Matching Machines to Programs

Dave Wilson
Consultant, Strategic Systems Group
Meiko Scientific
650 Aztec West
Bristol BS12 4SD.

Email: davol@uk.co.meiko

Tel: 0454 616171

Abstract

The problem of reconciling machine capabilities with software requirements is re-examined and the thesis advanced that the efficient exploitation of parallel machines calls for greater sophistication in the software development toolset than can be supported solely by enhancements to existing low-level tools.

The methodology which underlies Meiko CSTools is described in order to illustrate the scope and capabilities of this approach to software development

1 Introduction

Over the past few years, users of computers in the scientific and engineering communities have come to recognise that many of the systems they wish to study cannot be satisfactorily modelled on sequential machines. Furthermore, there has been a widespread willingness to look to parallel computers for a solution to this problem. The analogies with human work are clear and persuasive. If a sequential computer resembles a tireless solitary craftsman who eventually completes all the tasks allocated to him, a parallel computer is like a small factory full of people, all working at once. Nevertheless, there is no clear consensus on the best way to program such machines and users frequently find themselves uncomfortably close to the raw hardware.

Any solution to these difficulties must satisfy three criteria:

- Programming must be straightforward and direct

- Programs must be immune to changes in the underlying hardware technology.
- Programs must be capable of exploiting the hardware efficiently.

While work progresses on standardising high-level language support for both shared memory and distributed memory programming models, it seems to be concerned for the most part with matters of detail. The overall nature of program development for parallel machines is considered either too trivial or complex to be of concern and the shape of the target machine is unclear.

Much of this stems from the fact that, in attempting to arrive at an understanding of how parallel machines should be programmed, we have adopted a design framework which derives from the *special case* of the single processor. Although we cannot entirely escape this methodology it is my intention to review its essential characteristics and to show how it is best adapted to the demands of parallel programming.

The history of technology has many examples of similar transfers of technique from an existing to an emergent discipline. If the first digital computers had been parallel machines we would have developed a software technology very different from the one we are embarking on now.

Our experience is primarily with distributed memory, message-passing parallel computers and I will concentrate on this architecture. This, however, should be intrepreted as a convenient simplification and not as a denial of the applicability of these observations to the shared memory model.

2 Programs

As users of computers we write programs - usually in some high-level language, compile them and link them with appropriate libraries to create executable files. Provided we present these executables to the right hardware running the right sort of operating system, we get programs which work - or at least *start* to work.

If we consider the transformations which a program undergoes as it moves from source to execution, they can be summarised as follows:

1. Compile - constrain to instruction set.
2. Link - constrain to run-time environment.
3. Load - constrain to specific hardware.

Interestingly, while there are volumes written on compilation techniques, the design of linkers is considered to be of secondary importance and loading is almost always treated as entirely trivial and pragmatic. This is perhaps understandable. It has become customary to view linked executable files as ready to be passed directly to the execution environment. Of course, what actually happens is that the operating system layer responsible for the initiation of program execution reads the file header and discovers the size of the program code, it's stack requirement etc. It then attempts to allocate the necessary memory resources. loads the program and initiates it's execution.

Most parallel programming methodologies we have come across seem to have been designed to a brief which requires that the relative importance of the operations listed above is kept as similar as possible to that found in the sequential case. That is - that the program is fully specified in a set of source files and that after compilation and linkage an executable file is produced which contains all the necessary code and loading directives necessary to start the program running on a real parallel machine.

Our experience at Meiko has led us to reject this approach. Instead, we found our attention drawn to the loading operation as the most appropriate foundation on which to develop support for parallelism. The methodology which underpins Meiko's CSTools is one in which the loader is elevated to the status of a program in its own right - with sufficient control over the machine configuration to ensure, not only that the segmentation of each processor's address space is correctly managed but also that the necessary system services are provided where needed and that the network topology is well-matched to the communication requirements of the application.

In almost all real examples we have encountered, parallel programs are found to consist of a few basic sequential components, supported by a similar but distinct layer of system infrastructure. The actual number of times each component is replicated and the precise manner in which system services are provided are all secondary matters which are more properly associated with the abstract parallel machine and its method of implementation.

3 Machines

Conventionally we talk about *targetting* a program at a particular machine. This does not imply however that we are talking about a real physical machine. More often than not the machine consists of a set of interfaces to capabilities which it is assumed the real machine will provide. This is what is known as an *abstract* machine. Two immediate benefits follow from adopting this approach.

- Programs targetted at an abstract machine will, in general, be more readily ported onto different physical machines. This assumes, of course, that the abstraction is well-considered. The most familiar example is the Unix run-time environment.
- If the abstract machine is defined at a sufficiently high-level there is sufficient latitude for the hardware implementation to make effective use of advanced hardware technology. This makes it possible for manufacturers to compete in terms of the performance they can deliver to programs targetted at the same abstract machine.

One might summarise the concensus view of an appropriate abstract machine for parallel programs as follows.

- It allows a number of application processes to run at the same time.
- Processes can send messages to one another
- Processes might want to share memory too.

The *perfect* implementation of such a machine would provide state-of-the-art compute performance for *each* process and optimal communication bandwidth between any two processes regardless of the number of processes engaging in communication. Furthermore, the message latency between two processes would be independent of their location. And - I almost forgot - the machine must be scaleable too. In other words, all of this must remain true independent of the number of processes running in the machine.

Needless to say, such an implementation is not really feasible at present. And yet the time taken to transfer messages is so crucial to the performance of most parallel applications that a second-rate implementation of the abstract machine is likely to come a poor second to the same hardware specifically configured. This represents a difficult dilemma for many people engaged in developing real scientific applications on parallel computers.

If the target machine is transputer-based, it is hard to improve on the performance obtained by programming the application in Occam. This brings with it certain disadvantages, however. Since the target machine for Occam programs is devoid of any sort of system service layer, it is common for much of the application programming effort to be diverted into the provision of through routing and file serving utilities. Worse still, these system components must be confusingly

comingled with the application program. While it is true that this type of machine is efficient and direct in its provision of basic processing and communication resources, the relationship between program and machine is too close. As high performance processing and communication technology becomes available the Occam machine begins to look awkward and restrictive.

It appears that we are driven to adopt a somewhat inefficient, abstract machine as the target for software development. This is only true, however, if we consider the implementation of the abstract machine to be static and unchanging across all programs. There is no reason why the machine can't be configured BOTH to conform to the abstraction AND to achieve a close match with the program requirements.

CSTools is a software system designed to achieve this objective - which we term Adaptive Abstraction. In fact, though CSTools can be used to realise a number of alternative abstract machines, the term Adaptive Abstraction describes the ability to realise a specific abstract machine in a manner which is optimised for a specific piece of software.

A CStools program is developed and run in a familiar operating system environment on a sequential machine such as a workstation. As it executes it builds up a detailed model of the parallel program AND the machine on which it will execute. Once the model is fully evaluated the necessary hardware resources are committed, network connections made and the program is loaded and run on the parallel machine.

Before describing CSTools in more detail it is necessary to describe a little of the background which prompted its development.

3.1 The Computing Surface

The machine at the core of all Meiko products is the Computing Surface. A Computing Surface is a collection of processing elements. Each element is capable of operating independently but may choose to cooperate with other nodes in the system by exchanging messages. The Computing Surface is a multiple instruction multiple data (MIMD) machine, each element runs its own processes acting on their own data and accesses that of other elements by message passing. Each element has the processing power of a powerful RISC workstation (currently 15-40 MIPS, 5-40 Mflops depending upon the processor type and the application) and could be regarded as such. The intimacy of cooperation (current bandwidths are 4-8 Mbytes of data per second between each processor and its neighbours) and the scalability of communication (all of the processors in a system, whatever the size, can sustain these

data rates) ensures that large numbers of processors can be effectively employed in solving a wide range of problems.

Some elements, such as disc controllers and communications interfaces are specialised in nature, the bulk are (usually) dedicated to computation. A machine is built by combining appropriate numbers of processing elements of each type to suit the user, the application or class of applications.

Each Computing Surface element has four fundamental components, a processor, a memory system, a communications engine and a control interface. Optionally elements may have an interface to one or more I/O devices.

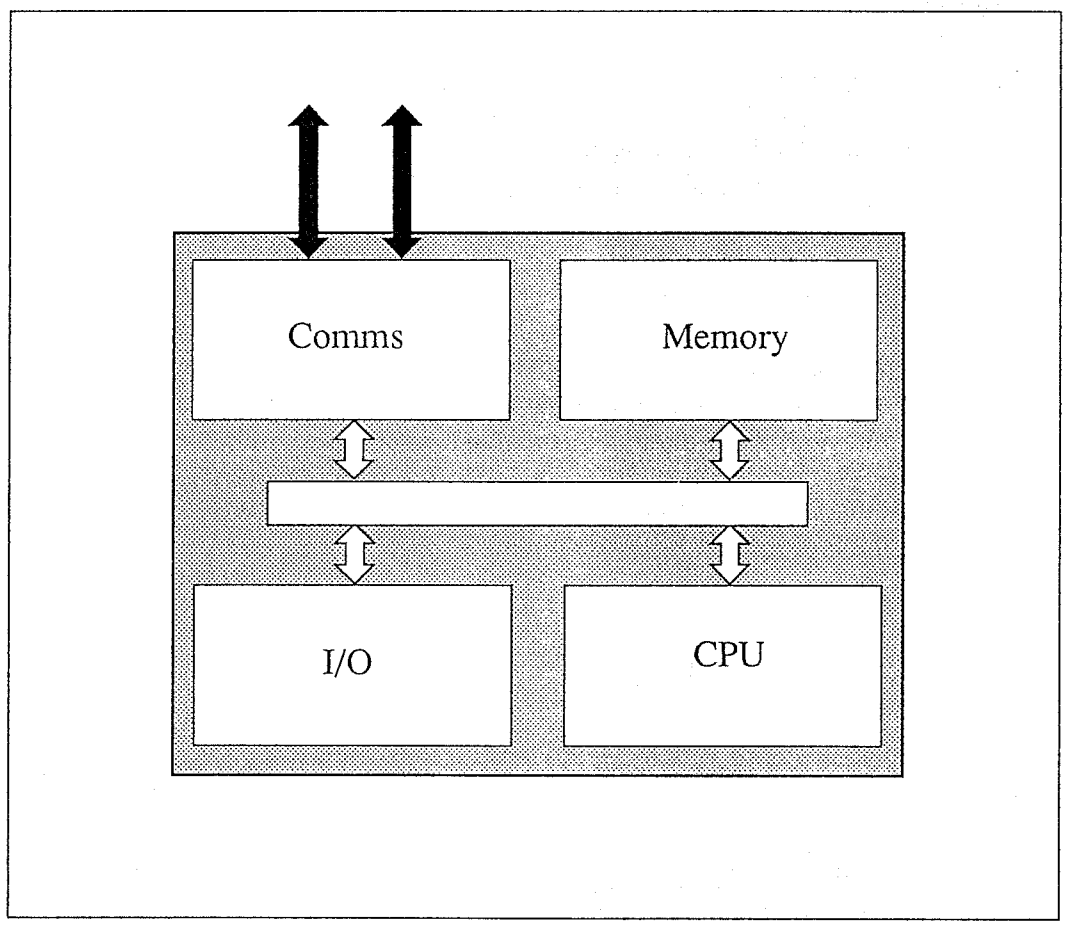

Fig. 1 Generic Processing Element

Compute elements may have Sparc, i860 or Transputer processors, each provides a different type of functionality. Sparc and i860 processors provide high performance general purpose computational capabilities with memory protection

and virtual memory. i860s provide very high levels of performance for vectorisable applications. Transputers provide excellent support for rapid context switching and are significantly cheaper, but have lower levels of general performance and no memory protection.

Elements are inter-connected via their communications engines, which are in turn connected via switches. The switches are joined by channels in the backplane, backplanes can be joined to form larger and larger systems.

Communications engines are currently made up of one or more transputers and enable elements to be connected in a wide range of topologies. General purpose arrays such as rings, trees, grids, torii and hypercubes are possible, as are application-specific topologies.

4 CSTools

CSTools evolved in response to the need to support software development within Meiko and to provide a development toolset for our customers. This represented an unusually diverse set of requirements. The major part of the work was undertaken during 1988/89.

Having decided that the software development tools should run in a standard sequential environment, we began by considering the kinds of representation which might be appropriate for modelling programs, communication channels, processors and memory spaces. Experience gained during the development of the Inmos Silicon Design System suggested that when you're not quite sure of the ground ahead of you, it is a good idea to structure the software as a general-purpose core onto which specific behavioural modules can be attached at will.

The core structure adopted to implement CSTools was an object-oriented class hierarchy in which methods are invoked via table-driven evaluation strategies. Although this sounds complicated it means that CSTools is free to evolve in two distinct senses. Firstly, new types of object can be incorporated into the repertoire of entities which CSTools manipulates. The most obvious example of this arises with the introduction of a new type of processing element. Secondly, the rules determining the interactions between objects can be modified simply by the replacement of the software module which defines them.

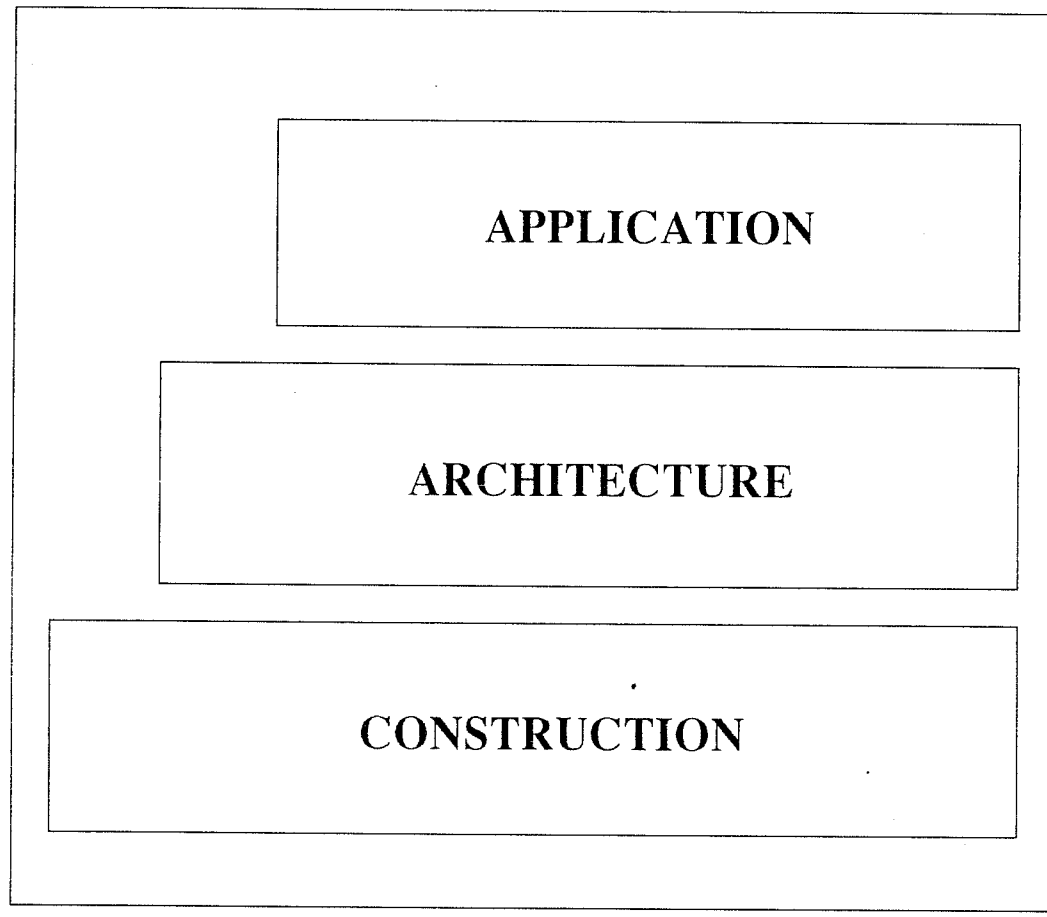

Fig. 2 CSTools - layers

The software can be viewed as built up of three layers. The boundary between the Application and Architecture layers is real. That between the Architecture and Construction layers is less distinct - but will become more clearly defined in the future.

The layered structure is important in that it enables us - at different times - to evolve new high-level interfaces on a stable foundation and to support existing high-level interfaces on new hardware.

4.1 Application Layer

This is the layer we wish to develop as the principal point of contact with Meiko machines, There will be a number of alternative, application-specific interfaces

at this level, each supporting what is considered to be the most appropriate view of the application domain. Where an established standard appears to meet this criterion, we will attempt to implement it - though it is important to recognise that this is not always possible. An obvious example of this is Fortran as a standard for scientific applications. The problem of targetting sequential Fortran, unchanged, at a massively parallel machine has not been solved. When it has, the solution will belong in the Application Layer.

To date, support for parallel applications has been provided in the form of a simple textual description - the *parfile*. In the example below, `fred`, `jane` and `bill` are the names of transputer-executable files which have been produced on the host workstation using a transputer cross-compiler. Each is essentially a straightforward, single-threaded sequential program.

```
par
processor 0 fred
processor 1 jane
processor 2 bill
endpar
```

CSTools interprets this as meaning that each of the three programs is to be run on a separate transputer and that any demand for file I/O is to satisfied by invoking requests on the host file system. In addition, if the application processes make use of message-passing procedures, the necessary network connections will be made such that there is a route between each possible sender/receiver pair. On transputers, these connections are made in such a way that the number of intermediate processors which the message must pass through is minimised for all routes. This is the default wiring strategy. It is possible to specify alternative interconnection schemes with the `networkis` keyword.

Parfiles represent a simple and direct method of describing parallel applications. They are quite adequate for applications in which the component processes are either quite small in number or are replicated over a single, regular topology. Where complex topologies are involved or the application is built up of non-identical sub-assemblies a more sophisticated programming tool is required.

We intend this to take the form of a graphical, symbolic editor which will be used to construct visual representations of applications. We believe this to be the most appropriate way to develop and fine-tune parallel programs.

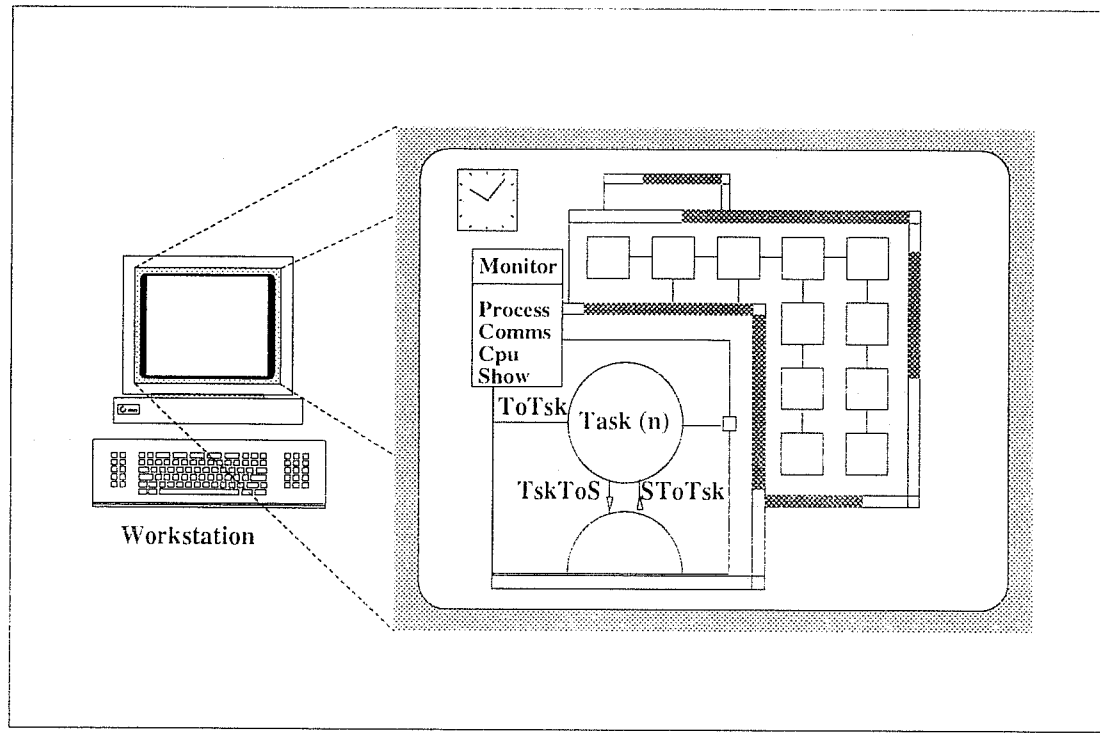

Fig 3. Visual programming interface.

Visual programming methods present several opportunities to broaden the scope of the application development toolset. These centre around the fact that a graphical representation can be used to display secondary attributes of the program such as system processes, the mapping of processes onto hardware and the interconnection topology.

Once execution has started it will be possible to observe the behaviour of specific hardware components. The development of complex parallel applications is greatly facilitated by the ability to pinpoint processing and communication bottlenecks. The monitoring of processor idle time and the density of link traffic can both be accomplished relatively easily but do not relate directly to the primary program structure. What emerges is that the need for two distinct views of the application.

The Program View

For this we require a visual language expressing processes and associated dataflows in a clear and economical way. This should be hierarchical - permitting the suppression of unecessary detail - and modular, in that it should be possible to construct applications from independently developed sub-assemblies. An important capability is to be able to specify and depict process replication over a variable range [1]. The visual representational scheme will stop at the level of sequential code

modules. Beyond this it will be possible to access the module source in separate overlaid text windows. In the program view, the principal monitoring tool is the parallel debugger, with which execution of one or more sequential threads can be studied at source level.

The Machine View
Here we are interested in processors and communication links and how application processes and dataflows have been mapped onto them. It will be possible to see how system modules have been used to provide access to standard services. In the machine view, the most appropriate monitoring tools are processor load meters and indicators of the density of link traffic.

To summarise, the visual programming tool is designed to deliver the following:

- Insight into the essentials of the real machine and how the application has been mapped onto it.
- Information on the the run-time behaviour of the application in the context of a particular mapping
- Control over succesive iterations of the program evaluation such that performance improvements can be attained.

4.2 Architecture Layer

At the Architecture Layer CSTools is primarily concerned with sequential program components which have been defined by the user and with the management of various associated attributes intended to control the machine configuration. Access to the Architecture Layer is via library procedures which are used to build a *seed* data-structure defining the parallel application. This data-structure is then evaluated and grown into a complete representation of the program which incorporates both the original user-defined components and standard building blocks used to support requirements for the various system services - access to files, inter-process communications, graphics etc.

The Architecture Layer implements the **policy** governing the provision of these services and how they are mapped onto a particular machine. Just as the Application Layer supports several alternative application interfaces, so the Architecture Layer supports alternative policies. Each corresponds to a distinct system architecture.

The current release of CSTools incorporates a documented library of procedures

known as CSBuild. This gives users direct access to the Architecture Layer. It implements a system provision policy with the following characteristics:

- Access to files is provided according to the client/server model.
- The file server runs on the host machine.
- Network communications are provided by the CSN (Computing Surface Network).
- Management of free store, and access to board-specific I/O interfaces is provided by a local client process.
- Processes running on the i860 processing element gain access to the above services via a client process running on the communications transputer.

CSBuild also includes procedures for defining the grouping of processes on specific computing elements and the physical interconnection of those elements. In this last respect, the CSBuild policy is too involved with the target machine.

To preserve a proper distinction between layers, we now realise that a more appropriate terminology is one which talks in terms of *application topologies* and the degree of *coupling* between processes. The specification of application topologies in particular, represents a more natural and direct interface to this layer than does the description of a particular hardware wiring. Whereas we currently allocate processes to processors and wire these together in a way which best serves the pattern of communication traffic *implicit* in the run-time code, it is preferable to allocate processes to nodes in an application dataflow graph and to leave it to the next layer of evaluation to decide how best to support these requirements.

At the Architecture Layer, policies are defined in terms of data-driven *strategies*. These are procedures which are invoked on encountering specific objects in the data-structure. In this respect, CSTools resembles a high-level linker.

The evaluation of a CSTools data-structure starts with the original user-defined processes. The requirements of these processes are expressed as named *imports*. A given import type will, in general, cause a specific strategy to be invoked. Some strategies will only be invoked where the import name matches that given in the strategy declaration; others are invoked on encountering any import of a given type, regardless of its name.

Imports are not simply attributes with which executables have been tagged. The majority are declared in library procedures defining access to system services and

are incorporated in executable modules as a result of the invocation of such services. Each import corresponds to a location in the process data segment which must be initialised with an appropriate value before the process executes. This mechanism is extremely versatile at binding a specific service implementation to the code which wishes to use it. For example, it is possible for a library procedure to import a reference to a structure defining a distributed file store. This enables the user process to establish the number of disk controllers on which the file store is implemented, as well as the network address of each controller. If several processes import this information, it is possible for a parallel application to access the filestore with optimal efficiency and without recourse to clumsy configuration files.

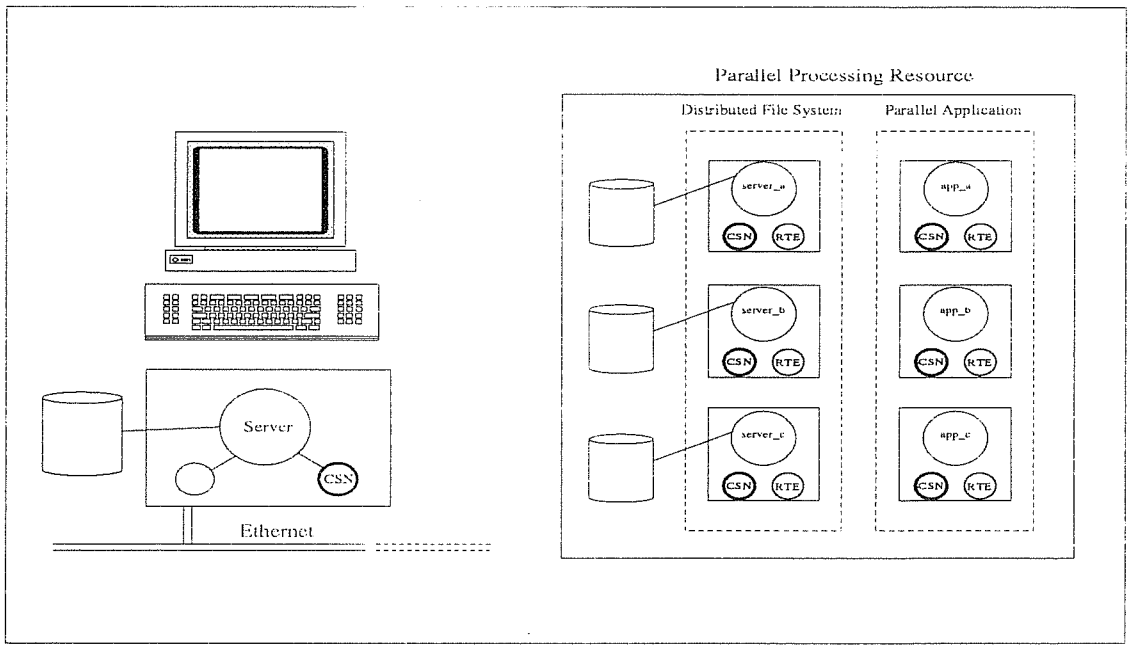

Fig. 4 Architecture Layer - distributed file store.

Inasmuch as both import and strategy declarations reference an identical set of names, the libraries used to build the sequential code modules and the policy defined in the Architecture Layer must match. Strategy declarations are only loosely bound to the core software and alternative policies are readily implemented. Each one corresponds to a particular system architecture.

What we now perceive as important is to develop a number of policies at this level which implement some of the more common structures encountered in scientific computational problems.

Task Farming
The controlling process, at some point invokes a procedure which states that a specific set of tasks is to be farmed out. The procedure encapsulates all the decisions involved in determining the number of slave processes and their network addresses.

Application topology There are many classes of problem where an identical process runs at each location in a regular topology. If this process declares an import which specifies the class of topology required, submission of the process to an appropriate CSTools utility is sufficient to generate the entire application.

4.3 Construction Layer

The Construction Layer provides the *mechanism* needed to implement the *policy* of the layer above. It defines how processes, data-structures and dataflows are realised on a specific set of resources. We do not provide a documented interface to this layer, since it is only sensibly invoked by higher-level procedures. It is important, nevertheless, to maintain the distinction between the provision of a given set of abstract machine interfaces and their realisation.

The CSBuild policy targets a machine which consists, in general of a heterogeneous mix of processors and environments. Processes which are to be run on the host or a similar workstation will find themselves supported by the host operating system. Initiating execution of these processes involves little more than the invocation of a suitable system call. Transputer processes, on the other hand must be explicitly provided with a number of physical memory segments for code, data and workspace and the location of every object in the transputer memory space must be unambiguously specified.

Before a parallel application can be loaded and run the necessary resources must be claimed and connections made to support the communication network. On the Computing Surface, these connections are set up by sending messages to specific processor elements over the supervisor bus - a global control bus which links every element. Once the connections are established part of the resulting network is used as the *boot-tree*. That is - it provides an initial route into the machine for programs and data.

5 Conclusions

CSTools targets a machine which consists in general, of a heterogeneous mix of processors and environments. This heterogeneity is central to Meiko's computing strategy. The range of choice of processor technology and the wide range of I/O interfaces now supported has allowed Meiko to configure systems to meet the requirements of applications as diverse as database management and theoretical physics. The computing surface architecture allows Meiko customers to scale their systems to suit their applications and to grow their systems over time. It has alowed customers buying our first T4 transputer-based machines to keep abreast of the latest technology and to update them first with T800 transputers and later with i860s.

However, this strategy requires that customers are spared the inconvenience of having to undertake modifications to software, every time new hardware is added to the machine. CSTools defines stable high-level interfaces which protect the customer's software investment while attaining significant improvements in performance. CSTools achieves this in what we believe to be a unique way.

Massively parallel computers are capable of delivering a staggering amount of raw computing power. The continuing development of this technology in line with Meiko's corporate goal of "achieving performance through concurrency" demands software tools that are equal to the task.

References

[1] West, A and Capon, P., 'A High level Software environment for Transputer based systems'
Proc. 12th. Occam User group, April 1990