Providing Structure for Medium-grained Distributed Object-based Computation

R. Durrant, G. J. Sweeney, M. Fleury, A. C. Downton, and A. F. Clark
Dept. of Electronic Systems Engineering,
University of Essex,
Wivenhoe Park, Colchester, CO4 3SQ, U.K
fleum@essex.ac.uk

Abstract

Pipelined Processor Farms (PPF) are a generic parallel-processing structure suitable for continuous-flow, multi-algorithm applications. PPF has been redesigned for a multimedia workload which is multifunctional and multi-modal. A dynamic object-based processing structure is implemented which will map onto mobile stations. The PPF processing model with asynchronous message-passing in a multi-threaded environment is in effect a version of the actor paradigm which is a dynamic form of encapsulated system. Stable software facilities for distributed object-based systems which may include reflection are also needed. The supportive facilities of Java remote method invocation (RMI) are currently being used by PPF. PPF and RMI have been wedded together in a software toolkit, providing a development environment which includes performance prediction, analysis, and guided code generation.

1. Introduction

The advent of distributed-object systems allows distributed computation to work in an environment based on object request broker (ORB) middleware, standardised to a convenient object bus. The end result could be a dynamic, possibly mobile, computational resource. It would appear that distributed computation programming environments have lacked the software facilities to do all that they might wish while conversely middleware has lacked a clear computing structure to fit into. ORBs have therefore hitherto been used as the glue between large-scale systems, for example linking speech recognition with speech understanding. In the medium term, if distributed computation is to be supported the ORB model may shift in the direction of medium-grained applications, less tolerant of communication overhead. Though many of the facilities for effective medium-grained distributed computation are now in place, computing structures for using these facilities effectively are not. Design infrastructures for performance prediction and analysis of an application of distributed computation as well as software tools to construct such applications are required.

Pipelined Processor Farms (PPF) [8] is a medium-grained distributed computation system that has begun conversion to an ORB environment. PPF already employs distributed objects in order to prototype soft, real-time applications intended for eventual implementation on a variety of target hardware [14], thus anticipating industrial interest in networked embedded systems [11]. Examples of such applications, which involve a continuous flow of data, exist in video coding [7], vision [6], and image processing [13]. The advantages of employing an ORB are standardised software and ease in constructing a high-level design.

Our current experience, within a PPF software toolkit, has been with the Java remote method invocation (RMI) [27] ORB, which supports dynamic object loading. Standardisation allows Java components written by others to be incorporated as computational units into PPF, and for other systems to employ a PPF as a computational resource. The PPF is a container structure into which are inserted differing compute-intensive applications. The ability to put aside low-level communication issues allows more attention to be paid to fashioning a dynamic system of object creation and configuration, leading towards a ubiquitous actor processing paradigm for distributed computation. The extent this is possible depends on the facilities supported by a particular ORB, and the performance, which will depend on factors such as choice of protocol (HTML/IIOP/DCOM), levels of indirection, class loading latency, and security if reflectance is employed.

This paper provides a case study showing how structure can be captured within an integrated software toolkit. It is useful first to examine why PPF has taken the route it has.

2. Distributed computation

2.1. Some existing systems

Distributed computation's main aim is to extract performance from a shared resource. A number of communication harnesses have been developed to steal cycles from workstations either in a consensual fashion [19] or from within an open network [35]. These activities were opportunist and were not promoted as a general solution.

Various experiments crystallized in the Parallel Virtual Machine (PVM) [18] for parallel computation on Unix-based systems, distributed or parallel. PVM is not formally object-aware but did represent a way of introducing the actor paradigm. Unfortunately, this aspect of PVM has not been uppermost, as PVM sometimes became a ready-made message-passing facility onto which to layer other parallel programming paradigms, for example Linda, and virtual shared memory.

PVM is a large-grained version of the actors paradigm [3]. An actor is a dynamically-created distributed object which communicates through method-invoking messages, though PVM sends only data-bearing messages which can cause the invocation of functions. In PVM and actor systems, messages are queued in 'mailboxes' and serviced in asynchronous fashion. PVM messages are tagged, removing to some extent the restriction to one FIFO queue. An actor can change or reflect its behaviour and state upon receiving a message and may generate an agent thread or customer to carry out work on its behalf, though in PVM, only heavyweight processes are spawned. Reflection in the sense of changing the methods that can be executed is not available in PVM. The actor may have a table of references to a local set of actors or acquaintances, though again in PVM a global table is maintained remote from the actor by a set of interconnected nameserver dæmons.

The P-RIO system [25], bearing some similarity to PPF, employs PVM as its 'middleware' and it is suggested may move to the Message Passing Interface (MPI) [21]. However, MPI1 abandoned dynamic spawning for performance reasons which in itself is a further indication that many applications of PVM simply exploited the convenience of an existing message-passing harness without the underlying model. For scientific applications which P-RIO targets, a surrogate parallel machine on a network of workstations (NOW) may be the correct direction. However, many emerging multimedia applications are multifunction [11], for example concurrently processing video data, audio streams, and Web browsing, and impose significant real-time constraints on latency, throughput, processor efficiency, and synchronisation of concurrent threads of execution. The speed of processing may also change according to the available bandwidth, required quality of service, and data-dependent computational requirements. Additionally, with computing embedded in mobile stations with restricted resources the mode of computation may need to alter, for example the form of encryption or transmission protocol can change.

Applications that have migrated from implementation on a dedicated parallel machine to a distributed environment bring the static, performance-oriented model of computation. In a distributed system, a flexible response to changing resources (and resource failure) is needed which is why we seek to keep and extend the underlying model of PVM computation. As ORB middleware has the potential to do all that PVM could do and more, this is the preferred route though the granularity trap is present for an ORB as it has been for PVM.

2.2. The PPF design pattern

The PPF design pattern [15], which is intended for applications with a continuous flow of data, consists of a pipeline of data farms each one of which can incorporate internal parallelism, Fig. 1. A pipeline with a single backplane [23] has the advantage that it can map onto a variety of hardware architectures. Each PPF has a farmer responsible for distributing work to a set of workers which may be on a subsidiary network. The deployment of worker tasks is determined by the relative per-stage workload, the desired global throughput, and the desired pipeline traversal latency. Feedback or feedforward loops enable synchronous constraints to be introduced but for practicality only one such loop may occur at each input site. The PPF pattern is an extension and generalisation of a number of data-farming schemes: some NOW-based [31]; some based on dedicated multicomputers [38]; some constructed as algorithmic skeletons [28]; and some implicitly invoked in parallel extensions of C++ [20].

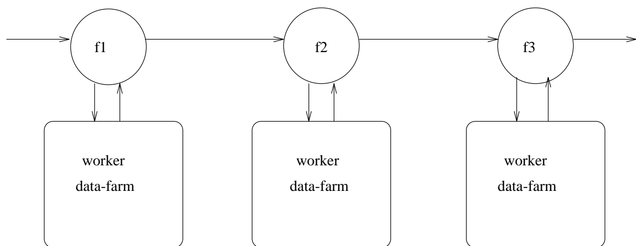

Figure 1. PPF design pattern

2.3. PPF before ORBs

The PPF programming model prior to the advent of ORBs [17] was close to the actor paradigm, though it originated as an asynchronous relaxation of CSP. Features such

as reflection and spawning of customer threads were not present in this version of PPF. As a 'C' compiler is the common factor across many real-time systems, the design not the implementation was actor-aware. Unlike PVM, PPF utilised multi-threading, with implementations for Solaris 1 light-weight threads library, and VxWorks real-time operating system. With the advent of kernel-level threads in Solaris 2 and WindowsNT, PPF multithreading could be efficiently deployed. A PPF worker module had a set of methods which can be invoked by messages from the farmer module. Messages were queued in FIFO buffers controlled by threads, though access to large data structures was managed by semaphores. Multicast messages across a farm and latterly urgent messages were also supported.

Communication in distributed PPF implementations has been through the socket application programmers interface (API), whereas kindred systems like PVM employ the semantics if not the actuality of remote procedure call (RPC) with data marshalling and exchange data representation (XDR). Fig. 2 [9] shows client invocation of a remote method via a proxy object. Unlike PVM, which provides a support network of user-level dæmon processes, and unlike RPC, PPF has hitherto not had supportive facilities. PPF has been implemented in homogeneous environments, though a backplane bus can provide enhanced bandwidth over the worker network in order to avoid bottlenecks. Where an application was transferred between different hardware architectures a new implementation of the model occurred. Conversely, RPC is intended for heterogeneous environments, but requiring system-level dæmons, a name-server to bind servers, and a stub compiler (`rpcgen`). Because RPC is synchronous, in the sense that the caller stalls until the results return, PVM preferred a socket-based system of communication. Unfortunately, it can be difficult to employ sockets in a generalised fashion, requiring individualised implementations.

In an ORB-based PPF, we seek the same structure as offered before by PPF but with additional dynamism.

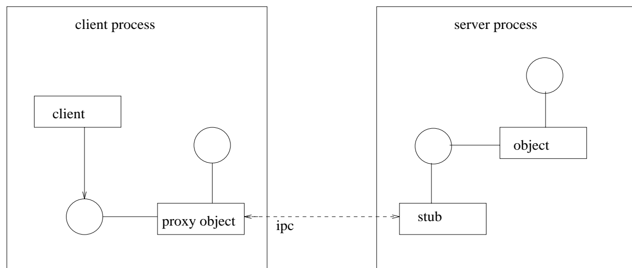

Figure 2. RPC paradigm

2.4. Suitability of the Actor paradigm

There are practical problems with a true actor model as advocated in [3] and which is shown in Fig. 3 [22], with two possible replacement behaviours, staying the same or changing. For example, in ABCL [39], an extra wait state is introduced which allows selection of specific messages from a queue. Without such modifications there is the possibility of unbounded delay. Message queues are assumed to be infinite, while in PVM message queues are implemented by a complex system of dynamically-created buffers. PPF is constrained by its design pattern and by the templating system, thus avoiding the implications of a complete actor model. The constraint is an aid to correctness checking. Message-passing as a model of parallelism [5], which is an intermediate step between actors and PVM, is in the Cantor language a fine-grained model of computation. In general, there needs to be guidance to the developer as to a practical granularity level for the architecture/hardware for which the application is targeted. Performance prediction, which can be cross-architectural, is an essential adjunct to a distributed computation development environment.

A true actor system may be more appropriate to a multi-agent adaptive system, for example through the Merle IV language [10], which changes behaviour frequently and intrinsically, while distributed computation is served by a modestly dynamic system.

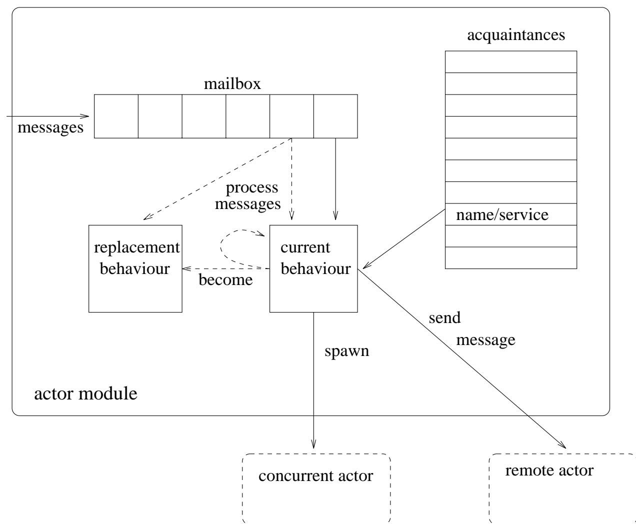

Figure 3. The Actor model

3. PPF in an ORB environment

3.1. PPF templates

The PPF structure has been captured in a set of templates[1] in which the worker module is designed as an actor and the farmer module acts as a work scheduler, encapsulating the parallel structure within the template. Implicit methods of encapsulating parallelism such as path expressions are possible. In opting for an explicit means – an object manager – PPF is similar to a number of distributed-object programming environments notably ALPS [37]. Intermediate mechanisms such as the synchroniser [2] are possible. The PPF template design and implementations have instrumentation built-in so that the performance can be monitored. When used to construct prototypes and on distributed systems, the trace data is stamped by a logical clock, while for performance tuning and on multicomputers, a global clock has been implemented. The ability to build in an infrastructure into a distributed object can be extended: to a fault-tolerance monitoring system responding to heart-beats from a watchdog object; and to reconfigurability. Applications with a time-varying workload will require changing the relative allocation of workers between the stages of a pipeline, though clearly this raises many organisation issues. In general, PPF now favours building in as much structure as possible anticipating future needs. The need to predict performance has resulted in a PPF toolkit into which templates are now embedded.

3.2. PPF toolkit

Though intended as a performance prediction graphical toolkit in fact the PPF Analyser, Predictor Template Toolkit (APTT) [16] has become an integrated system for distributed computation. The construction of APTT has made us aware that a distributed computation system should be integral to a supportive framework and not the other way round. In other words, the prediction and analysis support is not an afterthought, but an essential way of knowing the behaviour of the system. APTT is written in Java, raising the issue of how to implement the PPF programming model from within APTT. Java is eminently portable and intrinsically multi-threaded [26] allowing PPF buffering, communication, and computation threads. Current work is aimed at PC networks running Windows NT. Java RMI is an extension of RPC. There is a stub compiler (`rmic`), a security manager, server proxy objects (skeletons), and exported client stub objects. Unfortunately, from Section 2, unmodified RPC is synchronous and asymmetrical, being client-server. However, as client-server is a logical arrangement, in data-farms worker and farmer can alternate their role in the client-server relationship to remove the asymmetry.

[1] The term 'template' refers to a way of guiding the construction of PPFs and not in the C++ sense as a generic class.

APTT supports a design cycle, Fig. 4, aimed at producing pipelines with a single primary data flow constructed from a set of parallel data farms. Sequential bottlenecks are masked by single processor stages. While the core functionality is provided by APTT, the normal supportive software tools can be employed, examples being the SNiFF class browser, [34] Quantify [29] profiler with object code instrumentation, and Purify memory debugger (though for C++ not Java). It is assumed in Fig. 4 that a significantly-sized (10k lines-of-code and above) legacy application is split into code sections and indeed a number of implementations have taken place in this manner, for example a face identification pipeline. However, it is possible to directly enter code and message structures into the current template builder.

Profiling the development code enables a tentative allocation of functions to different stages of the pipeline based on mean timing ratios. Normally, the potential for loop parallelism will determine the per-stage weighting of worker processes. A set of inner-loop timings enable distribution fitting by statistical or other means. Identifying a job as one iteration of the inner loop, jobs can then be grouped into tasks. Note that the grouping results in a change in distribution as the addition of distributions is a convolution.

Fig. 5 shows a snapshot of the predictor running a simulation of a handwritten postcode recognition system [6]. The pipeline backplane occupies the main window with details of the stage activity such as buffer and processor usage available from subsidiary windows. Processor activity is shown using colour by analogy with stop/go displays. The communication arrows change colour from black, through red to white to highlight 'hotspots'. The arrows also widen and contract. However, the cumulative mean bandwidth, not instantaneous bandwidth is displayed in order to avoid an over-animated display. The colour scaling is adjustable to centre on critical data rates as otherwise the variation across the whole bandwidth range is too low to show up. Pipeline traversal latency is also indicated in a persistent display. Component jobs are marked off at task boundaries, with the task latency determined by the slowest job. Though persistent displays convey more information, they need to be balanced with features marking progress, which is why the processor activity diagram and message motion arrows are included.

3.3. The PPF template system

RMI has a three-layered software architecture with transport level protocol (TCP/IP circuit-oriented) handling at the lowest level, and choice of communication model, (unicast point-to-point), at the intermediate level below the application layer. Because RMI is specialised for Java applications

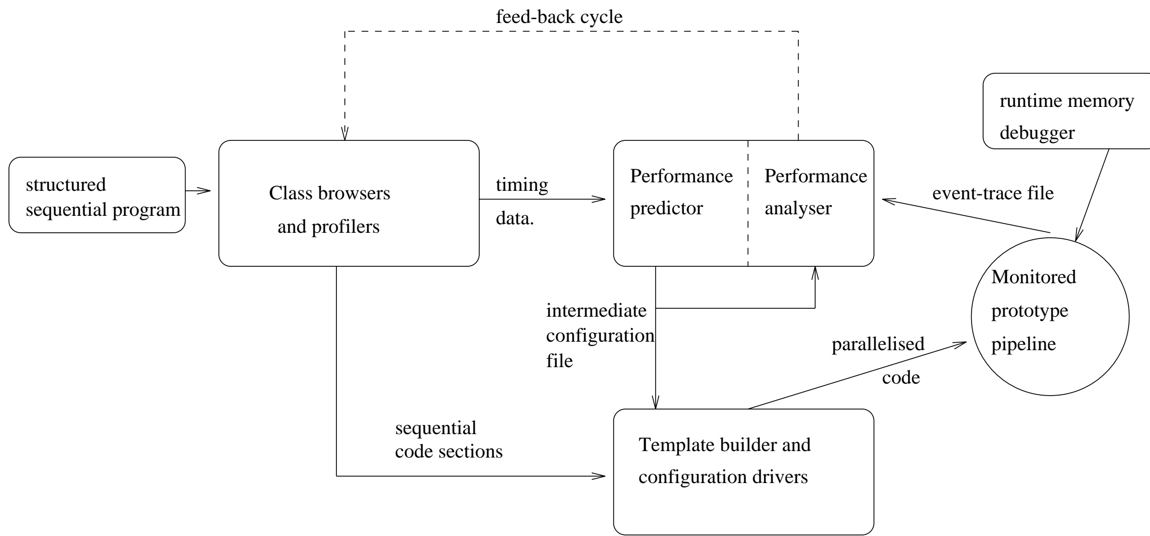

Figure 4. Design cycle

remote object loading, dynamic class and stub loading are supported in PPF.[2] Polymorphic loading can occur, whereby a sub-type of a declared type is loaded. On each processor, a registry must be run to act as a nameserver for remote object method invocation. The registry is not persistent as remote object garbage collection takes place by means of reference counting. These facilities are nearer to the dynamic, possibly mobile, model of distributed computation that is the favoured course for PPF.

In the PPF template, the Java vector class, which can be made to transparently grow and shrink as work requests are serviced, removes the need to provide concurrent access management. To avoid synchronous delay the implemented design requires the data farmer to poll the remote worker processes acting as servers. Conveniently, this fits a polling queueing model of performance estimation. Java's preemptive priority-based thread scheduling can be adapted to provide a responsive structure.

In Fig. 6, the threads (ovals) have high priority given to communication facilitating threads while worker tasks threads, each with a local message queue, run at reduced priority. In fact, each module contains a number of worker task instances which can either provide different functionality or multiple instances of the same functionality. Parallel slackness [36], whereby if one instance is blocked another instance of the same worker task can take over, is a method of masking communication latency originally introduced to allow one architecture to simulate another. There are limits to the number of threads that can be gainfully employed [1]. Therefore additionally, buffering of messages containing work requests can mask latency. Allocation of work is arranged by a work manager, which provides an ordering structure. Scheduling of threads is divorced from ordering through a highest priority user-level scheduling thread. The scheduling thread suspends itself for a given time interval. When it is rescheduled by the Java operating system, the scheduling thread adjusts the priority of the worker task threads before suspending again. The RMI server includes the skeleton stub created by `rmic`.

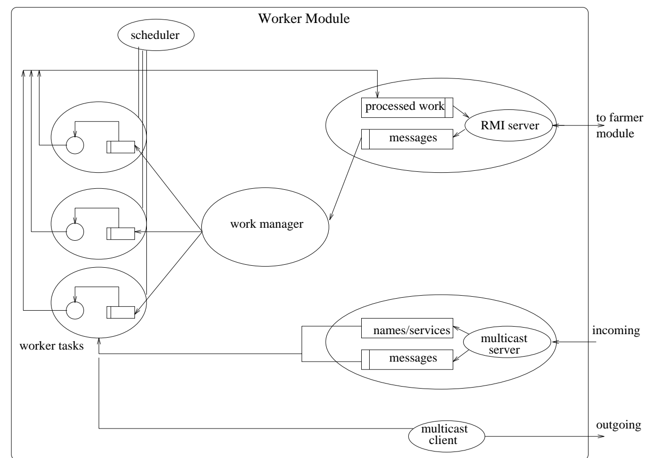

Figure 6. Java RMI Worker Module

On inception, worker modules multicast their names and

[2] Dynamic class loading is also a feature of mobile agent systems constructed under RMI.

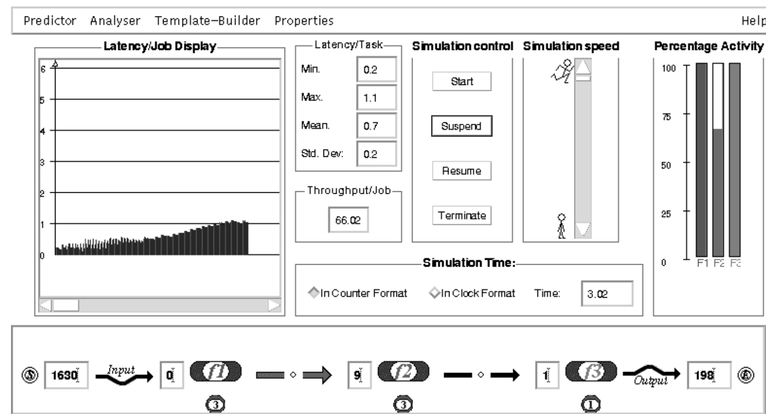

Figure 5. APTT predictor window

associated addresses. Associated with each name can be the computational services offered. Multicasting is provided through the Java socket API. The intended multicast recipient is the farmer module. If and when a worker module becomes available then the farmer module sends out requests for work processing. Multicast names and services are also collected by worker modules, allowing co-operative, intra-farm, processing to take place. A rather limited example of co-operative processing in work farms is the exchange of border regions during image processing. Multicasts are employed for distribution of start-up parameters. In some applications, such as the H.263 hybrid video encoder [30], global data is distributed after every round of processing, e.g. the current quantisation level.

In some actor systems [39], urgent messages are sent as a way of avoiding blocking at the head of the queue. There is also a scheduling issue when several different functions are performed by a worker module if a FIFO message queue is enforced. However, this is more of a problem for a hard real-time system than it is when deadlines are less strict in a soft system.

Once sufficient work messages have been sent out, the farmer module polls the worker processes for processed work. There are a variety of servicing schemes [33] for arranging polling such as waiting for work before proceeding, or waiting for a set number of processed items, but these are beyond the scope of this paper. On distributed-memory multicomputers, a farmer would not normally poll for work. Instead, worker processes request work on completion. The advantage of casting the relationship in this way is that the worker processes can balance their own workload. However, polling has been applied to bandwidth sharing by dynamic link-swapping in an otherwise fixed connection system [12].

As an alternative to work requests, an object not designated as remote can be serialised or reified and sent as a standard application parameter from the farmer module to the remote server, the worker module. This allows the remote worker after suitable casting to execute or revert the methods of that object. Due to polymorphism it is not necessary for the receiving worker to be aware of the implementation of a method. In this way, the worker module acts as a meta-object acting transparently to the methods embedded in the worker tasks which are the object itself. At the moment, this is only possible because RMI is confined to Java and hence the internal representation of objects is fixed. In a similar way, stubs, which are designated as remote, can be dynamically sent to a farmer module, the client, thus aiding in configuration of the PPF. These reflection facilities have previously been available only in actor systems, e.g. ABCL, Merle IV, and 3-Lisp, designed before the onset of mobile stations. In effect, reflectance is a form of code reuse or inheritance for dynamic systems.

The template design decouples the (farmer) client from worker (server) by polling, moving away from a synchronous model of computation. Parallel slackness and buffering are user-level techniques employed by us to reduce communication latency thus widening the class of applications suitable for RMI. Though we have called our design a template in fact a more apt description would be a software component, in the sense that a component encapsulates dynamic behaviour in contrast to objects which are passive. The correct features of any component design might be arrived at through categorical data theory [32]. Ideally, any object can be the subject of the dynamic structure provided by the component. The performance implications of using that object should be available through performance prediction. Complexity theory is one way to make an estimate. Alternatively, as we have done in APTT, a stochastic estimate based on queueing theory or simulation can be made,

which can subsequently by extrapolated to other machines, with scaling according to appropriately benchmarked parameters.

3.4. Code generation

Code generators (CG) [24] enable application code to be entered in a schematic manner in a graphical format. [4] for Linda applications is an example of a code generator which is an enhanced text editor. The APTT code generator allows a graphical PPF to be constructed in a guided fashion from a menu of farmers and workers. Each module has a set of properties associated with it, which appear in a pop-up menu. An example property would be a label identifying which application code extract is to be executed. Snippets of code are entered through a text editor allowing subsequent changes. The CG end-user is presented with a dataflow model. A set of triggers are needed to start processing. The layout of messages is entered in a separate part of the CG editor. Once the construction process is complete there are commands to compile and run. Since the entry process is high-level, the end-user is not confined to RMI as the Java snippets could be embedded in (say) MPI implemented in Java. In fact, there is nothing about the structure that confines data-entry to Java, as 'C' could be entered into the editor.

3.5. Adapting to networked computation

We have given some thought to inception and synchronisation of a networked PPF computation. One plan is to start with one master farmer module which is always present and a pool of farmer and worker modules which can be chained together. As currently implemented, the first farm is formed dynamically to save time. When processed work is ready to go on to another PPF stage, the first farmer inspects for farmer multicasts and selects a ready farmer. A more flexible arrangement is shown in Fig. 7. The pipeline manager (PM) is contacted from a pool of PPF elements on a known port/multicast address. The PM then ascribes the 'gender' (either farmer of worker) to a PPF element. Having set-up the PPF, Fig 8, the PM reflects its behaviour to become a PPF monitor. Each instance of a farmer is characterised by the type of data manager: at the start and end of a pipeline a file data manager is assigned, while in intermediate stages a message reception and relay manager is needed.

As described in Section 3.3, RMI can pass objects as parameters to a server provided that the objects are serialisable, allowing in effect PPF to dynamically load code onto a worker, though there are issues of casting. For the purposes of automatic configuration, objects designated as remote can be passed from server to client though in this case only the stub object arrives.[3] In principal, an application can be assembled on a PPF from a central source, or the application can be extended by passing stubs (having first sent naming information). A further issue to address is security, as code distribution is much like virus distribution.

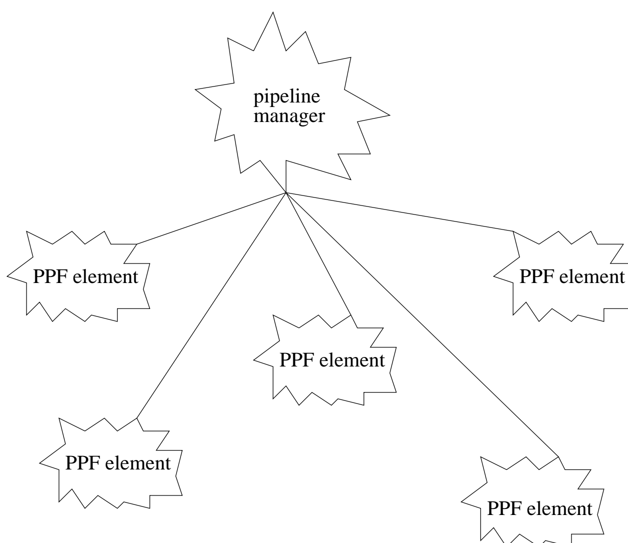

Figure 7. Inchoate PPF

4. Conclusion

Pipelines of Processor Farms (PPF) is a schematic design method for distributed computation. PPF is concerned with real-time multimedia processing. ORB-style processing is a continuation of a direction PPF had already taken towards encapsulated parallel systems. However, we have altered the way the RMI ORB would normally be used to suit medium-grained processing, widening the problem domain that can be tackled. A core design pattern has been presented for which application studies already exist. We have embedded the distributed computation design within an integrated toolkit, APTT, to include performance prediction, analysis, and code generation. If PPF is to adapt to the new mobile computational environment then dynamic pipelines will be needed. The fact that PPF modules are based upon a modified actor model makes that conversion easier. Dynamic pipelines are possible under RMI to some extent, but careful thought needs to be given to what design patterns and middleware facilities are needed.

[3] As the stub object is strictly constrained, it becomes necessary to provide a customised security manager.

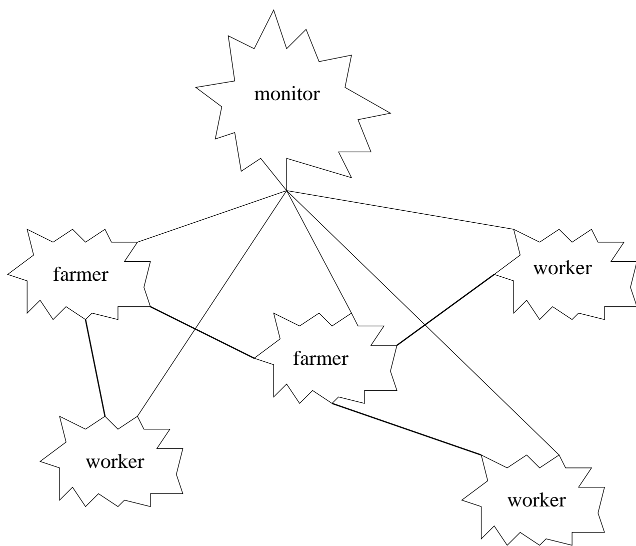

Figure 8. PPF after configuration

References

[1] A. Agarwal. Performance tradofss in multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[2] G. Agha, S. Frolund, K. W. Y., R. Panwar, A. Patterson, and D. Sturnman. Abstraction and modularity mechanisms for concurrent computing. *IEEE Computer*, pages 3–13, May 1993.

[3] G. A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT, Cambridge, MA, 1986.

[4] 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.

[5] W. S. Athas and C. L. Seitz. Multicomputers: Message passing concurrent computers. *IEEE Computer*, 21(8):9–24, August 1988.

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

[7] A. C. Downton. Generalised approach to parallelising image sequence coding algorithms. *IEE Proceedings Part I (Vision, Image and Signal Processing)*, 141(6):438–445, 1994.

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

[9] J. Feghhi. *Web Developer's Guide to Java beans*. Coriolis, Albany, NY, 1997.

[10] J. Ferber and P. Carle. Actors and agents as reflective concurrent objects: A Mering IV perspective. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(4):1420–1436, 1991.

[11] J. Fleischmann and K. Buchenreider. Integrated engineering. *IEEE Computer*, 32(2):116–119, 1999.

[12] M. Fleury and A. F. Clark. Performance prediction for parallel reconfigurable low-level image processing. In *International Conference on Pattern Recognition*, volume 3, pages 349–351. IEEE, 1994.

[13] M. Fleury, A. C. Downton, and A. F. Clark. Karhünen-Loève transform: An exercise in simple image-processing parallel pipelines. In C. Lengauer and M. Griebl, editors, *EuroPar'97*, pages 815–819. Springer, Berlin, 1997. Lecture Notes in Computer Science 1300.

[14] M. Fleury, A. C. Downton, and A. F. Clark. Co-design by parallel prototyping: Optical-flow detection case study. In *IEE Colloquium on High Performance Architectures for Real-Time Image Processing*, pages 8/1–8/13, 1998.

[15] M. Fleury, N. Sarvan, A. C. Downton, and A. F. Clark. A parallel-system design toolset for vision and image processing. In D. Pritchard and J. Reeve, editors, *EuroPar'98*, pages 92–101. Springer, Berlin, 1998. Lecture Notes in Computer Science 1470.

[16] M. Fleury, N. Sarvan, A. C. Downton, and A. F. Clark. Methodology and tools for system analysis of parallel pipelines. *Concurrency: Practice and Experience*, 1999. In press.

[17] M. Fleury, H. P. Sava, A. C. Downton, and A. F. Clark. A real-time parallel image-processing model. In $6^{\underline{th}}$ *International Conference on Image Processing and its Applications IPA'97*, volume 1, pages 174–178, 1996.

[18] G. A. Geist and V. S. Sunderam. Network-based concurrent computing on the PVM system. *Concurrency: Practice & Experience*, 4(4):293–311, 1992.

[19] D. Gelernter and D. Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with Piranha. In *International Conference on Supercomputing*, pages 417–427. ACM, 1992.

[20] A. S. Grimshaw, W. T. Strayer, and P. Narayan. Dynamic, object-oriented parallel processing. *IEEE Computer*, pages 33–46, May 1993.

[21] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT, Cambridge, MA, 1994.

[22] D. Kafura and J.-P. Briot. Actors and agents. *IEEE Concurrency*, 6(2):24–29, 1998.

[23] 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.

[24] T. G. Lewis. Code generators. *IEEE Software*, pages 67–70, May 1990.

[25] O. Loques, J. Leite, and E. V. Carrera E. P-RIO: A modular parallel-programming environment. *IEEE Concurrency*, 6(1):47–57, January-March 1998.

[26] S. Oaks and H. Wong. *Java Threads*. O'Reilly, Cambridge, 1997.

[27] R. Orfali and D. Harkey. *Client/Server Programming with Java and CORBA*. Wiley, NY, 1997.

[28] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, London, 1998.

[29] Pure Software Inc., 1309 South Mary Ave., Sunnyval, CA. *Quantify User's Guide*, 1992.

[30] H. P. Sava, M. Fleury, A. C. Downton, and A. F. Clark. A case study in pipeline processing farming: Parallelising the h.263 encoder. In *UK PARALLEL '96*, pages 196–206. Springer, London, 1996.

[31] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. The Enterprise model for developing distributed applications. *IEEE Parallel and Distributed Technology*, pages 85–95, August 1993.

[32] D. B. Skillicorn. Structured parallel computation using categorical data types. In A. Y. Zomaya, editor, *Parallel Computing: Paradigms and Applications*, pages 145–161. International Thomson Computer Press, 1996.

[33] H. Takagi. Queueing analysis of polling models. *ACM Computing Surveys*, 20:5–28, 1988.

[34] TakeFive Software GmbH, Stichting Mathematisch Centrum, Amsterdam, the Netherlands. *SNiFF+ Release 2.2 User's Guide and Reference*, 1996.

[35] M. M. Theimer and K. A. Lantz. Finding idle machines in a workstation-based distributed system. In *Proceedings of the* $8^{\underline{th}}$ *International Conference on Distributed Computer Systems*, pages 112–122. IEEE, 1988.

[36] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[37] P. Vishnubhotia. Synchronization and scheduling in ALPS objects. In $8^{\underline{th}}$ *International Conference on Distributed Computing Systems*, pages 256–264, June 1988.

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

[39] A. Yonezawa, editor. *ABCL: An Object-oriented Concurrent System*. MIT, Cambridge, MA, 1990.