Solving the Linda multiple `rd` problem

Antony Rowstron and Alan Wood*

Department of Computer Science, University of York,
York, YO1 5DD, UK.

Abstract. Linda is a co-ordination language that has been used for many years. From our recent work on the model we have found a simple operation that is widely used in many different algorithms which the Linda model is unable to express in a viable fashion. We examine a function which performs the composition of two binary relations. By examining how to implement this in parallel using Linda we demonstrate that the current methods are unacceptable. A more detailed explanation of the problem, which we call the *multiple* `rd` *problem* is then presented, together with some other algorithms which have the same problem. We then show how the addition of a primitive to the Linda model, `copy-collect`, extends the expressibility of the model to overcome this problem. This work builds on previous work on the addition of another primitive called `collect`[1]. The parallel composition of two binary relations is then reconsidered using `copy-collect` and is shown to be more efficient.

1 Introduction

Linda is an asynchronous model of concurrency, which allows parallel programs to be developed which are highly decoupled; in other words each process knows little or nothing about the other processes during computation.

Whilst working on the implementation of parallel image processing algorithms in Linda[2] it became clear that the Linda model was unable to support a certain operation, which we will refer to as the *multiple* `rd` *problem*. The operation however is one whose existence is often needed in parallel algorithms, where information is stored in tuple spaces for many processes to non-destructively access in parallel.

In order to demonstrate the problem we focus on the implementation of several algorithms to perform the composition of two binary relations. However, before considering the example, a brief overview of Linda is presented.

2 The Linda Model

The Linda model is described in detail in many papers[3]. The Linda Model is intended to be an abstraction, and as such is independent of any specific machine architecture. This has meant that alternatives and extensions to the

* {ant,wood}@minster.york.ac.uk

basic Linda model have been proposed and investigated. The extensions that are used currently in the York Linda kernel[4] are:

multiple tuple spaces The addition of multiple tuple spaces has been discussed for some time. Schemes based on hierarchies of tuple spaces have been suggested[5, 6], which involve the concept of active and frozen tuple spaces. The addition of multiple tuple spaces is achieved by incorporating a `tuple space` type and a primitive to create a new tuple space. For instance in `ISETL`-Linda[7], a type `bag` represents a tuple space, and the primitive `NewBag` creates values of type `bag`.

`collect` **primitive** A new tuple-space primitive, `collect`[1]. This primitive by its very nature requires multiple tuple spaces. Given two tuple space handles (`ts1` and `ts2`) and a tuple `template`, then `collect(ts1, ts2, template)` moves tuples that match `template` in `ts1` to `ts2`, returning a count of the number of tuples transferred.

3 Parallel Composition of Two Binary Relations

3.1 Introduction

In order to demonstrate the multiple `rd` problem, we will consider the implementation of an algorithm for the parallel composition of two binary relations. We will use a strategy that appears natural to the problem, and then consider how to implement it to overcome the multiple `rd` problem.

A binary relation is defined to be a set of ordered pairs. Given two binary relations, S and R, their composition, $R \circ S$, is defined to be:

$$\{(a, b) \mid (a, x) \in R \text{ and } (x, b) \in S\}$$

We assume that the elements of each set are held in separate tuple spaces, with each tuple representing an ordered pair. After performing the composition, a new tuple space will be created containing the resulting tuples. This is demonstrated in Figure 1.

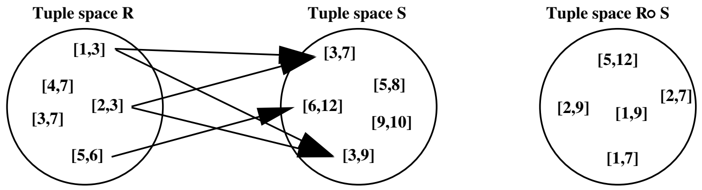

Fig. 1. Composition performed between two tuple spaces

3.2 Implementations

It is clear that an algorithm to do this can be implemented in parallel – every tuple in tuple space R is compared with every tuple in S in parallel. If the second item in a tuple from R matches the first item of a tuple in S, then a result tuple is placed in the result tuple space. At first sight it would appear that this problem is particularly well suited to Linda because of the matching element.

The most obvious solution is to have one worker for each element in the set R. Each of these workers will take a tuple from tuple space R. They will then attempt to find all tuples in tuple space S where the first element of the tuple is the same as the last element of the chosen tuple from tuple space R. It should be noted that any tuple in S may be needed by more than one worker process and any one process may need more than one tuple from S. Initially, the obvious idea is to use a `rd`. However, because more than one tuple from S may be required by a single process, repeated uses of the `rd` primitive might always return the same tuple. Therefore, how do the different workers access all the distinct tuples in tuple space S?

There are two basic approachs to the problem. We will outline each approach, and show the weaknesses that are inherent within them. In each case we present the ISETL-Linda[7] code for each worker.

Using tuples as semaphores The first solution to the problem is to use a tuple that acts as a semaphore (a lock tuple) on the tuple space S. Each worker takes a tuple from R, then attempts to grab the *lock* tuple in S. There is only one such tuple in the tuple space S, and therefore it acts like a binary semaphore. Once a worker has the tuple, it has unrestricted access to the tuple space S. The worker creates a template using the second element of the tuple read from R as the first element of the template. This template is then used by a `collect` to *move* all[2] the tuples that match the template in S to a local tuple space. Therefore, only tuples that are going to be used are moved from S. The worker then `in`s each tuple from the local tuple space and replaces the tuple back into S, and also outputs the composition result into a tuple space C. Finally, once all the tuples in the local tuple space have been processed and replaced into the tuple space S, the *lock* tuple is replaced in S. This means that S contains all the tuples it did when the worker started. It should be noted that the tuple which acts as the semaphore can only be replaced in the tuple space when the tuple space is in its original state. If the tuple is returned before this is true then the other processes can not guarantee duplicating all the tuples that they should. The `ISETL`-Linda code for a worker is shown in Figure 2.

It should be noted that although the `collect` primitive is used in this solution it is possible to create an implementation that is similar, but just uses the standard Linda primitives if the implementation supports a predicated

[2] In this case we can say all because the tuple space will be *inactive* if all processes adhere to using the semaphore tuple.

```
comp_worker := func(R,S,C);

  local my_val, my_ts, my_comb, dummy;

  my_ts := NewBag;
  my_val := lin(R,|[?int,?int]|);

  dummy := lin(S,|["lock"]|);

  todo := lcollect(S,my_ts,|[my_val(2),?int]|);
  while (todo > 0) do
    todo := todo - 1;
    my_comb := lin(my_ts,|[my_val(2),?int]|);
    lout(C,[my_val(1),my_comb(2)]);
    lout(S,my_comb);
  end while;

  lout(S,["lock"]);

  return ["TERMINATED"];
end func;
```

Fig. 2. Worker using binary semaphore or lock tuple

version of `in`. If the implementation does not support the predicated `in`[3] then only the second approach (detailed below) can be used.

This solution is unacceptable because it creates a sequential bottleneck. If more than one process wishes to perform the operation at the same time then it produces a bottleneck, as first one process gets the semaphore and then the next. In the worst case, as is the situation here, the whole program degenerates into a sequential program. All the workers wish to access the tuple space at the same time but only one at a time can.

Streams The second approach is to use what is called a *stream*. This involves the addition of a new unique field to each tuple. This unique field then makes each tuple in the tuple space different. As long as the workers know how the unique field is generated (for example using a counter) each worker can access each tuple using a `rd`, and the accesses can occur concurrently. This is because all tuples are now distinct, and hence the `rd` primitive will never match multiple tuples as the unique field is specified in the template. The `ISETL`-Linda code for a worker is shown in Figure 3.

With this solution all the workers can perform the searching of the second tuple space in parallel. However, there are two problems with this approach that makes it unacceptable:

[3] Some implementations do not because of semantic problems with such a primitive.

```
comp_worker := func(R,S,C,NumTupS);

  local my_val, my_comb;

  my_val := lin(R,|[?int,?int]|);

  while (NumTupS > 0) do
    my_comb := lrd(S,|[NumTupS,?int,?int]|);
    NumTupS := NumTupS - 1;
    if (my_comb(2) = my_val(2)) then
      lout(C,[my_val(1),my_comb(3)]);
    end if;
  end while;

  return ["TERMINATED"];
end func;
```

Fig. 3. Worker using streams

- Every tuple in the tuple space requires a unique field to be added, and all the processes using the tuples must be aware of the unique field and how it is generated. This removes the natural use of a tuple space as the data structure, by adding another structure (a stream) to the tuples within it. In order to achieve this either the producer must be aware of the need to add this unique field, in which case the cost of adding the field is minimal, or the tuples have to be preprocessed before use, which incurs additional and unwanted time costs.
- It also largely negates the advantages of the tuple matching abilities of Linda. *Every* tuple in the stream structure *must* be read. If there are several thousands of tuples in a tuple space and only one percent actually match, then the cost of reading every tuple is enormous. If the implementation does not support a predicated version of `rd` then additional checking of the returned tuple will be required to see if the fields match. Even if the majority of tuples match, any tuple that does not match introduces unnecessary and unwanted time costs.

Table 1 presents the execution times on our system[4] of a sequential version of the problem (using tuple spaces), a version using a binary semaphore and a version using streams. In each case the cardinality of R was five, the cardinality of S was 50 and the number of elements that each element of R should have matched with in S was four, producing a set C with cardinality of 20. It should be noted that the time to alter the data structures and spawn the workers is not included in the execution times.

As can be seen none of the parallel versions represent a noticeable speed up of the sequential version. We call the problem observed here the *multiple* `rd`

Version	Execution time in ticks (arbitrary units)
Stream version	11670
Sequential version	7885
Semaphore/Lock version	7143

Table 1. Experimental results

problem.

3.3 Generalisation of the problem

The multiple `rd` problem is not something that is exclusive to the parallel composition of two binary relations. We now present a fuller description of the multiple `rd` problem, and indicate some other areas where the problem can be observed.

This problem arises when multiple processes wish to *non-destructively read* a subset of the tuples in a tuple space. The repeated use of a `rd` in this situation is incorrect as it may result in the same tuple being read more than once. Within the "standard" Linda model the only solution to this problem is to use some sort of a binary semaphore or to use a stream, or some hybrid of the two methods. With the semaphore approach if a process wishes to read a subset of the tuples in the tuple space the process moves all the required tuples to a temporary tuple space, and then destructively reads them using an `in` and returns them to the original tuple space. If more than one process wishes to "read" a subset of the tuples in a tuple space then those *tuples* have to be "locked" whilst the copying is taking place because if any other processes were to try to copy the same tuples at the same time then both processes may fail to get a complete copy of all the possible tuples. The alternative, using a stream structure, requires the addition of a unique field to all the tuples and the reading of all tuples in a tuple space in order to detect all potential matches.

What makes this multiple `rd` problem more frustrating is that it would seem possible that several non-destructive reads of a tuple should be possible in parallel. Within the Linda model there is no notion of synchronisation between primitives, and hence two Linda primitives can be executed concurrently, and indeed the York Linda kernel[4] supports concurrent primitive operations.

We now consider other areas and algorithms where the multiple `rd` problem can be observed. Work on the parallel implementation of many different image processing algorithms in Linda has shown this to be a common problem. If, for example, a binary image is stored in a tuple space many image processing operations require to access only the pixels which have a value of one (or are "on"). Many low and intermediate level image processing operations use a repetitive process applied to every pixel, indicating that there will be many workers, sharing the image (for example the Hough transform). How can these multiple workers access the pixels with a value of one, without either checking every pixel[4] or locking the image? This is another example of the multiple `rd` problem.

Another example is in the use of persistent tuple spaces to store information. It has for a long time been suggested by some that persistent tuple spaces are a natural way to store information. Let us consider an example where a tuple space is being used to store information, perhaps tuples that contain names and addresses. What happens if several people concurrently wish to access the information? In the database world this would represent no problem: I wish to retrieve the addresses of all people whose name is "Smith". In a large database there are likely to be more than one tuple (record) which contains this name. The entire tuple space (database) would have to be locked or traversed in order to allow the retrieval of these tuples (records). This is clearly unacceptable.

4 The `copy-collect` primitive

In order to overcome the multiple `rd` problem we are investigating the use of an additional Linda primitive; `copy-collect`. It is similar to `collect`[1]:

copy-collect (ts1, ts2, template) This primitive *copies* tuples that match `template` from one specified tuple space (`ts1`) to another specified tuple space (`ts2`). It returns the number of tuples copied. This differs from a `collect` in that a `collect` *moves* the tuples, and therefore is a destructive operation on the source tuple space, where as `copy-collect` *copies* the tuples from the source tuple space, and is therefore non-destructive. If the source tuple space (`ts1`) is inactive (in other words no other processes are performing operations which modify its contents) then all the tuples which match the template will be copied from the source tuple space (`t1`). If a modifying operation is performed on the source tuple space at the same time then the outcome is nondeterministic, as it would be if an `in` and a `rd`, for example, where performed on the same tuple at the same time.

Linda is an asynchronous system, and therefore at the model level many operations can occur in parallel. Therefore, if two process happen to do a `rd` at the same time on the same tuple there is no reason why they could not be serviced concurrently. The `copy-collect` primitive is the same, many different processes can concurrently do a `copy-collect` and all these can be serviced concurrently. Therefore, many processes can *independently* and *asynchronously* produce a copy of a number of tuples in the same tuple space.

It might appear as though the primitive moves the bottleneck of using an explicit semaphore to lock a tuple space into the implementation - it appears to perform the same operations as indicated in the code fragment presented in Figure 2, except in the kernel. This of course can be the case. However, in our implementation the tuple space is distributed over many different processors

[4] In this case, if the image dimensions are known the image coordinates for each pixel acts as a unique field.

which can perform the operation in parallel. Therefore, the implementation is in effect locking several small sections of the tuple space and performing the operation in parallel. Hence, the effects of the bottleneck have been reduced. The cost of performing a `copy-collect` is similar to the cost of performing an `in` that blocks[4]. The primitive has been added to the York Linda kernel[4, 8].

5 Implementation - revisited

We can now reconsider the multiple `rd` problem, by examining how `copy-collect` would be used for the parallel composition of two binary relations. The new worker is shown in Figure 4. As with the first implementations each worker takes a tuple from tuple space R, and then uses the new primitive to copy all the tuples which it will require from tuple space S to a local tuple space. This is done by creating a template that will match all the tuples that are required and then performing a `copy-collect`. The copied tuples are then destructively read from the local tuple space, without affecting the source tuple space.

```
comp_worker := func(R,S,C);

  local my_val, my_ts, todo, my_comb;

  my_ts := NewBag;

  my_val := lin(R,|[?int,?int]|);
  todo := lcopycollect(S,my_ts,|[my_val(2),?int]|);
  while (todo > 0) do
    todo := todo - 1;
    my_comb := lin(my_ts,|[my_val(2),?int]|);
    lout(C,[my_val(1),my_comb(2)]);
  end while;

  return ["TERMINATED"];
end func;
```

Fig. 4. Code segment showing the worker using `copy_collect`.

Table 2 shows the execution times for the `copy-collect`, stream and semaphore version using the same data sets as for the first results presented in Table 1. As can be clearly seen the execution time for the `copy-collect` is significantly smaller than the other versions, and represents a speed up over the sequential version.

Table 2 also contains an additional result for a *coarser approach*. It had been suggested by an expert Linda programmer that the general approach that we were considering was a poor one, as it was too fine grained and consequently we should use a more coarse grained approach, where the contents of the set S are coded as a single tuple. All the workers start up, read a tuple from tuple space R as before but then read the single tuple in S, using some sort of local data structure to store the returned tuple and for the calculation of the matching elements. One of the underlying principles of our work is the abstraction away from large data structures with single processes and the use of a tuple space as a (distributed) data structure in its own right. Hence, although we could understand this view, we felt that it compromised this principle. However, we examined this approach and the execution time for the coarser version is shown in Table 2. As can be seen the execution time is comparable to the semaphore/lock version (and sequential) but the `copy-collect` version is still considerably faster.

Version	Execution time in ticks
Stream version	11670
Coarser approach	7244
Semaphore/Lock version	7143
`copy-collect`	4579

Table 2. Experimental results II

This has shown how the new primitive would be used, and it can be seen how it would work for all cases where a multiple `rd` is required, and therefore solves the *multiple* `rd` *problem*. In general, a worker creates a "local" copy of the tuples that it requires using a `copy_collect` and then destructively reads them from that local tuple space. For example, given a tuple space containing an image with each pixel a separate tuple (`[x_coord, y_coord, value]`) the command: `copy_collect(image_ts, local_ts, |[?int, ?int, 1])` would copy all the tuples with a value of one into the local tuple space.

6 Alternative proposals

As with any abstract model there have been many proposals to alter the model. One that is particularly of relevance here is the proposal for another primitive; `rd()all`[9][5]. The informal semantics of `rd()all` are:

rd (template)all(function) This primitive will apply the *function* to all tuples in a tuple space that match the *template.*

Anderson[9] notes that there are specific problems with such a primitive. The suggestion is that the operation is not atomic, so essentially a cycle is created where a tuple is fetched, the function applied to it, and then the next

[5] The same primitive appears to have been suggested under a number of other names including `rd*`.

tuple fetched. He states that this is due to the implementational difficulties of creating an atomic primitive. However, such a primitive raises much deeper questions whether it is perceived as atomic or non-atomic. What happens if the `function` removes tuples from the tuple space that the `rd()all` would match? What if the function adds tuples to the tuple space? Is there any reason why the function should not be executed in parallel? It would also imply that the primitive has the "interesting" ability to *livelock*, especially if it is not atomic. It is also unclear what information the primitive actually returns. However, it should also be noted that `rd()all` does not require multiple tuples spaces, and could be incorporated into systems with or without them.

There are however, a number of interesting things that this primitive does provide, such as the ability to unify across multiple templates. However, again the exact semantics are not specified, and this could be difficult to implement.

The `copy-collect` primitive is much simpler. It does not attempt to fold communication and computation into the same primitive. It also returns information which is very valuable to the programmer. The information allows the number of workers `eval`ed to be controlled for example. If there are many tuples that match, more workers will be required than if fewer tuples match.

7 Conclusion

We have demonstrated the multiple `rd` problem, and have shown how the addition of a new primitive to the model can overcome the problem.

Since the focus of this paper is on the *need* for `copy-collect`, we have not given details of the implementation of the primitive. However, the primitive has been implemented in our distributed kernel. The cost in terms of messages between the different distributed sections of the tuple space is comparable to an `in`. It has been suggested that `copy-collect`, because of the duplication of the tuple spaces, may require large amounts of memory to cope with all the tuple duplication. Although currently physical duplication does occur in our implementation, a tuple storage method where tuples are not duplicated has been designed.

In order to ensure that the `copy-collect` performs as the informal semantics indicate, then the implementation *must* support the *ordering of* `out`*s*. That is if a single process creates a local tuple space and then performs two `out` operations, the tuple space must never contain the second tuple and not the first. We think that this is a logical thing to assume as an `out` is a non-blocking primitive and can therefore be considered atomic. However, some implementations[10] do not support the ordering of `out`s.

Recent work has shown that `copy-collect` has other uses as well as solving this problem[11]. One of the biggest problems for fine grained parallel programming using the Linda model is the need to add *extra* synchronisation. This extra synchronisation can often rapidly become a bottleneck, causing very poor scalability. We have successfully used the primitive to act as a means for polling the condition of processes, so each process maintains it own state tuple in a common tuple space. A `copy-collect` allows a broker process to interrogate the states tuples, without affecting the workers, and determine if termination has occurred.

Acknowledgements

During this work Antony Rowstron was supported by a CASE studentship from British Aerospace Ltd, and the EPSRC of the UK. The authors would like to thank Andrew Douglas for his comments, and Nicholas Carriero and David Gelernter for their advice on how they would implement the example algorithm.

References

1. P. Butcher, A. Wood, and M. Atkins. Global synchronisation in Linda. *Concurrency: Practice and Experience*, 6(6):505–516, 1994.
2. A. Rowstron and A. Wood. Implementing mathematical morphology in ISETL-Linda. In *IEE 5th International Conference on image processing and its applications*, pages 847–851, 1995.
3. N. Carriero and D. Gelernter. *How to write parallel programs: A first course*. MIT Press, 1990.
4. A. Douglas, A. Wood, and A. Rowstron. Linda implementation revisited. In P. Nixon, editor, *Transputer and occam developments*, Transputer and occam Engineering Series, pages 125–138. IOS Press, 1995.
5. D. Gelernter. Multiple tuple spaces in Linda. In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE '89: Parallel Architectures and Languages Europe. Volume II: Parallel Languages*, pages 20–27. Springer-Verlang, Lecture Notes in Computer Science Volume 366, 1989.
6. S.C. Hupfer. Melinda: Linda with multiple tuple spaces. Technical Report YALEU /DCS/RR-766, Yale University, 1990.
7. A. Douglas, A. Rowstron, and A. Wood. ISETL-LINDA: Parallel programming with bags. Technical Report YCS 257, University of York, 1995.
8. A. Rowstron, A. Douglas, and A. Wood. A distributed Linda-like kernel for PVM. In J. Dongarra, M. Gengler, B. Tourancheau, and X. Vigouroux, editors, *EuroPVM'95*, pages 107–112. Hermes, 1995.
9. B. Anderson and D. Shasha. Persistent Linda: Linda + Transactions + Query Processing. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, volume 574 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.
10. Scientific Computing Associates. *Linda: User's guide and reference manual*. Scientific Computing Associates, 1995.
11. A. Wood and A. Rowstron. Deadlock and algorithm design: Stable marriages in Linda. *Submitted to Parallel Processing Letters*, 1995.