Constructing Generic Data-Farm Templates

M. Fleury, A. C. Downton and A. F. Clark

Department of Electronic Systems Engineering, University of Essex

Wivenhoe Park, Colchester, CO4 4SQ, U.K

tel: +44 - 1206 - 872795

fax: +44 - 1206 - 872900

e-mail fleum@essex.ac.uk

Summary

The data-farm template is a software component for the Pipelined Processor Farm (PPF) software architecture. The PPF data-farm template is a flexible, semi-manual method of prototyping continuous-flow applications. The template design originates in a set of abstract design principles. The PPF template is instrumented to provide either support for logic debugging at an intermediate development level on a network of workstations (NOW) or performance tuning on a target multicomputer. General implementational problems arise. The paper demonstrates the feasibility of implementing the design on a Unix-based NOW. Sufficient detail is included to guide other implementations and to make comparison between thread packages. Descriptions of two target system implementations follow.

1 Introduction

High-level architectural patterns with design reuse in mind [1] are a recent informal feature of the object-oriented approach to software development. For example the model-view-controller and presentation-abstraction-controller [2] are two abstract patterns for organising graphical user interfaces. However, adoption of pattern-oriented software has not been widespread in

parallel computing because of an emphasis on performance and hence on bespoke solutions. This paper is concerned with the Pipelined Processor Farm (PPF) [3] pattern. A pipelined processor farm is a pipeline each stage of which can incorporate parallelism (Fig. 1).

The PPF pattern can be supported by lower-level software components, called templates in this nomenclature to avoid confusion with higher-level patterns. The template is at an equivalent level to the JavaBean or DCOM component [4] but with an informal interface, being available in source code form. A template implementation need not reproduce every feature of the design and can be built on existing software facilities, thus easing the implementational burden. Templates are used through a text editor in the manner of the Linda program builder [5]. The programmer can slot in sequential code sections, and form messages, provided the message-passing structure is preserved. Other parts of the structure are transparent to the programmer such as message buffering, and event-trace instrumentation. Template construction along with parallel configuration is intended as one part of a coherent design scheme, which is in the course of development [6].

The PPF pattern is suitable for continuous-flow embedded systems, which commonly have a time constraint imposed upon their output. A survey established the generality of the PPF architectural pattern in sample applications, e.g. [7, 8, 9, 10]. Notice that these examples, in the vision and signal-processing domain, are significantly-sized, multi-algorithm applications. Realistically, it is expected that such applications will first be written by a team of programmers in a constrained sequential environment using structured programming. A parallelizing compiler would not resolve the appropriate granularity [11] throughout the irregular algorithmic components typical of the problem domain. Interestingly, a combination of task parallelism and data parallel modes, for software engineering reasons, has been independently proposed [12]. A language combining implicit parallelism and message-passing constructs would be an alternative route to PPF pattern implementations. In this sense, the PPF pattern is a candidate but as yet unformed feature of some parallel language.

The PPF pattern specifies the decomposition of each stage's workload in various ways: by

means of temporal multiplexing; through algorithmic parallelism; and through data parallelism.[1] Farming through data parallelism enables the pipeline traversal latency to be reduced and permits incremental scaling of the farm throughput. It is therefore more flexible and popular as a design technique than the other two forms of decomposition, and hence forms the basis of an initial template design. Examples of medium-scale, data-dominated, vision and image-processing applications parallelized along PPF lines are [13, 14, 15, 16, 17, 18].

Several versions of a data-farm pipeline have been developed: on a Unix-based network of workstations (NOW); on a general-purpose multicomputer, the Transtech Paramid [19] multicomputer; and on a microprocessor network supported by the VxWorks realtime operating system (o.s.) [20] which provides a Unix-like programming environment. The Unix environment being of widest availability is given priority in the paper. In fact, without a fast network the NOW is likely to be used as an intermediate stage in development concerned with logic debugging. The application is subsequently transferred onto some target machine (of which the Paramid is but one example), where performance tuning can take place. The VxWorks system falls across the two stages of development: a multicomputer can be emulated through a conventional network; and a version of VxWorks exists in which the same software will run as a tightly-coupled multiprocessor implemented through a VME bus and global memory modules. Soon improved support for networked computation, such as a customized ATM switch [21], may allow a NOW to be used both for intermediate logic debugging and target performance tuning. However, it will often be the case that a core version of an application is maintained, but subsequently adapted for a variety of client systems. Corporate purchasing policies [22] imply that NOWs will be the favoured core system.

The point of research in this paper is to demonstrate that the programmer can be presented with *the same* data-farm template whether working at an intermediate level of code production or on performance tuning. The programmer simply inserts sections of sequential code within

[1] In this paper, the term data parallelism indicates the form of parallel decomposition and does not refer to data-parallel languages such as High Performance Fortran.

the existing template structure. The implementation descriptions establish the feasibility of providing a set of data-farm templates with this scheme in mind.

Within the given problem domain, PPF can also be viewed as a software engineering approach to parallelism. As such, a complete design cycle has been tested with support from graphical performance prediction and evaluation tools. More detail on this aspect of PPF can be found in [23].

2 Design Principles for the Template Implementations

The abstract design of the data-farm template was motivated by engineering utility [24]. Communicating Sequential Processes (CSP) [25] was selected as a model of parallelism, in part because it has been successfully disseminated amongst the programming community, which in itself is a practical consideration. CSP presents a static process structure, that is there is no need to support dynamic process creation. Communication between two processes is solely via a channel by means of a *rendezvous*, which is a synchronous mechanism. Otherwise processes can be scheduled in an arbitrary interleaving, though CSP provides a process algebra which in principle can establish correctness.

CSP's process algebra implies two important features from the efficiency standpoint: low-overhead context switching by means of multi-threading; and the ability to alternate responses in a nondeterministic fashion. For ease of programming, internal and external communication between threads should be transparent and symmetric. These aims were also engendered in the transputer design and associated programming language, `occam`. However, the intention of the present design was not to emulate the transputer virtual machine, as in [26], but to incorporate the model in a looser fashion, relaxing those features not critical, and adding features to enable smooth operation of the data farm.

2.1 Modifying the CSP Model

In CSP, channels are a means by which the normal semantics of a programming language may be extended in a seamless way to include communication between processes. In [27], the absence of a channel from programming languages is viewed as an historical accident due to memory costs. Channels form an implicit name space, without the need for a name-server.

In a number of implementations of CSP there are no compiler checks to prevent a programmer writing from both ends of a CSP channel. Use of a template explicitly designed out this possibility. There is also a problem of excessive 'plumbing' inherent in the CSP channel, which results in the need for the programmer to keep a check on a large number of channels and ensure the messages correspond. Again, a template alleviated this problem.

CSP, as implemented in `occam2`, is not sufficient for data-intensive applications such as low-level image processing, as excessive memory-to-memory data movements may be needed between threads. Unfortunately, on recent hardware, improvements in memory access significantly lag, and may obviate, gains in processor speed. An indication of the problem is that multiple caches, interleaved memory and decoupled architectures are all aimed at ameliorating memory latency. Therefore, shared memory was added by us to the CSP model. Support for shared memory has also been added in `occam3` [28], intended for the T9000 series transputer. Similarly, the need to relax the `occam` specification has been recognized in [29], where semaphores, resources, events, and buckets are alternative synchronization mechanisms to the channel.

For our template design, counting semaphores had several advantages as a means of controlling access to shared memory. In PPF-style applications, semaphores are not needed extensively so that the danger of unforeseen interactions from disparate parts of the program is not present. Access to a critical region is not denied by a semaphore if data will not be compromised. An implementation of semaphores requires one process queue, a counter and a locking mechanism. The monitor construct was not included for a logical reason: it allows only one active call at any one time; and for a practical reason: its operation may be hidden

from the programmer [30]. Other contention access primitives, for example the serializer [31], though convenient for the programmer were not suitable as resources are used inefficiently.

CSP does not include complex communication structures. An asynchronous multicast from the farmer process to its worker processes was deemed necessary to reduce message traffic. The multicast can also act as a means of synchronization for computational phases as well as physical reconfigurations. In this implementation, the multicast does not initiate any reply messages, thereby restricting circular message paths. Care was taken that normal communication could not overtake multicast communication, as a multicast will often contain start-up parameters.

Message records were added as a useful structuring device, the equivalent of `occam`'s protocols. To enable reuse the communication structure was made transparent to the type of application messages. A tag message precedes each data message giving the length of the message for intermediate buffers and its type for the application code.

2.2 Realtime Issues

For soft (i.e. without time-critical deadlines) realtime applications, two priority pre-emptive context switching was sufficient but necessary. In the data-farm, the higher priority is needed to respond to communication events if it is possible to provide an asynchronous response. Once the communication event has been serviced the responding thread deschedules. As a base-level facility, implementable on most platforms, round-robin context switching was also set-up within the template. Scheduling of the ready queue is by a FIFO mechanism. As two levels of priority and a FIFO queueing policy are not sufficient for hard realtime applications [32] describes an alternative CSP-based realtime kernel. Round-robin scheduling can also be viewed [33] as unsuitable for such applications as it reduces response time.

If there are many potential inputs, alternation may be a hindrance because of the need to monitor all the inputs [34]. However, for embedded applications in the PPF pattern the number of inputs was not expected to become large enough to make necessary a tree of multiplexed requests and unlike hard realtime systems deterministic response was not required.

The data-farm paradigm is deadlock-free [35], thus avoiding the principle disadvantage of non-determinism.

Buffers were employed at the user process level to mask communication latency and to increase bandwidth. Input buffers reduce the time spent waiting for work and output buffers smooth out access to the return channel. Ideally, a one slot local buffer is enough to mask communication but in practice a few more slots were needed because of variance in task computation time and communication hold-ups. CSP's synchronous communication was retained. However, buffers act as agents [36] for the application which make it appear to the application that there is an asynchronous send and a blocking receive.

Additional communication structure was provided to enable data-farm template instantiations to be grouped in a pipeline, with options for I/O if the data-farm in question is a terminal stage. The same buffering module is employed between pipeline stages but more slots than needed for local buffers are normally necessary to smooth flow between the stages. A similar buffered pipeline design methodology has already been developed in [37]. A method of predicting the number of buffer slots for local- and inter-stage buffering is discussed in [6]. Additional data-structures may be needed if arriving message data needs to be re-ordered before passing to the data-farm [18] but the details are outside this paper's remit.

2.3 Scheduling policies

Demand-based data farming within the template was, in most cases, a way of scheduling work with limited loss of efficiency. At start-up time, a static scheduling phase was needed to fill buffers. Indeed, for constant task computation times, the time to fill all buffers should exceed single task computation time. Otherwise, task size can be determined by the predictive methods originally identified in [38]. The same use of order statistics approximations is suitable for PPF data farms, as has been broached in [39].

2.4 Instrumentation

Instrumentation, recording communication events, is a built-in feature of the template. Experience [40] shows that instrumentation is difficult to include at a later stage and that a static design will need to be tuned after an initial implementation. In different circumstances, instrumentation has been included in a number of environments such as Jade [41].

Correct termination of the data-farm template is necessary for both the collection of outstanding results and the gathering in of trace files. It was anticipated that the farm might need to be reconfigured if the workload altered during the course of a run. On termination, the data farmer employs a sink process, which is broadly in line with the methods discussed in [42].

2.5 Related work on templates and data farms

This paper examines templates from an object-oriented perspective. However, [59] introduced algorithmic skeletons which are a high-level template written in a functional language. The skeleton provides a parallel control structure but hides implementational detail from the programmer. A model for calculating the cost of the parallelisation is provided. A number of similar structured approaches to parallel programming exist, for a comparative review refer to [60]. Coincidentally closest to the PPF approach, in the sense that farms and pipelines occur amongst the skeletons, is the Pisa Parallel Programming Language, for example recently in [61]. The categorical data type (CDT) framework [60] for list programming is a related higher-level model which further step-backs from the control structures necessary for a particular parallel architecture and being polymorphic is without a preferred granularity. An attraction in principle of the skeleton/CDT approach is a formal and more complete software development scheme.

An influential performance model of the data farm occurs in [62], though aimed at store-and-forward communication. [63] is an elaboration of this model and also contains a thorough review of data farms, making the link with skeletons.

3 Implementation Concerns

Implementation of the design, in an object-oriented sense, is a process of establishing idioms, that is low-level features that are desirable but not present in the underlying software.

3.1 Limitations of PVM and MPI

At first sight, communication harnesses appear to be a natural way of implementing a data-farm template in a distributed and parallel environment.

However, the *de facto* standard, PVM [43], lacks CSP's nondeterministic operator and has no support for internal concurrency. PVM has a restricted set of message-passing primitives but in version 3.3 a system of buffers involving memory-to-memory copying restricts performance. As PVM has an underlying dynamic model of parallelism, dæmons are also necessary to spawn additional tasks and to act as name servers. Where user-level dæmon processes act as communication intermediaries a performance burden arises from the extra messaging needed to communicate between user application and dæmon. Broadcasts in PVM 3.3 are not true broadcasts but one-to-many transmissions. Different versions of PVM may exist on target machines for compatibility reasons. The version of PVM available for the Paramid machine is restricted to a host/worker configuration. As all communication is routed over a SCSI link performance is limited.

The message-passing standard, MPI [44], which is specified as thread-safe, also does not supply a nondeterministic operator. In MPI, multicast is available but by way of semi-dynamic process groups. MPI has a proliferation of communication primitives that can lead to confusion. Whatever the advantages in portability, it may be unclear which message-passing modes are efficiently implemented on the target machine. MPI's derived datatypes, intended to improve performance, are too low-level for many tastes.

Our attention turned to what native facilities were available to more closely implement the main features of the data-farm template: multi-threading, and communication. A customised implementation gave the option to use a true broadcast if the LAN supported that function.

3.2 UNIX Implementation Issues

On Unix systems, the cost of context switching for a heavy-weight process in a worst-case scenario might include swapping the user context from disc and cache flushing. Threads (light-weight processes) are a way of reducing the response time in either interactive or real-time settings. Communication can either be by means of a virtual circuit or by datagrams. The socket is an abstraction through which the programmer can interact with the networking software, typically by binding the socket to a source and destination address, by establishing a connection when a circuit is required, and by sending messages via the designated socket.

On the SunOS 4.1 o.s., the socket application programming interface (API) [45], included from BSD Unix, was combined with light-weight processes (LWP)[2]. The result made possible implementation of most of the required features of the data-farm. Remote procedure call (RPC) was not chosen as a basis for the implementation because of the known overheads, which in [47] were shown to be an order of magnitude above a procedure call on the same machine. The BSD version of Unix implements the socket API directly in o.s. kernel space. The main weakness of the SunOS thread system is that all threads, and indeed all processes, share the one kernel instance. Therefore, it was necessary to employ asynchronous communication to ensure that a LWP does not prevent a context switch by blocking on a communication call. However, asynchronous communication is reliant on signal-handling which in standard Unix implementations occurs as a result of a context switch internal to the process.

3.3 Target Machine Implementation Issues

The Paramid parallel processor is built up from twin-processor modules, using a T805 transputer communication processor and an i860 [48] computation engine. From the user perspective, the Paramid appears as a multi-user transputer machine with attached accelerator onto which jobs are allotted on a first-come-first-served basis by a host-based scheduler. Interpro-

[2]Light-weight process was Sun Microsystem's name for a thread. IEEE POSIX standard (P1003.1c) threads (`pthreads`) [46] are implemented on Solaris 2. The signal handling and scheduling schemes vary in `pthreads` but are not irremediably different from the viewpoint of an implementation of a data-farm.

cessor communication is effected in the first instance by the i860 interrupting the transputer (via the transputer event pin) to signal a request. The transputer inspects a common memory area in order to service the request, releasing a software lock after fulfilling the request. In Inmos parallel 'C' [49] for the transputer there is a set of thread library calls. Interaction with the hardware communication-link engines from within a thread is well defined on the transputer and posed no special problems. There is a choice of point-to-point physical channels or virtual channels. The virtual channel system (VCS) [50] enables direct global communication at a small cost from link sentinel support software.

VxWorks is a Unix-like single-user o.s. for real-time development work, which comes into the class of priority-driven o.s. with enhanced time services [33]. There are no heavy-weight processes only threads with optimised context-switching and response to events. The data-farm template modules can be written in 'C', cross-compiled on a PC running the Windows 95/NT o.s. and loaded and linked on attached 68030K boards.[3] The 68030 microprocessor [51], has an instruction set with test-and-set and compare-and-swap, suitable for implementing semaphores which indeed are a built-in feature of VxWorks. The 68030K boards are linked by an Ethernet LAN, with VxWorks providing a source-compatible BSD 4.3 socket API for using the network.

3.4 Generic Implementation Issues

Fig. 2 is a generic multi-threaded structure which implements the design principles. All of the features were explicitly put in place on the Unix version of the data-farm template. Urgent messages were implemented solely on the Unix system as they are not an essential feature. As I/O is efficiently buffered on Unix systems [52], the provision of a separate I/O thread may be nugatory. On the Paramid system, thread scheduling is implicit. Further, there was no need for message recovery for multicast messages. The VxWorks system version of sockets does not implement network broadcasts, but communication is asynchronous and thread-

[3] Versions of VxWorks are also available for a range of more recent microprocessors.

specific. Thread scheduling is user selectable, either priority-based with pre-emptive options or round-robin.

In designing the worker module an initial consideration was the nature of message traffic. Messages occur in two parts: a tag and the body of the message. The tag must include: the size of the message to follow; a type indicating whether a message is a broadcast or a request for processing; and a message number. The message number is intended to signal to the receiver which data structure to position for the accommodation of the second part of the message. The message number might also be used for other purposes. The body of the message should include a function number as the first field of the message, but otherwise the message record structure was undetermined. If a sequential version of a program exists, it may involve excessive data movements to form messages into logical message structures. The application programmer will need to balance utility with complexity.

The potential for a large number of different messages was the reason for the restriction to a rigid message format. Each work request message is serviced by one application thread function (Fig. 3). In 'C' it is possible to use an array of function pointers to which the function number forms an index. Though not entirely satisfactory, data and parameters are passed to the function as globals. The result is that each function can be referenced simply by a number. From the figure it will be seen that the application thread was divided between a public interface and a private part into which different functions can be slotted. This makes it possible to extract the functions from sequential code constructed with structured programming and place them into the slots.

Circular in-coming and out-going buffers service each application process (Fig. 4). Access-contention control was set up through semaphores. Separate buffer slots are kept for tags, otherwise there is a danger of the small slots needed for tags being expanded to handle larger messages. Buffers are automatically enlarged by dynamic memory allocation. To avoid the possibility of deadlock if a series of broadcast message were to arrive at asynchronous intervals, at least one extra buffer slot is needed over and above the number of messages sent out at loading time. This method is a variant of an algorithm which is proven in [53].

4 Unix-based NOW Implementation with Logic Debugging

4.1 Unix Threads/LWP

A thread system was first established on the worker and farmer modules using the LWP library. Context switching is within a Unix heavy-weight process. A process's stack is multiplexed by the LWP library between the various LWPs. The LWP library maintains minimal state for each LWP, for instance a program counter and the LWP's priority, so that if a LWP's time-slice is interrupted it does not complete the remainder of the time slice when it is rescheduled. This seemed appropriate for the data farm, but other thread systems (e.g. VxWorks) maintain more state. To allow for the rudimentary nature of the thread system, the Unix version of the template caught any stack overflows between LWPs.[4]

The data-farm design calls for pre-emptive communication threads and background round-robin thread scheduling. However, the SunOS LWP library is non-preemptive and priority-driven. A compromise was to set-up all normal threads to be switched by a round-robin schedule, but after polling for a communication event a communication thread immediately deschedules if no response occurs. Round-robin context switching was provided in the farm template by a software scheduler LWP:

```
while (TRUE)
     sleep(TIME_QUANTUM)
     reschedule active process queue
```

A time quantum of 100 μs was used in tests.

[4]Stack partitions can be red-zone protected by the LWP library.

4.2 Communication Structures

External communication was implemented by means of the BSD socket API. An effort was made to tune communication within the data farm by utilising some of the specialist features of the API in coordination with the LWP library. The `fcntl` system call enables socket communication to be made non-blocking, as was required in the design model. Reliable stream communication occurs via the underlying TCP/IP transport-level protocol. However, care was still needed in coding the template channel primitives as message contents could be lost if either the message system call unblocked or if the data stream delivered an incomplete section of the intended message. Standard Unix has a global error number, `errno`, which in this instance was important as it indicates message status. Therefore, the global `errno` was mapped to a local LWP copy. Conveniently, the LWP context in the relevant template threads was augmented to include `errno` by means of a LWP library facility. The template sockets were set so that Nagle's algorithm [54] was not applied (whereby small messages are delayed in order to avoid congestion on long-haul networks). The `fcntl` call also can set a socket to give an asynchronous response and this facility was employed for broadcast and urgent sockets. The socket was awakened by a BSD Unix I/O signal. Where more than one reception socket is needed then it is necessary to inspect each socket in turn as an I/O signal occurs on a per-process basis. In order to map signals to threads an `agent` was needed. An LWP `agent` sleeps at the highest priority until a signal arrives. The `agent` then makes a `rendezvous` with its associated LWP. The template broadcast thread or urgent thread services all pending communication before descheduling.

Internal communication between the template threads, principally between the buffer threads and an application thread, was implemented by wrapping the LWP `rendezvous` to resemble a CSP channel. Unlike CSP's channel, the LWP `rendezvous` is asymmetric. The designated sender passes the addresses of an input buffer and an output buffer. The receiver LWP is rescheduled when the sender and the receiver reach the `rendezvous` point. The sender is rescheduled by the receiver once it has processed the buffers in any way. In the template

implementation, the `rendezvous` was used simply for synchronisation, so as to avoid an extra memory-to-memory copy. Data were transferred not by the `rendezvous` buffer mechanism but by fast memory transfer through ANSI 'C' `memcpy` into global buffers.

4.3 Other Design Features

The template design specifies semaphore regulation of global buffers. However, Sun's LWP library supports a monitor data structure with associated condition variables, which can be signalled to. The action of the monitor is hidden through `mutex` variables. Counting semaphore wait (p) and signal (v) were fashioned from this primitive, illustrated in pseudo-code:

wait:

```
lock mutex

while (semaphore count is zero)
  {wait on semaphore condition variable}

decrement semaphore counter

unlock mutex
```

signal:

```
lock mutex

if (semaphore counter is zero)
   {set contention}
else
   {unset contention}
```

```
increment signal counter

if (contention is set)
   {signal on semaphore condition variable}

unlock mutex
```

No LWP is scheduled while another LWP is within either critical region, marked by the `mutex`, unless that LWP has been descheduled by `wait`. The queue to the semaphore is assumed to be FIFO.

The nondeterministic operator was simulated in the template by the socket API `select` system call. The system call is unblocked by only using `select` in its polling form. As more than one socket may become ready for communication a routine was added to shuffle the order of selection according to a suitable and ideally random re-ordering. The intention was to ensure fairness to the selection of inputs. An alternative which was easy to implement was to move the current socket tag to the end of a list and move all the other tags forward one step.

Message records were implemented by means of socket API vectored messages. Again, care was needed in writing routines for vectored messages as it is possible for one or more of the parts of the vectored message to arrive incomplete or not at all. The template code identifies which part of the vectored message has been dropped and picks up the rest of the message stream.

Broadcasts under BSD Unix are restricted to datagrams, for which delivery is not guaranteed. In tests, it was found that if successive broadcasts were sent there was a distinct possibility that broadcast frames would be dropped.[5] Therefore, the template incorporates

[5]The reception socket was again in unblocking and asynchronous mode.

robust checking. A recovery mechanism was necessary whereby after a timed interval a repeat request is sent. The timer was implemented by an alarm-clock interrupt. Each work message is stamped with the sequence number of the last broadcast to be sent. The broadcast thread maintains the sequence number of the last consecutive broadcast. It will then be evident if either a broadcast has been dropped or if a work message has overtaken a broadcast.[6]

4.4 Programmer's Model of the Template

The application programmer can prepare application code for incorporation into the template largely as if a worker process and the farmer process are single-threaded. The main exception is non-reentrant system calls [52] which should be avoided. The programmer can check the working of the application by reference to an event-trace which is timestamped (or event-stamped) by a global clock. [24] gives a simple example of this approach whereby the code for a one dimensional FFT was inserted in a worker template. Two farms were formed for the row/column processing stages of a two dimensional transform. The code for the intervening matrix transpose formed a centralised stage catered for in a single farmer. The whole formed a three stage 2-D FFT parallel pipeline.

4.5 Instrumentation and Visualisation

A scalar logical clock was not difficult to implement as it requires no extra message passing. The clock update algorithm required a minimum of calculation:

```
Initialise:
  Set logical_clock to zero.
  Set clock increment (usually to one).


Logical-clock procedure:
  If (message is a receive)
```

[6] Duplicate broadcasts pose no extra difficulty.

```
    {let logical_clock = maximum(logical_clock, received_time_stamp)}

Increment logical_clock

If (message is a send)
     {time-stamp message}
```

The farm system does not have simultaneous messages arriving, but were this so the tie would be broken by the process identities. The scalar clock may be extended to a vector clock, which allows ordering of internal events as long as the processes concerned are causally related (by a message). The components of the vector are the sending process's most recent scalar clock for all processes. The overhead from passing a vector of clocks with the most recent timings at any one process for all other processes grows with the number of processes, while the scalar logical clock keeps a record with minimal perturbation. Generally, one should bear in mind that the pattern of message passing on a distributed system might be different to that of the target machine. The intention is to catch unexpected orderings. A scalar logical clock also is employed in the ATEMPT trace system [55].

At this stage in development, the trace display was via the post-mortem visualizer, ParaGraph [56]. A standard trace file [57] format was used, compatible with ParaGraph. The format includes a broadcast field but does not include multicast, which is understandable as the destinations are difficult to specify if the record size is restricted. However, if several farms are employed within the PPF pattern, multiple per-farm multicasts can take place. These were emulated by creating multiple message records in the trace file. Multicasts were stamped with the source and a message-type code not used elsewhere. Post-processing changed the multicast message to a set of messages with the same timestamps but different destinations (ParaGraph does not assume a monotonic clock). Initialization and termination messages could also be removed at trace-file post-processing time.

Fig. 5 is a screen-shot showing a trace taken from a single-farm test run. The plotted lines represent messages from one process to another. The slope of the line indicates the direction of communication. Processor 0 hosts the farmer process, which initially loads three buffer slots with work on each of four worker processes. At each cycle of subsequent processing: a broadcast is sent; a request for work in the form of processed work is serviced; and an urgent signal is sent to one of the processes (on processor two).[7] Because broadcast messages, urgent messages and normal work messages are handled by separate threads within the worker process it is necessary to maintain distinct logical clocks for each of these threads.[8] At the end of processing, the trace files for each thread and each process are merged and subsequently sorted into order.

5 Target Machine Implementation with Performance Debugging

5.1 The Paramid System

Apart from multicast, most of the template communication primitives were already present in Inmos parallel 'C'. The existing system software (Fig. 6) includes servicing of run-time I/O requests on all modules by means of a run-time executive (RTE). I/O is multiplexed onto the SCSI link to the host. The multiplexor channels are set up conveniently by VCS software. Only one process running on the i860 can communicate with the system interface program running on the transputer. Naturally, this process was the worker application thread which was supplied with a public interface and a set of protected services, exactly as in the Unix version.

Instrumentation was the main sub-system missing from the Paramid system software (though console monitoring of link activity is available and useful). The existing interface

[7] The urgent signal is an optional facility provided to enable a response to interrupts.

[8] The size of each trace-record file data structure should be regulated in order not to perturb the application which could for example occur through excessive paging.

program was enhanced with a trace recorder and synchronized clock process (Fig. 7). To substitute the new interface program, the application object code is booted onto the i860 network and in a second loading phase the transputers are booted up. The interface program then restarts the i860. The local clocks are updated by periodic pulses from a monitor process. An adjustment algorithm compensates for local clock drift [23]. On receipt of a message from the i860 application program, a trace record is generated, timestamped by a call to the local clock. All the processes mentioned run at high-priority as it is important to service the i860. Where a trace is made on a transputer-based process, the clock should also run at high priority so as to reduce the interrupt latency, which for a single high priority process is 58 processor cycles (2 μs). If need be, an additional process is run at low priority [58] with the purpose of monitoring processor activity. The process simply counts each time it is activated before descheduling itself. If the processor monitor is called relatively frequently the processor can be assumed to be relatively idle. Internal monitoring of processes is not necessary if there is limited competition for the transputer's time. If the interface program could determine the destination or source of a message by its contents these arrangements would be enough. At present, the communication primitive on the i860 is augmented to include these details.[9]

5.2 VxWorks System

Many of the external communication features of the data-farm template were implemented in VxWorks exactly as in the Unix-based system. A data-farm worker module, consisting of application and buffering threads is spawned from an initialising thread. Remote spawning was accomplished by writing an iterative server as RPC daemons are unavailable in VxWorks. The internal state of the VxWorks system is almost completely user-accessible and in many cases user modifiable. For example, the o.s. clock is programmable and cache flushing is specifiable. In fact, the data-farm template structure is needed to avoid anarchic use of the facilities. Internal channels were emulated in VxWorks, by a `message queue` primitive which

[9]The Paramid shared-memory data structure can be changed usually without disturbing the pre-compiled kernel routines.

when single-spaced fulfils the same purpose. A wrapper was provided to make it appear that the channel is used for intra- and inter-processor communication. The queueing discipline on semaphores is selectable, though for compatibility with the data-farm template in the other environments a FIFO discipline was chosen.

6 Conclusion

The Pipelined Processor Farm (PPF) is a software architectural pattern likely to be of wide utility wherever there is continuous-flow of data. PPF has existed as an abstract design concept for embedded systems, indeed as a pattern in the object-oriented sense. In the wider parallel community, apparently the idea has only recently gained currency of combining task parallelism across a pipeline with data parallelism within each pipeline stage, perhaps because of a bifurcation between those favouring either implicit or explicit parallelism.

Again in the object-oriented mould, the PPF abstract design can be supported through software components, one of which, the data-farm template, has been explored in detail in this paper. The design of the data farm is guided by a model which will give reasonable efficiency across a range of accessible architectures. The model may be seen as a relaxation of an existing static model of parallelism, CSP.

The research in this paper is intended to show how a common design is feasible across three environments. In a distributed Unix environment a number of implementation problems have been solved by:

- combining threads with the socket API by means of asynchronous messages;
- providing a true broadcast with a recovery mechanism;
- mapping I/O signals onto the thread structure;
- forming semaphores from existing `mutex` primitives; and
- modelling: a nondeterministic operator via the `select` system call; message records through vectored messages; and internal messages through a `rendezvous` construction.

Within distributed Unix, communication software is couched in terms of an active client and passive server. However, in our design while the farmer and worker are active and passive the communication model's primitives are symmetric. Despite the difficulties, it is possible to superimpose the processing model onto the Unix environment.

The data farms do not exist in isolation but form part of the development support for PPF. In a distributed workstation environment the PPF processing model is envisaged in a prototyping role, providing a full range of development support facilities. In a dedicated parallel machine the support facilities would be available but usually would be discarded when the final application has been developed. A similar environment for development work is in a real-time Unix setting, as provided for example by the VxWorks o.s. Some extra features of this single-user environment are: kernel support for context-switching; convenient device interfaces; and enhanced interrupt response. Some developers may also want to use small-scale shared-memory machines. Message-passing by mailboxes is one mode of employing such machines. Thus, shared-memory machines represent a future port for PPF data farms.

To support prototyping, built-in instrumentation is necessary. Event tracing by means of logical clock timestamps can conveniently be employed for logic debugging as an accurate real-time clock is difficult to implement in a distributed environment. Event tracing through timestamps from a real-time clock can pinpoint hold-ups in application performance. As in the MPI standard, name-shifting is possible, whereby there are two sets of names for all library calls. As a result, access to the implementation is not needed if tracing is introduced or removed.

The PPF pattern may be applicable to meta-computing which implies the construction of a Java data farm software component based around a CSP class library. Other parallel design patterns are also possible.

Acknowledgement

This work is being carried out under EPSRC research contract GR/K40277 'Portable software tools for embedded signal processing applications' as part of the EPSRC Portable Software Tools for Parallel Architectures directed programme.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP'93 — Object-Oriented Programming*, pages 406–421, 1993. Lecture Notes in Computer Science volume 707.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns: Pattern-oriented Software Architecture.* Wiley, Chichester, UK, 1996.

[3] A. C. Downton, R. W. S. Tregidgo, and A. Çuhadar. Top-down structured parallelisation of embedded image processing applications. *IEE Proceedings I (Vision, Image, and Signal Processing)*, 141(6):431–437, December 1994.

[4] D. Krieger and R. M. Adler. The emergence of distributed component platforms. *IEEE Computer*, 31(3):43–53, March 1997.

[5] S. Ahmed, N. Carriero, and D. Gelernter. The Linda program builder. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 71–87. Pitman, London, 1991.

[6] M. Fleury, N. Sarvan, A. C. Downton, and A. F. Clark. A parallel-system design toolkit for vision and image processing. In *EuroPar'98*, pages 92–101. Springer, Berlin, 1998.

[7] M. N. Edward. Radar signal processing on a fault tolerant transputer array. In T.S Durrani, W.A. Sandham, J.J. Soraghan, and S.M. Forbes, editors, *Applications of Transputers 3*. IOS, 1991.

[8] S. Glinski and D. Roe. Spoken language recognition on a DSP array processor. *IEEE Transactions on Parallel and Distributed Systems*, 5(7):697–703, 1994.

[9] A-M Cheng. High speed video compression testbed. *IEEE Transactions on Consumer Electronics*, 40(3):538–548, 1994.

[10] A. Çuhadar and A. C. Downton. Structured parallel design for embedded vision systems: An application case study. In *Proceedings of IPA'95 IEE International Conference on Image Processing and Its Applications*, pages 712–716, July 1995. IEE Conference Publication No. 410.

[11] M. H. Coffin. *Parallel Programming A New Approach.* Silicon Press, 1992.

[12] I. Foster. Task parallelism and high-performance languages. In G-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, pages 179–196. Springer, Berlin, 1996. Lecture Notes in Computer Science Volume 1132.

[13] A. C. Downton. Speed-up trend analysis for H.261 and model-based image coding algorithms using a parallel-pipeline model. *Signal Processing: Image Communications*, 7:489–502, 1995.

[14] H. P. Sava, M. Fleury, A. C. Downton, and A. F. Clark. A case study in pipeline processor farming: Parallelising the H.263 encoder. In *UK Parallel '96*, pages 196–205. Springer, London, 1996.

[15] A. Çuhadar, D. Sampson, and A. Downton. A scalable parallel approach to vector quantization. *Real-Time Imaging*, 2:241–247, 1996.

[16] A. Çuhadar, A. C. Downton, and M. Fleury. A structured parallel design for embedded vision systems: A case study. *Microprocessors and Microsystems*, 21:131–141, 1997.

[17] M. Fleury, A. C. Downton, and A. F. Clark. Pipelined parallelization of face recognition. *Machine Vision and Applications*, 1998. submitted for publication.

[18] M. Fleury, A. C. Downton, and A. F. Clark. Co-design by parallel prototyping: Optical-flow detection case study. In *High Performance Architectures for Real-Time Image Processing*, pages 8/1–8/13, 1998. IEE Colloquium Ref. No. 1998/197.

[19] Transtech Parallel Systems Ltd., 17-19 Manor Court Yard, Hughenden Ave., High Wycombe, Bucks., UK. *The Paramid User's Guide*, 1993.

[20] Wind River Systems, Inc., 1010, Atlantic Avenue, Almeda, CA. *VxWorks Programmer's Guide*, 1993. Version 5.1.

[21] M. G. H. Katevenis, P. Vatsolaki, D. Serpanos, and E. Markatos. ATLAS 1: A single-chip switch for NOWs. In *Communication and Architectural Support for Network-Based Parallel Computing, CANPC'97*, pages 88–101. Springer, Berlin, 1997. Lecture Notes in Computer Science Volume 1199.

[22] J. S. Kowalik and K. W. Neves. Software for parallel computing: Key issues and research directions. In J. S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, pages 3–36. Springer, Berlin, 1993.

[23] M. Fleury, N. Sarvan, A. C. Downton, and A. F. Clark. Toolkit design for system analysis of parallel pipelines. *Concurrency: Practice and Experience*, 1998. Submitted for publication.

[24] M. Fleury, H. Sava, A. C. Downton, and A. F. Clark. A real-time parallel image-processing model. In *IPA'97*, pages 174–178, 1997. IEE Conference Publication No. 443.

[25] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[26] D. G. Patrick, P. R. Green, and T. A. York. A multiprocessor OCCAM development system for UNIX network clusters. In A. Bakkers, editor, *Parallel Programming and Java*. IOS, Amsterdam, 1997.

[27] G. Hilderink, J. Broenink, W. Vervoot, and A. Bakkers. Communicating java threads. In A. Bakkers, editor, *Parallel Programming and Java*. IOS, Amsterdam, 1997.

[28] G. Barrett. `occam3` reference manual. Technical report, Inmos Ltd., 1000 Aztec Way, Bristol, UK, 1992.

[29] P. J. Welch and D. C. Wood. Higher levels of process synchronisation. In A. Bakkers, editor, *Parallel Programming and Java*, pages 104–129. IOS, Amsterdam, 1997.

[30] H. Williams. Threads and multithreading. In *Java 1.1 Unleashed*, pages 121–163. Sams, Indianapolis, IN, 1997.

[31] C. E. Hewitt and R. R. Atkinson. Specification and proof techniques for serializers. *IEEE Transactions on Software Engineering*, 5(1):10–23, January 1979.

[32] E. Verhulst. Non-sequential processing: bridging the semantic gap left by the von Neumann architecture. In *Signal Processing Systems SIPS'97*, pages 35–49. IEEE, Piscataway, NJ, 1997.

[33] S-T. Levi and A. K. Agrawala. *Real Time System Design.* McGraw-Hill, New York, 1990.

[34] G. V. Wilson. *Practical Parallel Processing.* MIT, Cambridge, MA, 1995.

[35] P. Welch, G. Justo, and C. Willcock. High-level paradigms for deadlock-free high-performance systems. In *Transputer Applications and Systems '93*, pages 981–1004. IOS, Amsterdam, 1993.

[36] R. Milner. *Communication and Concurrency.* Prentice-Hall, New York, 1989.

[37] S-Y Lee and J. K. Aggarwal. A system design scheduling strategy for parallel image processing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(2):194–204, 1990.

[38] C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, 11(10):1001–1016, October 1985.

[39] M. Fleury, A. C. Downton, and A. F. Clark. Modelling pipelines for embedded parallel processor system design. *Electronic Letters*, 33(22):1852–1853, 1997.

[40] D. A. Reed. Performance instrumentation techniques for parallel systems. *Lecture Notes in Computer Science*, 729:463–490, 1993.

[41] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A high-level machine-independent language for parallel programming. *IEEE Computer*, 26(6):28–38, June 1993.

[42] P. H. Welch. Graceful termination — graceful resetting. In Bakkers A., editor, $10^{\underline{th}}$ *Occam User Group Technical Meeting*. IOS, Amsterdam, 1989.

[43] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine — A Users' Guide and Tutorial for Networked Parallel Computing*. MIT, Cambridge, MA, 1994.

[44] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. MIT, Cambridge, MA, 1994.

[45] W. R. Stevens. *Unix Network Programming*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[46] *Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) – Amendment 1:Realtime Extension (C Language)*. IEEE, New York, NY, 1995. Standard 1003.1c-1995, also ISO/IEC 9945-1:1990b.

[47] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[48] M. Atkins. Performance and the i860 microprocessor. *IEEE Micro*, pages 24–27,72–78, October 1991.

[49] Inmos Ltd., 1000 Aztec Way, Bristol, UK. *ANSI C Toolset User Manual*, 1990.

[50] M. Debbage, M. B. Hill, and D. A. Nicole. The virtual channel router. *Transputer Communications*, 1(1):3–18, August 1993.

[51] D. Tabak. *Multiprocessors*. Prentice-Hall, Engelwood Cliffs, NJ, 1990.

[52] S. Kleiman, D. Shah, and B. Smaalders. *Programming with Threads*. SunSoft/ Prentice-Hall, Upper Saddle River, NJ, 1996.

[53] A. W. Roscoe. Routing messages through networks: An exercise in deadlock avoidance. In T. Muntean, editor, $7^{\underline{th}}$ *Occam User Group Technical Meeting*, pages 55–79. IOS, Amsterdam, 1987.

[54] J. Nagle. Congestion control in ip/tcp internetworks. Technical report, Ford Aerospace and Communications Corporation, 1984. RFC 896.

[55] S. Grabner, D. Kranzlmüller, and J. Volkert. Debugging parallel programs using ATEMPT. In B. Hertzberger and G. Serazzi, editors, *High-Performance Computing and Networking International Conference*, pages 235–240. Springer, Berlin, 1995. Lecture Notes in Computer Science Volume 919.

[56] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, May 1991.

[57] P. H. Worley. A new PICL trace file format. Technical report, Oak Ridge National Laboratory, Oak Ridge, TN, USA, September 1992. Report ORNL/TM-12125.

[58] A. Bauch, T. Kosch, E. Maehle, and W. Obelöer. The software-monitor DELTA-T and its use for performance measurements of some farming variants on the multi-transputer system DAMP. *Lecture Notes in Computer Science*, 634:67–78, 1992. Proceedings of CONPAR '92 - VAPP V.

[59] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman, 1989.

[60] D. B. Skillicorn. *Foundations of Parallel Programming*. C.U.P., Cambridge, UK, 1994.

[61] M. Vanneschi. Heterogeneous HPC environments. In D. Pritchard and J. Reeve, editors, *Euro-Par'98 Parallel Processing*, pages 21–34. Springer, Berlin, 1998. Lecture Notes in Computer Science No. 1470.

[62] D. J. Pritchard. Mathematical models of distributed computation. In $7^{\underline{th}}$ *Occam User Group Technical Meeting*. IOS, Amsterdam, 1987.

[63] A. S. Wagner, H. V. Sreekantaswamy, and S. T. Chanson. Performance models for the processor farm paradigm. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):475–489, May 1997.

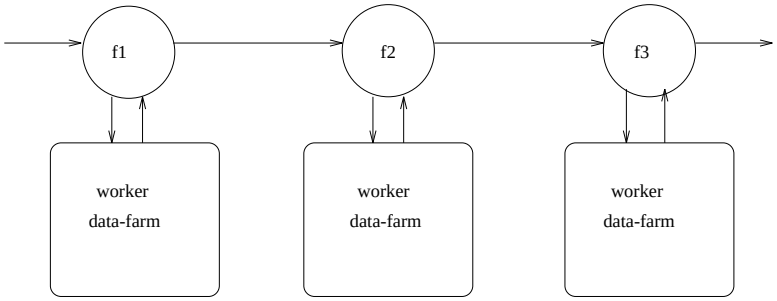

Figure 1: PPF Pattern

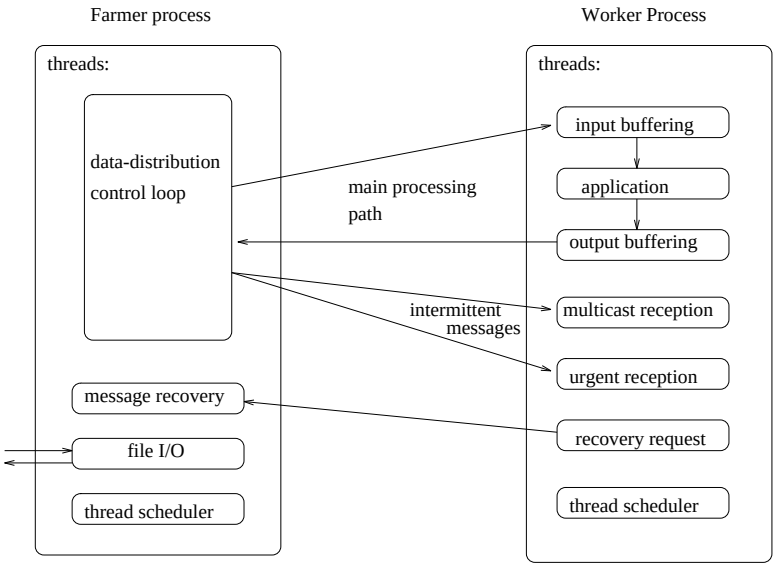

Figure 2: Simplified Layout of a Single Farm

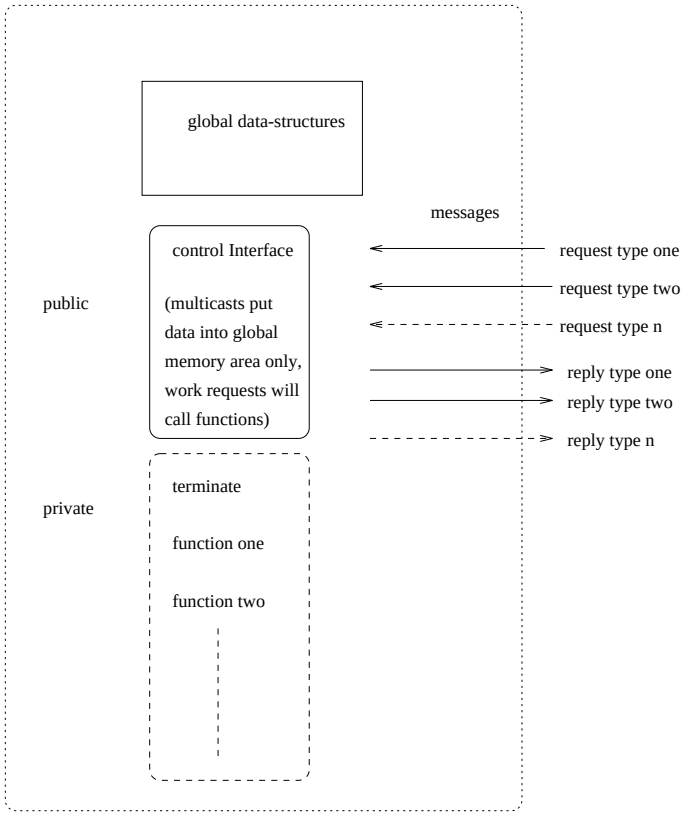

Figure 3: Generic Application Thread

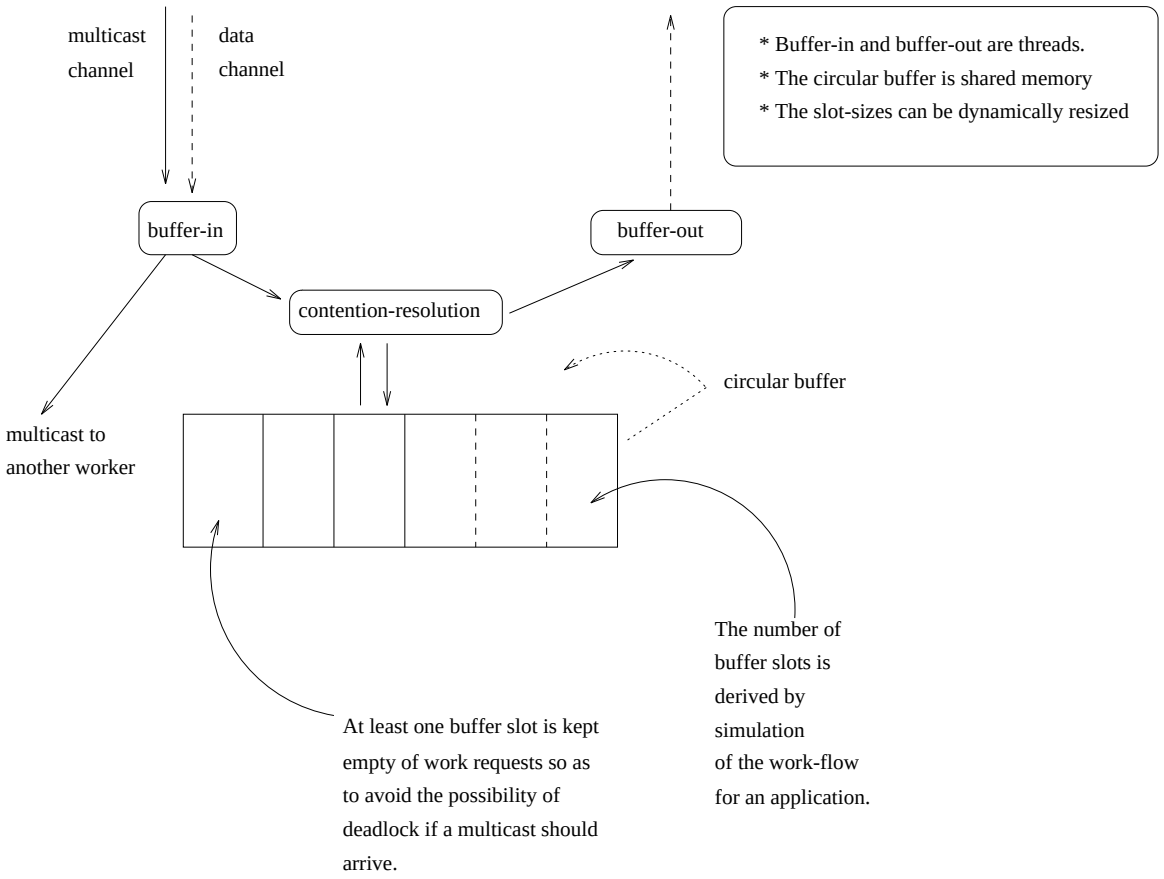

Figure 4: Generic Buffer

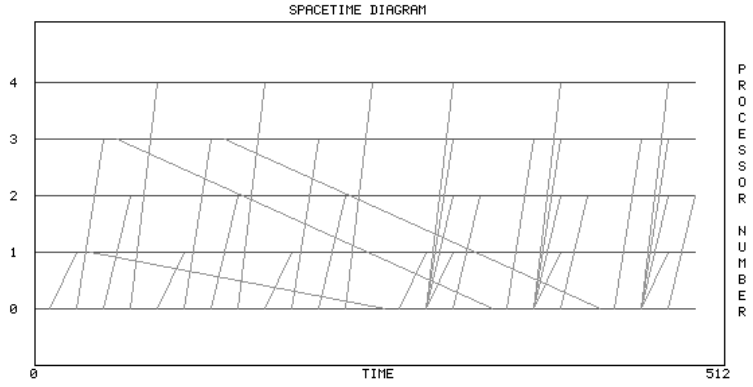

Figure 5: Logical Clock Trace of a Farm Test Run

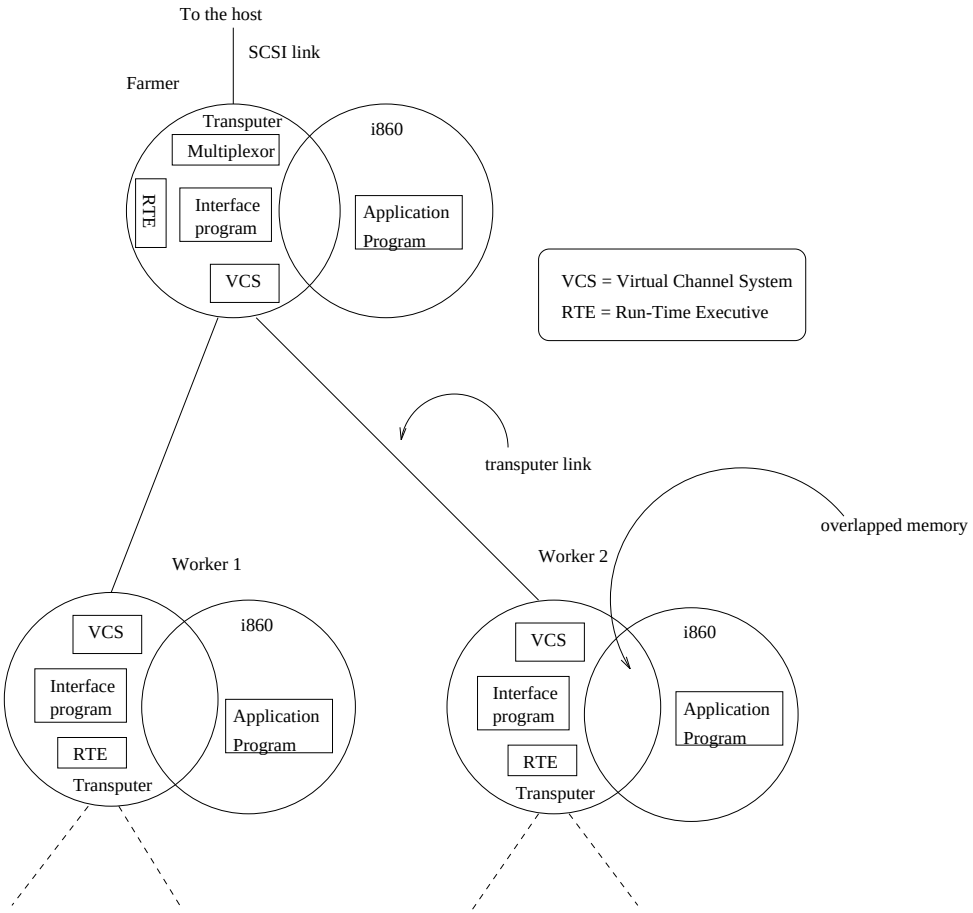

Figure 6: Paramid System Software

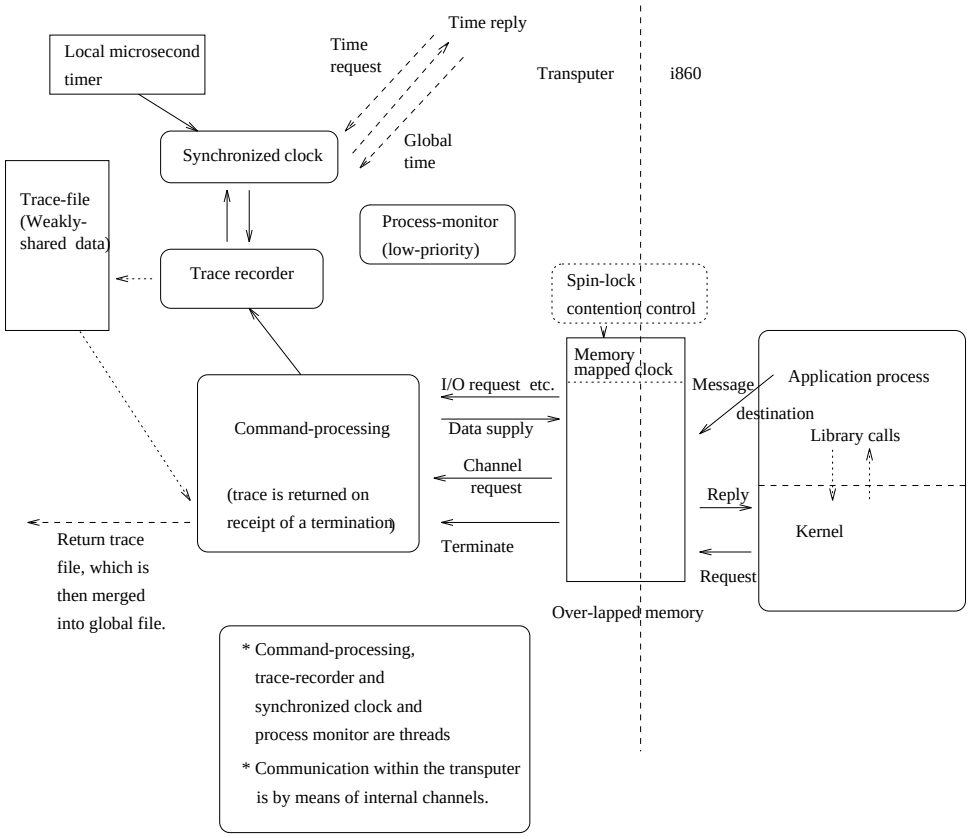

Figure 7: The Paramid Monitoring Layout