

Requirements for Parallel Programming in Object-oriented Distributed Systems

BRENDAN TANGNEY, ANDREW CONDON, VINNY CAHILL AND NEVILLE HARRIS

Distributed Systems Group, Department of Computer Science, Trinity College, Dublin 2, Ireland

In this paper we present some ideas on the functionality that should be incorporated into an object-oriented distributed system to support distributed and parallel programming. The work is based on practical experience in developing several substantial distributed and parallel applications on the Amadeus platform. Related work in the area is sketched.

Received April, 1994

1. INTRODUCTION

This paper discusses the functionality that should be offered by an object-oriented (OO) distributed system in order to support the programming of distributed, and parallel, applications in a hardware environment consisting of a collection of workstations connected by a local area network. The discussion is based on our experience of programming several substantial applications on the Amadeus platform (Cahill *et al.*, 1993a, b).

Although the description of the applications given here is in terms of their implementation on Amadeus we hold that the lessons learned, in terms of the functionality that an OO distributed system should provide, are of general applicability.

The layout of the paper is as follows, Section 2 describes the pertinent features of the Amadeus model, Section 3 describes how the application programs were structured to run on Amadeus. The key part of the paper is Section 4, in it we discuss the main features we think an OO distributed system should provide. The paper concludes with a look at some related work in the area.

2. THE PROGRAMMER'S VIEW OF THE AMADEUS PLATFORM

Amadeus is a distributed OO programming environment developed at Trinity College Dublin and is the reference implementation of the Comandos¹ platform. An overview of Comandos is given in a companion article in this issue and an in depth discussion of all aspects of the project can be found in Cahill *et al.* (1993) as well as in the other articles in this issue. This section focuses on those features of the Comandos *Virtual Machine Interface* and *Computational Model* that Amadeus implements which are relevant to the current discussion.

Comandos aims to provide support for distributed and persistent objects in a heterogeneous environment. Objects are generic and opaque and so can be bound to different language-specific object models. Objects are

passive—in that they do not change their own state—and are manipulated by active processing entities known as *activities*. An activity is a **multi-node** lightweight thread of execution which appears to move between nodes as it invokes various objects. When an activity visits a node, for the first time, it is said to **diffuse** to that node. Persistent objects are stored in the storage system (SS) and are brought/mapped into (virtual object) memory (VOM) when they are used. Communication and synchronization between activities is achieved by invoking on shared objects.

One of the goals of Amadeus is to support multiple languages. This is done through the provision of language-specific run-times which interface to a *generic run-time* layer (Cahill *et al.*, 1993). Two languages are currently supported: C** a slightly extended version of C++, and Eiffel**, a slightly extended version of Eiffel. The applications described in this paper were coded in C**.

The most relevant features of the Comandos computational model are elaborated on in the following paragraphs.

- **Jobs and activities.** Class method invocations can be 'forked' as heavyweight processes (*jobs*) or lightweight threads (*activities*).
- **Global objects.** Member functions marked with the C** keyword **global** can be accessed remotely. Such operations can be invoked **transparently** from any node in the system, i.e. Amadeus provides access transparency.
- **Persistence.** Object persistence, the ability of an object to outlive the program that created it, obviates the need for much of the traditional I/O code necessary to preserve information between executions of the program. C** objects are marked as persistent using the **permclass** keyword. (Note that an interesting feature of Amadeus is that persistence and global access are orthogonal to each other.)
- **Clusters.** One of the problems faced in the implementation of any OO system is to make the accessing of objects efficient. This is particularly problematic in a

¹ Comandos was partially supported by the Commission of the European Communities as ESPRIT projects 834 and 2071.

distributed system where target objects can be in local or remote memory, or on disk. Amadeus uses the concept of a **cluster** to group objects together. Objects within a cluster share a lot of the overhead state information that must be maintained to access objects (thus making small objects feasible) and are the unit of transfer between storage and memory. Furthermore, a cluster is only mapped at one node at any one time. If two activities invoke on objects within the same cluster then one, or both, of them will have to diffuse to the node at which the cluster is mapped. Currently clusters are not replicated nor is migration used once a cluster has been mapped. A single application (or activity) may of course use more than one cluster during execution.

- **Load balancing.** In Amadeus load balancing is provided by performing dynamic assignment of activities to nodes as they are created. The normal benefits of load balancing in a distributed system follow on from this.

2.1. Support for parallelism

Clustering and load balancing are central to the support for parallelism that Amadeus provides. Extensive use of both is made in the applications described below, so they are now described in a little more detail.

2.1.1. Explicit clustering

In loosely coupled systems it is not possible to provide 'performance transparency', i.e. while accessing a remote object may appear the same as accessing a local one there is a performance penalty to be paid. This issue is crucial to the way in which parallel programs are structured. This has also been recognized as an issue in shared memory systems with non-uniform access memory, e.g. Fleisch (1988). A commonly used solution to the problem is to force the programmer to explicitly partition the data within the application code.

In Amadeus the solution adopted is to utilize the concept of a cluster, which was introduced to diminish the overhead of mapping objects to and from storage. The idea is that objects within a cluster should exhibit a strong locality of reference. This allows different clusters within a parallel application to be placed on different nodes. A good assignment of objects to clusters will yield a good computation to communication ratio allowing for reasonable speed-up to be obtained.

The following cluster control primitives are provided to allow applications to explicitly control which clusters objects reside in. (Default action is taken by the system if they are not used.) Examples of their use are given in the next section.

- **newcluster.** This directs the system to start a new cluster and make it the current default cluster. All subsequent newly created objects are assigned to the current cluster.

- **setcluster.** This directs the system to change the default cluster. The cluster id returned on starting a new cluster is used to indicate which cluster to change to.
- **unmapcluster.** Clusters are automatically unmapped when the application that was using them terminates. This primitive is provided to allow an application to explicitly unmap a cluster before termination.

It is important to emphasize that the application is only responsible for expressing which objects go together, it is up to the system to decide where to place clusters. The system should ensure that with the possible exception of performance, there is no difference between an application running with all its activities and clusters on a single node or on multiple nodes.

2.1.2. Load balancing

The traditional advantage of load balancing is to redistribute (monolithic) jobs so as to improve resource utilization, etc. If, however, applications are structured as a number of cooperating activities then load balancing offers additional benefits in that the various activities may be assigned to different nodes allowing a **single application to execute in parallel**.

The mechanism is fair in that during periods of heavy load the application will be confined to one node and not impact on other users.

Clustering plays an important part in the effective load balancing of multi-activity programs (Tangney and O'Toole, 1991). Load balancing assigns a (preferred) node to each activity when it is created. When the activity faults objects (clusters) into memory they are placed at the preferred node. However when an activity invokes on objects that are already mapped, these objects are not moved, but instead the activity diffuses to the appropriate node.

If all the objects used by the application are in the same cluster then it will be mapped into memory at a single node and that is where all the computation will take place. Therefore the onus is on the programmer to partition the objects into clusters in such a way that activities spend most of their time executing within a single cluster with only occasional forays abroad to other clusters for synchronization.

Examples of this will be seen in the applications that follow.

3. APPLICATIONS

This section outlines how four different applications were structured to run on Amadeus. The programs include examples of both functional and data parallel algorithms as well as complex hybrid structures. The demands they make on the underlying system are also discussed.

3.1. Lehmer-Lucas

Numbers of the form $(2^p) - 1$ are known as *Mersenne*

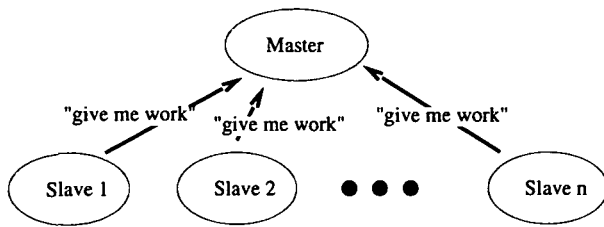


FIGURE 1. Simple master/slave topology.

numbers and the Lehmer–Lucas test is used to determine whether numbers of this form are prime or not (Knuth, 1981). The test is computationally intensive taking $O(p^3)$ time to discover if $(2^p) - 1$ is prime. The amount of communication involved in the solution is quite small when compared with the amount of computation required for non-trivial values of p .

The problem is a good example of the ‘give me work’ scenario, which frequently crops up in parallel programming. It can be parallelized along master/slave lines with each slave being given a single value of p to test for. The execution time of slaves differs dramatically so the master loops waiting for workers to complete handing, out more work as the slaves request it, i.e. slaves petition the master with ‘give me work’ requests.

3.1.1. Amadeus implementation

Figure 1 shows the master/slave configuration that was used in the Amadeus implementation of the Lehmer program. The master object was declared to be global so that workers could invoke on it from anywhere in the system. The master took responsibility for creating the workers which were automatically distributed around the system using load balancing.

It is necessary to ensure that the objects used by each worker, and by the master, are assigned to separate clusters so that load balancing can be effective in distributing the computation around the system. The master object uses the `newcluster()` call prior to initializing each slave’s objects. The clusters are then unmapped using `unmapcluster()` so that as each slave is created the load balancer is free to choose the node at which to place it. The following fragment of C++ shows this in practice.

```

for (int i = 0; i < no_slaves; i++) {
    cluster_id = amadeus.newcluster(SIZE);
    amadeus.setcluster(cluster_id);
    slaves[i] = new kehmer (i, this);
    amadeus.unmapcluster();
}
//.....
// loop checking for activities to terminate
// and starting new ones to process each number
//.....
while (i ≤ N) {
    for (k2 = 0; k2 < no_slaves; k2++) {
        if (ended (k2)) { // has the activity terminated?
            count << "Slave" << k2 << " is done\n";
            ii = next_prime(i);

```

```

        if (ii > N)
            break;
        cout << "Creating slave with " << ii << "\n";
        set_start(k2); // mark slave as active
        // launch new activity and remember id
        slave_fts[k2] = new activity (slaves[k2], ii);
        ii++;
    }
}

```

The structure of this program is very common in parallel applications, not just in its master/slave layout but in the initialization phase, done at the master’s node, followed by a computation phase done by many workers, spread around the system.

3.2. Ray tracing

A more substantial but similarly structured problem is that of ray tracing. Ray tracing is essentially an exercise in **data parallelism** with the program again being structured along classic master/slave lines. Output data (an image) is synthesized, by many workers, from two sets of input data—a model of the scene and a camera model.

The image can be broken up into a number of subsections (rectangles) each of which can be processed in parallel by a slave worker without any reference to its neighbours. Once the rectangle is complete it can be sent back to the master to be stored as a persistent object—or as was done in our case displayed directly on a screen.

A slave is created for each rectangular sub-image. Each slave needs its own copy of the input models and the number of slaves can be equal to, greater than or less than the number of processors. Conceptually, the master gives each slave an empty sub-image object and the slave manages the computation of that rectangle. The slaves return their completed sub-images for collation and/or display.

Thus far the application is identical in structure to the Lehmer program previously described. However the problem of ensuring good performance is more difficult to solve in this case than it was in the Lehmer example. There are two reasons for this and they crop up in many distributed applications.

- **Computation versus communication.** The speed-up obtained by the application will be limited by the ratio of the time spent actually computing the problem to that spent in communicating between the various activities. This ratio is dependent on the nature of the algorithm and the underlying communication cost of the system. In the ray tracer this manifests itself as a lower limit on the size of a sub-image that is worth farming out to a slave—too small an image implies too frequent communication and too high a communication to computation ratio.
- **Work redistribution.** As the total computation time of a parallel application is limited by the completion time of the last task it is necessary to distribute work as

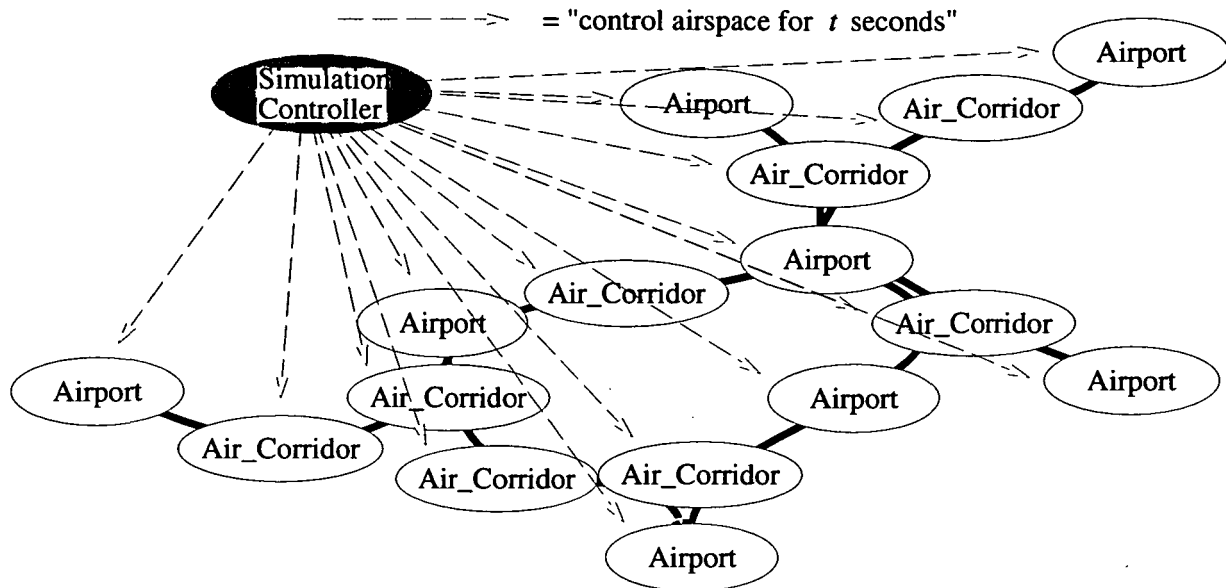


FIGURE 4. The structure of the ATC program.

being to facilitate a smaller granularity of parallelism, by increasing the number of active objects in the model, see Figure 4.

This application makes heavy use of persistence and transparent distribution. This allows the simulation to be stopped and re-started (possibly on different machines) transparently to the objects in the simulation.

In Amadeus the simulation is run in parallel for one time unit as follows:

- The simulation controller object invokes the `Control-Airspace` member function on each airport and `air_corridor` object as an *activity*.
- Each airport and `air_corridor` proceed in parallel, communicating with one another to hand-off flight-plans.
- Eventually the activities complete the computation for the clock period and the process repeats.

3.4. A problem for computational linguistics

The next problem comes from the area of computational linguistics. It involves finding a domain (D), and value in that domain (x, y, z) , such that the following formula³ holds for the binary relation F :

$$F(x, y) \Leftrightarrow F(y, x) * F(z, x) \\ \Leftrightarrow \neg F(z, y) \Leftrightarrow F(z, z) \Leftrightarrow \neg F(x, y)$$

The solution to the problem involves beginning with a domain containing the single element 1 and checking to see if the formula can be satisfied from that domain. If it cannot then the domain size is increased by 1 and the formula tested again. The algorithm for testing the formula is a 'little complex' and involves using the disjunctive normal form of the original formula plus a

³ Where the domain is the set of natural numbers and $*$ can be read as 'conjunction' and \Leftrightarrow as 'gives the same result as'.

substantial amount of backtracking. Full details of the algorithm can be found in Burke (1993). What is of interest to this discussion is that (i) the algorithm is horrendously expensive both in terms of computation time— $O(n^2(n-1))$ —and memory space required for large domains, and (ii) there are a number of different ways in which it can be partitioned to execute in parallel.

3.4.1. Amadeus implementation

The current Amadeus implementation is based on a Prolog coded version of the algorithm, i.e. it implements backtracking. This allowed easier development of the classes needed for the problem as well as verification of the algorithm against the original. Because of the OO approach, the classes developed provide a basis for investigating other strategies. Figure 5 shows the structure of the Amadeus implementation. Again explicit clustering and load balancing were used. One chosen allocation of object to nodes is shown, but this partitioning is by no means the only possible approach having been chosen to allow verification of the language port rather than maximizing parallelism.

This (slightly parallel) C++ version is *considerably* more memory efficient than the Prolog one. It can explore domains up to size 16 as opposed to 6 for the Prolog program. To go beyond this however, requires even greater exploitation of parallelism, e.g. putting a full pipeline on each node and implementing the kind of pruning required to work on domains large enough to contain solutions (analytically determined to be $n > 25$, which entails a search-tree of *depth* 15 000).

3.4. Performance

As the hallmark of any useful parallel system is improved performance, this section briefly reports on the results achieved for one of the applications described above, i.e. the ray tracer.

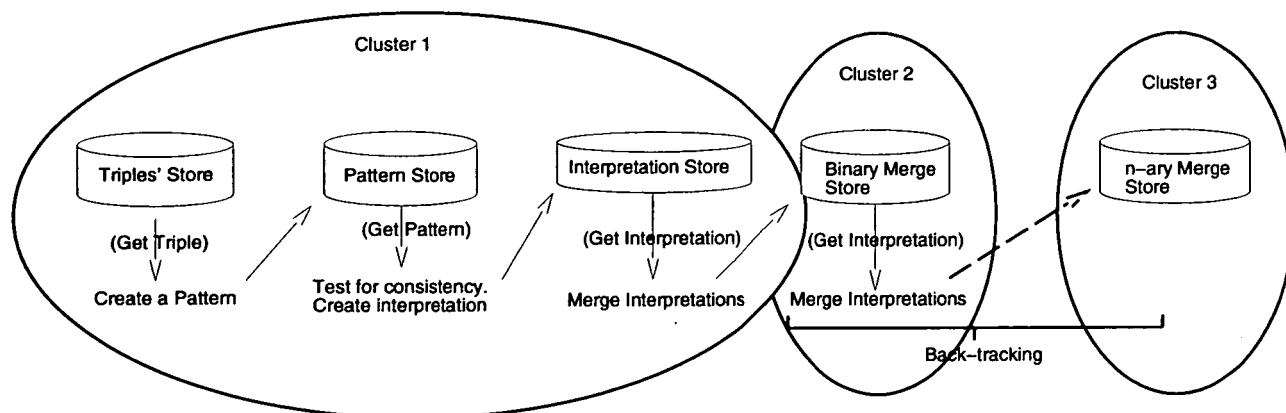


FIGURE 5. The structure of the CL program.

Figure 6 shows the speed-up curve obtained for the ray tracer running on a number of Sun workstations, SPARC Classics with 96 MByte of memory, while Figure 7 shows the speed-up curve obtained for a number of Digital DS2100 machines with 12 MByte of memory. A number of points are worth noting about these curves. Firstly both curves show that significant speed-up can be obtained from the system. Furthermore applications benefit in another way in that the problem domain size can be increased as more nodes, or more powerful ones, become available. This phenomena is known as scaled speed-up (Gustafson, 1988) and two examples of its effect can be seen in the curves. Super-linear speed-up is obtained on the early part of the DS2100 curve due to the extra memory made available over the (overworked) single node case. On moving to the more powerfully configured Sun machines the image data size can be greatly increased, from 600^2 to 1100^2 pixels. In both curves the speed-up begins to level off around six nodes. Further scaled speed-up can be obtained by increasing the data size so that there is enough work to keep all the nodes busy.

4. REQUIREMENTS

Using the examples just described as illustrations, this

section discusses what we hold to be more important requirements on a distributed programming platform.

4.1. Object-orientation

The many supposed advantages of the OO approach are well documented in the literature, e.g. Cox (1986) and Nierstrasz (1986). In brief the principles are that three key techniques (*encapsulation*, *inheritance* and *polymorphism*) are claimed to yield three important benefits, i.e.

- Re-use through the use of extensible classes.
- Clarity and simplicity of code as a result of polymorphism and inheritance.
- Robustness of design in the face of changes to the specification.

In our experience these advantages are true in practice. We can see examples of the first two from the ray tracer, which had the advantage (from this perspective) of having a previous incarnation of a C program running on hyper-cubes. The earlier program was 4000 lines of C compared to only 2000 of C++. Because this reduction in code-volume is due to code re-organization and re-use, rather than recoding, the result is empirically simpler: it is not a matter of terseness to the point of obfuscation. In

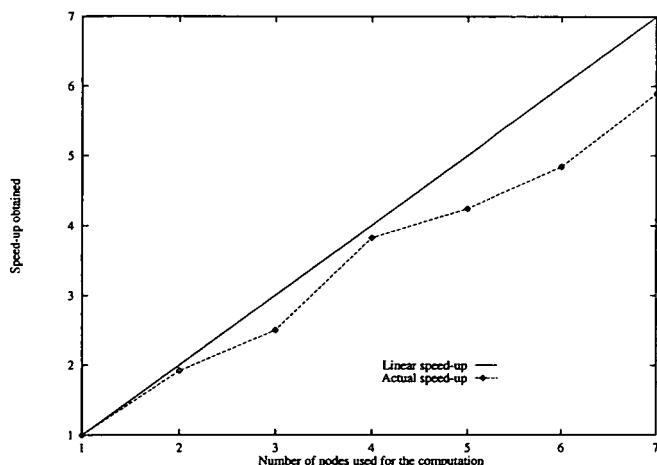


FIGURE 6. Sun speed-up curve.

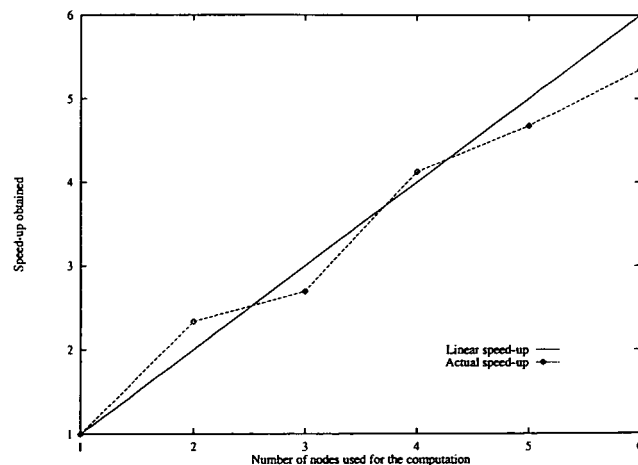


FIGURE 7. Dec speed-up curve.

the case of the other programs under discussion there can be no direct comparison, as they were written using OO from the beginning, however the programmers are confident that the principle holds for these too.

In parallel, and distributed, programming the issue of design-robustness is crucial if the sequential and parallel facets of the design are to be separated. (In Amadeus this is done through the use of global objects, which isolates the global communications strategy and allows it to be altered without changes propagating to the non-global objects.) Parallel program designs must be robust because, at present at least, mapping an algorithm onto the underlying hardware is a difficult task and may require a number of iterations. A good example of this is in the ray tracer where the initial model of the dynamic work reallocation involved the slaves reporting their progress to the master at intervals. It transpired that it was better with the current implementation of Amadeus (i.e. less prone to bottlenecks) to poll all the slaves whenever one finished. The algorithmic politics are not of interest here, what is significant is that these policy decisions could be investigated quickly and easily because the changes required were localized (one member function in each of the master/slave classes).

Inheritance and polymorphism make it possible to build a type hierarchy which embodies various parallel strategies, e.g. master/slave, ring or pipeline computations. Programmers can specialize (i.e. derive) from these classes when writing new applications, and if the chosen base class is inappropriate they can substitute another without impacting on the rest of the design.

4.2. Transparency

The pros and cons of transparency, and the appropriate level of transparency to provide, in distributed systems, are well documented in the literature (Popek *et al.*, 1983). All the applications described previously rely heavily on the fact that access to remote objects, be they in memory or secondary storage, is fully transparent.

In comparison to some parallel systems, e.g. Intel Corporation (1986), applications are not tied to any specific node address or number of nodes. A multiple activity program will run just as correctly with all its activities assigned to the same node as it would if they were assigned to separate nodes.

4.3. Global objects

At present parallel programs are frequently custom-made for each parallel architecture. Our approach to parallel programming is that the parallel structure of a program can be made orthogonal to an (OO) design, if adequate support is available from the language and/or run-time. This has the benefit of separating the parallel concerns from the sequential ones.

The first and most important extension is to allow interaction between objects in non-shared memory. In the programs we have implemented Amadeus' global

objects are communication points between groups of non-global objects.

In the programs discussed in this paper global classes are a minority. In the ray tracer, for example, the only global classes are the master and slave classes (two out of 33 classes), while in the ATC only airports and air-corridors are invocable globally (two out of 16 classes). While global classes constitute a minority of the total number of classes, global objects make up an even smaller minority of the actual object instances: less than 30 compared with several million in the case of the ray tracer. Because the ATC is a simulation the numbers of objects it produces is open-ended. However, the numbers of *global* objects are fixed at start-up (at around 100) while the non-global objects number more than 500 and are produced continuously. The Computational Linguistics application is, if anything, more extreme: a handful of global objects and a virtually unbounded number of smaller objects representing the formulae.

These applications lend support to an approach whereby non-global is the default. In a loosely coupled system the alternative strategy of making global the default incurs far too much overhead to justify the benefits to the programmer. However it is very important to ensure that extending a class to be global should be *easy*, e.g. inserting a keyword.

4.4. Persistence

As noted previously, the addition of persistence to the programmers arsenal reduces the amount of routine I/O code that must be written. As a concrete example, the programs discussed in this paper required *no* code to output to disk and very little code to perform input. (Where input code was written it was to allow non-Amadeus data to be brought into the persistent store.)

4.5. Load balancing

Although explicit placement of activities was used to achieve the speed-up curve shown in Section 3.5 similar results were obtained using automatic load balancing in a quiescent system. In effect load balancing combined with transparency allows a distributed system to be used as a cost effective parallel processor which supports multiple simultaneous users.

4.6. Explicit clustering

Application level control of clustering was used in all of the applications described to ensure that computation could actually be spread over multiple nodes and we hold that the programmer must be able to exercise such control over the grouping of data.

In the applications discussed the `unmap()` primitive was used extensively at the end of the initialization phase to allow worker's data to be defaulted in at the nodes assigned to each worker. An alternative way to achieve the same effect would be to provide for migration of

clusters. This would necessitate marking the cluster as being fixed or movable, a property that could be expected to change during the course of execution. Migration of clusters is, however, problematic if an activity is actually executing within the cluster.

A related issue is that of object migration. A case can be made that individual objects should be able to move after they are created, e.g. aircraft objects in the ATC. Our current approach is to do long-term reclustering based on statistics gathered by the garbage collector as we are wary of allowing too much object movement during execution of programs.

4.7. Replication (and fragmentation)

Encapsulation, one of the primary planks of the OO credo, encourages designers to stress the hiding of internal data in order to limit dependencies between conceptually unrelated pieces of code. In systems with multiple address spaces (i.e. most parallel and all distributed systems) the sharing of encapsulated objects automatically becomes an issue. Actually, this is not a problem with encapsulation *per se*, but with the physically divided address space, because access times are markedly different between local and remote access.

In non-OO code, however, the data that needs to be shared would probably be global⁴ variables in the executable in each address space. So, given that (uncompromised) OO is worth pursuing, what can be done to bridge this gap between having each object as an indivisible encapsulated unit and the desire to have local access to it *on several nodes*?

In common with many others we believe the answer lies in transparently replicated objects. If you have one object and multiple nodes there are three cases:

- Normal (local to one node).
- Replicated (local to several nodes).
- Fragmented (*parts* of object are local to several nodes).

To understand this in concrete terms, consider the ray tracer, a simple functional computation. It has input data and output data. This is easily modelled as an input data object, a computational object and an output data object, as seen in Figure 8. Parallelizing this will usually involve placing a copy of the computational object on each node but where should the input data object go? Obviously, each node needs it locally, yet it remains conceptually one object. The answer is to maintain the impression of a unique input object while (transparently) *replicating* it.

One can make a similar case for the output object, parts of which are being computed on each node, yet it too remains conceptually one object. It could be (transparently) *fragmented*, though this is much more

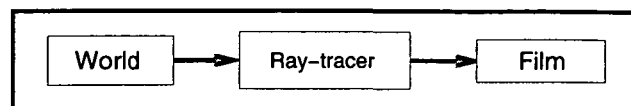


FIGURE 8. Simplest view of ray tracer.

complicated than replication, although this is the approach favoured by Makpangou *et al.* (1991).

Figure 9 shows one way in which replication and fragmentation could be utilized in the ray tracer example.

Our experience suggests that fragmentation can be foregone in favour of separate objects per fragment, with some other object collating the pieces. We suggest, therefore, that while transparent replication is a requirement for parallel programs on distributed systems, fragmentation may be useful but is not proven to be so by our examples.

4.8. Multi-cast

If replicated objects are to be supported it follows on automatically that some form of multi-casting must be provided.

Furthermore, in all the applications discussed in this paper, the situation arises where the same invocation is applied to a group of objects of the same class. In the ray tracer this occurs when the master is inquiring about the progress of the slaves, determining which is the slowest. In the ATC it occurs when the simulation controller initiates a simulation period in all the airports and air-corridors. These situations, too, could be better handled by some form of multi-cast.

4.9. Low overhead of run-time

It is important that the functionality provided by a system such as Amadeus should not impact performance of sequential computation. In Amadeus and other OO paradigms this means that it should be possible to have objects which have none of the overheads associated with persistent and global properties. These are two reasons why this is an important criterion for a parallel programming platform.

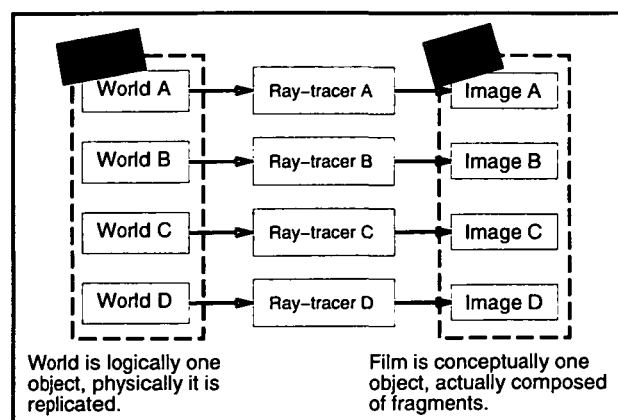


FIGURE 9. Replication and fragmentation possibilities in the ray tracer.

⁴ Here we mean *global* in its traditional sense and not as a C++ keyword.

Firstly, it is obviously of paramount importance to the parallel programmer that the performance of the serial parts of the program should not be adversely affected by the use of the parallel-enabling facilities.

Secondly, it encourages one to develop in the paradigm *from the beginning* in order to leave open the possibility of incorporating distribution and persistence facilities later. If it is not the case, then programmers or designers have to justify using persistence and global objects from the beginning of the project. If such justification were not forthcoming the project would proceed in development in some other paradigm, and build up inertia against moving from it (even if persistence and distribution were shown to be necessary later).

4.10. Application-level locking

In multi-threaded programs, such as the ones we have described here, it is important that synchronization be efficient. In systems which use system-level locking the cost of obtaining or relinquishing a lock is often two system calls. In such cases, the programmer will have to choose the lesser of these two evils:

- Lock when necessary and pay the performance price.
- Lock less often, resulting in unnecessary lock-out.

Performance tuning of programs with such sub-optimal locking strategies is hard to do and increases with the numbers of parallel processes competing for the locked resource, leading to sharp declines in performance.

We encountered this problem in the ray tracer where the activity performing the computation in the slave needs to synchronize with the activity requesting that it split its work.

4.11. Summary

Obviously the list of items just discussed is not exhaustive and, depending on the application domain, other issues such as fault tolerance and security could be very important aspects of the system. Nevertheless we hold that the functionality just discussed should be central to an OO distributed system.

5. RELATED WORK

Our major research goal has been to define a language-independent layer providing the necessary support for programming parallel and distributed applications in loosely coupled distributed systems. This language-independent layer allows a number of existing OO languages to be used to write such applications. To this end we have tried to identify the key requirements on the system.

Many other researchers have focused on the problem of providing support for distributed and parallel programs in distributed systems. In many cases this support has taken the form of a new language in which

applications can be written: examples include Emerald (Black *et al.*, 1986) and Orca (Bal and Tannenbaum, 1988). In Emerald a program consists of a collection of distributed objects and processes—where a process is a (potentially) distributed thread of control. An important feature of Emerald is its support for object mobility—an object may migrate at any time. Control over migration is provided by a number of language primitives to, for example, *move* an object. The parameter passing modes *by-move* and *by-visit* are also provided to allow a parameter to a remote invocation to be passed along with the invocation request. More recently the Orca language has been proposed based on the so-called shared data-object model which hides the location of objects from the programmer but uses object replication to maintain performance. The Amber system (Chase *et al.*, 1989) is an example of a system supporting the use of an existing language to program parallel applications in a distributed system.

In the case of Amber the language is C++ extended with primitives for thread management and object mobility similar to those of Emerald. Emerald is targeted at supporting a single language at a time in a homogeneous distributed system. Extensions to Amber provide support for load-balancing.

Parallel programming in distributed systems has also been addressed in the context of distributed shared memory systems—a good example being Munin (Bennett and Zwaenepoel, 1990). Munin provides a number of mechanisms to support shared data including the delayed updates based on release consistency and type-specific memory coherence based on the identification of a number of shared data object types including: write-only, private, write-many, result, synchronization, migratory, produced-consumer, read-mostly and read-write.

6. CONCLUSION

This paper has given an overview of the Amadeus system and described how a number of applications were structured to run on it. The lessons learned from the exercise, in terms of the general functionality that a distributed object oriented system should provide, have been discussed and related work in the area sketched.

ACKNOWLEDGEMENTS

The authors would like to thank all the present and perhaps more importantly all the past members of the distributed systems group for their contributions over the years.

REFERENCES

- Black, A., Hutchinson, N., Jul, E. and Levy, H. (1986) Object structure in the Emerald system. In *Proc. Conf. on Object-Oriented Programming Systems, Languages and Applications*, pp. 78–85, ACM, New York.
- Burke, G. (1993) *Implementing Mints' problem on Amadeus*. Technical Report DSG-05-02, Trinity College, Dublin, Department of Computer Science.

- Cahill, V., Baker, S., Starovic, G. and Horn, C. (1993) The Amadeus GRT: generic runtime support for distributed persistent programming. *OOPSLA 1993 Conf. Proc.*, **28**, 144–161.
- Cahill, V., Balter, R., Rousset de Pina, X. and Harris N. (eds) (1993b) *The COMANDOS Distributed Application Platform. ESPRIT Research Reports Series*. Springer-Verlag, Berlin.
- Chase, J. *et al.* (1989) The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proc. 12th Symp. on Operating System Principles*, pp. 147–158, ACM, New York.
- Intel Corporation (1986) *iPSC Hypercube Programmers Manual*. Intel Corporation, USA.
- Cox, A. (1986) *Object Oriented Programming: an Evolutionary Approach*. Addison Wesley, Reading, MA.
- Makpangou, M. *et al.* (1991) *Structuring distributed applications as fragmented objects*. Technical Report 1404, INRIA, France.
- Fleisch, B. (1988) Distributed shared memory in a loosely coupled distributed system. In *COMPCON Spring 1988*, San Francisco, CA.
- Gustafson, J. (1988) Reevaluating Amdahl's Law. *CACM*, **31**, 532–533.
- Bal, H. and Tannenbaum, A. (1988) Distributed programming with shared data. In *Proc. Int. Conf. on Computer Languages*, pp. 82–91, Miami, FL.
- Bennett, J. and Zwaenepoel, W. (1990) Munin: Distributed shared memory based on type-specific memory coherence. In *Proc. 2nd Symp. on Principles and Practice of Parallel Programming*, pp. 168–176, Seattle, WA.
- Knuth, D. (1981) *The Art of Computer Programming. Vol. 2. Seminumerical algorithms*, 2nd edn. Addison-Wesley, London.
- Nierstrasz, O. (1986) *What is the 'Object' in Object-oriented Programming?* Technical report, Centre Universitaire d'Informatique, Université de Genève.
- Popek, G., Walker, B., Chow, J. and Edwards, D. (1983) LOCUS: a network transparent, high reliability distributed system. *ACM Operating Syst. Rev.*, **17**, 169–177.
- Tangney, B. and O'Toole, A. (1991) On overview of load balancing in Amadeus. In *Proc. ISMM Conf. on Parallel and Distributed Computing and Systems*, pp. 144–148, Washington, DC.