Solving the Linda multiple `rd` problem using the `copy-collect` primitive

A.I.T. Rowstron[a] and A.M. Wood[b]

[a] *Computer Laboratory, University of Cambridge, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK*

[b] *Department of Computer Science, University of York, Heslington, York, Y01 5DD, UK.*

Linda is a mature co-ordination language that has been in use for several years. However as a result of recent work on the model we have found a simple class of operation that is widely used in many different algorithms which the Linda model is unable to express in a viable fashion. An example algorithm which uses this operation is the composition of two binary relations. By examining how to implement this in parallel using Linda we demonstrate that the approaches possible using the current Linda primitives are unsatisfactory. This paper demonstrates how this "multiple `rd` problem" can be overcome by the addition of a primitive to the Linda model, `copy-collect`. This builds on previous work on another primitive called `collect`[3]. The parallel composition of two binary relations using the `copy-collect` primitive can be achieved with maximal parallelism.

Key words: Linda, co-ordination models, multiple `rd` problem.

1 Introduction

Linda[5] is a co-ordination language, providing mechanisms to allow concurrent process to communicate. The process communication is asynchronous and is performed through tuple spaces, which are associative shared memories. Processes which communicate can be both spatially and temporally decoupled. Indeed, two process can communicate even if one has terminated before the other has started. The Linda model is described in more detail in Section 2.

While working on the implementation of parallel image processing algorithms in Linda[20] it emerged that there is a particular operation that the Linda model can not perform adequately. The operation is described as a *multiple* `rd`. In order to demonstrate the multiple `rd` problem an example is used, the parallel composition of two binary relations. The example is introduced in

Section 4. Two approaches are suggested that use the current Linda primitives in Sections 5 and 6, and through experimental results presented in Section 7 the problems of both approaches are shown. A new primitive is then introduced in Section 8, and how this primitive overcomes the multiple `rd` problem is then shown. The three approaches are then compared at a theoretical level, with the use of a simple model for each of the methods.

2 The Linda Model

A detailed description of Linda can be found in Carriero et al.[5]. The primary primitives are:

out(tuple) This places the tuple into a tuple space.
in(template) This removes a tuple from a tuple space. The tuple removed is associatively matched using the template[1] and the tuple is returned to the calling process. If no tuple that matches exists then the calling process is blocked until one becomes available.
rd(template) This primitive is identical to *in* except the matched tuple is not removed from the tuple space, and a copy is returned to the calling process.
eval(active-tuple) The *active-tuple* contains one or more functions, which are then evaluated in parallel with each other and the calling process. When all the functions have terminated a tuple is placed into the tuple space with the results of the functions as its elements.

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 basic Linda model have been proposed and investigated. The extensions that are used currently in the York Kernel I and II[22,8] are:

multiple tuple spaces The concept of multiple tuple spaces was introduced by Gelernter as part of Linda 3[10], and involved the addition of a type `ts` and a primitive `tsc`. The idea of adding multiple tuple spaces has led to many different proposals of how multiple tuple spaces could be incorporated within Linda[12,13], and many implementations include multiple tuple spaces in one form or another[8,19,11,17,14,15].

Multiple tuple spaces were introduced as an effective way of hiding information. Information within a tuple space can *only* be accessed by those processes that know about the tuple space. As the use of Linda has changed to incorporate different styles of distributed computing the need to hide tuples has become increasingly important to ensure that other processes

[1] Sometimes referred to as an *anti-tuple*.

neither maliciously nor accidently tamper with the tuples that other processes are using. When multiple tuple spaces are added to Linda there are two issues: whether tuple spaces are first class objects, and the relationship between the tuple spaces. If the tuple spaces are first class objects then a tuple space can be an element of a tuple, and treated like any other first class value. Generally most implementations do not support first class tuple spaces because making tuple spaces first class introduces many complex questions involving the semantics of the primitives, for example what does it mean to insert a tuple into a tuple space that another process has consumed? The tuple spaces can either be unrelated or they can be related creating some form of hierarchy[12,13]. Most implementations adopt a flat structure. As far as the work described in this paper is concerned the relationship between tuple spaces is not important, just the fact that multiple tuple spaces are incorporated into Linda.

The `collect` primitive [3]. 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. This primitive by its very nature requires multiple tuple spaces.

2.1 Linda Semantics

Linda was originally proposed as what might be characterised as an 'engineering solution' to the problem of coordinating parallel processes independently of their computational aspects. This resulted in an loosely, and informally, specified semantics for the Linda operations, and then Linda developed incrementally in an evolutionary manner from this informal basis.

However, as research interest, and implementational work, increased it became clear that the lack of a formal specification of Linda's operations was needed, not least because different implementors interpreted the informal definitions in different ways, giving rise to a number of inconsistent *de facto* 'semantics'. Consequently, many attempts (for example [2,7,6,13,16]) have been made to specify the meaning of the extant Linda operations *post facto*. This is a much more difficult task than defining the semantics of a system from scratch, without the demands of being 'backwardly compatible' with existing informal definitions. This difficulty is highlighted by the fact that, despite the many attempts, there is no generally accepted *formal* specification for Linda.

The main reason for the lack of generally accepted Linda semantics is that the 'design space' — the set of *reasonable* alternative interpretations of the Linda operations — is very wide (a recent survey [4] of some of the open questions identified a dozen more or less orthogonal choices which could be made). This indicates that there is unlikely to be a uniquely “correct” specification of Linda. In addition, most of the formal systems used in specifying the semantics of computational systems have some difficulty in capturing some of the intuitively fundamental properties of Linda — principally due to its combination of non-determinism and asynchrony — in a way which useful to a Linda implementor, or natural to a Linda user.

This lack of a generally accepted formal semantics has meant that as new proposals for additions or modifications to Linda are made, an *informal* description of their meanings is the norm, as will be the case in section 8 when we introduce the `copy-collect` primitive as a solution to the multiple-`rd` problem. If and when a suitable semantic formalism is developed, the informal descriptions will be more rigorously specified — in fact discovering how well proposed formalisms capture the intuitive meanings of such operations will be a measure of their quality.

3 The multiple `rd` problem

In this section, an expressive limitation of the Linda model is identified, which is referred to as the *multiple* `rd` *problem*. A multiple `rd` is defined as an operation where two or more processes are required to concurrently, and non-destructively read one or more tuples from a tuple space which match the same template, where there are at least two or more tuples that match the template, and at least two of the processes can be satisfied by the same tuples. The problem is that a multiple `rd` cannot be performed efficiently using the current Linda model if two or more tuples are concurrently and non-destructively read from a tuple space using the same template.

As an example consider a tuple space containing a number of tuples with each containing two fields, representing peoples' names such as $\langle \text{“Antony”}_{string}, \text{“Rowstron”}_{string} \rangle$. This tuple space is shared among many processes that may require access to the tuples. How would all the tuples representing people whose surname is *Rowstron* be retrieved by a process?

Initially, the answer would appear to be the repeated use of the `rd` primitive. The template $\langle| \square_{string}, \text{“Rowstron”}_{string} |\rangle$ will match a tuple whose surname is *Rowstron*. This will only work if there is a single tuple which matches the template. If all the names of an entire family are in the tuple space, or there are several unrelated people with the same surname stored in the tuple space, the repeated use of a `rd` will not work. The semantics of `rd` mean that if more than one tuple matches a template the tuple returned is chosen non-deterministically.

There are only two methods that enable many processes to concurrently and non-destructively access a tuple space using the standard Linda model. One method is to use a designated tuple as a binary semaphore and the other is to organise the tuples as a stream. Before these two methods are examined and evaluated another example containing the multiple `rd` problem is described.

4 Parallel composition of two binary relations

A binary relation is defined as a relation defined between two sets. The binary relation defines a subset, B, of the Cartesian product of the two sets. Therefore, given two sets, T and X, the Cartesian product, $T \times X$, is defined as:

$$\{(t, x) : (t \in T) \text{ and } (x \in X)\} \tag{1}$$

If the ordered pair (s_1, s_2) belongs to the set B, then the binary relation B is said to hold between the two values. This binary relation could for example be "less than", so $s_1 < s_2$. Given two binary relations, R and S, their composition, $R \circ S$, is defined as:

$$\{(a, d) : ((a, b) \in R) \text{ and } ((c, d) \in S) \mid b = c\} \tag{2}$$

In this example it is assumed that the ordered pairs in each set are held in separate tuple spaces, with each tuple representing a single ordered pair. After performing the composition, a new tuple space will be created containing the resulting tuples. This is shown in Figure 1.

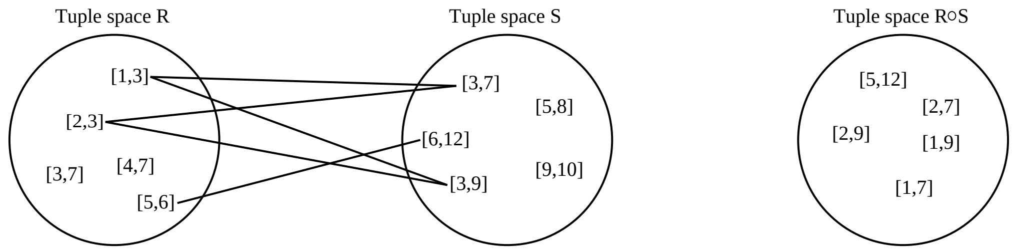


Fig. 1. Composition of two binary relations represented using three tuple spaces.

Due to the properties of the composition of binary relations it should be simple to implement in parallel, with every pair in tuple space R being compared with every pair in tuple space S concurrently. The results for each pair in tuple space R are independent of the results for any other pair in tuple space R. So a number of processes are used. Each process takes a pair from tuple space R, and checks the chosen pair with every pair in tuple space S. If the second element of the pair from tuple space R is the same as the first element in a pair from tuple space S, a new pair is produced. This new pair contains the first element of the pair from tuple space R and the second element from the pair from tuple space S. The finest grained parallel approach using this method will use a process for every pair in tuple space R. The detecting of the elements in tuple space S which match, is an associative look up which indicates that Linda should be ideal because of the associative matching properties it has.

The multiple `rd` problem is seen within the parallel composition of two binary relations because there are several processes that need to concurrently access the tuple space S non-destructively. The stream and semaphore methods of solving the multiple `rd` problem are now considered, using the parallel composition of two binary relations as an example.

5 Tuple semaphores

The first method considered for overcoming the multiple `rd` problem is using a tuple as a binary semaphore, or *lock tuple*. The lock tuple is a single and unique tuple that allows *user processes* to control access to a tuple space.

The general concept is that a process obtains the lock tuple, then *destructively* removes the matching tuples, using either `inp` or `collect`.[2] Once all the tuples have been removed they are replaced, and then the lock tuple reinserted. The removal of tuples is acceptable because only a single process can obtain the lock tuple and therefore access the tuple space at any one time[3] provided the tuple space is returned to the same state as it was when the lock tuple was removed no other process will be aware that the tuples have been removed and replaced.

In the case of the parallel composition of binary relations, the ISETL-Linda[8] code for a worker process is shown in Figure 2. Each worker process removes a tuple from tuple space R and then tries to remove the lock tuple in tuple space S. There is only one lock tuple in the tuple space S so all but one of the processes will block on the `in` primitive (line `A` in Figure 2). When a worker process retrieves the lock tuple, it has unrestricted access to the tuple space S.

The worker process creates a template using the second field of the tuple removed from tuple space R as the first element of the template. In this example, the template is then used by a `collect` to *move* all the tuples that match the

[2] If the Linda implementation supports neither of these or it supports them but does not provide `out` ordering (see Section 8) then the semaphore method can not be used, and streams *must* be used.
[3] Provided the all the processes accessing the tuple space adhere to the use of the lock tuple.

template in tuple space S to a local tuple space. The same operation can be performed using `inp`.

The worker process then removes each of the tuples from the local tuple space using the `in` primitive. The worker process then places the tuple back into tuple space S. Because of the fine grained nature of the worker processes used in the composition of binary relations, as the worker process returns the tuples to tuple space S, it also calculates any results and places them in the results tuple space C. If the computation "associated" with each tuple is more complex then either the worker process can place another copy of each tuple in a different tuple space for processing, once all the tuples from tuple space S have been replaced, or further processes, can be spawned to actually perform the calculations.

Once all the tuples in the local tuple space have been processed and replaced in S, the lock tuple is also replaced in S. This means that tuple space S contains all the tuples that were present when the worker process obtained the lock tuple. The tuple which acts as the semaphore can *only* be replaced when the tuple space is in its original state. If the tuple is returned prior to this then the other processes are not guaranteed to find all the tuples that they require.

```
comp_worker := func(R,S,C);
  local my_val, my_ts, my_comb, todo;

  my_ts := NewBag;
  my_val := lin(R,|[?int,?int]|);      -- Get the element from R
  dummy := lin(S,|["lock"]|);          -- Get the lock (A)
  todo := lcollect(S,my_ts,|[my_val(2),?int]|);
  while (todo > 0) do              -- Grab matching tuples in S
    todo := todo - 1;
    my_comb := lin(my_ts,|[my_val(2),?int]|); -- Process each
    lout(C,[my_val(1),my_comb(2)]);    -- Create result tuples
    lout(S,my_comb);                   -- Replace tuple in S
  end while;
  lout(S,["lock"]);                    -- Let the lock tuple go
  return ["TERMINATED"];
end func;
```

Fig. 2. A worker process using a tuple as a binary semaphore or lock tuple.

There are two reasons why tuple semaphores are *not* an acceptable solution to the multiple `rd` problem.

– Firstly, the solution requires the processes that use a tuple space to agree to use the lock tuple, and there is no guarantee that other processes will conform to this. Consider the example given in the introduction of this paper, involving a tuple space containing a set of names. There is no reason why several processes performing different and unrelated tasks may all require access to the name tuples within the tuple space concurrently. Suppose one process does not conform to the use of a lock tuple, either maliciously or by accident, then all the processes can no longer reliably have access to all the possible tuples.

– Secondly, such an approach creates a sequential bottleneck for the access of the tuple space as only one process can access the tuple space at any one time. Therefore, in the parallel composition of binary relations example, the only speed up achieved is the parallel reading of the tuples from tuple space R. The majority of the time that the program executes only a single worker is active, creating a sequential solution because only one process can access the tuple space S at any one time. Consequently, the use of this method is not normally acceptable because the sequential access produces bottlenecks.

6 Streams

The second approach is to use a *stream*. The basis of this approach is to remove the multiple `rd` problem of having many tuples which match the template to be used. This is achieved either by using information which is already in the tuple, or by adding a unique field to each tuple. This means that a unique template can be generated which will match a single tuple in the tuple space. Any processes which wants to use the tuples within the tuple space must be aware of the fields used within the tuple, and, if necessary, how the field is generated. Processes accessing the tuple space use the `rd` primitive to retrieve *every* tuple, and use a local check to see if the tuple is required.

Consider the example of the parallel composition of binary relations and assuming that the tuple space S contains the five tuples (as shown in Figure 1):

$$\langle 3_{int}, 7_{int}\rangle, \langle 6_{int}, 12_{int}\rangle, \langle 3_{int}, 9_{int}\rangle, \langle 5_{int}, 8_{int}\rangle, \langle 9_{int}, 10_{int}\rangle.$$

There is no unique field that allows each tuple to be independently chosen. Therefore a unique field is added to each of the tuples:

$$\langle 1_{int}, 3_{int}, 7_{int}\rangle, \langle 2_{int}, 6_{int}, 12_{int}\rangle,$$
$$\langle 3_{int}, 3_{int}, 9_{int}\rangle, \langle 4_{int}, 5_{int}, 8_{int}\rangle, \langle 5_{int}, 9_{int}, 10_{int}\rangle.$$

After adding the extra first field, each tuple contains a unique field, and the relationship across the tuples between the unique fields is known (an integer counter that is incremented by one for each tuple). This allows a process to access the tuple space using the template $\langle | index_{integer}, \Box_{integer}, \Box_{integer} | \rangle$

where *index* is a value between one and five in this example. Every worker process takes a tuple from tuple space R, and then reads *every* tuple from tuple space S, using the *index* field to match each tuple in turn. The worker process checks if the returned tuple is actually required and either discards it or uses it accordingly. If the implementation supports the `rdp` primitive then this removes the need to check the tuple *locally*, but all tuples still have to be checked. A template of the form $\langle| index_{integer}, R(2)_{integer}, \Box_{integer} |\rangle$ would be used, where *R(2)* is the second element from the tuple retrieved from the tuple space R. The `rdp` primitive would then be used returning either the matching tuple or a value to indicate it was not found. Every tuple still has to be checked. The ISETL-Linda code for a worker process using the stream method is shown in Figure 3.

```
comp_worker := func(R,S,C,NumTupS);
                    -- NumTupS - No. of tuples in S
  local my_val, my_comb;

  my_val := lin(R,|[?int,?int]|);  -- Get a tuple from R
  while (NumTupS > 0) do           -- Check all tuples in S
    my_comb := lrd(S,|[NumTupS,?int,?int]|);
    NumTupS := NumTupS - 1;
    if (my_comb(2) = my_val(2)) then -- Does the tuple match?
      lout(C,[my_val(1),my_comb(3)]);
    end if;
  end while;
  return ["TERMINATED"];
end func;
```

Fig. 3. A worker process using streams.

In this example it is necessary to add an extra field but sometimes a unique field is already present within the tuple. For example, when an image is stored in a tuple space, with each pixel being stored as a tuple of the form:

$$\langle \text{x-coordinate}_{integer}, \text{y-coordinate}_{integer}, \text{pixel value}_{integer} \rangle.$$

A process may want to access all pixels that are of a particular value. Here the obvious template would be $\langle| \Box_{integer}, \Box_{integer}, \text{pixel value}_{integer} |\rangle$. However, if many processes wish to perform the operation in parallel it will introduce the multiple `rd` problem. Assuming that the coordinate system used within the image will be known to the accessing processes, and there is only one pixel value for each coordinate, the coordinate fields within the tuple can be used as the unique fields. The processes can then use a stream approach, reading every coordinate to check if the pixel value is the one required, and discarding if it is not.

With the stream method all the worker processes can perform the accessing of a tuple space in parallel, however there are two problems that make this approach unacceptable:

- Firstly, it negates the advantages of the tuple matching abilities of Linda. *Every* tuple in the stream structure *must* be read. If there are many tuples in a tuple space and only a few are required, the time and communication costs of reading every tuple are considerable. This is compounded if the implementation does not support `rdp`, because additional processing within the user process of the returned tuple is needed to check if the tuple is one that is required.
- Secondly, every tuple in the tuple space requires a unique field to be added, and any process using the tuples must be aware of the unique field and how it is generated. This dilutes the natural use of a tuple space as the data structure by adding another structure (a stream) to the tuples within the tuple space. 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 it is minimal, or the tuples must be pre-processed to add the unique field before being used.

Even if the producer can add the extra field, and so no pre-processing of the tuples is required, the communication and time costs of checking every tuple explicitly using either `rd` or `rdp` is unacceptable unless the majority of tuples within a tuple space match the template.

7 Experimental results

In order to show the problems of both the binary semaphore and stream methods the execution times of parallel composition of binary relations using both these methods are considered. The experimental results presented in this section are obtained using ISETL-Linda executing on a Transputer based Meiko CS-1 parallel computer using the York Kernel I[9]. For the experiments the cardinality of the tuple space R is set to five; the cardinality of tuple space S is 50. For every pair (represented as a single tuple) in tuple space R there are four pairs (again, represented as single tuples) in tuple space S that match, therefore the cardinality of the composition tuple space C is 20. The worker processes are altered to enable them to be instructed on how many tuples are processed from tuple space R. Thus, a single worker computes the results for all five pairs in tuple space R, whereas five worker processes each compute the results for a single pair from tuple space R, as in the example code segments Figure 2 and 3. This is used to show that the semaphore method forces sequential access to tuple space S, whilst the stream approach allows parallel access to tuple space S.

The execution time *does not* include the time taken to spawn the worker processes, and does not include the time taken to create the tuple spaces S and R. In the stream approach it is assumed that the producer added the unique field to the tuples as they are created, thus avoiding the need to pre-process the tuples to add the unique field. All execution times are given in ticks, where 15625 ticks is the equivalent of one second.

The performance for both the methods of solving the multiple `rd` problem are compared against a sequential version. The code for the sequential version is shown in Figure 4. The sequential version takes each tuple from tuple space R, then uses the `collect` primitive to destructively move to another tuple space every tuple from tuple space S in which the first element of the tuple is the same as the second element of current tuple chosen from tuple space R. The moved tuples are then destructively read using the `in` from the other tuple space, processed and then placed back in to tuple space S. The result tuples are placed in tuple space C. The sequential version uses the tuple spaces to store data structures as do the parallel versions.

```
comp_worker := proc(R,S,C,NumTupR);
       - NumTupR is the number of tuples in R
  local my_val, my_ts, my_comb, todo, loop;

  my_ts := |{}|;
  for loop in [1 .. NumTupR] do      -- For all tuples in R
    my_val := lin(R,|[?int,?int]|); -- Get the a tuple
    todo := lcollect(S,my_ts,|[my_val(2),?int]|);
    while (todo > 0) do           -- Process the matched tuples
      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;
  end for;
end proc;
```

Fig. 4. The code for the sequential composition of binary relations.

7.1 The binary semaphore method

Figure 5 shows the execution times taken for the version using the lock tuple method when the number of worker processes are varied from between one and five. Also shown is the time taken for a sequential version of the program. The timings are given in ticks, which are arbitrary units of time.

The sequential version is slightly faster than the parallel version using the lock

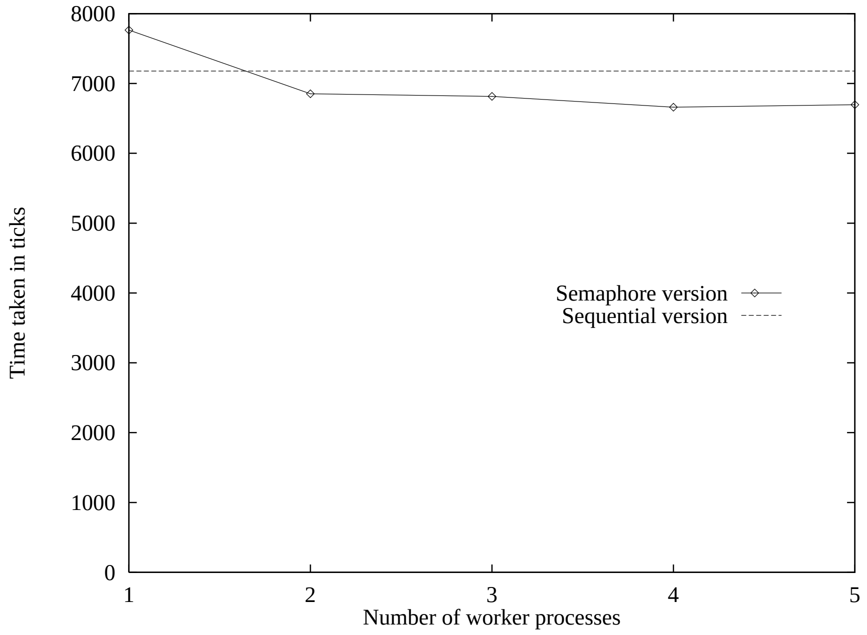


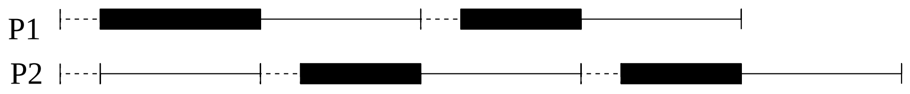
Fig. 5. Execution time for the parallel composition of binary relations when using the binary semaphore method.

method and one worker process. This is because the sequential implementation is similar to the lock method except that a lock tuple is not used, as only one process can ever access the tuple space. This means the the difference in the execution times represents the cost of fetching and replacing the lock tuple.

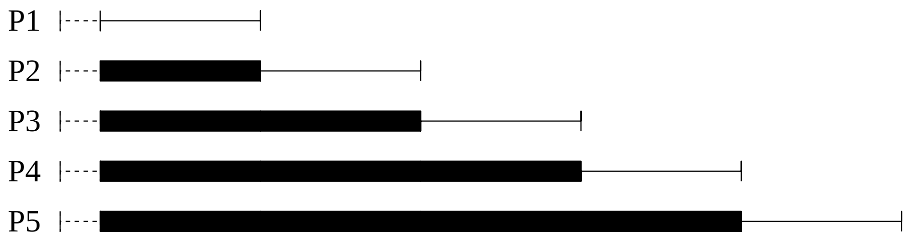
When two worker processes are used the execution time is slightly less than the execution time of the sequential version. This is achieved because of the parallel access to the tuple space R. The fetching of a tuple from tuple space R is the only work that can be parallelised; the access to tuple space S is forced to be sequential.

There is no performance gain by increasing the number of worker processes above two. For a parallel algorithm of this nature it would be expected that the addition of further worker processes would lead to a shorter execution time. In this case, when there are two worker processes one will consume three tuples from tuple space R and the other will consume two tuples from tuple space R. Five worker processes will each consume only one tuple from tuple space R, implying that as the number of worker processes are increased the work each worker has to do drops. This is shown in Figure 6 where the solid line represents the time a worker process is accessing tuple space S, and the dotted line represents the time when the worker is accessing tuple space R. The solid thick black lines represent the time when a worker process is blocked awaiting the lock tuple. The length of the time taken by the longest worker is the execution time of the program. Figures 6(a) and 6(b) show that the time taken by the longest worker process when both two and five worker processes are used is the same. If three or four worker processes are used then the longest worker process will again take the same time.

As the number of worker processes increase there is no performance increase because there is nothing more that can be achieved in parallel. When there are five worker processes, each will consume only one tuple from tuple space R. However, the program takes the same length of time as that with two worker processes because there is no gain in having more than two worker processes access tuple space R in parallel. This is shown in Figure 6, where the solid lines represent the time when a worker process is accessing tuple space S, the dotted lines represent the time that the worker process is accessing tuple space R, and the solid black bars represent the time when the worker process is blocked awaiting the lock tuple. The length of time taken by the longest worker represents the time the program takes. Hence, by looking at Figures 6(a) and 6(b) it can be seen that the time taken by the longest worker process in each case is the same.



(a) Execution pattern using two worker processes.



(b) Execution pattern using five worker processes.

Fig. 6. Execution patterns for two and five worker processes using the semaphore method.

7.2 The stream method

Figure 7 shows the execution times taken for the version using the stream method when the number of worker processes are varied from between one and five. Again the time taken for the sequential version is also shown and the predicted execution times are also shown. The predicted execution time is calculated on the basis of the time taken for one worker process.

The results show that the execution time is dependent upon the number of

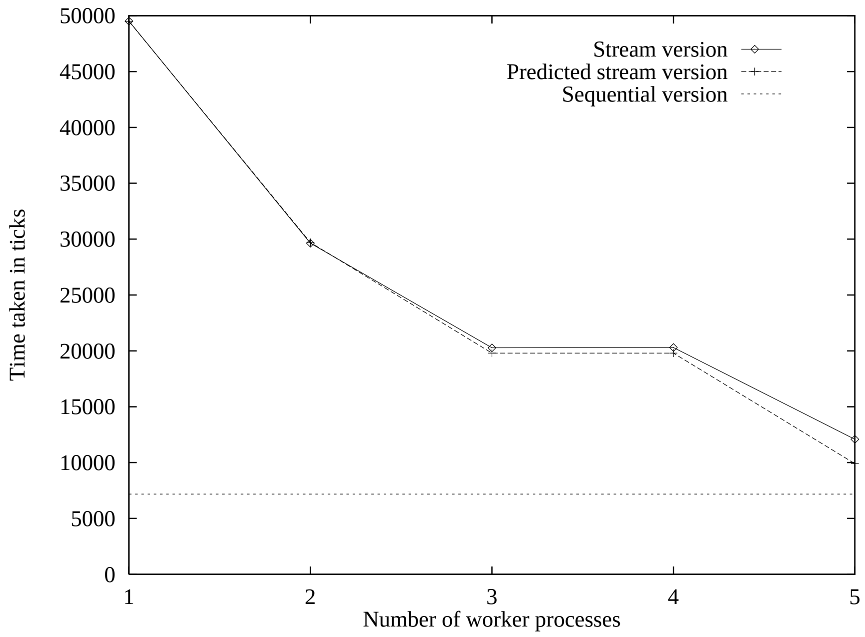


Fig. 7. Execution time for the parallel composition of binary relations when using the stream approach.

worker processes used. The solution is parallel so the time taken depends on the worker process or processes which consume the most tuples from tuple space R. When either three or four worker processes are used then in both cases the most tuples from tuple space R a single worker process consumes is two. Subsequently the execution times when using three or four worker processes are same.

Although there is a speedup as the number of worker process is increased, the execution times are always greater than the sequential version. This is because *all* the tuples have to be read in regardless of whether they are actually required. The cost of reading all the tuples, regardless of whether they are to be used is an expensive operation.

7.3 Experimental conclusions

The experimental results show the dilemma that a programmer faces with the multiple `rd` problem within the Linda model. The binary semaphore method provides no speed increase as the number of worker processes used increases but is slightly faster than the sequential version when two or more worker processes are used. The stream method shows a speed up as the number of worker processes are increased but it takes a longer time to execute than the sequential version, even when five worker processes are being used.

What makes this multiple `rd` problem more frustrating is that it is intuitively the case 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 kernels[9,22] support concurrent primitive operations.

8 The `copy-collect` primitive

In order to overcome the multiple `rd` problem the addition of a new primitive to the Linda model has been proposed, a relative of the `collect` primitive[3]. The informal semantics of the `copy-collect` primitive are:

n = copy-collect (ts1, ts2, template) This primitive *copies* tuples that match `template` from one specified tuple space (`ts1`) to another specified tuple space (`ts2`). A count of the number of tuples copied (`n`) is returned. Tuple space `ts1` is known as the source tuple space and tuple space `ts2` is known as the destination tuple space.

To determine how many tuples are copied a series of rules are used. These rules are:

(i) If a `copy-collect` primitive and no other Linda primitives are performed using the source tuple space concurrently, then *all* the tuples that match the template will be copied to the destination tuple space.

(ii) If a `copy-collect` primitive and a `rd` primitive are performed using the same source tuple space concurrently, and one or more tuples exist that can satisfy both templates, then *all* the matching tuples will be copied to the destination tuple space and the `rd` primitive will return some matching tuple.

(iii) If two `copy-collect` primitives are performed using the same source tuple space concurrently, and one or more tuples exist that can satisfy both templates, then *all* the matching tuples will be copied to the destination tuple space for each of the `copy-collect` primitives.

(iv) If a `copy-collect` primitive and an `out` primitive are performed concurrently, and the `out` primitive is placing a tuple into the source tuple space that matches the template used in the `copy-collect` primitive, then the result is a non-deterministic choice between copying the inserted tuple or not. All other matching tuples will be copied.

(v) If a `copy-collect` primitive and an `in` primitive are performed using the source tuple space concurrently, and one or more tuples exist that can satisfy both templates, then the `copy-collect` primitive either copies all the tuples, or all the tuples minus the *matched* tuple that the `in` primitive returns (the choice is non-deterministic).

(vi) If a `copy-collect` primitive and a `collect` primitive are performed concurrently using the same source tuple space, then the number of tuples copied is non-deterministic within the bounds of zero to the maximum number of tuples present that match the template. The number of tuples that the `collect` primitive will move will be the number of tuples present that match the template.

If any primitive occurs concurrently with a `copy-collect` primitive that does not use either a template which matches one or more tuples that the `copy-collect` primitive template matches, or the source tuple space, then there is no interference between them. The exception is when the primitive is either a `collect` primitive or a `copy-collect` primitive performed on the destination tuple space with a template that matches one or more of the tuples being copied. Then each tuple placed into the destination tuple space is non-deterministically copied or moved by the `collect` primitive or `copy-collect` primitive being performed on the destination tuple space. When a value is returned by a `copy-collect` primitive the copied tuples *are* present within the destination tuple space.

The `copy-collect` primitive will never live-lock — it will always complete and return a value. Rule 4 states that if an `out` primitive occurs concurrently with a `copy-collect` primitive then the inserted tuple may or may not be included in the copied tuples. Is it possible for one process to perform many primitives concurrently with another processes performing a `copy-collect` primitive? Within Linda there is no notion of time associated with a primitive. Therefore, it can be assumed at the model level that all primitives take the same time. Hence, the maximum number of `out` primitives that can occur concurrently with a `copy-collect` primitive is the number of user processes minus one. This means that the `copy-collect` primitive will always complete provided there are a finite number of processes. Pragmatically a `copy-collect` primitive may take longer than a single `out` primitive and therefore, several `out` primitives may occur concurrently with the `copy-collect` primitive. Due to the semantics given above for the `copy-collect` primitive it is up to the implementor to ensure that the `copy-collect` primitive completes and does not live lock.

The `copy-collect` primitive assumes that there is `out` ordering[9], which means that if a *single* process performs two sequential `outs` to the same tuple space, the second tuple can not appear in the tuple space *before* the first tuple. The example processes shown in Figure 8 demonstrate why `out` ordering is required. Let us assume that the tuple space `ts1` is only accessible from the two processes shown. Process One places two tuples into the tuple space `ts1`. Process Two performs an `in` on one of the tuples and then performs an `inp` on the other. When using *out ordering* then there is only one outcome of these two processes, Process Two removes the tuple $\langle \text{``}DONE\text{''}_{string} \rangle$ and then the variable `x` is assigned the value one, and the tuple $\langle 10_{integer} \rangle$ is copied to `ts2`. However, if *out ordering* is not used then the `copy-collect` might copy no tuples.

Process One	Process Two
`out(ts1, [10]);`	`in(ts1, |["DONE"]|);`
`out(ts1, ["DONE"]));`	`x := copy-collect(ts1, ts2, |[?int]|);`

Fig. 8. `out` ordering example.

If `out` ordering is not used, then there is no way for a programmer to ensure that all the tuples are present within a tuple space before the `copy-collect` is performed. Enforcing `out` ordering is quite achievable, even in distributed implementations. For more information see Rowstron et al.[21].

9 Using `copy-collect` to solve the multiple `rd` problem

The `copy-collect` primitive has the functionality to overcome the multiple `rd` problem. The primitive allows several processes to concurrently *copy* the tuples they require. Consider again the example where a tuple space is used to store a number of tuples containing peoples names. All the people with the same surname are required. Using the `copy-collect` primitive it is possible to extract into a separate tuple space all the tuples with the same surname. Hence, to extract all people with the surname *Rowstron*, a `copy-collect` primitive is performed using the template $\langle| \square_{string}, \text{“Rowstron”}_{string} |\rangle$.

Figure 9 shows the use of the `copy-collect` primitive to overcome the multiple `rd` problem. The shared tuple space is called *image_ts* and the processes are called P_A, P_B and P_C. Each process creates a tuple space (*image_ts_*P_x) to which only they have access. They then perform a `copy-collect` primitive using the source tuple space as *image_ts* and the destination tuple space as *image_ts_*P_x. Once the tuples have been copied to the destination tuple space each process can retrieve each tuple in turn using the `in` primitive. Each process knows the number of tuples in the tuple space because the `copy-collect` primitive returns a counter of the number moved, and by destructively removing them ensures that the same tuple is never read twice. Because the tuple space cannot be accessed by any other process the destructive removal of tuples does not affect any other process. The example in Figure 9 has two of the processes requiring all the pixels which are set (the third field set to one) and the other process requiring all the pixels which are not set (the third field set to zero).

The parallel composition of two binary relations used as an example in the previous sections, is now used to show in more detail how `copy-collect` primitive overcomes the multiple `rd` problem in more detail. The worker process using the `copy-collect` method is shown in Figure 10. The general structure

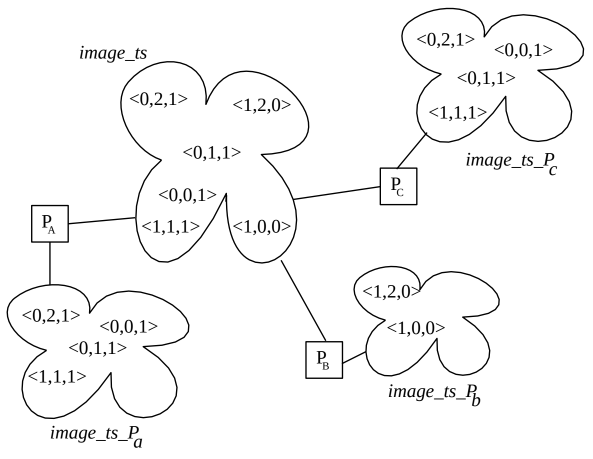


Fig. 9. Using the `copy-collect` primitive to solve the multiple `rd` problem.

of the approach is the same as in the solutions given in the previous sections, with each worker process removing a tuple from tuple space R. The worker process then creates a template using the retrieved tuple for use with the `copy-collect` primitive. The second field of the retrieved tuple from tuple space R is used as the first field of the template. The second field of the template is left as a formal of type integer. A `copy-collect` primitive is then performed which copies the tuples from tuple space S to a tuple space which the worker process creates. The counter returned by the `copy-collect` primitive is then used to control an iterative loop which destructively reads the tuples from the tuple space and creates the result tuples in tuple space C.

```
comp_worker := func(R,S,C);
  local my_val, my_ts, todo, my_comb;

  my_ts := NewBag;
  my_val := lin(R,|[?int,?int]|);  -- Get the tuple from R
  todo := lcopycollect(S,my_ts,|[my_val(2),?int]|);
  while (todo > 0) do              -- Process all matched tuples
    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. 10. The worker process using the `copy_collect` primitive.

10 Experimental results

The experimental results presented in this section as in the previous sections are obtained using ISETL-Linda running on a Transputer based Meiko CS-1 parallel computer using the York Kernel I. The `copy-collect` primitive was added to the run-time system by Douglas[9], and the implementation is naive. As in the previous section the worker process was altered to enable the number of tuples from tuple space R to be consumed to be specified.

For the experimental results the same configuration for R and S is used as used in the experimental results presented in the previous sections. Figure 11 shows the execution times for the worker processes for computing the composition of two binary relations using `copy-collect`, the best execution time of the other two methods (using a lock tuple with four worker processes), and the expected execution time for the `copy-collect` method is shown.

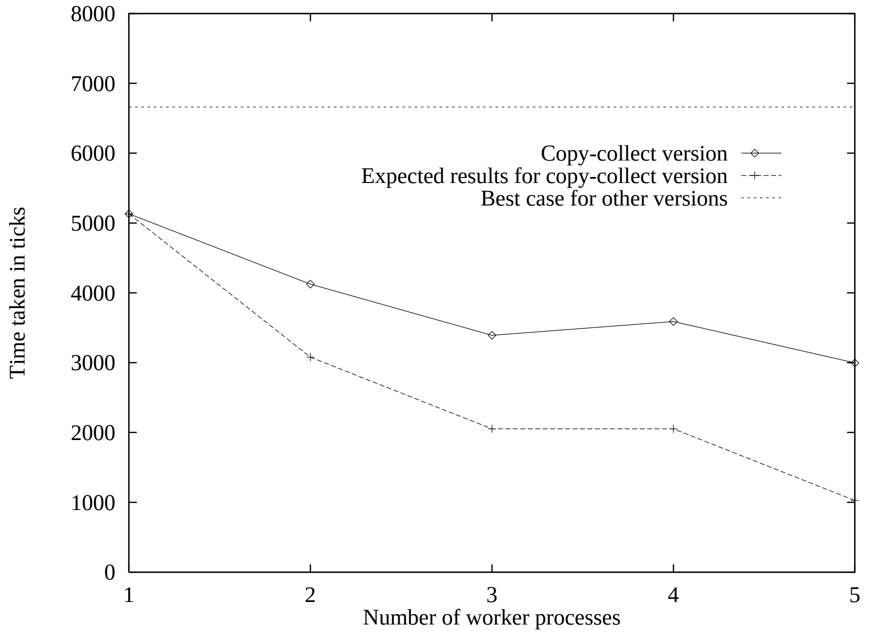


Fig. 11. Execution time for the parallel composition of binary relations when using the new `copy-collect` primitive.

Figure 11 shows some interesting results. Firstly, the execution time using a single process is less than the best time achievable using any number of worker processes for any of the other methods. This is because the *number* of tuple space operations that have to be performed is significantly less. The time taken when three and four worker processes are used is similar, for the same reason that in the stream method the time taken for three and four worker processes is similar. The expected results are calculated using the time taken for the single worker process and dividing it by the number of tuples in tuple space R, which is five. As with the stream method, because there is parallel access it is expected that with five worker processes, each processes a single tuple from tuple space R, and therefore, the time each worker takes should be one fifth of the time the single worker process takes. The actual execution times are greater than predicted because the underlying run-time system is "saturating"; in other words the run-time system is receiving more requests than it can process, so becomes a bottleneck. However the performance even with the run-time system saturating is twice as fast as the best time produced by the semaphore or the stream methods.

Sections 8 and this section demonstrate how the `copy-collect` primitive solves the *multiple* `rd` *problem.* A worker creates a "local" copy of the tuples that it requires using a `copy_collect` primitive 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]|)` copies all the tuples with a pixel value of one into the local tuple space. Given the tuple space containing tuples representing first names and surnames. Each process would then use the `copy-collect` primitive with the template $\langle|\square_{string}$, "Rowstron" $_{string}|\rangle$, which will copy all the tuples with a surname of "Rowstron" to a separate tuple space, where they can be destructively processed.

So far with all the experimental results the execution time for a specific cardinality of binary relations S and R have been considered. Now the effect of making more tuples in tuple space S match each tuple in tuple space R is considered. For this the cardinality of tuple space R is again fixed at five. Figure 12 shows the execution times for five worker processes when the number of tuples in the tuple space S that match *each* tuple in tuple space R is increased from one to 50 (the cardinality of tuple space S is 50).

As the number of tuples in tuple space S that each tuple in tuple space R matches increases there will be an increase in the computation time within each worker process associated with the calculation and placement of the result tuples in tuple space C. Hence, although it might be expected that the stream method should take a constant time because all tuples in tuple space S are always read by every worker process, the actual time increases slightly. This increase is attributable to the extra computation that the worker processes perform. The time taken for the semaphore method increases uniformly with the addition of the extra tuples in tuple space S that match each tuple in tuple space R. The reason why the execution time increases at a greater rate than the other methods, is that the other methods are parallel. So when the number of tuples that match each element in tuple space R increases by one, each of the five worker processes process one more tuple. If the method is parallel then this is performed concurrently. Because the semaphore method is sequential the five tuples are processed sequentially. This leads to an increase in the

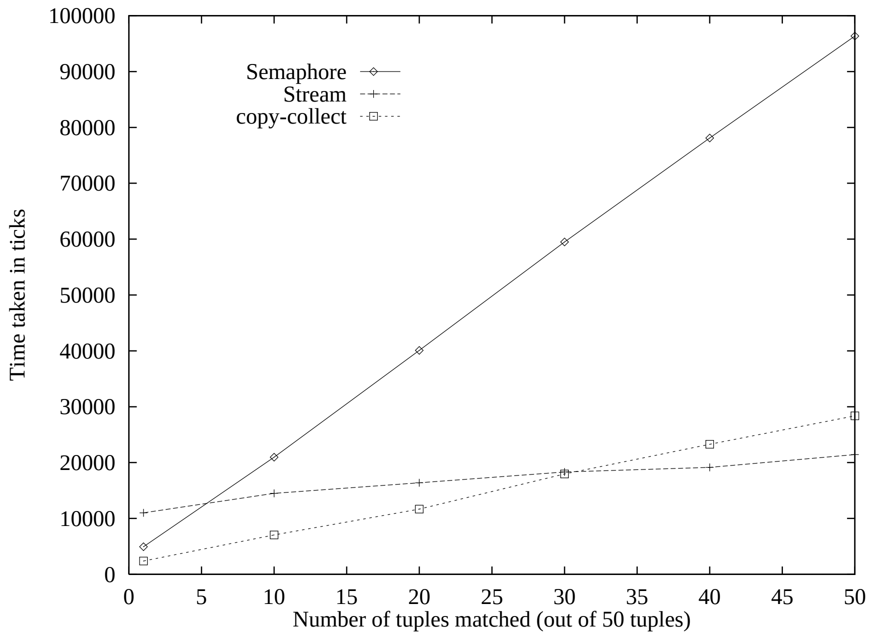


Fig. 12. Execution time for the parallel composition versus the number of pairs in tuple space S which each pair in tuple space R matches.

execution time which is five times larger than the increase for the parallel methods for every extra tuple from tuple space S which matches each element in tuple space R. The execution time for the `copy-collect` method increases as the number of matching tuples in tuple space S increases. As more tuples match, not only is there the extra computation costs associated with each extra tuple processed by the worker processes but there is extra communication as the number of tuple space operations is proportional to the number of tuples in tuple space S that match each tuple in tuple space R.

However, the performance of the stream method seems to perform better when approximately 70% of the pairs within the tuple space S match each pair in tuple space R. The exact performance of the different methods depends on the underlying Linda run-time system implementation. In this case the kernel is saturating at this point, because all the worker processes wish to perform the `copy-collect` primitive simultaneously. In the York Kernel II[22,21] the performance of the `copy-collect` method is always better than a stream approach because of a more efficient implementation approach for the `copy-collect` primitive (see Section 12).

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.

11 Alternative proposals

As with any abstract model there have been many proposals to alter the model. Two of particular interest are; One the proposal for another primitive; `rd()all`[1] [4] . 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.*

This primitive can potentially be used to solve the multiple `rd` problem because it applies the function to *all* tuples that match the template within the tuple space. Therefore, in the parallel composition of binary relations one could consider creating a worker which removes a tuple from tuple space R and then performs a `rd()all`. A worker process using this primitive is shown in Figure 13.

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

  my_val := lin(R,|[?int,?int]|);  -- Get the tuple from R
  rd(|[my_val(2),?x]|)all(lout(C,[my_val(1),x]);
  return ["TERMINATED"];
end func;
```

Fig. 13. The worker process using the `copy_collect` primitive.

Anderson[1] 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 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 *live-lock*, 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.

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

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 valuable to the programmer. The information allows the number of worker processes `eval`ed to be controlled for example. If there are many tuples that match, more worker processes will be required than if fewer tuples match.

In Objective Linda[15] the primitives are bounded. This is achieved by adding a maximum and minimum number of tuples that the primitive can return. By specifying a maximum number of *infinite-matches* all matching tuples are returned. This therefore can be used to overcome the multiple `rd` problem. However, the primary difference is that the `copy-collect` primitive (and `collect` primitive) *do not* return tuples. The tuples are copied or moved from one tuple space to another and therefore, their storage is still controlled by the system that manages the tuples. The bounded primitives in Objective Linda return bags of tuples, which have to be stored locally within the user process. In Objective Linda this leads to special local data structures with special operations, including a primitive to count the number of items in the local data structure.

12 Implementation of `copy-collect`

Having demonstrated the *need* for the `copy-collect` primitive, we now briefly consider how the primitive can be implemented in an efficient manner. The primitive has been implemented in all our distributed kernels[22,9].

There are two concerns about implementing the `copy-collect` primitive. One is that the primitive apparently implies duplication of the matched tuples. Because of the nature of the primitive there is the chance that large numbers of tuples will be physically duplicated. Pragmatically this depends on the implementation of the run-time system, and how tuples are stored within it. It is quite possible that all tuples which are identical are stored in the same place within the run-time system which enables the tuple to be tagged as belonging to several tuple spaces, without being physically duplicated. This approach is to be adopted in York Kernel III, a distributed run-time system.

The second concern is that the `copy-collect` primitive may lead to the unnecessary movement of tuples around a run-time system. York Kernel I[9] implements a `copy-collect` which never moves tuples internally during the execution of a `copy-collect` primitive. This is a distributed run-time system which places tuples within the distributed system based on their content rather than the tuple space to which they belong. Hence regardless of which tuple space a tuple belongs, it will always appear in the same place within the distributed kernel. Therefore, when a `copy-collect` primitive is performed tuples never move from one place within the distributed run-time system to another place.

However, the bulk movement of tuples has been shown in some circumstances to be desirable. The York Kernel II[22,21] is a two layer hierarchical kernel, which does move tuples *under certain circumstances*. The intelligent movement of tuples around a distributed kernel can lead to large speed increases over "traditional" implementations. The York Kernel II uses *implicit* information to enable it to classify every tuple space as either a *local tuple space* or a *remote tuple space* on the fly and without extra communication or programmer guidance. A local tuple space is one which can only be accessed by one process and a remote tuple space is one that many processes can access. By making this distinction it enables the York Kernel II to transparently move packets of tuples around the system to gain optimum performance by placing the tuples as close to the processes which can consume them as possible. It should be noted that making the distinction between local tuple spaces and remote tuple spaces does not alter the semantics of the Linda model or of the `copy-collect` primitive, a user sees no distinction between a local and a remote tuple space. When a `copy-collect` primitive is performed, under certain circumstances, it leads to the bulk movement of tuples between local and remote tuple spaces and vice versa.

13 Conclusion

In this paper the multiple `rd` problem has been characterised, and two possible solutions using the facilities of the standard Linda model — the stream method and the binary semaphore method — have been shown through experimentation to be inefficient. The stream method requires every tuple in a tuple space to be read whilst the binary semaphore method yields a sequential solution. It is concluded that the standard Linda model is unable to provide a satisfactory concurrent solution to performing the multiple `rd` operation.

Consequently, a new primitive has been proposed, called `copy-collect`, which has been shown to overcome the deficiencies in the standard model solutions. Experimental results have been presented which demonstrate the improved performance of the `copy-collect` based method compared to streams and binary semaphores.

The results suggest that when the number of matching tuples is high, the stream method may be faster than using the `copy-collect` method. This is because the kernel being used to gain the experimental results is saturating. However, this is an implementation matter, and given an improved implementation, such as York Kernel II, the `copy-collect` method will always perform better than the either the stream or semaphore approach[18].

Finally, although a deliberately simple example program is used in this paper in order to clarify the presentation, the multiple `rd` problem occurs in a wide range of more complex algorithms and situations — it is possible for the problem to occur in any algorithm where information is stored that is required by several processes. A more substantial example can be found in [18] which describes the parallel implementation of image processing algorithms in which the processing of an image is to be shared by multiple processes.

Acknowledgement

Whilst performing this research Antony Rowstron was supported by an EPSRC CASE grant in conjunction with British Aerospace Military Aircraft Division. He would also like to thank The Olivetti & Oracle Research Laboratory, Cambridge, for current funding. The authors would also like to thank the reviewers for there helpful comments and Andrew Douglas for his comments.

References

[1] B. Anderson and D. Shasha. Persistent Linda: Linda + Transactions + Query Processing. In J. 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.

[2] P. Butcher. A behavioural semantics for linda-2. Technical Report YCS-137, Department of Computer Science, University of York, 1990.

[3] P. Butcher, A. Wood, and M. Atkins. Global synchronisation in Linda. *Concurrency: Practice and Experience*, 6(6):505–516, 1994.

[4] D. Campbell, H. Osborne, and A. Wood. Characterising the design space for LINDA semantics. Technical Report YCS-277, University of York, 1997.

[5] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[6] P. Ciancarini, R. Gorrieri, and G. Zavattaro. Towards a calculus for generative communication. In E. Najm and J. Stefani, editors, *Proceedings of the IFIP Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 289–306, 1996.

[7] P. Ciancarini, K. K. Jensen, and D. Yankelevich. On the operational semantics of a coordination language. In P. Ciancarini, O. M. Nierstrasz, and A. Yonezama, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, pages 77–106. Springer-Verlag, 1995.

[8] A. Douglas, A. Rowstron, and A. Wood. ISETL-LINDA: Parallel programming with bags. Technical Report YCS 257, University of York, 1995.

[9] 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.

[10] 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*, volume 366 of *Lecture Notes in Computer Science*, pages 20–27. Springer-Verlag, 1989.

[11] W. Hasselbring. *Prototyping Parallel Algorithms in a set-orientated language.* PhD thesis, University of Essen, 1994.

[12] S. Hupfer. Melinda: Linda with multiple tuple spaces. Technical Report YALEU /DCS/RR-766, Yale University, 1990.

[13] K. Jensen. *Towards a Multiple Tuple Space Model.* PhD thesis, Aalborg University, Department of Mathematics and Computer Scien ce, 1993.

[14] K. Jeong. *Fault-tolerant parallel processing combining Linda, checkpoint ing and transactions.* PhD thesis, New York University, 1996.

[15] T. Kielmann. Designing a coordination model for open systems. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models, Proceedings of Coordination ' 96*, volume 1061 of *Lecture Notes in Computer Science*, pages 267–284. Springer-Verlag, 1996.

[16] R. D. Nicola and R. Pugliese. A process algebra based on LINDA. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 160–178. Springer-Verlag, 1996.

[17] B. Nielsen and T. Sorensen. Implementing Linda with multiple tuple spaces. Technical report, Aalborg University, Department of Mathematics and Computer Sci ence, 1993.

[18] A. Rowstron. *Bulk primitives in Linda run-time systems.* PhD thesis, Department of Computer Science, University of York, 1997.

[19] 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.

[20] 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.

[21] A. Rowstron and A. Wood. An efficient distributed tuple space implementation for networks of heterogenous workstations. Technical Report YCS 270, University of York, 1996.

[22] A. Rowstron and A. Wood. An efficient distributed tuple space implementation for networks of workstations. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96*, volume 1123 of *Lecture Notes in Computer Science*, pages 510–513. Springer-Verlag, 1996.