`COPY-COLLECT`: A new primitive for the Linda model

Antony Rowstron, Andrew Douglas and Alan Wood
Department of Computer Science, University of York

Abstract

Linda is a model of communication for concurrent processes. This paper proposes the addition of a new primitive to the Linda model, called `copy-collect`. An informal semantics of the new primitive is presented and an outline of the *multiple* `rd` *problem* which provides the justification for adding a new primitive. A description of how the new primitive can be implemented on several different implementations is also provided.

1 Introduction

The originators of Linda[1] describe it as a *co-ordination language*; its principal purpose is to orchestrate interprocess communication. Communication between different processes involved in a (concurrent) computation can only occur through a *tuple space*, which can be thought of as a bag into which tuples are inserted and withdrawn by the processes. Linda is only concerned with process co-ordination, and thus is always embedded into some other *host* language e.g. `C`, Prolog, Lisp, etc. which provides the conventional computational mechanisms.

Recent work has shown a particular operation (which we call the *multiple* `rd` *problem*) to be difficult to perform using the current Linda model. The addition of a new primitive to the model, `copy-collect` appears to rectify the problem. We discuss the new primitive, the multiple `rd` problem and the effects on implementations of the new primitive. However, first an overview of the Linda model is presented.

2 The Linda Model

The Linda model is now well known and a detailed description can be found in [2]. The main 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.

Some Linda systems support two other primitives, `inp` and `rdp`. These are non-blocking versions of `in` and `rd`. Instead of blocking they return a value to represent that no tuple was found.

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. Two extensions that are fundamental to this work 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[3, 4] as well as one with a mixture of flat and hierarchical tuple spaces[5]. The new primitive proposed in this paper, `copy-collect`, requires multiple tuples spaces. However, it makes no assumptions about the relationship between tuple spaces. All that is required is that a unique identifier (or handle) exists for each tuple space.

`collect` **primitive** Syntactically[2] the `collect`[6] primitive can be represented as: `collect(ts1, ts2, template)` where `ts1` and `ts2` are two tuple spaces handles and `template` is a template. The `collect` primitive *moves* tuples in `ts1` that match `template` into `ts2`, returning a count of the number of tuples moved.

3 The `copy-collect` primitive

The `copy-collect` primitive represents a natural extension of the `collect`[6] primitive. Where as the `collect` primitive moves tuples, the `copy-collect` primitive *copies* tuples. The informal semantics of `copy-collect` are as follows:

`copy-collect(ts1,ts2,template)` This primitive *copies* all available tuples that match the given `template` from one specified tuple space (`ts1`) to another specified tuple space (`ts2`). It returns a *count* of the number of tuples copied.

[1] Sometimes referred to as an *anti-tuple*.
[2] Using a C-Style syntax.

One of the advantages of Linda over other communication models is its simplicity. However, the addition of new primitives is, we believe, justified it *extends* the expressibility of the model. The problem which `copy-collect` solves is the *multiple* `rd` *problem*.

4 The multiple `rd` problem

The multiple `rd` problem occurs when many processes wish to concurrently read a subset of the tuples in a tuple space. In order to demonstrate the problem a simple example is used.

Figure 1 shows a tuple space and two processes. The tuples in the tuple space represent a binary image, with the tuples taking the form of $[x\_coord, y\_coord, value]$. The two processes perform some simple image processing operation. They both require to use only the pixels which are set (in other words the tuples where the third field is one).

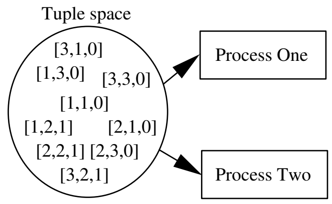

Figure 1: Example of where the multiple `rd` problem can exist.

The problem is how to allow the two processes to read the necessary tuples in an efficient manner. First thoughts would suggest the use of a `rd`, with a template of the form $[?integer, ?integer, 1]$. However, Linda does not guarantee that a `rd` will always return a different tuple, and so this solution must be rejected. Using the current Linda model there are two ways to solve this problem – which we call the *multiple* `rd` *problem*.

Streams In this approach each tuple has to have a unique field (or a set of fields). In this example, the integer pair (x_coord, y_coord) could be taken to be the unique field. If no unique field is present then one has to be added, for example numbering the tuples from one upwards, and then storing the value of the last value used in a tuple so the consumer can work out when all the tagged tuples in a tuple space have been checked.

All the tuples are then read using a `rd` using the unique field to generate a template for each tuple as required. Because each tuple is unique only one tuple will match with the template. The skeleton of the procedure needed to do this is shown in Code 4.1, where `image_ts` is the tuple space handle for the tuple space depicted in Figure 1.

The example code demonstrates how the unique fields are used. *Every* tuple is read and if the third field contains a one (representing a set pixel)

Code 4.1 Example code using streams.

```
for (x = 1; x < 4; x++)
  for (y = 1; y < 4; y++)
    image_ts.rd(x,y,?value);
    if (value == 1)
      do_operation(x,y);
```

the operation is performed. In this case the size of the image has to be known in order to allow a consumer to generate all the possible pairs for (x_coord, y_coord) and either all pixels need to be present or the system supports predicated version of `rd`.

Semaphores The semaphore approach is reliant on the implementation supporting either `inp` or `collect`. A unique tuple is placed in the tuple space, which acts as a lock tuple or semaphore. Before a process can use the tuples in the tuple space it must first grab the lock tuple (using an `in`). Once it has the lock tuple, it can destructively remove the tuples from the tuple space either by repeated use of `inp` or by using `collect`. The removed tuples are stored in another, temporary, tuple space. Once all the matched tuples have been destructively removed from the original tuple space, they are replaced by removing the ones that are in the temporary tuple space. Then finally the lock tuple is returned to the tuple space. The skeleton of the procedure needed to do this is shown in Code 4.2. This uses `inp` but could also use `collect`.

Code 4.2 Example code using streams.

```
count = 0;
image_ts.in("lock");
while (image_ts.inp(?x,?x, 1)) do {
  local_ts.out(x,y,1);
  count++;
}
for (i = 1; i < c; i++) {
  local_ts.in(?x,?y,1);
  image_ts.out(x,y,1);
  do_operation(x,y);
}
out("lock");
```

The code given here is one of several ways of performing the operation and is suited to fine grained use (that is the time cost of performing do_operation(x,y) is about equivalent to a Linda operation).

The stream approach is unacceptable for two reasons. The addition of a unique field may not be easy. If the producer of the tuples is aware of the consumers problem then they can be added easily, but the consumer may then have to be told how the unique field was generated. If the producer was not aware of the consumers problem (because it is part of a library or it was written by someone who was not aware that the resulting tuples would be used in such away etc) then this involves the preprocessing of all the tuples, which can be expensive and unnecessary. The second reason why the stream is unacceptable is that *every* tuple must be `rd` regardless of whether it is to be used. If there are millions of tuples and only a small percentage are required then it is *very* expensive to fetch all the tuples and then use just a small number of them.

The semaphore approach is unacceptable because if there are many processes wishing to access the tuple space, they have to do so one at a time. They will all compete for the lock tuple. One process will get it and the others will block until it is replaced, and then one of the other processes will get it and so on. This clearly makes the accessing of the tuple space sequential and subsequently completely unacceptable in a parallel system.

How does the `copy-collect` primitive allow the multiple `rd` problem to be solved? The tuples in the main tuple space that are required are *copied* using `copy-collect` to a local tuple space and then are destructively read from the local tuple space. The outline of procedure needed to do this is given in Code 4.3.

Code 4.3 Example code using `copy-collect`.

```
count = copy-collect(image_ts,local_ts, (?int,?int,1));
for (lp = 0; lp < count; lp++)
  local_ts.in(?x,?y,1);
    do_operation(x,y);
```

The template used for the `copy-collect` matches only the tuples in the tuple space which are required. These tuples are copied to the local tuple space. The count of the number of tuples copied is used to control the number of tuples destructively read from the local tuple space. The `copy-collect` operation can be performed by many processes concurrently.

We have used an image processing problem as the example. However the multiple `rd` problem occurs whenever a tuple space is used to store information that is required to be read by many processes concurrently. The tuple space may contain a list of names (such as employee names), an image or even collections of tuples containing HTML documents!

5 Performance

Having shown how the new primitive solves the multiple `rd` problem in a practical way, we will now consider how the three approaches (semaphore, stream and `copy-collect`) perform in a more general way.

In order to achieve this let us consider the general case, where there is a tuple space, $\mathbb{T}$, which contains N tuples. Given a template, t, there are n tuples which match the template. There are $\mathbb{P}$ processes wishing to perform the multiple `rd` concurrently. Initially, let us consider the case where $\mathbb{P} = 1$. How many Linda primitives are required in order to the process to `rd` all the elements n and at the end leave tuple space $\mathbb{T}$ in its original state.

Stream method If there are N tuples then each tuple will be read once. Therefore, the number of Linda primitives required is:

$$\text{No. of Linda primitives} = N \tag{1}$$

Semaphore/lock method As already mentioned there are many ways of implementing this. In the best case for a `collect` based implementation, the number of primitives required is:

$$\text{No. of Linda primitives} = 3 + 2n \tag{2}$$

and the best case for an `inp` based implementation, the number of primitives required is:

$$\text{No. of Linda primitives} = 3 + 4n \tag{3}$$

copy-collect method The number of Linda primitives required will be the primitive to copy the tuples to a local tuple space (1) and the primitives to remove the tuples from the local tuple space (n). Therefore, the number of Linda primitives required is:

$$\text{No. of Linda primitives} = 1 + n \tag{4}$$

Let us now consider the what happens as $\mathbb{P}$ becomes greater than one. Obviously the number of primitives required in total will all rise proportionally with the number of processes. However, let us consider the number of primitives that can *not* be done concurrently, or in other words let us consider the number of primitives that are parallelizable. We will assume the we have an ideal Linda implementation.

The stream and `copy-collect` versions will not alter. All the processes can *concurrently* perform the operation. There is no reason why an ideal kernel could not process many non-destructive tuple space operations concurrently. However, the semaphore/lock version will rise proportionally with the number of processes, as each will perform an `in` on the lock tuple and then $\mathbb{P}-1$ will block. Therefore, in the case of the semaphore version the number of unparallelisable primitives is:

$$\text{No. of Linda primitives} = (3 + 2n)\mathbb{P} \tag{5}$$

Figures 2 and 3 show how the non-parallelisable primitive counts vary. In both cases the size of $\mathbb{T}$ was 100, the size of $\mathbb{P}$ and n was varied. Figure 2 shows how the stream and semaphore solutions perform, and Figure 3 shows how the stream and copy-collect versions perform.

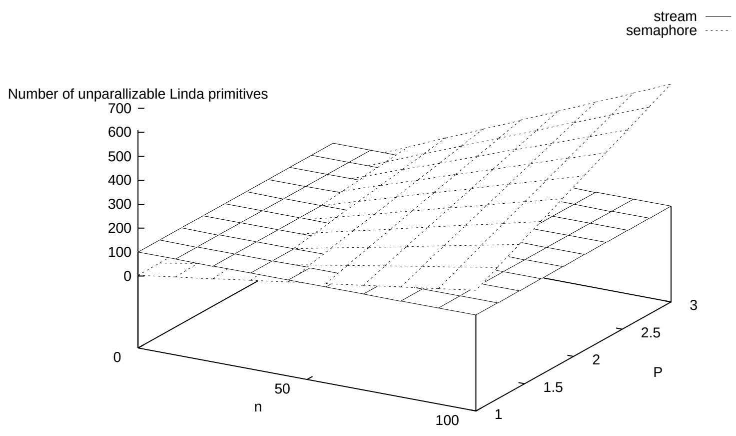

Figure 2: Comparison of primitive counts for the stream and (collect) semaphore solutions to the multiple `rd` problem

Simple analysis indicates that when one or more processes wish to perform a multiple rd, a `copy-collect` approach is always better except when all the tuples in a tuple space match, and in that case a stream is better. It is better because we wish to minimise the number of tuple space operations, as each tuple space operation has an overhead attached to it.

6 Implementation of `copy-collect`

When considering the addition of a new primitive to the Linda model there are a number of important issues. The primary issue is that the new primitive should extend the expressibility of the model to overcome a perceived problem. The other issue is how efficiently can the new primitive be implemented.

When considering `copy-collect` there are three efficiency issues which need considering. These are, in order of importance:

Communication It is necessary to ensure that the amount of communication that is required is kept to a minimum. The mass movement of tuples around a kernel would clearly be unacceptable.

Memory space The `copy-collect` primitive produces a *copy* of a number of tuples. If there are many processes producing copies of millions of tuples using `copy-collect` then there is clearly a concern about memory usage.

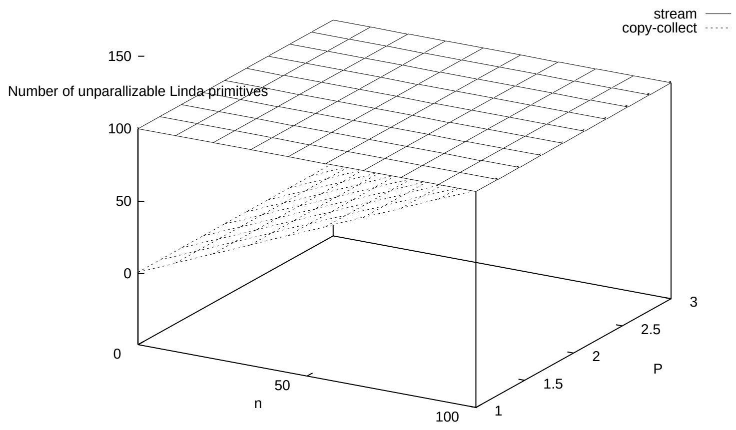

Figure 3: Comparison of `copy-collect` and stream solutions to the multiple `rd` problem

Level of parallelism It is unacceptable to merely move the problem with the solution at a Linda program level into the kernel in many situations. We wish to attempt as far as possible to increase the level of parallelism.

We will now consider each of these issues in greater depth.

6.1 Communication

Implementations where communication is a consideration fall into one of a number of categories. These are:

Single server In this approach the server is a single process. Other processes send messages to it and receive replies from it. An example of such an implementation is Glenda. The actual `copy-collect` primitive requires no more messages than an `in` or `rd`. A message to the server containing the template, and a message back containing the number of tuples copied.

Distributed server Distributed Servers use a number of tuple space managers to manage tuples and tuple spaces. These tuple space managers are normally distributed across a number of processors. Distributed servers generally fall into one of two sub-categories – those that allow duplication (that is, when `out(ts,t)` occurs, the tuple `t` is duplicated a number of times) and those that do not (that is, when `out(ts,t)` occurs, the tuple `t` will appear only a single time).

For `copy-collect` to be efficient, there is an essential requirement that the tuple space identifier is *not* used in the placement decision. This way, tuples appear on tuple space managers, regardless of the tuple space identifier, and the lookup mechanism is also independent of the tuple space identifier. When this requirement is satisfied, copying tuples between tuple spaces becomes an operation *local* to a tuple space manager – there is no communication requirement between tuple space managers.

We can make a distinction between two kinds of programming system: open systems, where a parallel program is written which may communicate with other programs, and closed systems, where the communication pattern of the entire program is given in the one piece of code.

In the open systems context, tuple placement which uses tuple space identifiers would be difficult, because tuple space identifiers must be communicated in some way to the other program. When this happens, it would also be necessary for the hashing algorithm for that tuple space to be sent with the identifier. No system, at present, will do this (although it is a consideration which must be met).

In a closed system, tuple placement which uses tuple space identifiers is a real possibility, as placement mechanisms are often generated at compile time, or *evolve* during run time. It is quite possible, in a system which implements `copy-collect`, to enforce our requirement. In fact, if compile time optimisation is used, it may well be possible to incorporate a hybrid approach, where our requirement is enforced on those tuple spaces where a `copy-collect` might happen, but to implement a more relaxed regime on those tuple spaces that do not interact.

Where tuple duplication is allowed, a simple mechanism is used whereby a tuple is placed with more than one tuple space manager. Lookup is also a simple mechanism, where a template is directed to a number of tuple space managers in order to find the required tuple. Usually, some kind of arbitration is necessary before a tuple can be returned to the process making the request. When implementing `copy-collect` in such implementations, it will not be necessary to communicate tuples between tuple space managers, assuming that placement is performed without reference to the tuple space name (that is, tuples are always placed on particular tuple space managers, regardless of the tuple space name). Some communication may be needed for arbitration purposes, but in an efficient implementation, this will be no more than is required for an `in` or `rd`. Examples of this kind of implementation are given in [7] and [8].

Where no duplication is allowed, hashing is the most commonly used method in tuple placement and retrieval. Each `out`ed tuple hashes to a tuple space manager, and each template will hash to either a single tuple space manager, or a set of tuple space managers where the tuple can be found. Once again, making the assumption that the tuple space name is not used in the hashing process, no tuple movement is required – the operation is simply a matter of re-tagging the appropriate tuples. Examples of this kind of implementation are given in [9] and [10]. In these implementations, a `copy-collect` primitive requires no more communication than an `in` or `rd` operation.

It is obviously impossible and unnecessary to consider all the particular implementations that have created. However, we hope that we have demonstrated that in many cases the communication cost of adding the `copy-collect` primitive is equal to a `in` or `rd` that blocks.

6.2 Memory space

Another important issue is memory usage. This largely depends on the data structure being used to store the tuples. There is however, no meson why a data structure can not be used where the tuples are tagged with the tuple space identifiers which contain them. In this way there is no need to actually duplicate the tuples.

6.3 Level of parallelism

In a single server approach `copy-collect` acts very like the semaphore approach. The server services one primitive at a time, so in the worst case you can have an implementation that mimics the worst aspects of both Linda level approaches. The server locks the entire tuple space (by accessing only one primitive at a time) and then could if a poor data structure was being used check every tuple to see which matched the template and then duplicate them. Therefore, every tuple is being checked and the process is sequential.

A distributed server however, can remove the sequential access to the tuple space. If each server holds some of the tuples then the checking and duplication can be carried out in parallel by the different servers. Therefore, the sequential accessing of the tuple space is parallelised. Also better tuple storage data structures can remove the necessity to check all the tuples, reducing the number needing to be checked.

6.4 Conclusion

A brief overview has been given of some of the important implementational issues. Current work on our PVM kernel[10] shows that it is possibly possible to use extra information that `copy-collect` provides to increase performance (particularly over networks where communication is expensive). This is because `copy-collect` is a tuple space operation. It gives you information about which process is likely to be using the copied tuples.

7 Conclusion

We have proposed a new primitive called `copy-collect` which has been added to our Linda kernel[9, 10]. The primitive can be used to overcome the multiple `rd` problem. We have shown how using current implementation strategies `copy-collect` can in many cases be implemented efficiently. Work is currently underway considering how `copy-collect` allows more efficient implementations.

Acknowledgements

During this work Antony Rowstron was supported by a EPSRC CASE studentship with British Aerospace Military Aircraft Division. Andrew Douglas was supported by an EPSRC grant number GR/J12765.

References

References

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

[2] N. Carriero and D. Gelernter. *How to write parallel programs: A first course*. MIT Press, 1990.

[3] 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.

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

[5] K.K. Jensen. *Towards a Multiple Tuple Space Model*. PhD thesis, Aalbrog University, Department of Mathematics and Computer Science, 1993.

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

[7] C. Faasen. Intermediate uniformly distributed tuple space on transputer meshes. 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.

[8] S. Ahuja, N. Carriero, D. Gelernter, and V. Krishnaswamy. Matching language and hardware for parallel computation in the Linda machine. *IEEE Transactions on Computers*, 37(8):921–929, 1988.

[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] 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.