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

ANNUAL REVIEW OF SCALABLE COMPUTING
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/4263.html

with the monitor or the message solution. Another simple example is N-task synchronization with a tuple:

```
... before ...
IN (tuplename? X)
OUT (tuplename, X + 1)
RD (tuplename, N)
... after ...
```

i.e., initially a tuple (tuplename, 0) was released into the tuplespace. After completing the "before" part, each task increases the numerical value by 1, and then waits for the tuple (tuplename, N) to appear, at which point the synchronization completes and everyone proceeds to the "after" part. Again this structure can be readily explained to users, in contrast to the semaphore, message or monitor solutions.

For yet another example, below is the reader-writer problem programmed using a single tuple:

```
{ PROCEDURE Writer;
      LOCAL Readers: INTEGER;
      IN ('Control, − ? Readers);
      { IF Readers > 0
        => { OUT ('Control, FALSE, Readers);
             IN ('Control, FALSE, 0)
      }       }
      ... writing ...
      OUT ('Control, TRUE, 0)
      }
{ PROCEDURE Reader;
      LOCAL Readers: INTEGER;
          Flag: BOOLEAN;
      IN ('Control, TRUE? Readers);
      OUT ('Control, TRUE, Readers +1);
      ... reading ...
      IN ('Control? Flag, Readers);
      OUT ('Control, Flag, Readers −1)
}
```

ANNUAL REVIEW OF SCALABLE COMPUTING
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/4263.html

The corresponding monitor and lock solutions, requiring separate calls for read and write lock and release, are considerably harder to teach.

We now turn to the method for process creation. The fork-and-join construct provides a simple way to retrofit parallel execution to a sequential language: the fork designates two sequences of statements in a program as parallel threads, and the join shows where the two threads merge back into one. In the Unix and C practice, forks cause the return of task IDs which the programmer must keep track of, again requiring some system programming expertise to manage any complex parallel executions. In our own work, we adopted an alternative form of fork-and-join that eases the programmer's load, by imposing the requirement that the main thread always waits for the termination of the most recent thread started by it. It follows that, if there are several parallel threads, then a succession of waits would wait for new-thread terminations in a last-in, first-out order. This makes it unnecessary for the programmers to have knowledge of thread IDs, and imposes a nesting structure for multiple parallel threads.

By adopting the fork and joint, the BaLinda model deviates from the Linda model, which uses the EVAL construct that ties task and tuple generation: EVAL (expr1, expr2,. . ., exprN) spawn N tasks, each returning one value to a resulting tuple. The main thread can wait for all N threads to end by executing an IN or RD on this tuple. To illustrate the difference of the two approaches, consider a simple parallel multiplication program which spawns N tasks for N factors which are then multiplied into a common tuple:

```
...
OUT ('Product, 1.);
{ FOR I = 1 TO N
    DO EXEC OUT ({ LOCAL X,Y: REAL;
                   X = Factor (I);
                   IN ('Product ? Y);
                   OUT ('Product, X * Y)
                 })
{ FOR I = 1 TO N
    DO IN (–)
}
IN ('Product ? Result);
```


© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/4263.html

Here EXEC OUT (), analogous to the Linda EVAL, spawns one task to compute the single expression in the (), and returns a single field tuple, which is then removed using an IN in a subsequent loop. In more standard BaLinda, the second loop is replaced by an N-task SYNCHRONIZE:

```
...
OUT ('Product, 0, 1.);
{ FOR I = 1 TO N
      DO EXEC { LOCAL X,Y : REAL,
                    X = Factor (I);
                    IN ('Product? Y);
                    OUT ('Product, X * Y)
}                 }
SYNCHRONIZE (N);
IN ('Product ? Result);
...
```

3. Objects from Functions

The BaLinda object system provides the CLASS construct for data encapsulation through a slight change to the FUNCTION construct. Whereas a function returns a result after a function call, a call on a CLASS creates a package of data and functions by returning a pointer to the execution environment, which constitutes a newly instantiated object. It is then possible to reenter the class environment and resume execution therein. Further, the CLASS specification permits some of the internal routines to be visible from outside the package, i.e., public functions or methods. In this way, it specifies entry points into an object such that, after its creation, further calls may be made on the object to execute its enclosed functions and change its state.

Consider a Node object containing an element, two pointers, and some functions:

```
{ CLASS Node < − New : INTEGER;
      LOCAL { Left, Right } : Node;
      PUBLIC
      Element : INTEGER = New,
```

ANNUAL REVIEW OF SCALABLE COMPUTING
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/4263.html