Parallel Simulation Using the Linda Language

Petr Hanáček

Abstract

The paper deals with the examples of the simulation techniques using the Linda programming language. The Linda programming language (or system) works in a distributed environment. The distributed environment is thought of as a set of processors which run in parallel and which do not share a common memory. The processors communicate only via communication links. Such a system is often called a loosely coupled multiprocessor system.

Keywords:

parallel simulation, distributed computing, event-driven simulation, demand-driven simulation, Linda language.

1. THE LINDA LANGUAGE

Linda is a language that was developed at Yale University and it is copyrighted by Scientific Computing Associates, Inc. Linda, however, lacks common features of usual programming languages. Linda does not define such things as variables, statements, and a syntax of programming structures. All of these common features are provided by some language that is called a base language. The Linda system is obtained by adding the small number of Linda operators to any of the sequential languages such as Pascal or C. Linda operators are used for the creation of processes running in parallel, for communication purposes, and for a synchronization of processes. The number of Linda operators is small, and they are quite simple. So, it is easy to understand them and use them. Linda is based on an associative memory model. An elementary memory unit is called a *tuple*. The tuple is an ordered collection of fields (called elements), and it is similar to a record in the relational database theory. Each element of the tuple has a type associated with it. The type is one of the valid types allowed in the base language. Linda defines three types of elements. *Constants* are thought of in the same meaning as constants in the base language. *Actuals* are names of variables that are used as input parameters of Linda operators. *Formals* are names of variables preceded by a question mark. Formals are used as output parameters of Linda operators.

The *tuple space* is a collection of tuples. The tuple space can contain theoretically unlimited number of copies of the same tuple. The tuple space is a global shared object, and each process has access to it.

Following rules are given for matching two tuples:

- To be a candidate for matching, the number of fields in the tuple and the their types must be the same.
- Actuals match actuals, constants match constants, and actuals match constants if they are of the same type and if they have the same value.
- Actuals match formals and constants match formals if they are of the same type (a formal has no value).

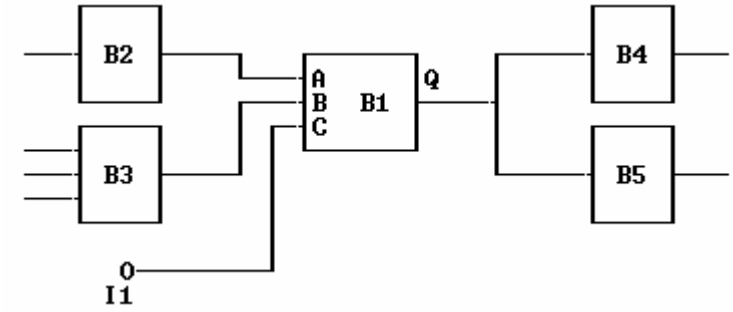
2. LINDA OPERATORS

Linda defines only six operators: out(tuple), rd(tuple), in(tuple), inp(tuple), eval (list_of_arguments).

- *out(tuple)* - this operator is used for putting the tuple into the tuple space. The process that performed the operation is not blocked. For example, the operation out('count',2) will put the tuple ('count',2) into the tuple space.
- - *rd(tuple)* - the operator rd is used for reading values of elements from the tuple (placed in the tuple-space) that matches the argument. The process that executes this operation is blocked until the matched tuple is found. For example, the operation rd('count',?x) will match the tuple generated in the previous example from the tuple space. After finishing the operation the variable x will contain the value 2.
- *in(tuple)* - this operator is similar to the rd() operator with one exception; the matched tuple is removed from the tuple space.
- - *rdp(tuple), inp(tuple)* - these operators are the predicate versions of the rd() and in() operators. They are non-blocking operators. They return True if the matched tuple is found in the tuple space, otherwise they return False.
- *eval(list_of_arguments)* - this operator allows to create processes executed in parallel. An argument can be an element or a closure. Closure is a pair consisting of the values of all free variables defined within a function along with the text (code) of the function body. Linda evaluates all closures in parallel and the result is placed into the tuple space. For example eval('result',load(a),load(b)) will evaluate two functions load in parallel and the tuple ('result',value1,value2) will be placed into the tuple space. The value1 is the result of the function call load(a), and the value2 is the result of the function call load(b).

3. SIMULATION IN THE LINDA LANGUAGE

Using of the Linda language for the simulation purposes will be shown in two examples. The model that we use consists of five blocks. Each block has several inputs and one output. The connection of blocks is shown on the following picture:



We will show program in Linda language which describes the B1 block. This block has three inputs A, B, C and one output Q. Value of the Q output is dependent on the input values and the model-time. We will show both the continuous simulation approach and the synchronous demand-driven simulation approach.

4. CONTINUOUS SIMULATION IN THE LINDA LANGUAGE

In the case of continuous simulation we will use two types of tuples. The first one is keeping the current model-time, the second one is keeping the value of output of each element.

```
Tuple ("Time", Step, Time);
Tuple ("Value", "ElementName", Value);
```

The program which describes the B1 element is an endless loop. The body of the loop in each cycle reads the model time, reads the input values of the element, calculates the output value and sends it to the tuple space.

```
void B1 () {
   for (;;) {
     rd ("Time", Step++, ?Time); // Reading model time
     in ("Value", "B2", ?A);     // Reading input A
     in ("Value", "B3", ?B);     // Reading input B
     in ("Value", "I1", ?C);     // Reading input C
     Q =function (A, B, C, Time);// Calculating output
     out ("Value", "B1", X);     // Sending output value
     out ("Value", "B1", X);     // twice - fan-out is 2
     }
   }
```

5. SYNCHRONOUS DEMAND-DRIVEN SIMULATION IN THE LINDA LANGUAGE

In the case of continuous simulation we will use two types of tuples. The first one is keeping the value of output of each element, the second one is used for the synchronization purposes.

```
Tuple ("Value", Step, "ElementName", Value);
Tuple ("Complete", ValueA, ValueB, ValueC);
```

The program which describes the B1 element is called only when the output value of this element is requested. The program first checked, if this value is yet computed. If not it requests evaluating of the three elements connected to its inputs (B2, B1 and I1 elements). When this is done, program reads results, calculates its own output value Q and sends it to the tuple space.

```
logval B1 (Step) {
   if (rdp ("Value", Step, "B1", ?X)) return X;
       // See if block is already evaluated
   eval ("Complete", B2(Step), B3(Step), I1(Step));
       // Create 3 processes for evaluating the input blocks
   in ("Complete", ?A, ?B, ?C);  // Read results
   Q =function (A, B, C, Time);  // Calculating output
  out ("Value", Step, "B1", X);  // Sending output value
  return X;                      // Returning output
  }
```

6. CONCLUSIONS

The previous examples are very simple, but they show how to solve some simulation problems using the Linda language. Using this language is not only another approach to the already solved problems. It allows to speed up simulation many times because allows user-transparent division of the computational load between many processors in the distributed computing environment.

7. REFERENCES

[1] Jagannathan, S.: Optimizing Analysis for First-Class Tuple-Spaces, in Advances in Languages and Compilers for Parallel Processing, Pitman Publishing, London 1991, ISBN 0953-7767

[2] Shekhar, K., H, Srikant, Y., N.: Linda Sub System on Transputers, in Transputing 91, IOS Press 1991

[3] Carriera, N., Gelernter, D.: The S/Net's Linda Kernel, ACM Trans. Comp. Sys. (May 1986)