

4. Active Versus Passive Objects

Objects are defined by first specifying the generic structure and content in a class definition, and then by making instances of the class occupying memory allocated from the heap. Execution activities in an object are triggered by calls on its methods. With multiple objects existing independently, parallel execution can be readily introduced. However, this does not occur by default but by design, and decisions need to be made on a specific parallelization mechanism.

Consider the following example, involving pre-existing objects X, Y and Z:

```
X calls Y
X waits for result      Y executes
X receives result      Y idle
further processing
X calls Z
X waits for result      Z executes
X receives result      Z idle
...

```

Despite the existence of three separate objects, no parallel execution takes place between them. In comparison, we can instead have

```
X calls Y
X continues without Y result      Y executes
X calls Z                          Z executes
X continues without Z result
X demands results of Y & Z
X idle                              Y & Z reply
X resumes                          Y & Z idle
...

```

In other words, whether multiple objects produce parallel execution would depend, among other things, on the communication protocol used between them.

The requirement here is somewhat different from the spawning of parallel procedures, because a procedure can return results by assigning them to variables in the main thread. In contrast, the execution effects of an object method remain in the object, unless some explicit step to retrieve them, after

verifying their availability, is carried out. Some relevant mechanisms must be provided in the parallel object language to do this subsequent confirm then retrieve.

One simple mechanism is for a called method to return a result token or “future” to the caller immediately, so that the caller can continue with its own execution while holding this token for later use. The token is a promise to deliver a result at some later time, and before the result becomes available, any access on it would suspend the thread making this unsuccessful demand. When the result does become available, the thread demanding it will be resumed. Copies of the token could be passed to other threads so that a number of tasks can synchronize by blocking on a common result. Sequences and nestings of futures, with access of results between different sequences and nestings, can produce versatile parallel program structures.

However, copying of future tokens and passing them through complex program structures carries risks and could result in deadlock, such as two futures waiting for each other or child future suspending on the parent. Because a future token can be either a pointer to a piece of unfinished computation in another object, or a data item returned to the caller’s data space at the end of computation, it has an uncertain scoping status. A thread entitled to copy the result after X has received it from Y may not be entitled to point into Y during the computation. Further, future tokens, while they can be used for sequentialization, are not atomic, and do not provide information exchange during task execution, since a result is returned only upon termination. They are not convenient ways to produce repeated exchanges of information between two executing threads.

A more versatile technique is to return a task ID which may be used to check the execution status of a method, to look into the task’s data space, to retrieve the result if execution has completed, and to control its execution such as terminating it or assigning it different execution priorities. While this gives the programmer much freedom and power, some sophistication and knowledge are required to use it effectively. In our experience, most application programmers would not be able to handle such a tool well because it requires a background of system programming. There is again the risk of deadlock when task IDs are passed to other threads. (In Sec. 7, speculative processing, we shall present some ideas on a simple execution control mechanism designed for application programmers, without involving system programming knowledge, that deal with some of the need.)

Our view is that a call on an object method should be treated just like a procedure call: normally the caller's execution suspends during the procedure call and continues upon return, but the programmer can choose to spawn the call as a parallel thread, and can use its result after the thread has ended. If the method is a procedure, then the call changes the object state, but the caller has to retrieve the result by further (nonparallel) calls on public portion of the object:

```
make object X
spawn X.method
other processing
wait for end of thread
use X.public
```

If the method is a function, the result is returned upon function exit and may be assigned to a variable in the caller's environment:

```
spawn Y = X.method
other processing
wait for end of thread
use Y
```

Concerning the right way to wait for the end of a parallel thread, our view is that in an application program it is inappropriate to deal with system programming information, and most programmers do not have the required expertise to manage task statuses; it should neither be necessary nor even possible to know task IDs, and the spawning function should not return any such information. The wait (SYNCHRONIZE in the BaLinda languages) should simply wait for the end of the most recently spawn thread. When multiple threads are spawned, one merely waits for them individually in a last in first out order, or as a group using SYNCHRONIZE (N) where N is the total number of threads most recently spawned from the current thread.

The question of whether SYNCHRONIZE should retrieve the execution result, in addition to confirming its availability, is an intriguing one. Consider the example of adding up a set of results returned from a list of objects, evaluating in parallel (EXEC is the BaLinda task spawning command):

```
X = ListHead;
Count = 0;
Sum = 0;
```

```

{ Loop: EXEC X.method;
    Count = Count + 1;
    { IF NOT (NULL (X.next))
      => { X = X.next;
          Loop
        }
    }
}
{ FOR I = 1 TO COUNT
  Sum = Sum + SYNCHRONIZE
}

```

Each SYNCHRONIZE retrieves the result of one most recently spawned thread, until all the results have been included. (It should also be possible to store the retrieved result of SYNCHRONIZE (N) into an array but this has not been provided at present.)

The alternative of storing the result of each X.method in an array requires us to know beforehand the maximum list length so that the array could be declared, or to have some way of acquiring additional storage if the array turns out to be too short. Alternatively, the X.method results have to be stored in a linked list. Acquiring the result with SYNCHRONIZE where it is needed makes things more flexible. Further, we may have a multi-dimensional list, and wish to evaluate branches in parallel:

```

{ FUNCTION Loop < - X, Sum, Count;
  Sum = Sum + X.method;
  { IF NULL (X.Left)
    => { IF NULL (X.Right)
        => { { FOR I = 1 TO Count
              DO Sum = Sum + SYNCHRONIZE
            }
          Sum
        }
      T => Loop (X.Right, Sum, Count)
    }
  T => { IF NULL (X.Right)
        => Loop (X.Left, Sum, Count);
    }
}

```

```

T => { EXEC Loop (X.Right, 0, 0);
      Loop (X.Left, Sum, Count+1)
    } } }
...
Sum = Loop (ListHead, 0, 0);

```

Each call on Loop does one method execution in an object to add to the current sum, and causes further execution in left and right branches if they exist, proceeding in parallel if both branches are present. The number of tasks spawned off on a downward path is counted, and upon reaching the leave, the task results are retrieved and added. Each branch may do the same spawn/retrieve process.

Though in this case it is still possible to assign returned results to an array, work is required to number the elements uniquely because tasks are independently generated in the various branches and each must write to a different array element. It is simpler to sum everything in a shared atomic location such as a tuple:

```

{ PROCEDURE Loop < - X, Sum, Count;
  LOCAL Total;
  Sum = Sum + X.method;
  { IF NULL (X.Left)
    => { IF NULL (X.Right)
      => { IN (? Total);
        OUT (Total + Sum);
        SYNCHRONIZE (Count)
      }
      T => Loop (X.Right, Sum, Count)
    }
  T => { IF NULL (X.Right)
    => Loop (X.Left, Sum, Count);
    T => { EXEC Loop (X.Right, 0, 0);
        Loop (X.Left, Sum, Count + 1)
      }
  } } }

```

...

OUT (0);

Loop (ListHead, 0, 0);

IN (? Sum)

However, the competition for the Index tuple in the main thread from different branch threads may become a bottleneck. In other words, using SYNCHRONIZE to recover the task results, we avoid having to hold them until we are ready to use them, because otherwise we might have store them in locations inconvenient for the tasks producing them. SYNCHRONIZE, like a future token, links to a piece of separate, possibly unfinished, computation (the most recently spawned), but does so without using a variable ID and the link cannot be copied and passed on, thus avoiding some of the hazards of future tokens.

Note that both of the above two programs are tail recursive: they do not spawn several recursive calls, wait for their results, and combine them before proceeding to other processing, but end one recursion with another recursion, with later recursions picking up the earlier results using SYNCHRONIZE. This is valuable both for compiler optimization replacing recursion by iteration, and parallel task efficiency by delaying SYNCHRONIZE operations and reducing the chance of having to wait. For the same reason, BaLinda K does not strictly require all parallel threads to terminate before the end of the block spawning them. In the two examples, tail recursive calls cause the deletion of the stack section of the current recursion, even though a parallel thread may be open; however, formally the parent recursion does not end until the child ends, when a leave is reached, by which time all parallel threads spawned in all the recursions would be verified as having terminated with SYNCHRONIZE.

A simple way of comparing our parallelization scheme with the future scheme is to say that EXEC creates a future but does not provide a token to point to it, while SYNCHRONIZE touches the most recently created future so that waiting for uncompleted computation can occur.

Next we consider the issue of the object “body” or initialization code. A class definition has the following typical structure:

```
{ CLASS A ...;
    PUBLIC ...;
```

```

LOCAL ...;
  body
}

```

When an instance of A is made, the variables declared in the PUBLIC and LOCAL statements are assigned storage, as are the methods defined there; if the body is present, then the code specified is executed to initialize the object, before a pointer to the object is returned to the caller permitting access to the new A object. Such an object is passive, since no execution will take place inside it unless calls on its method are received.

However, consider the follow scenario:

```

spawn X = makeinstance (A)
other processing
SYNCHRONIZE

```

While “other processing” is occurring, X already points to the storage area allocated to the object, and the public information is already visible, even though the body may still be executing and changing the information. It is possible for calls to be made on X, so that there is parallel execution between the body of X and its called method. X is no longer a passive object; it contains execution of its own, which may interact with calls on it from the outside.

A pointer to an active object, like a future token, is a link to an unfinished piece of computation, but whereas accessing a future token would block a thread until computation completes, an object has no definitive final state and the public part of its content can be accessed even as computation proceeds in its body or methods. However, doing this safely would require some atomic facility, and as discussed in the next section, this is done using tuples.

One could also wait for the object initialization code the end with a SYNCHRONIZE, after which the object is no longer active, and then call its methods like a normal passive object. This merely makes object initialization and execution outside the object concurrent between the class call and the SYNCHRONIZE.

5. Objects and Atomicity

An object encapsulates a relatively self contained set of data and code. In parallel processing problems, it is natural to see an object as a unit of atomicity, which arises from the imposition of sequential entry into the object, i.e., only