Three Steps to Distribution: Partitioning, Configuring and Adapting

Judy M Bishop
Computer Science Department
University of the Witwatersrand
Johannesburg 2050

1. Introduction

In the community concerned with distributed Ada, partitioning and configuring are emerging as two actions that are required when a single Ada program is to be run on several processors.

> **Partitioning** is the process of dividing the program up into units which could run on independent processors.
> **Configuring** is the process of allocating those units to the processors, not necessarily in a one-to-one fashion.

The current feeling is that the programmer will have to be aware of the partitioning process during the design of his program, but that configuring should be an independent phase, relying only on the results of the partition and the hardware available [Wellings 1987].

Our experience is that this is not the whole picture. By remaining configuration independent, the programmer may assume that each partitioned unit can interface with any other in the normal way, that is, through shared memory or via procedure or entry calls. If the target configuration consists of multi-processors **without shared memory** and with **limited physical links** between them (such as a transputer array) then the program will actually have to modified before it can run. Furthermore, we have found that a program that runs on n processors may need to be changed when more processors are used. We therefore propose an intermediate phase as follows:

> **Adapting** is the process of modifying a partitioned program to ensure that partitioned units can still communicate as required on a given hardware configuration.

All the three steps should be able to handled by tools in the future. This position paper provides further motivation for the adaptation process, with examples, and describes a tool which can adapt certain kinds of Ada programs by means of source translation.

2. The need to adapt programs

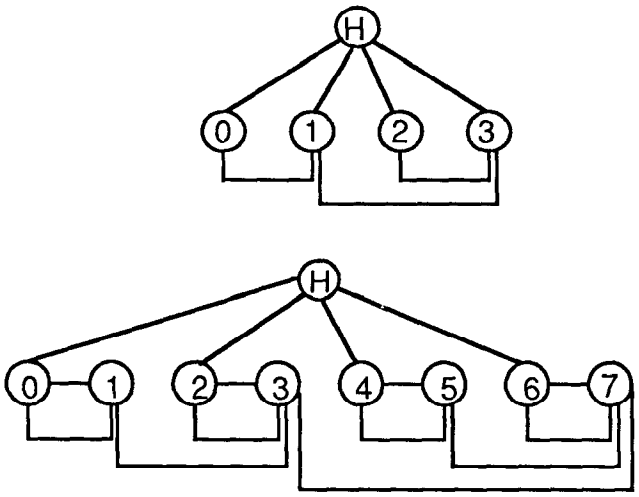
The realisation that distributing a program involves more than partitioning into virtual nodes, and configuring these onto the available processors, was a result of work done on programming transputer arrays at Southampton University. Transputers are processors that have their own on-chip memory and that communicate with each other via serial full-duplex links. Thus a program destined for a transputer array cannot make use of shared memory to communicate between units which may end up on different transputers. This is problem number one, and falls squarely into the scope of the design-cum-partitioning phase. The programmer will have to be aware of the memory structure of the target multi-processor configuration and to design the program accordingly. It may be that a tool could be devised which will take a shared memory program and convert it to a non-shared memory one, but we have not tackled this problem yet.

Assuming that a program has been partitioned into virtual nodes, and that communication between these nodes is via subprogram or entry calls, one would imagine that a Configurer would have the freedom to place the

nodes on a given set of processors in the most efficient way. This is problem number two. Processors, such as transputers, will have a fixed number of physical links. Therefore, any processor may communicate directly with only a certain number of others. If it is necessary to communicate with a further processor, then the communication has to be routed through one of those that are directly connected. In other words, routing software has to be added. The need for routing software arises for two distinct reasons:

1. **Broadcasting.** Often one node, such as a host, has to broadcast data or commands to all nodes, but is not physically connected to all of them.
2. **Overloading.** A processor may be able to communicate directly with a certain number of neighbours, but when the processor configuration is increased, it needs to communicate with more, and cannot do so directly.

As an example, consider the problem of sorting by dimensional collapse. A host node distributes values among all participating processors. Each processor sorts its sublist, and then merges it with that of its rightmost neighbour. The left neighbours become passive, and the merging process is repeated until a single sorted list emerges. If one uses 1 host processor and 4 workers, then the communication between nodes is described in Figure 1a. Given that each processor has four physical links, a partitioned version of the program could be configured in a one-to-one way with the hardware, and there is no need for adapting. If the number of worker processors is doubled to 8 then the communication should follow that of Figure 1a, but in fact the physical link limitation causes a problem with broadcasting, which is solved by the configuration in Figure 1b. The communication between the host and the 8 workers is split into four "crescents" going from H–0–1, H–2–3 etc. When the number of processors is increased to 32, then overloading occurs, since processors 15 and 31, which already have 4 links each, will have to communicate. The problem can be overcome by routing through 14 and 30.



1a. Possible configuration with 5 processors

1b. Possible configuration with 9 processors and some links replaced by indirect routing on existing links

Figure 1 Communication configurations for processors with four physical links

An approach to this problem is to assume that routing software will be resident on each processor, and that the compiler will generate code to make use of it. Disadvantages of this approach are that there may not be room for generalised routing software on each processor, and that the program which is actually running will have a synchronisation pattern which is quite different to that of the source program, and not visible or controllable by the porgrammer. Testing and debugging distributed systems is difficult enough, and such a layer of opaqueness will add considerably to the complexity. Moreover, for embedded systems, and for large scale simulations where performance is important, it is desirable that run-time software be kept simple and small [Hey *et al* 1986].

3. The Adapter approach

The approach we have taken is to recognise that the re-routing of links usually follows set patterns, depending on the initial communication configuration. Thus broadcasting can be replaced by sets of crescents, overloading by using neighbours, and so on. An initial set of such patterns has been identified and embodied in a methodology described in Bishop *et al* [1986]. This methodology has now been translated into a tool which will take an Ada source program, analyse its structure in terms of virtual nodes, add to it the necessary routing software to enable it to run on a configuration with limited links, and produce new Ada source for a distributed version of the program.

To obtain the distributed version, tasks are designated as virtual nodes, packages are encapsulated inside new tasks, forming additional virtual nodes, and communication is set up between the virtual nodes so as to keep the number of communication lines to a minimum, say two or three. Thus the communication algorithms that have been developed will be applicable to a wide range of multiprocessor system configurations.

The basis of the communication is the insertion of a controller task in every virtual node. Calls across virtual nodes are redirected to the controller task, which is then responsible for sending the information on in the form of a discriminated record (one version for every parameter set). Very little change has to be made to the original program, since entry and subprogram calls are syntactically equivalent. In fact the only change within the body of a package or task would be a RENAMES clause, to redirect calls to a controller. For example, if a package contains a call:

```
collector.sign_off (id, penetrations);
```

where the task `collector` is in a different virtual node, then the following declaration will be inserted in the package:

```
DECLARE collector : controllers RENAMES processor(id).control;
```

where `processor` is an array of records of which one field is the control task for each processor; the other fields reflect the original units of each virtual node. The methodology is described in full, with examples, in Bishop *et al* [1987].

4. Advantages of adapting by source translation

The major advantages of adapting programs by source translation are that it is possible, cheap, and immediately available. It does not rely on expensive compiler technology. Naturally, a distributed compiler is required, but by distributing the control of the communication between virtual nodes to special controller tasks within each virtual node, the demands made on the compiler are kept to a minimum. These demands are the abilities to

- generate code in a multiple address space
- handle a rendezvous between two directly connected nodes.

The Ada runtime system that will be resident at each site will not have to take care of a rendezvous with remote sites further down a line: this is all incorporated at the source level.

A further advantage is that our particular methodolgy allows both packages and tasks to be virtual nodes. Tasks are important for replication. The client-server models under consideration rely heavily on multiple instances of the same code. Arrays of tasks, or even dynamically created tasks, will be an essential component of such programs. Packages are essential for encapsulating data types, providing state machines and generally as the building blocks of large Ada systems. The Adapter methodology copes with packages in the full, including generic and library packages. An important advantage of our Adapter approach is that library and generic routines can be incorporated into the distributed system without alteration. This is vital if essentially system packages such as `text_io` or mathematical software are to be available, not to mention the advantages that accrue from re-using tried and tested packages. This point has not been addressed by most other writers [Mudge 1987, Hutcheon *et al* 1987].

5. Status of the project

The first version of the automatic Adapter has been written and tested. It consists of a front end, which scans the original concurrent program, detecting the presence of tasks and packages and their interactions. Such interactions stem from the presence of WITH and USE clauses, as well as from the normal visibility present through the order of declaration. In fact, visibility of this latter sort is the only basis for interaction between tasks, there being no task equivalent of the WITH statement.

The front end builds up a data structure representing the interaction between units of the program and also writes the actual text out to a temporary file. The back end of the Adapter then uses the data structure to drive the merge of this text with that of the skeleton control and communication routines stored on a permanent file.

Attention is now being given to extending the repetoire of patterns that the Adapter can recognise and transform, and to introducing a limited (and probably interactive) form of configuration language. At the moment, the Adapter handles programs that consist only of actors or servers, not transducers. The problem with transducer tasks – those that have entry points and entry calls – is in handling an accept via a controller task. A feasible method would be to have a synchronisation variable which becomes true when a process signals to its controller that it is ready for an accept. This variable then causes a corresponding conditional accept in the controller to become open. The difficulty is that a call for such as accept may have to circulate if no process is ready for it, and this may clog up the ring, causing deadlock. Work on further algorithms is proceeding.

Acknowledgements

This work was done in conjunction with ESPRIT Project 1085 (Reconfigurable Transputer Processor Supernode), and Craig Faasen helped with development of the Adapter program.

References

Bishop J M, Adams S R and Pritchard D J, *Distributing concurrent Ada programs by source translation,* Software – Practice and Experience, **17** (12) 859–884, 1987.

Hey A J G, Jesshope C R and Nicole D A, *High Performance simulation of lattice physics using enhanced transputer arrays*, in **Computing in High Energy Physics**, 363–369, Elsevier Science Publishers (North Holland) 1986.

Hutcheon A D, Snowden D S and Wellings A J, *Programming and debugging distributed real-time application in Ada*, First International Workshop on Real-Time Ada Issues, in SIGAda, VII (6) 64–66, May 1987.

Mudge T, *Units of distribution for distributed Ada,* First International Workshop on Real-Time Ada Issues, in SIGAda, VII (6) 64–66, May 1987.

Wellings A J, *Issues in distributed processing (Session summary),* First International Workshop on Real-Time Ada Issues, in SIGAda, VII (6) 57–60, May 1987.