Global Synchronisation in Linda

Paul Butcher*, Alan Wood†, Martin Atkins‡
Department of Computer Science
University of York
Heslington
York YO1 5DD

Abstract

This paper examines the "Distributed Summation" problem, and its solution in Linda. A number of problems with the current set of primitives are examined, and a new primitive, `collect`, is proposed.

1 Introduction

The coordination language Linda [1, 4, 5] constitutes a simple, efficient and (most importantly) portable inter-process communication mechanism. Linda is based upon the concept of tuple spaces, which are bags of tuples (records of typed fields). Six simple primitives are provided on tuple spaces: `in`, `out`, `rd`, `eval`, `inp` and `rdp`. Many recent implementations also provide a tuple space creation primitive `tsc`, and we will assume that such a primitive is available throughout the rest of this paper.

The semantics of `in`, `out` and `rd` are well understood, and will not be covered any further in this paper. The reader requiring more details is referred to the literature. `Eval`, the process creation primitive, is significantly more contentious. However this issue is outside the scope of this paper.

`Tsc` has not been around long enough for a consensus on its semantics (or indeed its name) to be reached. The ideas presented in this paper simply require that tuple spaces may be dynamically created — most current implementations of `tsc` provide all the necessary features. In this paper, we shall assume that `tsc` returns a handle to a tuple space, which may be stored in a variable of type `TS`. Tuple space handles may be placed in other tuple spaces and may therefore be passed between processes. The other Linda primitives may be prepended with such a tuple space variable to specify which tuple space they refer to, for example:

```
TS      tsNew = tsc();

tsNew.out("hello");
tsNew.out("foo", 5);
tsNew.in("foo", int ?);
```

*Currently with Harlequin Ltd, Cambridgeshire, UK, email: `paulb@harlequin.co.uk`
†To whom correspondence should be addressed, email: `wood@minster.york.ac.uk`
‡email: `martin@kins.demon.co.uk`

If called without an explicit tuple space handle, Linda primitives refer to the "local" tuple space (i.e. the tuple space containing the calling process).

In addition, we shall assume that tuple spaces may be garbage collected (as they are in Lucinda [3]), so explicit deletion is unnecessary. If this is not the case, the modifications necessary to the given examples are trivial.

Inp and rdp are called the "predicate" primitives. If there is a matching tuple present in tuple space, they return TRUE and behave in exactly the same manner as in and rd respectively. If there is not a matching tuple in tuple space, they return FALSE. Although they have been part of Linda since its conception, a number of doubts have been expressed about their inclusion in the language (see [8]). Firstly, they seem to duplicate much of the functionality of their non-predicate forms. For example, in may be implemented in terms of inp as follows:

```
while(!inp( <template> )
        ;
```

Secondly, the predicate primitives require *global synchronisation* in order to work. Whenever the value FALSE is returned, the primitive is making a statement about the global state of tuple space. Such statements are expensive in a distributed environment. One of the perceived advantages of Linda is the fact that it encourages the construction of programs that do not rely on having such global knowledge.

In this paper, we will discuss the situations where global synchronisation is necessary and demonstrate that inp and rdp do not provide an appropriate solution. To this end we will consider a particular problem, which we have chosen to call the "Distributed Summation" problem. We will then propose a new primitive, collect, which we contend provides all the functionality of inp while avoiding some of its disadvantages.

Aside: Note that rd and rdp are "unnecessary" in the sense that they may be implemented in terms of in and inp respectively (e.g. rd becomes an in/out pair). Although making rd and rdp primitive does improve efficiency, they add nothing to the expressive power of the language.

2 Example Problem — Distributed Summation

Imagine that we wish to solve the following problem. We have n values $\{x_1, \ldots, x_n\}$, and we wish to find their sum (i.e. $x_1 + \ldots + x_n$). These additions are to be performed in parallel, preferably extracting the maximum possible level of parallelism.

A well-known algorithm that solves this problem involves forming a binary tree of processes, each process taking two inputs and producing one output (the sum of its inputs). The structure of this solution is shown in figure 1.

The tree consists of $n-1$ processes, and the calculation completes in a time $t \propto log_2 n$. Each process terminates after it produces its output, and termination of the calculation occurs when the "root" process terminates (i.e. the result is calculated).

2.1 The Linda "Solution"

At first glance, the "standard" Linda multiple worker model fits this problem well. We simply create a number of worker processes which in two numbers, add them together, and out the

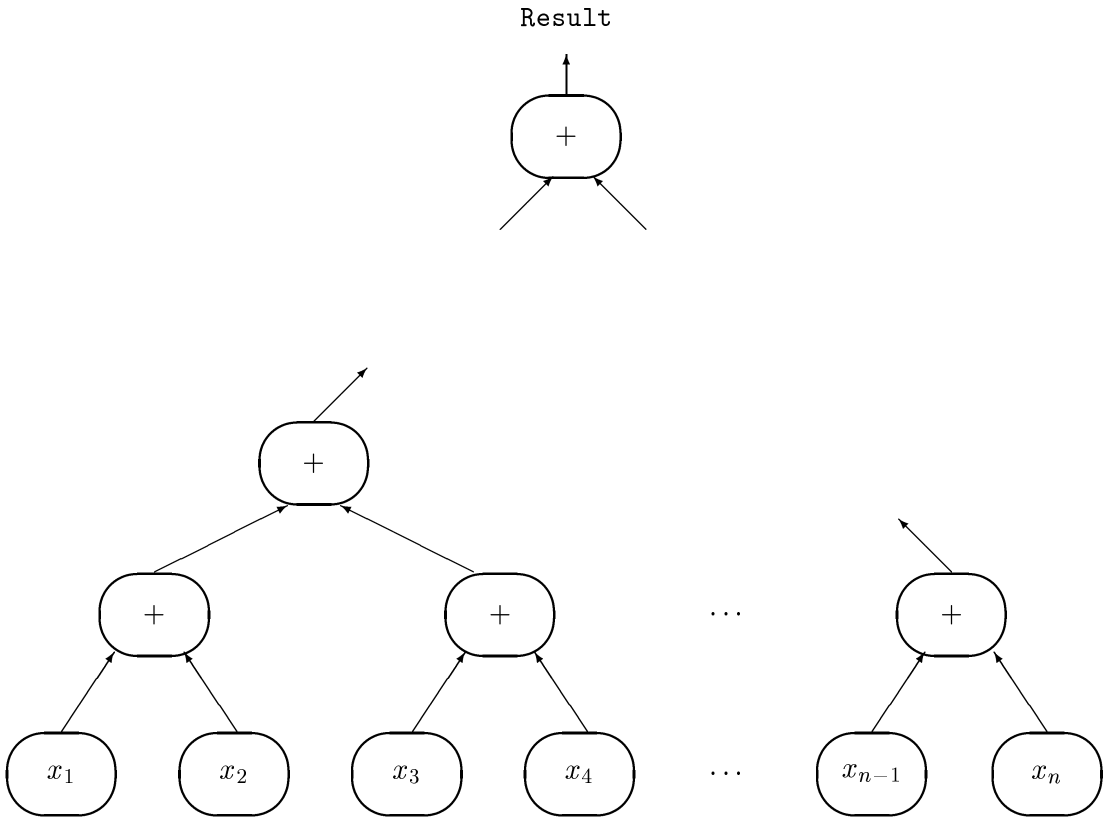

Figure 1: The Distributed Summation Problem.

```
void Worker(void)
{
        int     x, y;

        while(TRUE) {
                in(?x);
                in(?y);
                out(x + y);
        }
}
```

Figure 2: An *Erroneous* Linda Solution to Distributed Summation

result (see figure 2). This solution, while at first sight simple and elegant, has two fatal flaws. It provides no way to detect termination, and it may deadlock. Imagine that we have n values in tuple space, and n worker processes. We may find ourselves in the situation where each process has executed the first `in` of its loop (i.e. has removed one tuple from tuple space), but has blocked on the second `in` as there are no more values in tuple space. This is analogous to the *dining philosophers* problem, where each philosopher has grabbed one chopstick, and everyone starves as a result.

None of the "obvious" solutions to this problem work. For instance, changing the `in(?y)` to an `if(!inp(?y)) return;` statement, merely provides the possibility of every process dying (while this removes deadlock, it doesn't result in any work being done!). Similarly, busy waiting with `inp` isn't guaranteed to solve the problem — artefacts of the run-time system could cause livelock.

Were Linda to provide an atomic "multiple-`in`" operation, the above solution would not deadlock (analogous to providing an atomic "grab-left-and-right-chopstick" in the dining philosophers problem). Such a primitive is, however, no aid to termination detection. Atomic operations of this kind can be extremely expensive to implement in a distributed environment, and have usually therefore been avoided.

We may emulate an atomic "multiple-`in`" in Linda by implementing a "lock" tuple. Before starting the calculation, we place a single copy of the tuple `("lock")` into tuple space. The interior of the loop then becomes:

```
in("lock");
in(&x);
in(&y);
out("lock");
```

This solution guarantees freedom from deadlock, but at a huge cost to realisable parallelism: the `("lock")` tuple becomes a bottleneck. Termination detection is still impossible with this method.

Examination of the algorithm in figure 2. shows that deadlock can only occur when $P_t \geq n_t$, where P_t is the number of processes in the system at time t, and n_t is the number of values present in tuple space at time t. Notice that n_t decreases as t increases, and therefore so must P_t. We may, in fact, be more specific — the efficiency of the algorithm will be greatest if $P_t = n_t/2$. One possible algorithm which makes use of this fact is shown in figure 3.

This algorithm avoids deadlock by having the worker processes selectively die, ensuring that the correct number are always present to handle the number of values present in tuple space. Each worker is given its number, `n`, and a mask, `m`, which it uses to determine if it should die after performing each calculation. At the end of every iteration, the mask is shifted right by one bit, ensuring that half the workers die out. The only worker that stays alive for the entire calculation is worker 0, and this worker may detect termination by detecting $\mathtt{m} = 0$.

An important feature of this solution is that `AddTuples` *must know the value* of n (the number of tuples present in tuple space) before the calculation begins. It is, of course, possible to determine this value using `inp`, by removing all the tuples from tuple space and counting them before the calculation starts, then replacing them all. This, however, requires n calls to `inp`, n calls to `out` and counts the tuples serially. More importantly, this method is completely unsuited to a parallel environment since the tuples could be summed during the (serial) counting process!

```
BOOL Worker(int n, int m)
{
        int     x, y;

        while(TRUE) {
                in(?x);                 /* Perform the addition    */
                in(?y);
                out(x + y);
                if((n & m) == 1)        /* See if we should die    */
                        return TRUE;
                if((m >>= 1) == 0)
                        return TRUE;
        }
}

int AddTuples(unsigned int N)
{
        unsigned int    n, m = 1;

        if((N % 2) == 1) {              /* Ensure that N is even   */
                out(0);
                N++;
        }
        n = N / 2;
        while((n >>= 1) != 0)           /* Calculate the mask, m   */
                m <<= 1;
        for(n = 0; n < N / 2; n++)      /* Create the workers      */
                eval("Worker", Worker(n, m));
        for(n = 0; n < N / 2; n++)      /* Wait until termination  */
                in("Worker", TRUE);
        in(?n);                         /* Get the result...      */
        return n;                       /* and return it           */
}
```

Figure 3: A Linda solution when n is known

It is possible, rather than using the above algorithm, to code the tree solution given in figure 1 (by creating $n-1$ processes and annotating the tuples with their "level" in the tree), but this solution also requires knowledge of n. Chandy and Misra have given a parallel 'tree' solution in UNITY [2, p.131] to an isomorphic problem (finding the *maximum* of a set of numbers). However, their solution does not consider termination detection since UNITY is concerned with *deriving* the *property* of termination by deducing that a fixed-point is reachable, rather than developing operational means of *detecting* this condition. Termination detection was one of the concerns which led to the `collect` primitive proposed in section 3.

2.2 Composition of Programs

A pertinent question to ask of the above problem is "is it realistic?". Obviously on some systems it may not be efficient to attempt to parallelise such a fine-grain operation as addition of integers. However, by replacing the addition with any commutative and associative binary operator of arbitrary complexity,[1] we may however produce a plausible situation. A more important question is whether it is reasonable to assume that n is unknown. Presumably the values in tuple space have been produced by some earlier calculation, and it seems reasonable to assume that we may calculate n while the values are being generated. This value may then be passed to the next stage of the program.

The above is true, however, if and only if we have access to the entire source code of the program. For Linda to be useful for general software engineering it must support code re-use (whether in the form of libraries of routines, pre-compiled program modules or otherwise), i.e. it must be possible to take a section of code and use it as a "black box", unaware of its internal workings. Requiring every such routine to always create "accounting" information (such as a count of tuples) for all data it places into tuple space is unreasonable, both for reasons of efficiency and practicality. How, for example, do we know what form of accounting information will be required later?

Consider as an example a routine that generates pairs of numbers in tuple space. It could certainly create a tuple giving the number of pairs created, but now imagine that the consumer of these tuples was only interested in pairs of the form `(1, ?)`. A count of total tuples would be useless. The producer could certainly produce further tuples containing counts of pairs indexed by the first field, but what if the consumer uses the second field as the index? The number of possibilities rapidly becomes prohibitive.

Even more seriously, requiring the producer of information to supply such accounting information restricts parallelism between the producer and the consumer. The consumer must wait until the accounting information is available before it may process the data.[2] We can therefore see that, in the context of realistic large-scale program development with Linda, it is necessary to provide some sort of primitive that will allow us to make statements about the global state of tuple space. This is to enable us to detect termination and avoid deadlock.

[1] In fact, what we require is an Abelian monoid $\langle S, \diamond \rangle$, where S is a set containing an identity element i, and $\diamond$ is a commutative, associative, binary operator defined over S. The operation we are performing is similar to `reduce` or `fold` in a functional language.

[2] Parallelism between producer and consumer may, of course, be realised by having the producer supply *intermediate* accounting information. The same disadvantages apply to this approach, only more so.

3 An alternative — Collect

This paper proposes the addition of a new primitive to Linda: `collect()`. The syntax of `collect` that will be used in the examples, (which are based on a C-language embedding of Linda), is:

```
int collect(TS destination, <template>);
```

Its (informal) semantics is that `collect` moves all the tuples which match `<template>` into the tuple space `destination`, and returns the number of tuples collected.

More formal (and full) semantics for this primitive are currently being investigated. However, for the purposes of this paper we need only one property: given a *stable* tuple space (one in which no `in`s nor `out`s are occurring), `collect` will will remove *all* tuples matching the template. The intricacies of a formal semantics centre on the meaning of 'all tuples' in a non-deterministically active tuple space.

`Collect` will normally be used with a newly-created tuple space as its first parameter. This destination tuple space would be 'invisible' to other processes (which would therefore be unable to modify it) unless the creating process explicitly passed them the tuple-space's handle. This is a crucial property since the count returned by `collect` must reflect the state of the destination tuple space for as long as is required by the process containing `collect`. However, by using an existing tuple space as a parameter, `collect` can be used as a convenient 'block move' operation. In this mode, `collect` can be used to perform a kind of *inverse* `collect` which replaces the collected tuples in the source tuple space.

When several concurrent processes execute `collect`s whose templates are such that they 'compete' for the same tuples, then the common tuples "collected" are partitioned between the `collect`s non-deterministically. Thus the `collect`s are guaranteed not to get the "same" tuple (which would imply an erroneous replication of tuples). However, none are guaranteed to get *all* the tuples (i.e there is no "locking" of tuple space by `collect`), nor to get *any* tuples.

Finally, although not used in this paper, `collect` is extendable to act on tuple spaces other than the 'main' or 'default' space by prepending it with the tuple space name in the same way as the other Linda primitives.

3.1 Distributed Summation with collect

At its simplest, we may solve the distributed summation problem in exactly the same manner as that given in figure 3, and use `collect` to determine n before calling `AddTuples`. More satisfactorily, we may use `collect` to give us parallelism between the producer and `AddTuples`, as shown in figure 4.

The implementation of `AddTuples` used is exactly the same as that given in figure 3. This solution assumes that the producer will inform us of its termination (i.e. it will inform us that no more values will be produced) by maintaining a tuple of the form `("done", f)`, where `f` is `TRUE` if the producer has terminated, `FALSE` otherwise. This is a reasonable assumption, since detecting termination of the consumer will always depend upon detecting termination of the producer.

This solution successively `collect`s all the values produced so far, and adds them. The result of this summation is added to an accumulator. When the producer indicates termination, the value in the accumulator is returned.

```
int ParAddTuples(void)
{
        TS      dest;
        int     f, result, c, accumulator = 0;

        do {
                rd("done", f);
                if( (c=collect(dest = tsc(), int ?)) == 0 )
                        continue;
                dest.eval("Result", AddTuples(c));
                dest.in("Result", ?result);
                accumulator += result;
        } while(!f);
        return accumulator;
}
```

Figure 4: Distributed Summation with `collect`

Note: Depending upon the relative speeds of the producer and consumer, it may be advantageous to have the consumer delay between iterations (otherwise it may only pick up one or two values each iteration). It would also be possible to implement a heuristically variable delay dependent upon the count returned by collect. One could even imagine several instances of `ParAddTuples` running in parallel.

3.2 Implementation of `collect`

Figure 5 shows a possible implementation scheme for a processor mesh. This assumes a similar tuple space distribution strategy to that described in [5], where each processor has an `in`-set (the columns in this diagram) and an `out`-set (the rows). If processor p performs a `collect`, the destination tuple-space handle, together with the template, is broadcast to all the processors in p's `in`-set. Each of these processors then broadcasts a "move" instruction to all the processors in their `out`-sets, causing all the tuples matching the template to be moved to the destination tuple space (some arbitration will be required at this point to ensure that all copies of the new tuple space are consistent). Finally each processor in p's `in`-set sends a count back to p.

`Collect` requires that each node that may contain a matching tuple must be visited (as with `inp`). No tuples need be sent over the network in reply however, only a count of matching tuples.

4 Discussion

4.1 Cost of `collect`

As hinted above, the cost of performing a `collect` is similar to that of a single `inp` that returns `FALSE`. A "failed" `inp` has to query, and receive a reply from, each node in the system

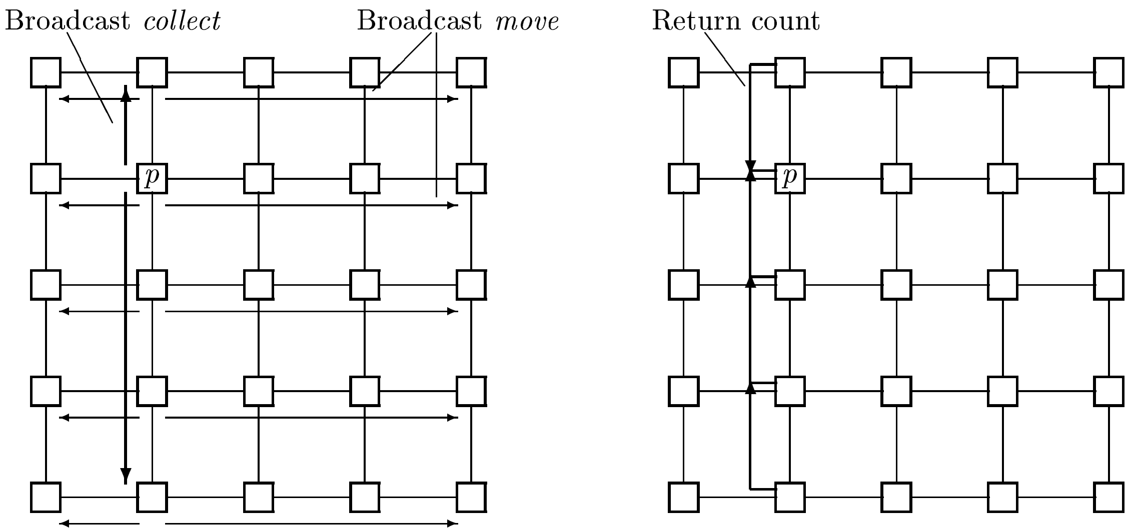

Figure 5: Implementation of `collect` on a processor mesh

that may have a matching tuple. `Collect` similarly has to query each such node. The only extra overhead of `collect` is that of moving the tuples from one tuple space to another. Note that in many systems this will not require transmission of any data between processors, and may not even require data copying in local memory (simply requiring the manipulation of pointers).

4.2 Equivalence of `collect` and `inp`

We would like to show that `inp` and `collect` are equivalent in that they may be implemented in terms of each other. The difficulty is, however, that no semantics for `inp` has been defined. As mentioned in section 3 a full semantics for `collect` has yet to be finalised, but the basic property mentioned there is all that is required for this paper. We shall therefore *assume* that `inp` satisfies at least that property. That being the case, Figure 6 shows `inp` implemented by `collect`, and figure 7 shows `collect` implemented by `inp`.[3] This demonstrates that by replacing `inp` with `collect` we have not lost any expressive power by deleting `inp` from Linda, and adding `collect`. However, adding `collect` brings significant advantages to the model.

The full semantics will provide `collect` with an appropriate notion of *boundedness* — that is `collect` will always terminate (on finite tuple spaces). Given this, the symmetry in the 'cross-implementations' breaks down since `collect` implemented using `inp` may not terminate in situations where the `collect` *primitive* does. The validity of `inp` implemented using `collect` is maintained for the cases where `inp` terminates, however. We regard this as an important improvement on `inp`. `Collect`'s semantic advantage, therefore, comes from the fact that it supplies us with more useful information than `inp`, and terminates when `inp` may not.

[3]Unfortunately, the syntax of C is not powerful enough to perform such a "cross implementation" as tuples are not first class. Imagine appropriate code in the sections marked `<template>`.

```
BOOL inp(<template>)
{
        TS      ts
        int c = collect(ts=tsc(), <template>);
        if(c == 0)                      /* No matching tuples      */
                return FALSE;
        while(--c > 0) {                /* Put c-1 tuples back     */
                ts.in(<template>);
                out(tuple);
        }

        ts.in(<template>);              /* Get the one that's left */
        return TRUE;
}
```

Figure 6: `Inp` implemented in terms of `collect`

```
int collect(TS ts, <template>)
{ int c = 0;

         while(inp(<template>)) {
                ts.out(tuple);
                c++;
        }
        return c;
}
```

Figure 7: `Collect` implemented in terms of `inp`

4.3 Other possible primitives

Two other primitives, `alt` and `forall` were considered instead of `collect`. This section describes these primitives, and discuses the reasons for discarding them. In addition we consider the proposed Linda variant, Melinda.

`Alt` is similar to the construct of the same name in occam, taking as its arguments a number of templates (rather than the single template taken by other Linda primitives). A tuple matching one of these templates is removed from tuple space. A possible variant is `altp`, a predicate form of `alt`, which either removes a tuple matching one of the templates and returns `TRUE`, or has no effect on tuple space and returns `FALSE`.

It is easy to see that the blocking variant of `alt` solves none of the problems discussed above, as it may easily be implemented in terms of `in`, `out` and `eval`. The non-blocking form is simply syntactic sugar for a sequence of calls to `inp`.[4]

`Forall` allows us to iterate over all the elements of a tuple space matching a particular pattern. Many different syntaxes and semantics are possible for `forall` (indeed this is part of its problem), and it would not be possible to cover all variants here. The version we shall consider takes the form of a control construct (similar to `while` or `for` in C). If, for example, we wanted to add 1 to the value of every tuple matching the pattern `(int ?)` we could use the code:

```
int     x;

forall(?x)
        out(x + 1);
```

If there are n tuples matching the pattern, then n processes will be created, each one running the code `out(x + 1)` with `x` set to the value removed from tuple space. The first thing to note is that this form of `forall` doesn't provide us with an easy solution to distributed summation, because each process only gets one tuple. By changing the semantics of `forall` so that each process is given a number, and then having process 0 communicate with 1, process 2 communicate with 3, etc. we may solve this particular problem, but this approach is not easily implemented, and will not generalise well. Alternatively, we may implement `collect` in terms of `forall`, and use the solution given in figure 4, but `collect` implemented in this way is extremely inefficient.

A number of difficult semantic problems occur when considering `forall`. If the tuple space being accessed is not altered by any other process while the `forall` is in progress, then its semantics are well defined. If, however, the tuple space is being modified, we must ask precisely which tuples will be found by `forall`. Does it lock the tuple space against modifications until it has terminated? Or does it just lock all tuples matching the given template? If locking is not used, what happens if a tuple that was present in tuple space when the `forall` started is removed before it finishes? How about tuples added after it starts? These issues are important because the processes running under the `forall` have

[4]Things become slightly more complicated if we want "fair" alternation, but the effect is essentially the same.

access to the tuple space, and may perform operations on that tuple space (as in the above example). These processes need to know what they are likely to find.

While it is obviously possible to choose a particular semantics for `forall`, none of the choices are clearly superior to the others, and none are particularly intuitive. In addition, a small change in the semantics will have a dramatic effect on the behaviour of a program using the primitive. For these reasons we rejected `forall` as a primitive to help solve the problems addressed by `collect`, which is considerably cleaner and more precise (and thus easier for a programmer to understand).

Melinda (or Linda 3) is a proposed extension to Linda described in [6] and expanded upon in [7]. Its main advantage over "standard" Linda is the inclusion of multiple tuple spaces in the language.[5] Unlike the implementation of multiple tuple spaces used in this paper (in which `tsc` returns a *handle* to a new tuple space) Melinda's `tsc` returns a *suspended* or *frozen* empty tuple space image. This suspended tuple space is of type `image ts`, and may be "enlivened" by placing it in another tuple space (resulting in a tree structure for tuple spaces similar to the UNIX^{TM} filing system). Melinda allows "live" tuple spaces to be removed from other tuple spaces, resulting in them being re-frozen. Although no Linda primitives may operate on frozen tuple spaces, Melinda suggests that the base language should provide functions to allow such frozen images to be translated into data structures (e.g. lists, arrays, etc.) understood by that language.

At first sight this freezing of tuple spaces gives us a facility similar to `collect`. A process wishing to use the tuples in a particular tuple space removes that tuple space, and converts it to a host language data structure for processing. This method, however, results in the processing being carried out serially.

Alternatively, we may remove the tuple space from its original scope and re-enliven it in a "private" scope, allowing a number of processes (those with access to the private scope) to work on it. Note that this approach still gives us no help with determining the number of tuples present in the tuple space, and is in essence the same as "locking" the tuple space. More insidiously, allowing tuple spaces to be removed in this manner may result in calls to `out` blocking. Where previously we only had to consider deadlock when calling `in` and `rd`, we now have to consider it on calls to `out` as well.

5 Conclusion

We have proposed a new Linda primitive `collect` to replace the original Linda primitives `inp` and `rdp`. While `collect` may be implemented in terms of `inp`, we have argued that it is both more efficient (allowing significantly greater parallelism) and semantically cleaner.

Acknowledgements

The research described here was funded in part by SERC Research Grant GR/G31000. During this work Paul Butcher was supported by a SERC studentship, and Martin Atkins was supported by DRA Malvern. Thanks are due to Dave Scholefield for useful discussions on the semantics of `collect`.

[5] Melinda provides a number of other features (e.g. archival of "live" processes and tuple space access rights) with their own advantages and disadvantages. These features are not considered here.

References

[1] Butcher, P. *A Behavioural Semantics for Linda-2.* Department of Computer Science, University of York. Technical Report YCS–137, July 1990.

[2] Chandy, K.M. and Misra, J. *Parallel Program Design: a foundation.* Addison-Wesley, 1988

[3] Butcher, P., and Zedan, H. *Lucinda — A Polymorphic Linda.* In Banâtre J.–P. Ed. Proceedings, Research Directions in High-Level Parallel Programming Languages, Lecture Notes in Computer Science, LNCS-574 Springer Verlag, 1992

[4] Carriero, N., and Gelernter, D. *Coordination Languages and their Significance.* Yale University, Department of Computer Science. Research Report YALEU/DCS/RR–716, July 1989.

[5] Gelernter, D. *Generative Communication in Linda.* ACM Transactions on Programming Languages and Systems, 2(1) pp 80–112, January 1985.

[6] Gelernter, D. *Multiple tuple spaces in Linda.* in: Odijk, E., Rem, M., and Syre, J.-C. Ed. PARLE '89: Parallel Architectures and Languages Europe. Volume II: Parallel Languages, June 1989. Springer Verlag, Lecture Notes in Computer Science Volume 366, pp 20–27.

[7] Hupfer, S. *Melinda: Linda with Multiple Tuple Spaces.* Yale University, Department of Computer Science. Research Report YALEU/DCS/RR–766. February 1990.

[8] Leichter, J. *Shared Memories, Buses and LANs — Linda Implementations Across the Spectrum of Connectivity.* Department of Computer Science, Yale University. Research Report YALEU/DCS/TR–714. July 1989.