

# Chapter 1

## ACTIVE OBJECTS: A SOFTWARE STRUCTURE FOR CLUSTER-BASED SYSTEMS

C. K. YUEN

School of Computing  
National University of Singapore  
Kent Ridge, Singapore 119260  
email: yuenck@comp.nus.edu.sg

### Abstract

Active objects provide a two-level parallel processing structure with tight coupling for parallel functions spawned within an object and loose coupling between objects, matching the two-level structure of cluster-based system. This article describes the idea of active objects in the BaLinda parallel programming model, and examines a number of issues relating to the behaviour of active objects, including parallel object calls, atomicity and speculative processing.

### 1. Introduction

Within the general area of parallel processing, parallel programming languages provide the vital link between algorithms and hardware/operating system, but despite many years of work, this aspect has remained to be a weak link. One could hardly think of a single parallel language which has all the desirable attributes of wide availability, efficient implementation, ease of use, good supply of library programs, and large numbers of trained programmers.

This backwardness is due to a combination of factors, the most basic reason being that general concepts and standard constructs of parallel programming took some time to be properly developed, because the subject was not seen as a distinct area that requires its own unique concepts. Its practice has often been taken as an *ad hoc* matter of fitting modules of application programs into the operating environment of a given parallel machine. In consequence, language developments were too closely connected with specific problems and hardware, and lacked broad applicability. With continuing parallel hardware development, it was difficult to reach a steady state in language development.

Further, parallel language designs were often based on concurrency concepts borrowed from the study of operating systems, and lacked the simplicity and user-friendliness required of more general-purpose languages. Finally, parallel languages have not had the kind of organized international efforts which took place for the more conventional languages such as Algol, ADA or Common Lisp, since such cooperative efforts must be based on widely shared ideas about the subject of discussion.

This condition is exacerbated by the historical situation that the groups with the most resources and doing the most advanced work in parallel processing usually needed the system for just a few applications, e.g., weather forecasting, nuclear modelling or chess playing, or just one domain, e.g., Fortran programs. For such groups, program portability and common execution models would only hamper their freedom in machine design, or would require inefficient layers of system software to achieve compatibility among different machines, and so is not something to be given high priority. These influential groups have tended to set the prevailing paradigm, against the search for common models. For all too long, parallel processing meant programming in unfamiliar languages on unfamiliar machines. Without program portability, compilers and application software rarely get the desired level of debugging, optimization and cost recovery that can only come from the wide use of the same program on many machines by many users. It is therefore understandable that parallel computers almost always had difficulty to use, yet expensive, software that deter the adoption of parallel processing solutions.

In fact, we could use the analogy that parallel system developers have a tendency to see their products as Rolls-Royces, immune to the concerns of the mass market. To a certain point this is no doubt true, but the thinking also isolates Rolls-Royces from the kind of performance improvements and cost reductions that mass market cars have gone through, until they start to offer luxury models that, for a fraction of the cost, provide almost the same comfort and reliability and so become serious threats to Rolls-Royces. With single chip processors now offering multi-MFLOPS number-crunching capabilities, it is difficult to convince the users to change to parallel machines.

We do see recent signs of enlightenment in the situation. In the IBM SP2, we have a widely available, reasonably general-purpose parallel machine based on a familiar processor architecture, and the publicity it enjoyed in running the chess program beating the world champion provided an additional boost. The good, at least for time, receptions given to Linda, PVM, MPI and Java

when they came on the scene, showed that the industry has at least the desire for a common parallel programming model, though it might still be unclear about what it wants. The search for a common model has not yet reached its goal, but at least it is very much alive.

## 2. BaLinda: Fork and Join with Tuples

What kind of parallel programming models would be simple to use by application programmers? For people with extensive experience in system programming, it might seem that any tool can be learnt and “ease of use” is purely subjective. The experiences of application programmers are very different. They find most parallel tool complex and hard to use, and until we can cater to their needs, widespread use of parallel systems will remain unachievable. You cannot just ask Toyota owners to service their own Rolls-Royces.

An important part of a parallel programming model is inter process communication, and among the various competing models for IPC, the idea of Linda tuplespace is more intuitive: if you want to supply information to others, simply execute OUT (exp1, exp2,..., expN) to release an N-field tuple; if you want to receive information, use IN (exp1, exp2,..., expM? name1, name2, ..., nameN-M) or RD (exp1, exp2, ..., expM? name1, name2, ..., nameN-M), which retrieve from the tuplespace an N-field tuple whose first M fields match the results of the M expressions in the IN/RD, and then store the values of the last N-M fields into the N-M variables specified in the IN/RD. (This is a simplified form of Linda, adopted for both implementation and educational reasons.) The relation between the tasks is easily understood, as is the idea that if no matching tuple is found the task executing the IN/RD suspends until another task OUTs the required tuple.

For example, when teaching operating system concepts like mutual exclusion, which is programmed using either a semaphore or a tuple:

DOWN (semaphore)	IN (token)
... critical region ...	... critical region ...
UP (semaphore)	OUT (token)

we have found that application programmers readily take in the idea of a token held by the task currently in the critical region and released upon exit (witness the one-sentence explanation given here). In contrast, the idea of the semaphore is far more abstract, and indeed cannot be explained with brevity to a novice. Similar though somewhat lesser difficulties would be encountered