and Theriault R, Distributed Ada – a case study, **Distributed Ada – Developments and Experiences**, 17–60, Cambridge University Press 1990.

Wellings A, Workshop Report, ARTEWG Seattle, *Ada User* **10** (4) 204–207, 1989.