Designing and Instrumenting a Software Template for Embedded Parallel Systems

M. Fleury
Dept. of Electronic Systems Engineering, Essex University
Colchester, U.K
H. P. Sava
Dept. of Electronic Systems Engineering, Essex University
Colchester, U.K
A. C. Downton
Dept. of Electronic Systems Engineering, Essex University
Colchester, U.K
A. F. Clark
Dept. of Electronic Systems Engineering, Essex University
Colchester, U.K

Abstract

This paper considers the design of a reusable software template for a parallel data-farm which uses demand-based load-balancing. A feature of the farm is integral instrumentation. A design example is given for a hybrid processor message-passing machine (the Paramid) in which monitoring is accomplished by an instrumented interface program. Other aspects of the design are use of buffering to mask communication latency, an asynchronous multicast provision, and a controlled interface to the worker functions. Trace material is discussed from two examples when the template design was used to monitor real-time, continuous-flow applications. The template is a component of the Pipelined Processor Farm (PPF) methodology.

1 Introduction

This paper describes the design of a software 'template' which supports construction of embedded parallel applications that utilise multiple data farms. A template restricts the scope of the software thereby making the design easier to develop. The data farm is intended to enable rapid prototyping of applications developed according to the Pipelined Processor Farm (PPF) methodology [1]. PPFs are suitable for the class of continuous-flow embedded systems, which commonly have a time and/or ordering constraint imposed upon the output. The methodology proposes three forms of decomposing the workload in a component farm: by means of temporal multiplexing; through algorithmic parallelism; or through data parallelism. Demand-based farming utilising 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 provides the basis of this initial template design.

Unlike some other template designs [2, 3], our design includes instrumentation as an integral part of the software. Experience [4] 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 [1]. In `Tmon` [5] instrumentation is included but requires a hardware monitor to provide time pulses, which may improve the accuracy but naturally limits the portability. `Tmon` also requires user annotation of the source code, whereas the present system intercepts communication events by means of a monitoring sub-system, implemented as part of the template. The PIE Environment [6] includes instrumentation provision in software but is less constrained in its applicability and is aimed at machines supporting a global address space. The concept of providing performance evaluation facilities as an integral part of the environment is present in Jade [7], though otherwise the similarity stops as this environment attempts to hide the details of parallelism from the programmer.
The design principles for our data-farm template can be summarized as:

- a demand-based load-balancing algorithm;
- latency regulation by the use of buffering, which is transparent to the message size and type;
- a multicast mechanism (on a per-farm basis), which can be called on at any time;
- graceful termination, which also allows a reset for reconfiguration purposes;
- a controlled interface to the functions offered by each worker process;
- instrumentation of all communication events.

These criteria are developed in succeeding sections. Though the major part of this paper is couched in the form of a case study, in fact most of the material presented is general (i.e., Sections 2.2, 2.3, 2.4 & 2.6) and can be transferred to other message-passing environments. Reference is made to two applications: a handwritten postcode recognition application and an H.263 image coder parallelization. Output from a trace visualizer for the two applications is included in Section 3. The final part of the paper, Section 4, summarizes and draws some conclusions.

2 An Instrumented Template on a Hybrid Machine

The initial target architecture for this implementation was the Paramid parallel processor from Transtech Ltd. [8] which is a distributed-memory message-passing machine built up from twin-processor modules. In our case, we were

using an eight-module machine, with transputer communication processor (running at a nominal 30MHz, with link speed set to 20Mb/s and 4Mbytes RAM) and i860 computation engine (50MHz and 16Mbytes RAM). From the user perspective, the machine appears as a transputer machine with attached accelerator onto which jobs are allotted on a first-come-first-served basis by a host-based scheduler. The i860-XP is a superscalar RISC processor with pipelining of the integer and floating point computational streams. The i860 [9] is primarily suited to the processing of regular structures, such as matrices. Interprocessor 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.

2.1 System Software Organization

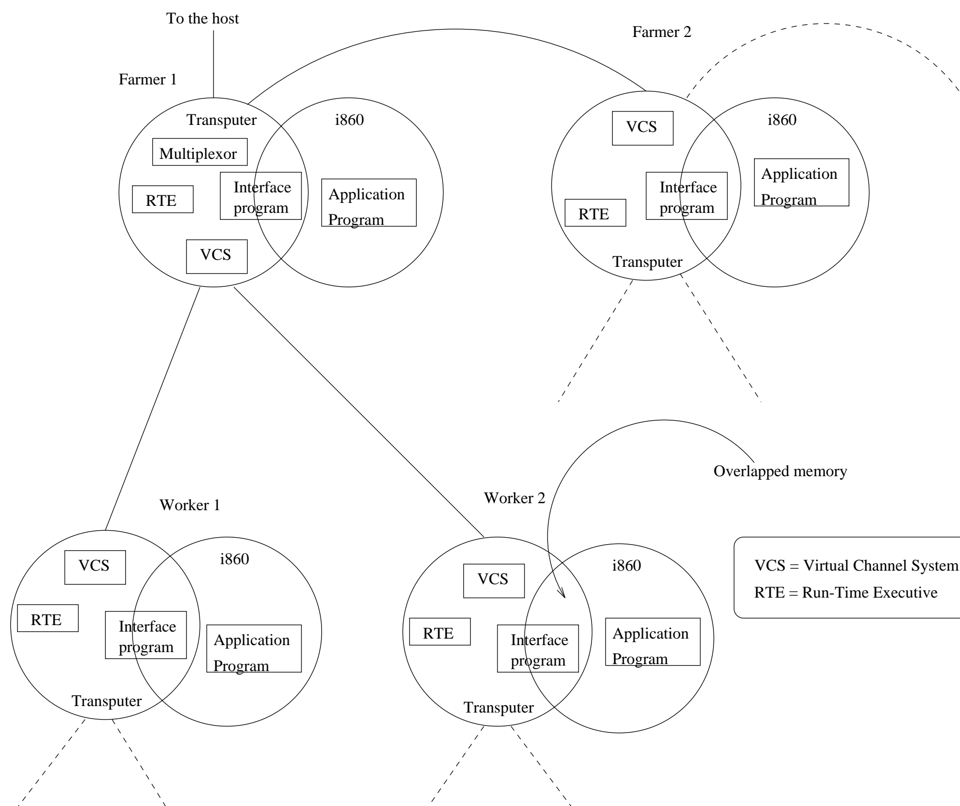

Figure 1: System Software

The existing system software (Figure 1) 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 means of a virtual channel system (VCS) which acts on each processor as an external communication link sentinel. VCS software is important in limiting the complexity of the communication software that is provided by the application programmer; but it also makes direct communication as required for multicasts awkward to guarantee (Section 2.4). Ideally, one would like to have both virtual and actual communication in the same application. The transputer provides efficient internal concurrency by manipulation of a process stack [10] but only one process running on the i860 can communicate with the system interface program running on the transputer. This restriction neatly meets the design methodology in that one process can be written with a public interface and a set of protected services, which thereby take the place of multiple processes on the i860 (Section 2.2). Note that 3L Parallel 'C' and Inmos Parallel 'C' for the transputer both distinguish between tasks on the same processor, which may communicate only by channels, and processes (equivalent to threads [11] or light-weight processes) which are internal to tasks and can communicate either by common memory or by channels.

Two versions of the data-farming template are necessary for this machine: one in which all application processes are sited on the i860s; and another in which the data-farmers are placed on transputers. The precise arrangement is application-specific as it depends on the degree of centralized processing necessary. In the H.263 encoder application [12] there were serial bottlenecks which made it important to place the data-farmers on the i860s: this produced a performance gain of a factor of 2 compared with placing the farmer processes on a transputer. In contrast, the three data farmers used in the postcode recognition application [13] were all placed on transputers as their computational load was small. This still allowed the real-time, ordering and latency constraints of the postcode specification to be met.

2.2 The Generic Worker Module

In designing the worker an initial consideration is the nature of the 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 multicast 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. It 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 is 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 is a reason for confining oneself to a rigid message format. Unlike `occam`, the 'C' computer language does not offer guidance in this respect. C++ has the

difficulty that there is not yet an ISO standard. Each work request message is serviced by one worker-module function (Figure 2). For reasons explained in Section 2.4, multicast messages generate no processing. 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. This means that each function can be referenced simply by a number. From the figure it will be seen that the worker module is 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.

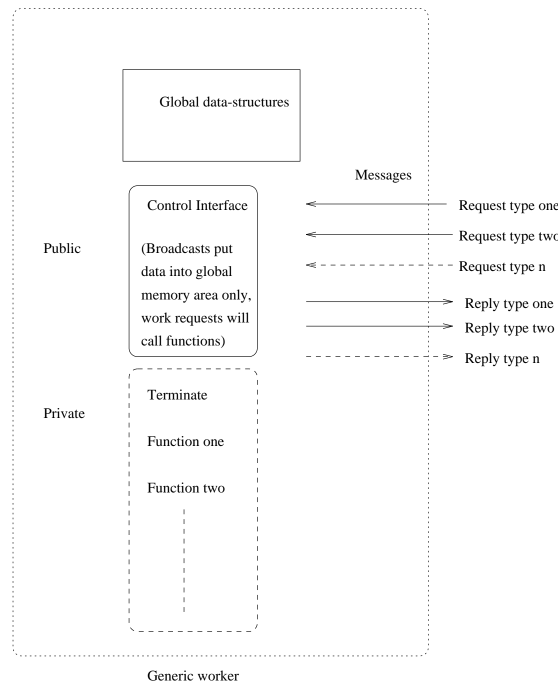

Figure 2: The Worker Module

2.3 The Buffer Design

While there has been considerable exploitation of parallel slackness (by means of concurrent or pseudo-parallel processes) as a way of masking communication latency, rather less attention has been given to using buffers for the same purpose [14]. Historically this situation arises because the XPRAM model [15] was part of an attractive plan for a universal programmer's model for parallel machines that could work on the two broad classes of MIMD, shared- and distributed-memory machines. Providing internal concurrency may not however be cost-free. If overlapped register windows are used, the cost of context switching may become large as the register set is exhausted. If a process

is descheduled on one processor at a time when a process on another processor wishes to communicate, multiple delays can occur [16]. Buffering on the other hand was an important part of an early pipelined design where there are non-deterministic flows between the pipeline stages [17]. There is a substantial literature on queueing theory which can be applied selectively to particular buffering problems [18].

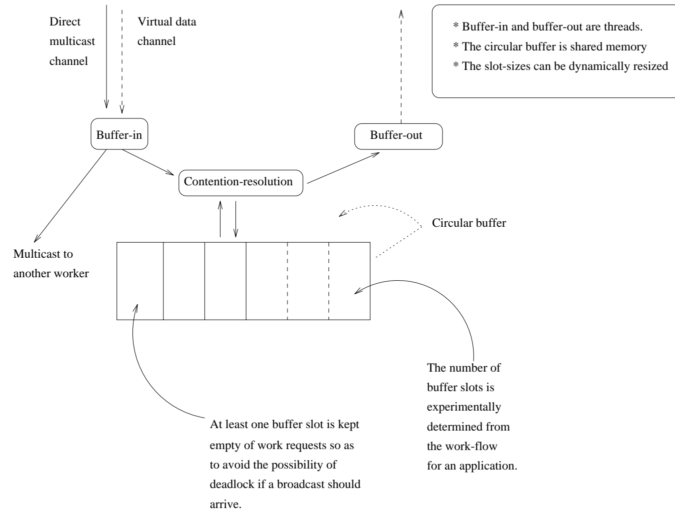

Figure 3: A Generic Buffer

In the template design, in-going (Figure 3) and out-going buffers service each application process. Additionally, buffers are placed between the stages of the pipeline. All buffers have the same generic form though they may differ in the number of buffer slots. To avoid internal data movements a shared address space is needed. The control of access contention is a standard problem in concurrency [19]. On transputers, software semaphores can be utilised by missing out any assembly language statement which might allow a context switch by the firmware. So that the buffer slot size should remain transparent to the message arrival size an initial buffer slot size is set which can be expanded by dynamic memory allocation on the arrival of too large a message. Separate buffer slots are kept for tags, otherwise there is a danger of the small slots needed for tags being expanded to provide for larger messages. At present, there is no means of reducing the buffer size if that size should, in the general case, turn out to be too large. Diagnostic software for memory usage (such as `Purify` [20] on sequential, possibly multi-threaded, machines) is not readily available for parallel machines, though one would need to expand the basic facilities of `malloc` debugging and array bound checking in order to check inter-task memory usage interaction. To avoid the possibility of deadlock if a series of multicast message were to arrive at asynchronous intervals, at least one extra buffer slot is provided over and above the number of messages sent

out at loading time. This method is a variant of an algorithm which is proven in [21]. Trust in the implementation was established by sending multicasts at randomly (with a uniform distribution) determined intervals against a backdrop of continuous message traffic. The number of multicasts at each distribution time was also randomly determined.

2.4 The Multicast Sub-System

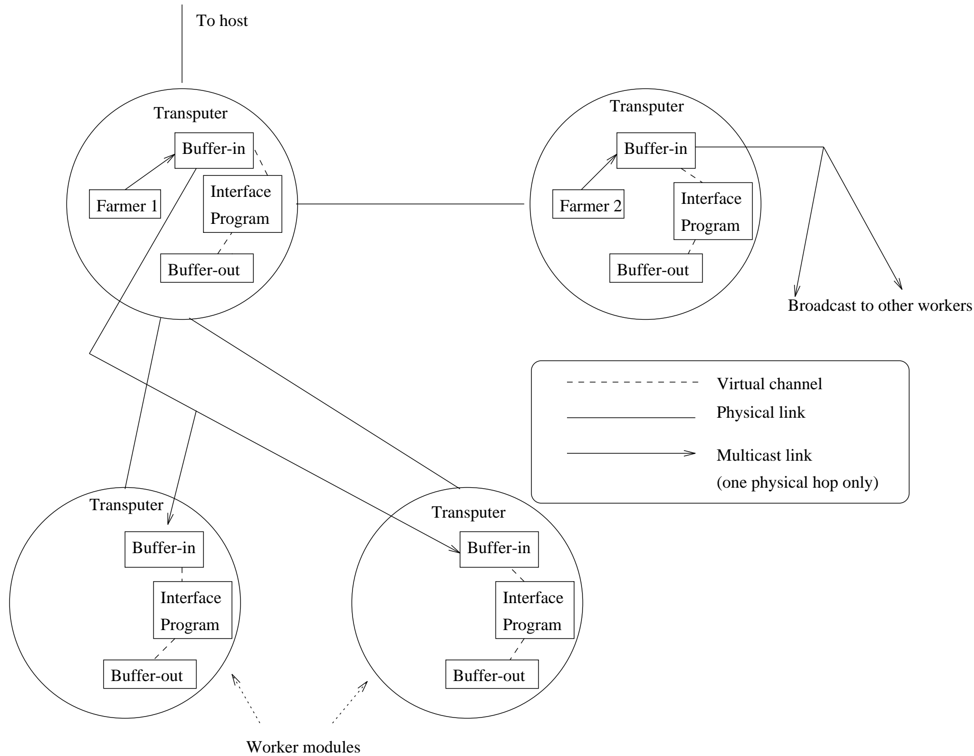

Figure 4: The Multicast Sub-System

In the expanded PPF methodology, multicasts can occur within each processor farm. Broadcasts (such as for initial parameter passing) obviously occur by inter-farmer communication. A multicast reduces message traffic, though in fact it breaks the paradigm of preventing worker to worker communication. When a VCS is employed it becomes necessary to specify point-to-point links and, if need be, set weightings to guide channel placement on the physical links available. The use of a tree-topology makes this easy to do but since a farm is not limited to a particular topology this is not generally the case. The tree topology also makes routing of multicasts trivial to arrange, since the in-going buffer need 'know' only how many ports it should send the multicast out to.

Other topologies may require the use of time-to-live (TTL) counters to avoid endless circulation. If multicasting is to be asynchronous then the possibility exists of a multicast being blocked because of a circuit between farmer and worker. In Section 2.3 an extra buffer slot is prescribed, but two further restrictions are needed to guarantee deadlock avoidance: a worker process can only act as a sink of multicast messages; and each work request can be met by one processed-work reply.
An alternative is barrier synchronization [22], but this option is rejected because of the overheads. Either all messages have to be absorbed by the farmer before broadcasting, in which case a per-worker process message count is needed, or a supervisory kernel is needed to wait for all workers to reach a barrier point.

2.5 The Monitoring Sub-System

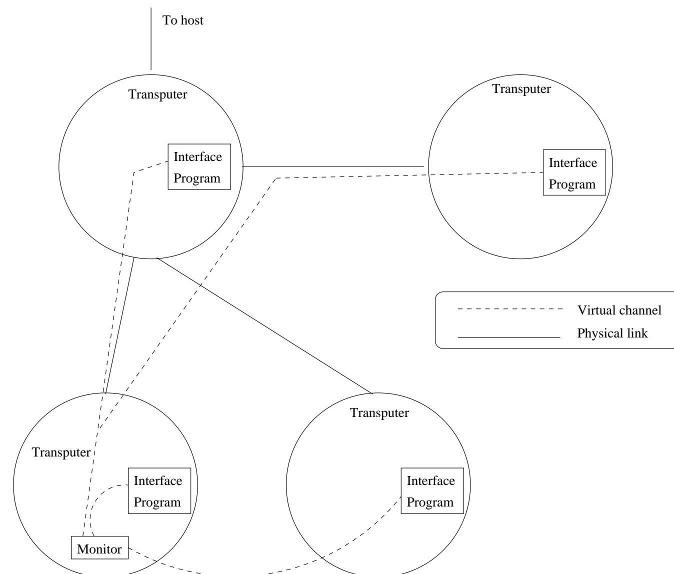

Figure 5: The Monitoring Sub-System

In the template version in which all application processes are placed on i860s, a monitor process synchronizes the clocks of all processes (Figure 5) by means of virtual channels. The monitor process can be placed on any processor, which avoids overloading the processor directly linked to the host. Virtual links are sufficient as the software global clock relies on the relative time differences between local clocks and a centrally maintained clock in order to synchronize to the central clock. Optionally at the start and definitely at the end, the monitor process synchronizes with the master data-farmer, at which time trace collection occurs. If transputer-based farmers are employed, the monitor process can be subsumed in the farmer. This is less satisfactory if the objective is to provide

transparent monitoring, as the communication primitives must directly make the trace. PICL [23] communication calls were mimicked for this eventuality. In fact, the PICL trace file format [24] is also used as this enabled us to test the post-mortem output on the ParaGraph visualizer [25]. The PICL 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 but which made it necessary to emulate multicasts by creating multiple message records in the trace file. Multicasts were stamped with the source and a 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 post-processing time.

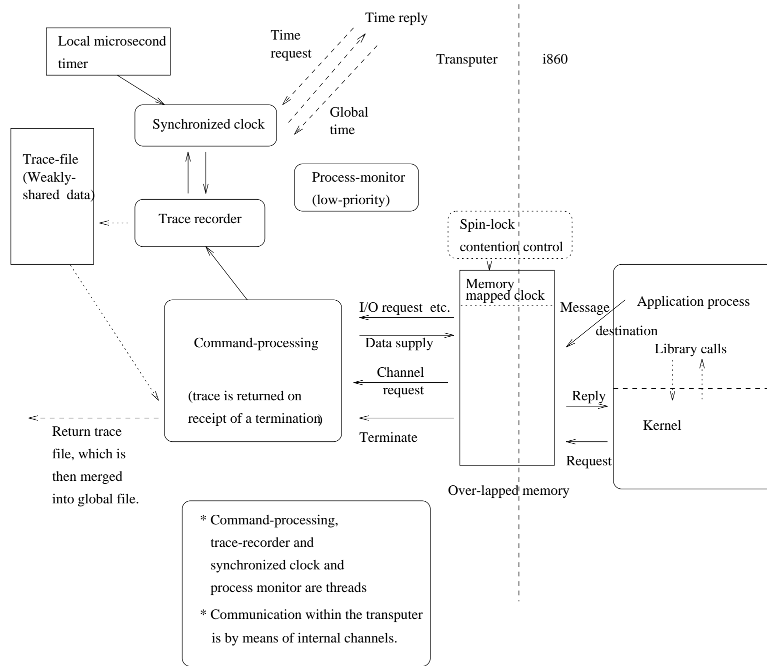

Figure 6: The Monitoring Layout

The interface program was enhanced with a trace recorder and synchronized clock process (Figure 6). 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 the monitor. An adjustment algorithm is used to compensate for local clock drift. On receipt of a command from the i860 application program, a trace record is also 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 run at high priority so as to reduce the interrupt latency, which for a single high priority process is 58 processor cycles ($2\mu s$). If need be an additional process is run at low priority [26] 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 processor'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. (The Paramid shared-memory data structure can be changed usually without disturbing the pre-compiled kernel routines.)

The synchronization algorithm, discussed in detail in [27], involves sending three messages. Unlike a generalized tracing system, initiation and subsequent maintenance of the clocks can be performed from a central point, the monitor process. In order to reduce disruption to the pattern of messages during normal working, all worker processes are synchronized at approximately the same time by a round-robin poll. The monitor computes the relative time difference between an averaged central time and the local times. The local clocks will receive the estimated difference at a later time. Between synchronization points, local clocks are adjusted by a local estimate of the relative drift between the clocks. Because crystal clocks are used linear drift is a good approximation (for experimental evidence see [28, 4]). The interval between synchronization pulses, before drift causes an error greater than the resolution of the intended visualizer, is calculated by an heuristic adaptation of a method due to [29]. No ordering errors were generated for the postcode or H.263 applications when the trace files were fed through ParaGraph's consistency checks though the runs lasted for several minutes. The time taken up by the clock synchronization messages was in the region of 5% for an application with mean per-message computation time of 0.1s for 1000 messages. Larger mean computation times (with the times forming a truncated Gaussian distribution) result in a lower percentage cost.

2.6 Other Features

Correct termination of the farm is necessary both for the collection of outstanding results and the gathering in of trace files. It is anticipated also that the farm may need to be reconfigured if the workload alters 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 [30]. The first function in the worker modules is reserved for termination.

Pipelines are developed in an incremental fashion by adding one farm at a time. To allow a farm to be developed in isolation, source and sink processes may be needed. These process stubs collect files that can be used as a comparison with a correctly-running sequential version. If a feedback path exists then this development cycle may not be possible, as was discovered with the

H.263 encoder. However, for complex systems it is strongly recommended that an incremental testing procedure is thought out before commencing.

The implementation of the guarded indeterminate communication operator in the firmware of the transputer favours those channels found first in order of textual declaration. To provide 'fair' selection of input channels, a channel shuffling routine is provided. Experiment shows that for demand-based farming, avoiding locking out worker requests does improve performance when requests are closely synchronized.

3 Analysing the Results from Two Real-Time Systems

Our goal in providing visualization as a built-in feature was to diagnose the communication behaviour for differing regimes of the parallel application. Accuracy is not of primary concern for a top-down approach to performance tuning, but it was apparent that an accurate trace could also serve to debug an application (at a future date). Presently, real-time debugging messages can be turned on at the interface program.

```
Master3: No. = 287 read filename=ts50335.l countAddrLine=4 postcode=CO111RU, address[0]=WELHAMS WAY
Sending out rank vector no. 297.
Filename ts50346.u No. of characters is 6 sequence no. is 297
Master3: No. = 288 read filename=ts50336.u countAddrLine=4 postcode=CO57PU, address[0]=HALL COTTAGES
Sending out rank vector no. 298.
Filename ts50347.l No. of characters is 7 sequence no. is 298
Sending out rank vector no. 299.
Filename ts50348.l No. of characters is 7 sequence no. is 299
Master3: No. = 290 read filename=ts50338.u countAddrLine=2 postcode=CO44QS, address[0]=CLOUGH ROAD
Master3: no. = 290 read filename=ts50339.u countAddrLine=3 postcode=CO106BA, address[0]=GREGORY STREET
Master3: no. = 291 read filename=ts50340.u countAddrLine=3 postcode=CO148UB, address[0]=CLAYS ROAD
Master3: no. = 292 read filename=ts50341.l countAddrLine=3 postcode=CO168XT, address[0]=MYTCHETT CLOSE
Master3: no. = 293 read filename=ts50342.u countAddrLine=2 postcode=CO28JB, address[0]=HYTHE QUAY
Master3: no. = 294 read filename=ts50343.l countAddrLine=3 postcode=CO64PL, address[0]=PLOUGH LANE
Master3: no. = 295 read filename=ts50345.l countAddrLine=4 postcode=CO57BQ, address[0]=CHURCH GREEN
Master3: no. = 296 read filename=ts50344.u countAddrLine=3 postcode=CO167DD, address[0]=PRIMULA CLOSE
Master3: no. = 297 read filename=ts50346.u countAddrLine=2 postcode=CO28QR, address[0]=QUEEN ELIZABETH WAY
Master3: no. = 298 read filename=ts50347.l countAddrLine=3 postcode=CO153AR, address[0]=STANWYN AVENUE
Master3: no. = 299 read filename=ts50348.l countAddrLine=4 postcode=CO124HA, address[0]=PARKESTON ROAD
Dictionary stage terminated.
Farmer1 `wall-clock' time is 25.922048 secs.
This is a throughput of 11.573160 postcodes/sec.
With a success rate of at least one match of 79.666668.
```

Figure 7: Postcode Diagnostics

3.1 A Postcode Recognition System

The postcode recognition application is intended to read automatically handwritten British postcodes in time for the envelopes to be coded with the correct postcode (using a phosphorescent dot code) as they reach the end of a mechanical conveyor belt. Diagnostic output from the tail end of the application from the final dictionary search farmer (master 3) is shown in Figure 7. The throughput easily meets the specification even when a trace is included (10.60 postcodes/sec. with trace and 11.57 postcodes/sec. without). There is no direct comparison because due to the number of trace records generated the

run for a trace was limited to 100 postcodes. If one of the processes, in this case the initial postcode image extractor, sends a relatively large number of short messages it will fill its trace buffer up quickly. At visualization time the display can be cluttered by messages from verbose processes, though this may be solved by post-processing the trace file.

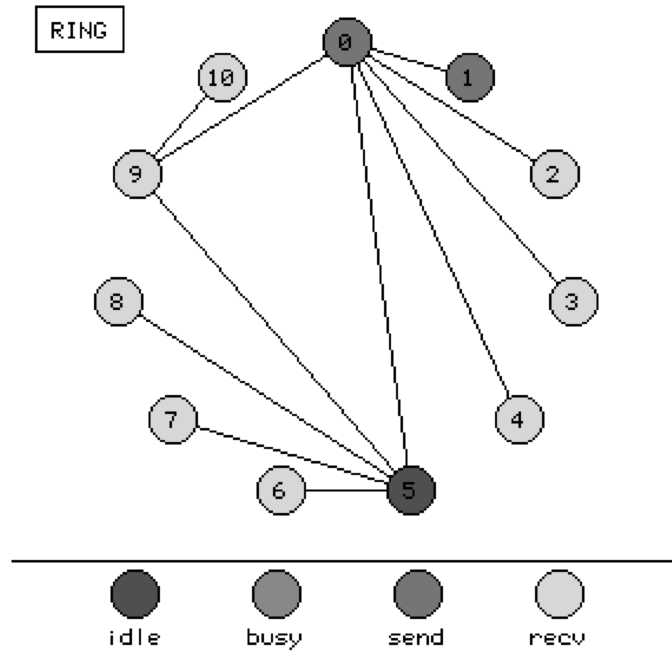

Figure 8: Animation of the Postcode Application

In the example chosen, there are three stages to the pipeline. The first pre-processing farm has three workers (processors 2, 3, & 4) and an initial image extractor (processor 1), with three workers (processors 6, 7, & 8) in the classification stage and one for the dictionary stage (processor 10). The arrangement is shown in a screen shot from ParaGraph's animation display (Figure 8). Note that, though the Paramid has eight modules, the three farmers are placed on transputers, giving eleven processes in all. Less clear is Figure 9, showing a time-space display at full magnification. Paradoxically, because the particular partition of the pipeline kept the processors running with limited idle time the display is cluttered. Had the buffer processes also been instrumented the impression would be cramped further. The trace does not show the operation of system software, which in some cases might be helpful. As communication is taking place simultaneously at various stages of the pipeline there is a good overlap. ParaGraph's display is not proportionate to the time taken by the application but is dependent on the display exigencies (as naturally some applications would take too long to display). The information from a display of

the type in the figure gives a broad-brush impression but for some applications traffic-flow statistics would need to be extracted from the trace as a basis for a complementary analytic analysis.

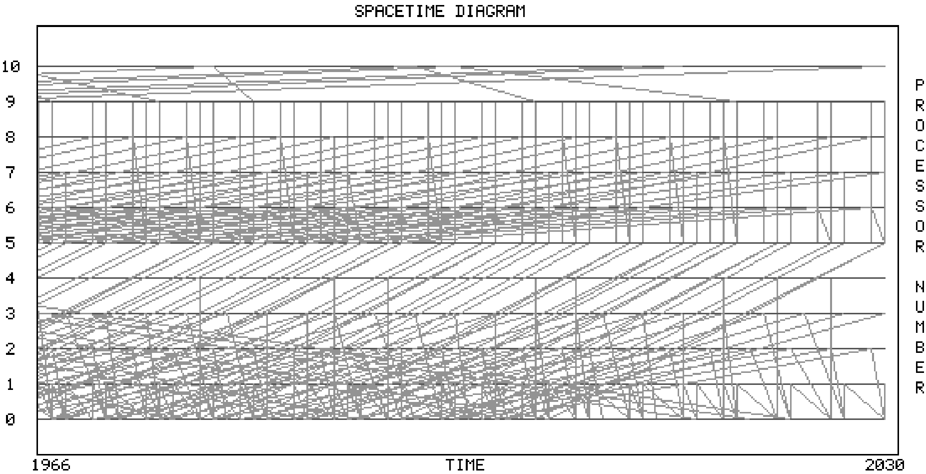

Figure 9: Time/Space Display of the Postcode Application

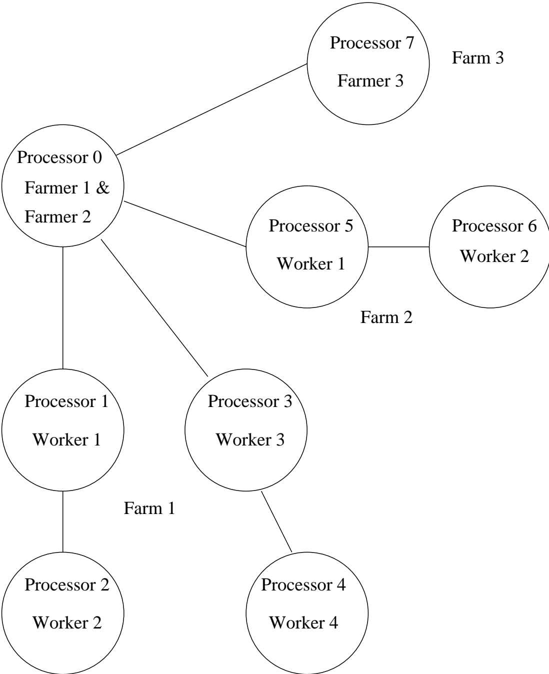

Figure 10: H.263 Physical Layout

3.2 The H.263 Video Encoder

The H.263 encoder, which is a standardized algorithm, is intended for real-time encoding of video frame sequences for very low bit-rate videophones or video conferencing [12]. The H.263 encoder was tested using three physical farms. However, because of the sequencing constraints imposed by the algorithm two of the data-farmers were combined into one, leading to the physical arrangement portrayed in Figure 10. Farm 1 has four workers, farm 2 has two workers and the worker on farm 3 is actually the farmer.

Figure 11 shows a portion of a trace for a typical run. The slope of the lines indicates the direction of travel. Unfortunately, for short messages even at highest magnification this is not apparent, as the first message to processor 7 shows. This is a tag message, with the body of the message following later. Running diagonally across the figure to meet at processor 0 are the messages from the first worker set, which farmer 1 cannot respond to until it has received work back (in the guise of farmer 2) from the second farm. The processor-monitor revealed a 30% idle time. The three farm arrangement was later abandoned in favour of a simpler setup.

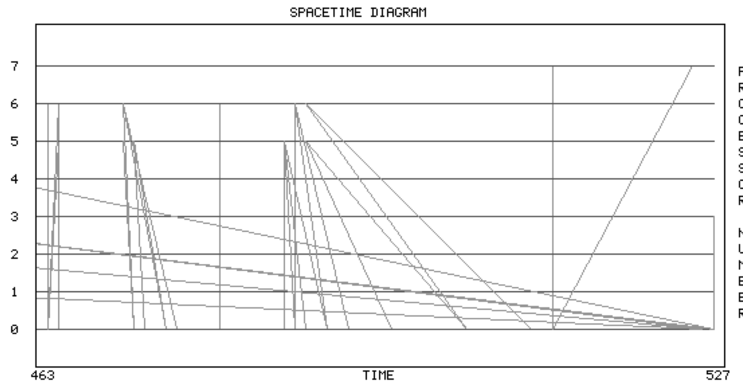

Figure 11: Time/Space Display for H.263

Because ParaGraph was intended for a hypercube machine it gives the principal topologies that can be embedded in a hypercube. The hypercube display (Figure 12) is helpful in the respect that non-hypercube communication is obvious on a colour screen. Thus, one can see whether a port would succeed. Though the range of displays and the display options offered by ParaGraph is very convenient and cost-effective, a more focussed approach would be helpful. ParaGraph is too unconstrained, offering the user limited guidance, which is a point also made in [31]. From the PPF perspective, construction of multiple tree topologies would be useful, showing message flow constrictions. If the displays were written in a language that was in the first instance interpreted (such as Java [32]) then display prototyping would be quicker. A further advantage is that the display formats could be user modifiable, without the extensive

parameterization of the X Window system.

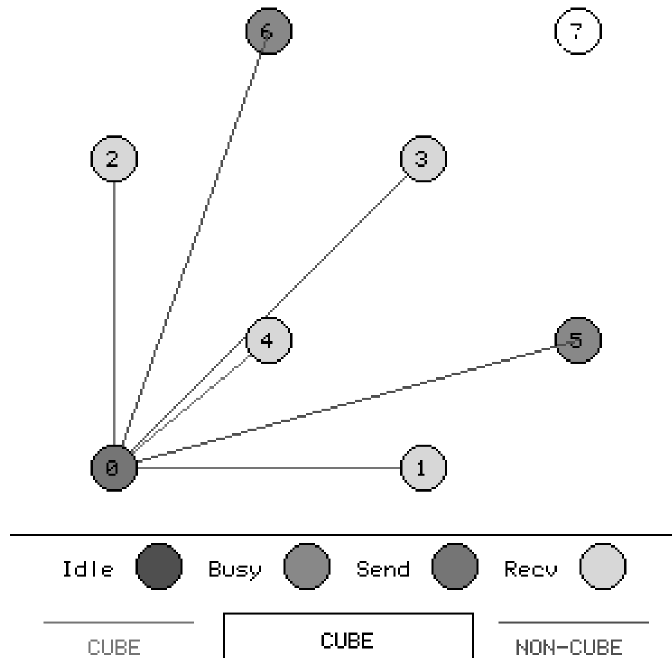

Figure 12: Hypercube Display Screen Shot

4 Conclusion

This paper has described the design of a farm template which includes software instrumentation as an integral part. The template is at a prototype stage and does not include measures to reduce the trace flow (such as semantic compression, throttling of the flow or user indication of events of concern). The principles behind the design of the template have been tried out in two applications involving a pipelined design. Interim results are shown in the form of trace displays. Data for the displays are collected by interface processes which are largely transparent to the application source code. The template has been constructed upon object-oriented design principles, which means that rather than produce generalized software all software consists of a collection of self-contained building blocks. The worker module with its controlled interface and private functions is not unlike earlier approaches, for instance the Actors' model [33]. It has been necessary to include a per-farm asynchronous multicast facility. Trace visualization is modified so as to add multicasts. The displays can be used to show communication traffic flow. The partition of the pipeline

is then adjusted accordingly. On the micro level the expected traffic flow is used to balance memory requirements for buffering, which is also an essential part of the template design. A customized visualizer, capturing the features of the PPF method, is a future intention. A template for the other two parallel decomposition paradigms also may be part of forthcoming work. Finally, if 'repeatable' runs can be achieved as part of a debugging cycle [34] then an integrated parallel debugger can share the timing data.

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] A. C. Downton, R. W. S. Tregidgo, and A. Çuhadar. Generalized parallelism for embedded vision applications. In A. Y. Zomaya, editor, *Parallel Computing: Paradigms and Applications*, pages 553–577. Thomson, London, 1996.

[2] 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.

[3] D. Feldcamp and A. Wagner. Using the Parsec environment to implement high-performance processor farm. In $28^{\underline{th}}$ *Annual Hawaii International Conference on System Sciences*, pages 212–221, 1995.

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

[5] J. Jiang, A. Wagner, and S. Chanson. Tmon: A real-time performance monitor for transputer-based multicomputers. In D. L. Fielding, editor, *Transputer Research and Applications 4*, pages 36–45. IOS, Amsterdam, 1990.

[6] Z. Segall and L. Rudolph. PIE: A programming and instrumentation environment for parallel processing. *IEEE Software*, pages 22–37, November 1985.

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

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

[9] M. Atkins. Performance and the i860 microprocessor. *IEEE Micro*, page 24, October 1991.

[10] D. A. P. Mitchell, J. A. Thompson, G. A. Manson, and G. R. Brooks. *Inside the Transputer*. Blackwell Scientific Publications, Oxford, 1990.

[11] A. D. Birrell. An introduction to programming with threads. Technical report, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, Cal., 1989. Research Report 35.

[12] H. P. Sava, M. Fleury, A. C. Downton, and A. F. Clark. A case study in pipeline processor farming: Parallelising the H.263 encoder, 1996. In this volume.

[13] A. Çuhadar and A. C. D. 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.

[14] W. N. Rea. Performance of task farming with transputers. In T. S. Durrani, W. A. Sandham, J. J. Soraghan, and S. M. Forbes, editors, *Applications of Transputers 3*, pages 792–797. IOS, Amsterdam, 1991.

[15] L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 943–972. Elsevier, Amsterdam, 1990.

[16] R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson. The interaction of parallel and sequential workloads on a network of workstations. In *ACM Sigmetrics Conference on Measurement and Modelling of Computer Systems*, pages 267–278. ACM, May 1995.

[17] S. Yalamanchili and J. K. Aggrawal. Analysis of a model for image processing. *Pattern Recognition*, 18(1):1–16, 1985.

[18] P. J. B. King. *Computer and Communication Performance Modelling*. Prentice Hall, New York, 1990.

[19] G. R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1):49–90, 1991.

[20] R. Hastings and B. Joyce. Purify. In *Winter Usenix '92 Conference*, 1992. Preprint.

[21] 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.

[22] H. F. Jordan. Problems in characterizing barrier performance. In M. Simmons, R. Koskela, and I. Bucher, editors, *Instrumentation for Future Parallel Computing Systems*, pages 185–200. ACM, New York, 1989.

[23] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A user's guide to PICL: a portable instrumented communication library. Technical report, Oak Ridge National Laboratory, Oak Ridge, TN, USA, August 1990. Report ORNL/TM-11616.

[24] 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.

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

[26] A. Bauch, T. Kosch, E. Maehle, and 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.

[27] M. Fleury, A. C. Downton, A. F. Clark, and H. P. Sava. The design of a clock synchronization sub-system for parallel embedded systems, 1996. In preparation.

[28] R. Cole and C. Foxcroft. An experiment in clock synchronization. *The Computer Journal*, 31(6):496–502, 1988.

[29] T. H. Dunigan. Hypercube clock synchronization. *Concurrency: Practice and Experience*, 4(3):257–268, May 1992.

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

[31] C. Pancake. Visualization techniques for parallel debugging and performance-tuning tools. In A. Y. Zomaya, editor, *Parallel Computing: Paradigms and Applications*, pages 376–393. Thomson, London, 1996.

[32] J. Gosling and H. McGilton. The Java language environment. Technical report, Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, Cal., 1995.

[33] G. Agha, C. Houck, and R. Panwar. Distributed execution of Actor programs. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 1–17. Springer, Berlin, 1992. Lecture Notes in Computer Science Volume 589.

[34] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(4):471–481, April 1987.