Linda Implementation Revisited

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

Abstract

Linda is a model for communication and co-ordination of parallel processes. The model provides a virtual shared memory called tuple space, and primitives to place tuples into and remove tuples from tuple space. The style of programming provided by Linda is quite different to the style of, say, occam.

We describe a new implementation of Linda across a network of transputers. We provide the four Linda primitives, `in`, `out`, `rd` and `eval`, together with a new primitive, `collect`, developed at York.

The implementation focuses on two issues. The first issue is that the ordering of `out` operations in a sequential process must be preserved if we want Linda to act as a co-ordination langauge. Our implementation provides this. The second issue is the implementation of `eval`, Linda's mechanism for spawning processes. We outline an implementation which provides arbitrary spawning of processes which execute concurrently, despite the restriction, enforced by the transputer architecture, of declaring a static number of processes at the compilation stage.

We provide a small example to show how Linda can be used to write parallel programs, and then outline current work being undertaken at York, which focuses on interpretive environments for high level parallel programming languages. A prototype Linda implementation and ISETL interpreter have already been developed.

1. Introduction

Transputer programming is not often as easy as one thinks. Its design principles, being based on formal process calculi, require certain restrictions on the way that programs are written, such as a static, previously declared number of communication channels and processes. This often results in spatial and temporal coupling of processes; processes know where and to whom they are sending messages, and the execution time of the sending and receiving processes must overlap.

This paper outlines recent research into the implementation of a shared memory style of parallelism, based on the Linda communication model [9], for a Meiko Computing Surface. Linda provides a simple model for communication between and co-ordination of processes. The approach is a highly de-coupled one, where communication is performed simply by placing messages into a virtual shared memory. A process sending a message knows nothing of the receiver of the message, and vice versa. In fact, the temporal existence of sending and receiving processes may be quite distinct[6].

We describe an implementation of Linda currently under construction at the University of York. The goal of this research is to implement two high level programming language environments. We embed the Linda model into these languages, and have them running processes across a set of processors, accessing a shared memory which is distributed between these processors. Both of these language environments are interpreted, and this is one aspect which has affected our design.

The paper proceeds by outlining the York variant of the Linda model, including a new primitive, `collect`, developed at York[4]. We point out a few problems which must be addressed by any implementation. We then outline the implementation in two phases: the first phase gives the tuple space implementation, and the second phase describes process creation and management. We give a comparison of our implementation with other Linda implementations, in particular with those given in [7], [8] and [16], which describe transputer implementations.

2. Linda

Linda provides primitives for communication between processes. This communication takes place using *tuple spaces*, which are shared memories with an associative lookup mechanism.

A tuple space is a *bag* of *tuples*, a tuple being an ordered sequence of heterogeneously typed objects. For example, `[1, true, "fred"]` is a tuple of three fields of type integer, boolean and string. Given a tuple `t`, we can access the individual fields through indexing. For example, `[1, true, "fred"](3)` has the value `"fred"`, while `[1, true, "fred"](8)` is undefined.

Corresponding to tuples, we have *templates*, which are used in the process of *associative matching*. A template is also an ordered sequence of heterogeneously typed objects. For example, $|$`[1, true, "fred"]`$|$ is a template of three fields. This template will precisely match the tuple `[1, true, "fred"]`. Templates are different from tuples because a field of a template may be typed, but have no value. For example, we can have a template $|$`[1, ?bool, "fred"]`$|$, which matches the tuple above, but will also match the tuple `[1, false, "fred"]`. The ?*type* notation represents a typed hole. Templates are first class objects and may be passed between processes.

A template will *match* a tuple if the following hold:

- The tuple and the template have an equal number of fields,
- Each field contains an item of the same type,
- Each field contains an object with an equal value, or the template field contains a hole.

The following functions are supplied by Linda, for manipulating tuple spaces:

- `out(ts, t)` takes a tuple space `ts` and a tuple `t`, and places the tuple `t` into the tuple space `ts`
- `in(ts, t)` takes a *template* `t`, and searches in the tuple space `ts` for a *tuple* matching `t`. If a suitable tuple is found, it is returned as the result. Otherwise, the process making the call is blocked until a suitable tuple enters the tuple space.

  The tuple returned is removed from the tuple space.
- `rd(ts, t)` takes a *template* `t`, and searches in the tuple space `ts` for a *tuple* matching `t`. If a suitable tuple is found, it is returned as the result. Otherwise, the process making the call is blocked until a suitable tuple enters the tuple space.

  The tuple returned remains in the tuple space.

- `collect(ts, tt, t)` takes two tuple spaces, `ts` and `tt`, together with a template `t`, and moves all tuples in `ts` which match `t`, into the tuple space `tt`. It returns a count of the number of tuples moved.

  The `collect` primitive replaces the functions of the original Linda `inp` and `rdp` operations [9].

2.1. *Multiple tuple spaces*

We allow multiple tuple spaces, each with a unique identifier, and provide a new primitive function, `tsc`, which returns a new *unique* tuple space identifier. In our system, each tuple space is an independent item, unlike Yale multiple tuple spaces [10], where each tuple space is *enclosed* within a larger one.

Multiple tuple spaces are ideal for local communication between a select set of processes, and hiding communication between processes. However, tuple spaces are first class objects, and tuple space identifiers can be passed between processes.

2.2. *Eval*

The `eval` primitive is Linda's way of spawning processes, and is given a tuple space and a tuple. A thread is spawned, to evaluate the tuple and place the result in the given tuple space. For instance, `eval(ts,[e, s, t])` will spawn a new process to evaluate the tuple `[e, s, t]`. The resulting tuple will be placed in the tuple space `ts`. It must also evaluate the three expressions `e`, `s` and `t` in parallel, if any of the components access tuple space. For this reason, some care has to be taken to ensure against unintentional deadlock through processes being blocked.

2.3. *Two important issues*

We now comment on two important issues which must be considered when implementing Linda. Both have been mentioned previously [13], in relation to C-Linda, a commercial Linda implementation [14]. We demonstate these issues using code fragments, written in the style of ISETL[3], a Pascal like imperative language.

2.3.1. *Ordering of* `out`*s*

Take the following fragments of code: The first is a series of `out` operations on a tuple space `ts`.

```
-- out four work tuples, followed by a ["Done"] tuple

out (ts, [tuple1]);
out (ts, [tuple2]);
out (ts, ["Done"]);
```

The second performs a number of operations on `ts`:

```
-- work on tuples in ts until the tuple ["Done"] appears

while (collect (ts, ts, |["Done"]|) == 0) do
   perform_some_stuff();
end do;

-- collect all the remaining tuples in a private tuple space

n := collect (ts, tt, |[?int]|);

-- and perform some work on them

if (n > 0) then
   clear_up (n, tt);
end if;
```

The pattern of this last fragment of code is often found in master/worker style parallelism. It loops for some time, working on tuples in a tuple space, `ts`. The end of tuple production is signalled by the appearance of a tuple `["Done"]`. Once this appears, we count and move the remaining tuples, perform some operations on these tuples and finish.

The important issue here is that, once the `["Done"]` tuple is detected in tuple space, the following `collect` operation will catch all tuples generated by the proceeding `out` operations. So, for a sequence of `outs` (in a sequential process) the tuple ordering must be preserved (ie the tuple outed by $\mathtt{out}_i$ must be available *before* that produced by $\mathtt{out}_j$ for all $i < j$.

2.3.2. Implementing `eval`

The second issue concerns `eval`, and is again demonstrated by a fragment of code. Take the following function:

```
myfunc := func ();
             do_some_stuff ();
             return "Finished"
          end func;
```

We can spawn a process using the `eval` primitive, which computes this function in parallel:

```
eval (ts, [myfunc()]);

do_local_work ();

in (ts, |["Finished"]|);
```

This code fragment starts a thread, does some work, and then waits for the spawned process to finish. If the computation performed by the `eval` were not to begin immediately, the `in` will block waiting for the result.

The second issue is that we must implement the `eval` primitive in a way that enforces the intuitive (some might say obvious) meaning of `eval` – that is, `eval` *must* spawn a new thread. As will be demonstrated, not all implementations of `eval` observe this inuitive meaning, which consequently leads to deadlock in some programs.

A similar issue is discussed by Narem [13], who points out that in the computation of `eval(ts, [f(),g()])`, we must compute `f()` and `g()` in parallel, otherwise deadlock might occur due to dependencies between the functions. Co-incidently, Narem also points out that, in the context of the C programming language, the functions `f` and `g` must not side-effect any global variables, as this amounts to communication between processes. We solve this problem by dealing with *closures*. A closure is a package, in which a separate environment is created for the separate thread.

3. Implementing Tuple Space

Consider a set of processors, $\{P_1, \ldots, P_n\}$, each with a local memory. We could consider each P_i to be, say, a *T800 Transputer* with 4 Megabytes of memory. We distribute tuple space as a set of *tuple space managers*, $\{T_1, \ldots, T_m\}$, distributed across the P_i is some way. Each T_j consists of a process T_j^P and a queue[1] of requests T_j^Q. The exact processor topology and distribution of tuple space managers is not fixed. However, each tuple space manager must be able to communicate with all the other tuple space managers.

User processes make tuple space requests by adding commands to the queue of a tuple space manager. We could approach this in two ways. The first requires a user process to know about only one tuple space manager, and for this manager to take care of placement of tuples within the tuple space. The second approach requires a user process to know about all tuple space managers, and for placement to be decided upon by the user process itself (using hashing). This latter approach has a slight cost advantage, as well as a certain flexibility since placement schemes are compiled with the application, rather than with the tuple space managers.

3.1. Tuple space distribution

The tuple space is distributed over the set of tuple space managers. Tuple placement and lookup is decided upon by a pair of hashing algorithms. The first hashing algorithm uses information which can always be determined, to produce a subset of the tuple space manager processes over which this *kind* of tuple can be placed. For example, given a tuple and a template, we can always determine the number of fields, and their individual types.

The second hashing algorithm uses the result of the first algorithm, plus some specific information which will determine precisely where to place the tuple in question. For a tuple, this information will always be obtainable, but for a template, the information required by the second hashing algorithm may not be determinable. We say that a tuple or template for which this information is determinable is *completely presented*.

As an example, when we perform an `out` operation, the tuple is *completely presented*; it contains no holes. The first hashing algorithm could give us a range of tuple space mangers T_x to T_y over which to place this kind of tuple. Then, the second algorithm could take the value of the first field of the tuple, produce an integer value representation of it, and then return this value, modulo $(y - x)$. Assuming that the value of the first field changes regularly, the distribution of this kind of tuple over the range of tuple space managers will be fairly even.

Performing an `in` operation follows a similar routine. The first algorithm will take a template and produce the range of tuple space managers over which this kind of tuple is distributed. Then, if the template is *completely presented* in its first argument (that is, the first argument is *not* a hole), we can determine precisely which tuple space manager will contain such

[1] Which may itself be implemented by a separate process.

a tuple. Otherwise, if the template is not *completely presented* in its first argument (that is, the first argument *is* a hole), we can leave requests for this tuple on the queue of all tuple space managers within the range specified by the first algorithm. Some arbitration is required so that only one matching tuple is removed from the tuple space and returned to the user process.

We note a number of things. Firstly, that hashing takes computation time, and we must choose hashing algorithms that take $\mathcal{O}(1)$ time to compute.

Secondly, since we are distributing tuple space across processors, and across tuple space managers, we want to provide hashing algorithms that give a fair and even distribution of tuples.

Finally, in some compiled languages, where the entire code is presented before runtime, we can perform static analysis of the code to determine the pattern of process communication. This can often result in a very fast Linda runtime system, where sequences of `in`s and `out`s are optimised. In any environment where processes and code are dynamic, such as in interpreted language environments, this kind of analysis cannot take place. In addition to this, we are proposing that tuple spaces and templates are first class objects, which can be passed between processes via tuple spaces. This makes static analysis much more difficult to perform[2]. We are, then, highly dependent upon hashing as the means of providing very fast placement and lookup of tuples within tuple space.

3.2. Servicing requests

We discuss briefly each operation with a view to its implementation:

- An `out` operation request causes hashing to take place. We require that the tuple is completely presented, and that hashing will choose a tuple space manager on which to place this tuple. A message to this tuple space manager, containing the tuple and tuple space name, is sent. The tuple will be *stored* in this particular tuple space manager, unless a suitable blocked process is waiting, in which case some other action will take place.

  The cost of an `out` operation will be the cost of sending *one* message.

- An `in` or `rd` is more complicated, since there are two cases; one for a completely presented template, and one for a template where hashing cannot determine an exact tuple space manager.

  In the case of a completely presented template, the template and tuple space name, together with the communication address of the requesting user process, will be sent to the hashed tuple space manager. There, a search will be undertaken to find a suitable matching tuple. A find will cause the tuple to be sent to the requesting process, and any action on the tuple (such as removal, if we are performing an `in`) will take place. No find will cause the process to be kept in a list of *blocked* processes. Any subsequent `out`s will cause this list to be searched, and any matching tuple will prompt the manager to send the tuple to the requesting process, and take action with the tuple.

  The cost of an `in` for a completely presented template will be *two* messages.

  In the case of an incompletely presented template, a request must be made to each of the tuple space managers in the set returned by the first hashing algorithm. However, responsibility must be taken by one process to see that *only one* tuple is sent to the requesting process. Therefore, the tuple space manager to which the request is sent (chosen arbitrarily from the set) keeps a record of the transaction, assigning it a *unique* transaction

[2] There is an analogy here with the static analysis of functional languages with higher order functions.

number, and passes this, together with the tuple and the tuple space name to the rest of the tuple space managers in the set[3].

Any find by one of these tuple space managers prompts the finder to send the tuple to the tuple space manager in charge. If the transaction is still in the list, the tuple is sent to the requesting process, cancellation messages are sent to the other tuple space managers involved in this transaction, and the transaction is removed from the list.

Any tuples matching this request, which are subsequently received by the tuple space manager in charge, will cause the tuple received to be re-`out`ed, since no record of the transaction will be present in the transaction list.

If a tuple is not found in one of these tuple space managers, the transaction is added to the blocked process list. Receipt of a *remove transaction* message causes the transaction to be removed from the list, if the transaction is present in the list, and is ignored otherwise. Receipt of a tuple matching a transaction will cause the transaction to be removed from the list, and the tuple is sent to the tuple space manager in charge.

Note that the keeping of transaction records means that only one tuple will ever be returned to the user process. Even when a cross-over of a matching tuple and a delete transaction message occurs, there is no inconsistency incurred since the delete message will be ignored, and the tuple returned will be re-`out`ed.

If the first hashing algorithm returns a range containing m tuple space managers, the *best case* cost (when a matching tuple is found on the tuple space manager in charge) of an `in` for a template which is not completely presented is 2 messages. There are two *worst cases*: the first, when a single match is found on a tuple space manager which is not the tuple space manager in charge, requires $2m$ messages (2 messages to send the request and receive the reply, $m-1$ message requesting a tuple from the other tuple space managers, 1 reply and $m-2$ requests to remove the transaction). The second worst case is when no match is found on the tuple space manager in charge, but a match is found on each of the other tuple space managers in the set. The cost here is $3m-2$ messages (2 messages to send the request and receive the reply, $m-1$ messages requesting a tuple from the other tuple space managers, $m-1$ replies and $m-2$ re-`out`ed tuples).

- A `collect` causes a global send to the set of tuple space managers which may contain tuples of this kind (that is, we use only the first hashing algorithm). Like the `out` operation, the tuple space manager to which the request is made is chosen arbitrarily from the set of tuple space managers, and is put in charge of summing the results and returning the count.

  During the execution of a `collect` primitive, no tuples are moved between tuple space managers, because placement of tuples depends only upon the value of the tuple, and not on the tuple space. In the case where the tuple space names are different, each individual tuple is `out`ed, which prompts lists of blocked processes to be searched. This blocked process list search is not required when the destination and host tuple space have the same identifier, as being present in tuple space is sufficient proof that no processes are blocked on this tuple.

  The cost of a `collect` will be $2m$ messages, where m is the number tuple space managers returned by the first hashing algorithm.

[3] An initial search of the tuple space manager receiving the request will be made, and a find will require no broadcast to the other tuple space managers.

3.3. Data structures

A naive approach to storing tuples and blocked processes would be to use lists. A list has advantages when it comes to adding items, which takes constant time. However, the action of searching tuple space for matching tuples, and matching tuples against lists of blocked processes might benefit considerably by a more thoughtful approach.

For example, if we take a tuple space to be a list of tuples, matching a template `t` against these tuples would take $\mathcal{O}(nm)$ time where n is a measure of the size of the template being matched and m is the number of tuples in the list.

Storing tuples in a hash tree would be much better. Adding tuples takes $\mathcal{O}(\log n)$ time, and finding a match takes $\mathcal{O}(\log n)$ to find the list of tuples to search, and $\mathcal{O}(nr)$ time for matching, where r is the number of tuples with the same hash value.

Some kind of *trie* tree representation [15] might be convenient. The structure of a tuple space takes a tree format, according to the size and type of the tuples it contains. Adding tuples means adding branches to represent those parts of the tree which don't exist already. A marker containing an integer represents the number of tuples of a particular size, type and value. In addition, we might include in this marker the list of processes blocked on this kind of template. This storage scheme has the same complexity as the hash tree, but has a smaller storage space requirement, and integrates the blocked process list into the tuple space.

At the moment, our implementation uses the naive data structures, although some progress on using hash trees has been made.

3.4. Tuple space garbage collection

No garbage collection of tuple space is performed in this implementation[4]. Such garbage collection would be a simple matter of tagging each tuple space with a reference count. With first class tuple spaces, it would be necessary to include in the reference count occurrences of tuple space identifiers embedded within tuples in tuple space. This would mean searching tuples when an `in`, `rd` or `out` operation is requested, to update the reference.

It is interesting to note that with a slight modification to our Linda model, it would be possible to use Linda in a persistent way. We can include within the model some kind of *global* tuple space available to *all* processes. In this tuple space, identifiers could be placed of tuple spaces which are required to be persistent. This kind of model would appear to be suitable for the kind of database applications where clients of the database come and go[5].

4. Eval

The `eval` primitive is Linda's way of spawning processes, and is part of Linda. However, unlike others, who view `eval` as acting *within* a tuple space[6], we take the view that `eval` is simply for spawning processes, and is more to do with the language of computation than it is to do with Linda.

We require, then, an *engine* which, given a computation, can spawn a *thread* which will perform the computation and place the result in a tuple space. More importantly, we want to give the impression of unlimited, dynamic creation of processes, while implementing it within

[4] By garbage collection of tuple space, we mean removing entire tuple spaces to which no references exist

[5] Although it doesn't implement the sophisticated persistence described in [2]

[6] Some propose *active* tuples – tuples which are within tuple space, evaluating themselve into *passive* tuples. Some go as far as to be able to `in` an active process, thus suspending its computation.

the strict regime enforced by the Computing Surface. Such an engine will consist of the following items:

- A single heap, shared by all threads,
- A list of runtime elements, containing stack pointers and code pointers,
- A main function which takes a closure, and computes its value,
- A single communication channel, together with a semaphore to control sharing, and a buffer to hold messages,
- A main function which will spawn threads, perform garbage collection and so on.

Each spawned thread will have an index to the list of stack pointers, which holds information about the state of its computation. All references to the heap will be embedded within this list, so that, at any time, the main process or any thread process can perform garbage collection.

It is a consequence of the transputer architecture that communication links must be declared in advance, before computation begins. Because all threads need a communication channel to the assigned tuple space manager, we must share this channel. The communication channel becomes a *transport*, using Meiko CSN terminology, together with a buffer which stores incoming messages. A semaphore will be required to control access to the transport. Sending messages is performed in the same way, although they must be tagged with a unique identifier belonging to a particular thread. Receiving incoming messages requires a process to check the message store for a message already received. Otherwise, the thread must wait until a suitable message arrives.

The main function waits for messages containing closures[7]. A thread is created using the `runp` function provided by the Meiko CS-Tools environment.

4.1. Floor management

Given a number of engines distributed over the set of processors, it is useful to have some kind of process performing load balancing of some kind. Each time a new thread is requested, it is requested from this *floor manager* process which returns a message containing an address of the *least loaded* engine. The *least loaded* engine is calculated by keeping a count of the number of threads executing at any time by a particular engine. Before a thread terminates, the result of computation is placed in the tuple space, and a message is sent to the floor manager to say that the thread has completed.

This is not true load balancing, as it does not take into account the amount of processing needed by a particular thread. Also, two engines running on the same processor can handle only as many threads as a single engine with a processor to itself. The floor manager process could be extended to cope with the latter kind of information.

5. Comparisons

Many implementations have been described, beginning with the uniform distribution model outlined in [9] and [1], and the hashing schemes of [5] and [1]. More recently, work has been

[7] A closure is an expression, together with any code that is required and the environment containing the values, at the time the closure was created, of any free variables

done by Zenith [16], Faasen [7] and Feng [8] on the implementation of Linda on transputer networks.

5.1. Zenith's implementation

In [16], Zenith describes an implementation of Linda on a number of transputers. His approach is based on the compilation of the C programming language with embedded Linda operations, into the occam programming language, which can be compiled to execute on transputers.

Zenith's approach is similar in some respects to ours. He proposes to divide tuples into "distinct subsets", based on the *kind* of tuples which appear. The kind of a tuple can be decided at compile time. Hashing is then used to determine where to place a tuple within a set of processors. Any tuples with no hash value get placed in a private, non-distributed hash table.

A *node* of Zenith's machine consists of a transputer to run the tuple space manager, together with two transputers to act as *computation nodes*[8]. Our approach is more flexible in the topology of processors, but could implement this approach if we wished it to.

It is not clear whether Zenith's approach will preserve the order of `out` operations. However, it is clear that our `eval` issue is not addressed in a satisfactory way. An `eval` operation is transformed into an `out` operation which is performed in parallel. This can be implemented in occam, presumably using the PAR construct. However, occam forbids replicated parallel processes without a constant bound. This causes a problem with code such as

```
for (i = 0; i < n; i++) {
   eval(ts, [i,P(i)]);
   }
```

as we cannot determine statically how many processes are required. The solution proposed is to transform the above code into the following:

```
for (i = 0; i < n; i++) {
   out(ts, [i,"P"]);
   }
```

and create *"some optimal number of the following process, dependent upon the number of processors available on the machine"*:

```
int n, m;
for (;;) {
   in (ts, [n, "P"]);
   out (ts, [n, P(n)]);
   }
```

That is, we place a number of tuples into tuple space, each representing *one* `eval`, and then create some processes to repeatedly `in` one of these `eval` tuples, peform the computation, and `out` the result. It is clear that it is possible that some processes could be performed sequentially and that if there were dependencies between the `eval`ed processes, deadlock might occur. For example, consider the following outline of a C program:

```
void worker (tuple_space ts, tt)
   {
     do_work_on (ts, tt);
   }
```

[8] The assumption behind this is that processors are cheap.

The `worker` function takes two tuple spaces. It might, for example, use tuples in `ts` to produce results in `tt`.

```
void main()
   {
      int n;
      tuple_space ts_arr[];

      do_some_work (&n, ts_arr);

      for (i = 0; i < (n-1); i++) {
         eval(worker(ts_arr[i], ts_arr[i+1]);
         }

      rest_of_main (n, ts_arr);
   }
```

The main function declares an array of tuple spaces, and an integer, which are set during in the `do_some_work` function. The loop then `evals` `n-1` processes, which act as a pipeline, using the tuple space array as its communication medium. If the `eval` transformation, described above, is used, deadlock will occur if the *"optimal number"* of processors started happens to be less then than `n-1`.

5.2. The approaches of Faasen and Feng et al.

These two papers both choose methods of tuple placement based on *uniform distribution*. Faasen chooses *intermediate uniform distribution*, where the processor topology is a fixed grid of processors. Feng et al. do not discuss topology, but use a similar approach. In common to both approaches is the use of *in* and *out* sets of processors. Each processor P is assigned a set of processors, P_{out} and P_{in}, where $P_{out} \cap P_{in} \neq \phi$. When an `out` operation occurs, the tuple is duplicated on each of the P_{out} processors. When an `in` is requested, a search is performed on each of the P_{in} processors for a matching tuple.

Faasen appears to suffer from problems due to arbitration between `in` operations (because of duplication of the tuple space, and the need to keep consistency). The approach he describes is the basis of a *Linda machine*, where such arbitration is performed in hardware.

The approach taken in this paper is to define tuple space manager sets at the tuple level. For each tuple/template, `t`, we define $T_{in}^{\mathtt{t}}$ and $T_{out}^{\mathtt{t}}$, where the cardinality of $T_{out}^{\mathtt{t}}$ is always 1, and the cardinality of $T_{in}^{\mathtt{t}}$ varies according to whether `t` is completely presented. These sets are determined at runtime, using the hashing algorithms.

6. A case study: parallel addition of a bag of numbers.

The following program performs the parallel addition of a bag of integers[9]. The function `AddTuples` has parameters for the number of integers contained in the bag, and the tuple space containing these integers. It spawns `(num+1)/2` threads, to evaluate the `worker` function. Each worker is passed two extra arguments, which are used to calculate how much work it must perform before it dies voluntarily.

[9]Of course, the program can be modified to perform any commutative binary operation.

```
AddTuples := func (int num, ts addt);

   local n;

   for n := 0 to ((num+1)/2)-1 do
     eval(addt, [worker(addt, n, num)]);
   end for;

   for n := 1 to (num+1)/2 do
     in(addt, ["WORKER_TERMINATED"]);
   end for;

   in(addt, [?n]);
   return n;
end func;
```

The function `worker` (see below) repeatedly loops, pulling two numbers from the tuple space, adding them, and placing their sum back in the tuple space. However, once the addition of each pair of numbers in the tuple space has taken place, we are left with half as many numbers to add, and too many processes to perform the addition. If these processes were left to compute, they might each grab one number from the tuple space, and block waiting for another which is not present.

We require, then, that half of the remaining processes die. The second parameter to the `worker` function is for this purpose. However, this task is not as simple as it first appears. The calculation following the addition (performed using only conditionals and bit manipulation for speed) establishes whether the thread lives or dies.

This *addition and die* loop continues until only one number remains in the tuple space. The addition is completed once each thread has placed a `["WORKER_TERMINATED"]` tuple into the tuple space.

```
worker := func(ts local, int position, int tuples);

    local work, v1, v2;

    work := 0;
    while (work = 0) do
       if ((position <> 0) or ((tuples and 1) <> 1)) then
          in(local, |[?v1]|);
          in(local, |[?v2]|);
          out(local, [v1 + v2]);
       end if;

       if ((position and 1) = 1) then
          work := work + 1;
       end if;

       tuples := (tuples + 1) / 2;
       if (tuples = 1) then
          work := work + 1;
```

```
      end if;
      position := position / 2;
   end while;

   return "WORKER_TERMINATED";
end worker;
```

7. The Future

The goal of our research is to implement two *very high level languages* which contain primitives for parallel processing. The two languages, ISETL and GOFER, both have interactive environments.

7.1. ISETL-LINDA

ISETL [3] is an interpreted imperative language whose main data structure is finite sets. Tuple spaces can be viewed as bags, which are similar to sets in many respects, except that items may occur many times in a bag. we are integrating the tuple space operations into ISETL by adding new types for bags and templates, and implementing operations on them which behave like the Linda primitives.

7.2. GOFER-LINDA

GOFER [11] is an interpreted functional programming language. We propose the extension of this language with operations to act on tuple spaces. It has been noted elsewhere that bags are a better basic data structure for parallelism than lists[12], and we hope to be successful in adding the Linda primitives toGOFER, for fast functional programming.

However, parallelism is essentially non-deterministic and stateful (especially where shared memories are concerned). We hope to use techniques already investigated to add Linda operations to GOFER in a way which preserves the properties given by functional languages.

7.3. Current progress

A prototype Linda implementation and ISETL interpreter have already been developed. Ultimately, both languages are being investigated for expressing image processing algorithms. Some of this work has already been undertaken in relation to ISETL-LINDA .

Acknowledgements

This work is partially funded by EPSRC Research Grant GR/J12765. Antony Rowstron is supported by an EPSRC CASE Grant with British Aerospace Military Aircraft Division.

References

[1] 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, August 1988.

[2] Brian G. Anderson and Dennis Shasha. Persistent Linda: Linda + Transactions + Query Processing. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel*

Programming Languages, volume 574 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.

[3] Nancy Baxter, Ed Dubinsky, and Gary Levin. *Learning discrete mathematics with* ISETL. Springer Verlag, 1989.

[4] Paul Butcher, Alan Wood, and Martin Atkins. Global synchronisation in Linda. *Concurrency: Practice and Experience*, 6(6):505 – 516, 1994.

[5] N. Carriero. *Implementing Tuple Space Machines*. PhD thesis, Department of Computer Science, 1987. Available as technical report number YALEU/DCS/RR-567.

[6] Nicholas J. Carriero, David Gelernter, Timothy G. Mattson, and Andrew H. Sherman. The Linda alternative to message-passing systems. *Parallel Computing*, 20:633–655, 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] M. D. Feng, Y. Q. Gao, and C. K. Yeun. Distributed Linda tuplespace algorithms and implementations. To appear in CONPAR'94, 1994.

[9] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages*, 7(1), January 1985.

[10] David Gelernter. Multiple tuple spaces in Linda. In E. Odijk, M. Rem, and J.-C. Syre, editors, *Parallel Architevtures and Langauges Europe*, volume 366 of *Lecture Notes in Computer Science*, pages 20–27. Springer Verlag, 1989.

[11] Mark P. Jones. An introduction to GOFER. 1993.

[12] G. Marino and G. Succi. Data structures for parallel execution of functional languages. In E. Odijk, M. Rem, and J. c. Syre, editors, *Parallel Architectures and Languages Europe, Vol. II*, number 366 in Lecture Notes in Computer Science. Springer Verlag, 1989.

[13] James E. Narem Jr. An informal operations semantics of C-Linda V2.3.5. Technical Report YALEU/DCS/RR-839, Yale University, 1989.

[14] Scientific Computing Associates Incorporated. Linda: An introduction and example. Available from Scientific Computing Associates Incorporated, One Century Tower, New Haven, CT 06510-7010, U.S.A, 1994.

[15] Dennis G. Severance. Identifier search mechanisms: A survey and generalized model. *ACM Computing Surveys*, 6(3):175–194, September 1974.

[16] Steven Ericsson Zenith. Linda co-ordination language; subsystem kernel architecture (on transputers). Technical Report YALEU/DCS/RR-794, Yale University, 1990.