Realising a Concurrent Object-Based Programming Model on Parallel Virtual Shared Memory Architectures*

Michael Fisher
Department of Computing
Manchester Metropolitan University
Manchester M1 5GD, U.K.
EMAIL: M.Fisher@doc.mmu.ac.uk

John Keane
Department of Computation
UMIST
Manchester M60 1QD, U.K.
EMAIL: jak@co.umist.ac.uk

Abstract

In this paper, we investigate the suitability of parallel architectures for the realisation of a novel object-based computational model [21] encapsulated within programming languages such as [11]. This model incorporates objects, groups, broadcast message-passing and asynchronous execution. As such it provides a high-level, architecture-independent representation for a variety of concurrent systems. The class of parallel architectures which we consider are logically shared but physically distributed memory systems.

1 Introduction

In this paper, we investigate the suitability of certain parallel architectures for the realisation, of a novel object-based computational model [21, 5] encapsulated within programming languages such as [11]. This model incorporates objects, groups, broadcast message-passing and asynchronous execution.

The programming model we consider is not new. It has been used both in Distributed Operating Systems [5] and Distributed Artificial Intelligence [21]. However, rather than being limited to those areas, we believe the model to be both general purpose and intuitively appealing. In particular, we have developed a programming language based on this model, which utilises executable temporal logic [11], and have shown how a variety of distributed and concurrent systems can be specified and implemented using this approach [8, 9, 10]. Further, we have outlined how the model can be used to utilise massive parallelism in particularly computationally intensive application areas, such as theorem-proving [12].

*This work was partially supported by UK SERC grant GR/J48979.

Thus, the model provides a high-level, architecture-independent representation for a variety of concurrent systems.

The basic elements of the model are objects. These are considered to be encapsulated entities, executing asynchronously to each other, and having complete control over their own internal behaviour. These objects, as well as performing their own internal execution, are able to

1. spawn new objects,
2. communicate with other objects, and,
3. terminate.

The mechanism of communication between objects is *broadcast* message-passing. However, as broadcast communication can *sometimes* be both inefficient and unwieldy, the notion of object *groups* [21] is central to the model, being useful both for restricting the extent of broadcast messages and for structuring the object space.

As two of the founding principles of the model are broadcast communication and asynchronous execution, an obvious question is: "why do we discard 'standard' approaches to communication and concurrency (such as point-to-point message-passing and synchronous concurrency) used in other models?". We consider point-to-point message passing to be *very* restrictive, particularly when developing open systems. In many cases, the particular receiver for a message is not known. Also, being tied to a particular receiver risks deadlock if that receiver 'dies' (i.e. its processor fails).

Asynchronous concurrency is potentially very powerful. Yet, although there are models which utilise this form of concurrency, such as Actors [14, 2], we believe that many systems are not expressive enough (both within objects and when structuring the object space) to exploit this inherent power. The Actor model itself, though based on asynchronous concurrency, not only uses point-to-point message-passing, but is also heavily biased in favour of fine-grained parallel computation [1].

Thus, we believe that our model, consisting of groups of asynchronously executing objects, communicating via broadcast message-passing, is both flexible and powerful. As such, it provides a high-level, architecture-independent representation for a variety of concurrent systems.

This paper is not intended to provide a detailed description of a particular language based upon this model. Rather, we consider the general problems involved in designing an implementation for a particular class of parallel architectures, namely Virtual Shared Memory (VSM) architectures. The paper is structured as follows. In §2 we present the characteristics of the programming model in more detail, while in §3 we outline several of the applications that have been developed based on this model. In §4, we review the features of the particular target architecture that we are investigating, namely VSM, while in §5 we consider the problems of implementing our chosen programming model on this class of architectures. Finally, in §6, we present conclusions and discuss our future work in this area.

2 The Programming Model

The basic tenets within our programming model are that

1. everything is an object,
2. all objects are concurrently active,
3. the basic mechanism for communication between objects is broadcast message-passing, and,
4. objects occur in *groups*, which restrict the extent of broadcast message-passing and provide structure within the object space.

Below, we consider these elements in more detail.

2.1 Objects

Objects, encapsulating both data and behaviour, are the basic entities within our model. Objects can either be 'simple objects' (i.e., the traditional definition of objects), or 'group objects' representing references to other objects. There are two basic elements of a simple object: its *interface definition* and its *internal definition*.

Interface Definitions

Individual objects only act upon certain identified messages. Thus, an object must be able to filter out messages that it wishes to recognise, ignoring all others. The definition of which messages an object recognises, together with a definition of the messages that an object may itself produce, is provided by the *interface definition* for that particular object. The interface definition for an object, for example '`stack`', may be defined in the following way

```
stack(pop,push)[popped,full].
```

Here, {`pop`, `push`} is the set of messages the object recognises, while the object itself is able to produce the messages {`popped`, `full`}.

Internal Definition

We will not consider the detailed internal definition of objects, but note that these could be given in a variety of forms, either imperative or declarative [11]. Indeed, such systems can be either homogeneous or heterogeneous in nature, e.g. one object might execute Fortran, another C++, another COBOL.

In addition to carrying out internal execution and spawning new objects, individual objects also receive messages broadcast from other objects and queue them. They can then consume one or more messages from this queue. The default behaviour is to consume messages from the queue up to either the next duplicate message, or the end of the queue. However, the object can utilise additional constraints (often meta-level) in order to define other strategies, e.g., consume single messages, all messages, or up to a message that matches a particular expression. Note that, objects are not necessarily driven by the consumption of single messages (c.f. Actors).

In principle, each object may incorporate concurrent execution. For example, an object may be sending several messages at the same time, consuming multiple messages (as above), and even spawning new objects concurrently. In practice, the extent to which this option is utilised depends upon the particular language used to implement the object's internal definition.

2.2 Concurrency

It is fundamental to our approach that all objects are (potentially) concurrently active. In particular, they may be asynchronously executing. Thus, our model captures a range of behaviour from standard sequential systems at one extreme, where enforced sequentiality must be represented, to multi-agent systems at the other, where objects may be asynchronously active, autonomous, and evolving.

2.3 Communication

The basic communication mechanism between objects is *broadcast* message-passing. Here, when an object sends a message it does not necessarily send it to a specified *destination*, it merely sends it to its environment, where it can be received by *all* other objects. Although broadcast is the basic mechanism, both multicast (achieved by imposing a group structure upon the broadcasted messages — see §2.4) and point-to-point message-passing (achieved by adding an extra 'destination' argument to the message) can be defined on top of this.

The default behaviour for a message is that if it is broadcast, then it will *eventually* be received at all possible receivers. Also note that, by default, the order of messages is not preserved, though such a constraint could be added, if required.

2.4 Groups

As mentioned above, *groups* are also objects. A group can be thought of as a simple object containing references to all the objects that the group contains (although groups need not be implemented in this way). As groups are objects too, any operation that can be performed upon an object can be performed upon a group. However, groups also understand other messages, in particular those relating to group manipulation, e.g., `add`, `remove`, etc. Thus, the basic properties we require of groups are that objects should be able to

- send a message to a group,
- add an object to a group,
- remove a specified object from a group,
- ascertain whether a certain object is a member of a group, and,
- construct a new subgroup.

There are other properties that might be useful, such as the ability to list the members of a group, but they are not essential.

A group is itself an object in the system. In principle, even the group of all objects (apart from itself!) might exist somewhere. To construct a new group, we just send a message to a group object with the appropriate definition (see below).

Group objects contain (references to) other objects which may, in turn, be either simple or group objects. Groups may also overlap. Thus, one object may be a member of several groups. When a group receives a message, the default behaviour is to pass the message on to all its members (although, again, this can be changed!). Messages can be sent to specific groups, as follows:

```
message(args)[group1,group2,...,groupn]
```

In principle, default messages are broadcast to the global group `all` but, in practice, messages of the form

```
message(args)[all]
```

are simply represented by

```
message(args)
```

While the basic purpose of groups is to restrict the set of receivers for broadcast messages, and thus provide finer structure within the object space, more complex applications can be envisaged. For example, the group object might enforce a different model of communication within that group. Thus, groups can effectively enforce their own 'meta-level' constraints on message-passing between objects, for example specifying that the order of messages is preserved.

3 Applications of the Model

We briefly outline some of the potential applications of this variety of programming model.

3.1 Distributed Artificial Intelligence

The first application area we consider is Distributed Artificial Intelligence (DAI) and, in particular, distributed problem solving. By defining objects as individual reasoning components, e.g. planners, and by utilising the grouping mechanism to organise the network of objects, powerful distributed problem solving systems can be developed (see [21] for examples of the use of this model). For example, hierarchical systems where one (*manager*) object partitions a problem so that it can be solved by other objects, can be constructed by ensuring that the protocol used between objects takes in to account the the manager's primary role in the group.

While the above approach is often used in real problem-solving systems, a more dynamic and flexible architecture can be defined by utilising the facility, in our model, for having multiple objects recognising the same messages. In particular, we can, in a way similar to the Contract Net approach [23], broadcast sub-problems to be solved to a group of objects. Each object might attempt to solve the problem in its own way, but the top-level object again waits until at least one solution has been returned from the set of objects. Note that the manager object in this case need not know exactly what other problem-solving objects exist.

Finally, individual objects might cooperate in order to solve more complex problems, i.e. problems that the objects could not undertake individually. Thus, a form of co-operative problem solving can be defined by ensuring that cooperating objects appear in a sub-group of their own.

3.2 Concurrent Theorem-Proving

Next, we consider a model for concurrent theorem-proving presented in [12]. This style of theorem-proving involves distributing formulae amongst objects, implementing a basic deduction mechanism within each object, and utilising broadcast communication between objects in order to pass intermediate deductions around. As an example, consider the following set of propositional Horn Clauses.

1. p
2. $\neg p \vee q \vee \neg r$
3. $\neg p \vee \neg q \vee \neg r$
4. $\neg p \vee r$

Now, assume that each clause is stored in a distinct object. To represent a form of theorem-proving, we impose the following behaviours upon these objects. Any object containing a proposition should pass (broadcast) that proposition to all other objects and, when objects receive propositions, they transform their own clause on the basis of this, broadcasting any new propositions generated and notifying other objects if a contradiction is produced.

Thus, as the objects above begin executing, the first object, containing only the proposition p, broadcasts the fact that p is true to the other objects. Once p has been received by the object containing clause 4, clause 4 is transformed to a positive literal r. This again causes new information to be broadcast, in this case the fact that r is true. After the r broadcast above reaches all the other objects, then the objects containing clauses 2 and 3 will both be transformed to literals. Finally, q is broadcast from clause 2 and the object containing clause 3 generates a contradiction.

This model of theorem-proving, though simple, provides an *adaptable* and *flexible* basis for large-scale theorem-proving [12].

Finally, grouping is used in certain cases to localise some of the deduction within the proof process.

3.3 Fault-Tolerance

We next consider an application which utilises the combination of broadcast message-passing and group structuring. Many real-life fault-tolerant systems, such as Distributed Operating Systems [4, 5], utilise the power of broadcast message-passing in order to provide duplication of important resources. One common approach is that of 'process pairs' [6]. Here, an 'essential' process in a distributed operating system, for example a network server of some kind, is 'shadowed' by a similar *secondary* object. This secondary object listens for exactly the same messages — this is possible since all requests are broadcast. Further, the secondary object *simulates* the behaviour of the primary object, with the exception that it does not broadcast any messages of its own.

Now, the secondary object also notes the primary object's activity. If the primary object has not responded to a request for a specific amount of time, the secondary object assumes that the primary object's processor has 'crashed' and takes over the duties associated with the service. This requires changing the object's interface so that it now broadcasts replies. At this stage, the secondary object has become the primary one and it spawns a new copy of itself to act as the new secondary one. In this way, fault-tolerance can be added 'seamlessly' to certain types of distributed systems.

3.4 Modelling Multi-Agent Societies

Since objects are autonomous, and may incorporate potentially complex internal reasoning mechanisms, we can view them as *agents*. By utilising both groups and broadcast message-passing, we are able to represent complex interaction, for example cooperation and competition, amongst such agents.

Imagine a scenario where agents are competing, for example by *bidding* for resources. In this case, cooperative activity within groups of agents can be organised so that the group as a whole puts together a bid. If successful, the group must (collectively) decide who to distribute the resource to. Thus, a number of groups might be cooperating internally to generate bids, but competing (with other groups) to have their bid accepted.

Within a given group, various subgroups can be formed. For example, if several members of a group are 'unhappy' with another member's behaviour, they might be able to create a new subgroup within the old grouping which excludes the unwanted agent. Note that members of the subgroup can still receive the outer group's communications, while members of the outer one cannot receive the inner group's communications. Although we have described this as a retributive act, such dynamic restructuring is natural as groups increase in size.

Although these examples have been based upon agents competing and cooperating in order to secure a certain resource, many other types of system can be represented. In particular, we can represent *agent societies*, where individuals cooperate with their fellow group members, but where the groups themselves compete for some global resource [10]. It is important to note that in all these examples, there is no explicit global control — individual agents perform local interactions with each other and their environment, and groups are effectively self-organising.

4 Virtual Shared Memory Architectures

The class of parallel architecture on which realisation of this programming model is considered is that of a logically shared but physically distributed memory, termed here *virtual shared memory*. It has long been recognised that physically shared memory machines suffer from a bottleneck (communication with the shared memory) above 20 processors.

Distributed memory machines are made up of sets of nodes with a communications means of some type. Each node has a store unit, and contains one or more processors. Such machines do not have the *shared memory* bottleneck but are regarded as more difficult to program. As a consequence, a hybrid scheme has been proposed where the system model is that of a logically shared memory on top of a physically distributed memory. In some instances of this model there is hardware support for shared memory.

4.1 Background

The Flagship machine [18] offered a global address space across distributed stores. Its successor prototype machine EDS [22] introduced hardware support for the shared memory model. Commercial machines offering VSM include the KSR family, and the Convex Exemplar. Both systems provide hardware-supported virtual shared memory.

The system model[1] for VSM machines is an extension to that of UNIX. A UNIX process couples an address space and a single thread of control. The VSM model is that of a set of threads operating within a shared address space (termed a *task*). The improvement over UNIX is that context switching time between threads within the same task is less than between processes.

Each thread may have its own *private* address space. Threads within a task communicate via the task address space. Potentially, each thread within a task can execute on a different processor, though each thread sees exactly the same task address space.

The VSM model usually supports a strongly coherent memory [15], i.e. a simulation of 'real' shared memory. When a thread attempts to write to a variable, an invalidate message is sent to all processors informing them to make all copies of this variable invalid, then the thread issuing the write is allowed to update the variable.

Whilst the system model of VSM is more convenient for the programmer, nonetheless the physical reality of distributed store means that for high efficiency most references from a processor must be to its associated store module [19]. The physical distribution results in there being different latencies to access local cache memory, local store, and off-processor store.

[1] We distinguish the *system model* (provided by VSM) from the *programming model* (the object-based computational model).

4.2 KSR Description

In the discussion, the KSR system [13] will be used as an example. The physical architecture is that of a set of processor-store pairs connected by a ring. A level-0 ring can have up to 32 such processor-store pairs. Multiple level-0 rings (up to 34) can be connected together, totaling 1088 processor-store pairs. Where there is more than one level-0 ring, a level-1 ring is included to provide communication between the level-0 rings. In such a configuration, there is a hierarchy of memories with increasing latencies: local cache, local store, off-node but on-ring store, and off-ring store.

5 Matching Programming Model to VSM

In this section, we consider how to match the programming model to the VSM system model. The abstractions offered by the VSM system model will be assessed according to their suitability to realise the programming model. It is easier to reason about an implementation if the conceptual gap between the programming layer and the system model abstractions used for implementation is small. In addition, we are concerned with efficiency: the programming model must be able to run with a high efficiency on target architectures. Our target architecture, at this stage of the investigation, has been a commercially available VSM machine. we consider prototype VSM architectures briefly in §5.7.

The characteristics of applications using the programming model are considered. From this a design for realising the programming model on VSM is discussed. An important aspect of the design is that the behaviour of an object should not need to be altered in any way, i.e. an object sends and reads messages in the way described earlier.

5.1 Characteristics of the Programming Model

The applications of the programming model have all been *symbolic*. Unlike conventional numeric systems, such applications do not allow a static process and data allocation strategy. They often require both dynamic process creation and dynamic data distribution.

The manifestation of these characteristics include:

- a large number of (sometimes small) processes
- processes all executing asynchronously

- frequent process creation and termination
- a large amount of broadcast message passing
- the use of grouping to provide locality

The facilities of the concurrent object-based programming model will be considered at two levels:

1. a global group of objects, without structure, which provides broadcast message passing;
2. object groups which structure the object space and offer a more sophisticated message passing model.

5.2 Broadcast message passing

The global group of all objects, `all`, is mapped to a task. Each object within the system is mapped one-to-one to a thread within this task space[2]. A thread can create another thread thus allowing object creation.

Broadcast message passing corresponds to a write to the global message queue (shared (task) address space) shared by all objects (threads). An object reads this message queue (shared address space) when it wishes to receive a message. Each object when created must be made aware of the address of the message queue. This design corresponds very closely to the description of an object sending a message not to a *destination* but to its *environment* where it can be received by all other objects.

When an object (thread) decides to send a broadcast message it issues a write request on an address in the message queue, all copies of that address are made invalid in all processors, the object (thread) then writes the message to the queue.

Various approaches can be taken to managing the object queue. For example, the programming model does not insist on preserving the order of messages. Consequently, each object could be given their own partition of the message queue to which only they can write.

5.2.1 KSR *poststore*

The KSR machine has a *poststore* operation. Using this, a variable[3] can be marked so that any thread that has previously read a copy is automatically updated with every new value of the variable by the hardware. The advantage of this is it helps to hide the latency caused by having to read a variable and then wait for it to be copied. In the programming model, if a majority of messages are read by all objects then this method is likely to be highly efficient. The amount of work to be done in propagating message copies not subsequently read would be offset by the latency hiding that it achieves. Each object (thread) could trawl over the message queue before computation begins thus ensuring that any subsequent message sends (writes to the shared address space) are propagated to them[4].

5.3 Object Groups

The simplest approach to object groups is to leave the design as it is, and merely ensure that the message queue is partitioned into group areas. An object sending a message to the group writes to the *group* partition allocated within the message queue. Subsequent reading of the messages can be handled as above.

The problem with this approach is that there is no way to ensure that an object that is not in a group does not maliciously or accidentally read messages intended *only* for the group (see earlier discussion of ‘unwanted agents’ in §3.4). It is possible to tag each object with a group identifier and then doing a check on the tag at each message read. This becomes more complex when overlapping groups are considered. This approach also adds overhead to each message read (and there are potentially very many) but is also conceptually unpleasing. A group should have access (by appropriate context mechanisms) to all and *only* the information designated by the application and the programming model.

In addition to this ‘violation’ factor, there is a further inefficiency issue: one of the objectives of object grouping is to minimise the impact of broadcast message passing.

A second approach is to map each object group into a task. On a KSR, it is possible to map part of address spaces between tasks via the UNIX memory mapper. Thus, each task (object group) has its own private address space for group messages, and inter-task address space (handled by the mapper) for broadcast messages. This approach would ensure privacy.

To ensure locality all objects (threads) within a group (task) could be tied to the same processor or same ring (we discuss overlapping groups in the next section). Even then, task switching on the KSR takes far longer than thread switching so again there may be serious latency problems.

A further difficulty is that, although in the conceptual model, groups are just objects (threads), in the design they are not associated with threads. The more general issue is that the KSR VSM does not offer any clustering mechanism beyond that of a thread within a task. We will discuss this further in §5.7.

[2] A similar approach was used to implement message passing abstract machines in [20].

[3] In fact, a 128 byte subpage.

[4] In practice, many numeric codes on the KSR do just this to ensure that all data is available locally before computation begins.

5.3.1 Overlapping Groups

Overlapping groups complicate the above model, in that now tasks will map parts of their address space to potentially many other tasks. However, assuming we have an efficient and secure way of handling object groups, then a simplistic way of dealing with overlapping object groups is to locate two overlapping groups on the same processor.

Unfortunately, in the case of many overlapping groups it is possible that, through transitivity, all object groups would reside on the same processor thus obviating any benefit of parallel execution. This is the classic trade-off between parallelism and communication — increasing one usually increases the other, decreasing one usually decreases the other. In the worst case, no parallelism means no communication, and (in almost all cases) no communication means no parallelism.

Given the above, some overlapping groups may have to reside on different processors. Extending this, each overlapping group could be tied, for example, to the same ring in a KSR-like architecture. Once again a pathological case could mean we only utilise one ring thus again limiting parallelism.

5.4 Dynamic Regrouping

A difficulty with the model of many small dynamic objects, and particularly dynamic overlapping groups, is that because there is no a priori information about object grouping nor of the message passing characteristics, it is not possible to statically arrive at a suitable distribution. It is possible to provide heuristics for when a new object group (possibly overlapping) should be created on another processor, or even for when an overlapping group should be dynamically moved to a processor with another group overlapping it which is receiving a proliferation of messages common to the intersection. Such heuristics are not difficult to determine and threshold values can be set.

The difficulty occurs when there are a large number of potentially transient objects (as with the programming model presented here). There may be a current burst of activity which reaches the threshold values and thus causes relocation of objects to occur - a form of dynamic regrouping. However, the object group causing the relocation may itself terminate thus once again leading to imbalance and relocation etc.

5.5 Point-to-point message passing

The recognition that point-to-point message passing can be viewed as an object group containing two objects enables the above group broadcast model to be utilised. In this approach, a new group is formed consisting of just two objects, with one of the object sending a *group broadcast* message and the other object in the group being the only recipient.

5.6 More Objects than Processors

If the number of objects exceeds the number of processors — as is very likely if the application depends upon a proliferation of small objects — more than one object (thread) can be assigned to a processor.

Conceptually this will work, and ensures that each object (thread) only has to issue reads to a message once, and this is seen by all objects (threads) associated with that processor. In practice, on the KSR-1 the context switching time for threads is higher than the latency time to access off-node data, even on another ring. Thus, a model involving a large number of objects is unlikely to run efficiently on the KSR because of the cost of thread switching.

More than one object can be assigned to a thread - thus retaining the one thread-per-processor model. The difficulty then becomes scheduling object execution at the user-level.

5.7 Discussion

The initial investigation has shown that VSM parallel architectures offer an elegant solution to broadcast message passing between objects. However, when we consider the full richness of the programming model difficulties arise.

The first difficulty (as manifested on the KSR) occurs when we consider large numbers of small objects with frequent switching between them. It is clear that KSR threads are too heavyweight to efficiently achieve such a model. This is not a difficulty *per se* of VSM, but many VSM machines have this system model.

A second difficulty relates to object groups. The standard VSM model is a two-level one of tasks and threads. Effectively, we require more levels within the system model corresponding to the abstractions within our programming model.

Finally, we note that our programming model requires rapid communication because of the frequency of message sends and particularly of broadcasts.

We now look briefly at other logically shared but physically distributed memory machines with more “suitable" architectural features. Essentially we are concerned three issues: support for large amounts of communication, support for fast thread creation and switching, and conceptually richer system models.

5.7.1 Communication

Flagship [18] with its fine-grained graph reduction abstract machine corresponds closely to our computational model in that it can cope with many small rewrites (fine-grained processes), much communication and dynamic load balancing. One of the features of programs running on Flagship was that although the communication network was never over-loaded, the majority of time during program execution was spent packing messages to be sent and de-packing them on arrival.

A consequence of this was that Flagship's successor machine, EDS, extended the processor-store pair node, to two processors and one store constituting a node. One of the processors carries out computation, the other handles all communication activity. It would seem essential, given the amount of message passing, involved in systems written in the computational model to have this notion of communication processor[5].

5.7.2 Thread Creation and Switching

The programming model has the capability of being able to create a number of small, potentially transient, objects. To efficiently handle such we need both extremely rapid object (thread) creation and thread switching. Though there are no commercially available multi-threaded parallel architectures (MTAs), they represent an interesting target for our computational model.

MTAs handle multiple threads (contexts) simultaneously [16]. When a thread issues a remote read to an off-node address, the latency is hidden by another thread being executed. The cost of thread switching must be lower than the latency — as has been discussed this is not so on the KSR.

The Tera multi-threaded architecture [3] is a logically shared memory MTA. Each processor can have up to 128 active program counters active at any one time. Sets of register banks are used to hold the contexts of active threads. Thread context switching occurs every clock cycle. To achieve this speed every processor has 128 of everything. Thread creation takes a few clock cycles. Tagged memory and registers with full/empty bits can allow message synchronisation [16]. As with the EDS machine (see above) specialised hardware for message handling can operate in parallel with computation.

With a multi-threaded architecture an object can be associated with a thread. For high efficiency there must be enough parallel activity to hide the latency. This should not be a problem in our model. For the Tera, problems may arise if we generate more than 128 objects on each processor as then there are no remaining thread slots. This suggests software intervention and inefficiency.

[5]We note that recent commercial multiprocessors (for example, the ICL GOLDRUSH and the Meiko CS-2) are made up of nodes with designated communication processors.

5.7.3 System Models

Given a fast context switching mechanism, and thus a one-to-one correspondence of objects and threads (note very small objects may be treated differently), then what our programming model requires of the system model is a way of grouping threads.

We wish to be able to deal with a set of threads with some common characteristic. The early stages of the operating system design for the EDS VSM machine postulated various layers of “process" abstraction (for more details see [17]. One of these abstractions was a *team* which was a collection of threads belonging to a task all residing on the same processor.

A team had a port associated with it to which messages could be sent to all threads within the team. In the EDS approach the idea was to optimise message storage and copying, and thread scheduling by making use of physical locality.

In our model, as we envisage both the dynamic relocation of objects (threads) to other processors and overlapping groups, the *EDS-team* is not sufficient. What we require is a new logical team abstraction, say *FK-team*, which allows logically related threads to be dealt with as a single logical abstraction. The *FK-team* is not tied to a single processor, but can range across a number of processors. Thus, there is no physical characteristic associated with the concept though we can optimise message copying by making use of threads all located on the same processor.

In the *FK-team*, we allow a thread to be in more than one *FK-team*. This enables us to realise overlapping groups. It also allows point-to-point message passing by placing a thread (object) in its own team (group) and sending to the team (group). Conceptually, we can even regard the object group `all` as being a team. Given the utility of broadcast message passing, then using shared memory is still likely to be more efficient particularly if there is hardware-supported operations such as the KSR *poststore*.

6 Conclusions

In this paper we have provided an initial investigation of the implementation considerations relating to a particularly flexible model of programming. While we have considered only VSM architectures, we believe that this programming model is so useful that implementation on a variety of architectures should be considered. Thus, our future work consists, not only of the development and testing of the programming model on a VSM architecture and the application of this implementation, but also of considerations of further implementation on alternative architectures. We need also consider issues such as the model's memory behaviour characteristics when run on a parallel machine, and garbage collection.

Finally, though our work is mainly involved with the development of particular programming models for extant architectures, we believe our investigations can have impact upon future parallel architectures. For example, our requirement for the possibility of a large number of small objects, together with the necessity to migrate these to other processors might require techniques analogous to hardware support for object-based languages, such as *dynamic grouping* [24]. The MIT-J-Machine is also of interest in this regard [7].

Acknowledgements

Thanks to Roo Ford, Rob Johnson, Ken Mayes and Dave Snelling for various discussions.

References

[1] G. Agha and C. Hewitt. Concurrent Programming Using Actors - Exploiting Large-Scale Parallelism. *Lecture Notes in Computer Science*, 206, 1986.

[2] G. Agha. *Actors - A Model for Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[3] R. Alverson, D. Callaghan, D. Cummings, et al., The Tera Computer System, Proceedings of 1990 International Conference on Supercomputing, ACM Press, Baltimore, MD June 1990.

[4] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[5] K. Birman. The Process Group Approach to Reliable Distributed Computing. Techanical Report TR91-1216, Department of Computer Science, Cornell University, July 1991.

[6] A. Borg, J. Baumbach, and S. Glazer. A Message System Supporting Fault Tolerance. In *Proceedings of the Ninth ACM Symposium on Operating System Principles*, New Hampshire, October 1983. ACM. (In ACM Operating Systems Review, vol. 17, no. 5).

[7] W. Dally, J. Fiske, J. Keen et al., The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms, *IEE Micro* 12(2), pp. 23-39, 1992.

[8] M. Finger, M. Fisher, and R. Owens. METATEM at Work: Modelling Reactive Systems Using Executable Temporal Logic. In *Sixth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE)*, Edinburgh, U.K., June 1993. Gordon and Breach Publishers.

[9] M. Fisher and M. Wooldridge. Executable Temporal Logic for Distributed A.I. In *Twelfth International Workshop on Distributed A.I.*, Hidden Valley Resort, Pennsylvania, May 1993.

[10] M. Fisher and M. Wooldridge. A Logical Approach to the Representation of Societies of Agents. In N. Gilbert and R. Conte, editors, *Artificial Societies*. UCL Press, 1995.

[11] M. Fisher. Concurrent METATEM — A Language for Modelling Reactive Systems. In *Parallel Architectures and Languages, Europe (PARLE)*, Munich, Germany, June 1993. Springer-Verlag.

[12] M. Fisher. An Open Approach to Concurrent Theorem-Proving. In *Proceedings of Workshop on Parallel Processing in AI (PPAI)*, Montreal, Canada, August 1995.

[13] S. Frank, H. Burkhardt and J. Rothnie, The KSR1: Bridging the Gap Between Shared Memory and MPPs, *Proc. of Compcon'93*, pp. 285-294, 1993.

[14] C. Hewitt. Control Structure as Patterns of Passing Messages. In P. Winston and R. Brown, editors, *Artificial Intelligence: An MIT Perspective (Volume 2)*, pages 433–465. MIT Press, 1979.

[15] W. Hussak and J. Keane, Representation of Coherency Classes for Parallel Systems, Proceedings of Fifth IEEE Symposium on Parallel and Distributed Processing, pp. 391-398, IEEE Press, 1993.

[16] K. Hwang, Advanced Computer Architecture, McGraw-Hill, 1993.

[17] J. Keane, An n-level Process Model, EDS Document, Department of Computer Science, University of Manchester, 1990.

[18] J. Keane, An Overview of the Flagship System, *Journal of Functional Programming*, January 1994.

[19] J. Keane, A. Grant and M. Xu, Comparing Distributed Memory and Virtual Shared Memory Parallel Programming Models, *Future Generation Computer Systems (11)*, pp. 233-230, 1995.

[20] J. Keane and M. Xu, Virtual Shared Memory-based Support for (Novel) Parallel Programming Models, Proceedings of *Programming Models for Massively Parallel Machines*, pp. 83-91, IEEE Press, 1993.

[21] T. Maruichi, M. Ichikawa, and M. Tokoro. Modelling Autonomous Agents and their Groups. In Y. Demazeau and J. P. Muller, editors, *Decentralized AI 2 – Proceedings of the* 2^{nd} *European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds (MAAMAW)*. Elsevier/North Holland, 1991.

[22] C. Skelton, C. Hammer, M. Lopez et al., EDS: A Parallel Computer System for Advanced Information Processing, in PARLE'92, Lecture Notes in Computer Science, 605, 1992.

[23] R. Smith and R. Davis. Frameworks for Cooperation in Distributed Problem Solving. *IEEE Transactions on Systems, Man and Cybernetics*, 11(1), 1981.

[24] I. Williams, M. Wolczko and T. Hopkins, Dynamic Grouping in an Object Oriented Virtual Memory Hierarchy. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pp87–96. Paris, France, June 1987.