Contents

Chapter 1

An Efficient Tuple Space Programming Environment

James B. Fenwick, Jr.[†] and Lori L. Pollock[‡]

[†]Department of Computer Science
Appalachian State University
Boone, North Carolina, USA

[‡]Department of Computer & Information Sciences
University of Delaware
Newark, Delaware, USA

Email: *jbf@cs.appstate.edu, pollock@cis.udel.edu*

1.1 Introduction

Writing efficient message passing programs can be difficult, error-prone, and tedious. An alternative paradigm of parallel programming in a cluster environment is distributed shared memory, in which a shared memory abstraction is presented to the programmer, despite the physically distributed memory. The shared memory abstraction provides an easier transition from a sequential to a correct parallel program. While a shared memory, parallelizing compiler provides an application programmer with an easy avenue to parallel computing, programming languages allowing the user to explicitly specify parallel constructs are prevalent. Sometimes it is difficult to express a parallel algorithm using a sequential language, even with annotations. Similarly, it may sometimes be easier to use parallel constructs and synchronization than to provide the compiler with sufficient annotations about data access patterns.

Tuple space is a structured distributed shared memory paradigm, as it offers programmers a shared space of structures as opposed to a linear array of bytes, and each structure is an individual shared unit. Explicitly created processes share a data space rather than sharing variables. Messages are not sent between processes,

but are instead placed in the shared data space for other processes to access. To reinforce this differentiation, messages in this paradigm are called tuples; hence, the shared data space holding these tuples is called tuple space[10].

The tuple space paradigm, or more succinctly just tuple space, provides parallel programmers with an abstraction that hides the specific underlying mechanisms implementing process creation, communication, and synchronization. The actual implementation of the parallel program on the target architecture is hidden from the programmer, and the architecture can be any number of platforms ranging from shared or distributed memory to a cluster of workstations. In short, tuple space offers the simplicity of shared memory programming and the benefits of distributed memory architectures without the false sharing and memory consistency concerns of unstructured distributed shared memory systems.

Unfortunately, any abstraction of this kind necessarily introduces a trade-off for the application programmer between ease of use and control over performance. Indeed, implementation of the tuple space paradigm on a distributed memory architecture has raised concerns regarding efficiency and performance. However, several researchers have demonstrated that distributed tuple space implementations can be efficient [14]. These experimental studies considered a wide variety of real applications that encompassed a large scope of parallel algorithm classifications.

This chapter presents an overview of tuple space programming, and techniques for effectively and efficiently compiling and executing tuple space parallel programs in a cluster environment. After describing the basics of translation necessary for correctness of tuple space programs in a distributed memory environment, techniques for optimizing compilation are described. Issues involved in implementing the shared data space abstraction of tuple space in a cluster environment are detailed, along with an overview of Deli, a UNIX-based distributed tuple space implementation.

1.2 Tuple Space Programming

Tuple space is central to the generative communication parallel programming model most notably embodied by Linda.[1] Linda is a coordination language consisting of a small number of primitive operations that are added to a base computation language. Example base languages include C, Fortran, and Lisp. The result is an explicitly parallel programming dialect of the base language (for example, C-Linda). The operations manipulate the fundamental objects of Linda, tuples and tuple space, to perform the communication and synchronization necessary for parallel programming.

[1] Linda is a registered trademark of Scientific Computing Associates, Inc., New Haven, Connecticut.

1.2.1 Fundamentals

Tuples and tuple space

A tuple is an ordered collection of typed fields, where the type of each field is dependent upon the underlying computation language and may be subject to restriction. Each field of a tuple is either an actual, which contains a data value, or a formal, which receives a data value. A field is distinguished as formal through the use of a syntactic element; for example, C-Linda uses the '?' character to indicate a formal field. Thus, ⟨`?i`⟩ is a single field tuple with the field being designated as a formal. The field variable `i` will receive a value from some tuple space process.

Tuple space is a shared memory because any process can reference any tuple regardless of where it is stored. Tuple space is a logically shared memory because it provides the appearance of a shared memory but does not require an underlying physical shared memory. Tuple space is also an associative memory, which means that tuples are accessed not by their address but rather by their content. The identification of tuples via an associative search is described in more detail in Section 1.2.3. The tuple space communication model is termed generative because a tuple generated by a process has an independent existence in tuple space. Any process may remove the tuple, and the tuple is bound to no process.

Gelernter noted two properties that distinguish tuple space from other parallel paradigms: space and time uncoupling [10]. Space uncoupling refers to the fact that tuple producers are unaware of where tuple consumers exist in the parallel machine. The reverse is also true; tuple consumers do not know from where tuples are generated. Time uncoupling means that tuple space processes do not have to coexist in order to communicate. This is possible because tuple space has an existence outside of any individual process.

Operations

Linda defines six operations that manipulate tuples and tuple space. These operations divide naturally into three classes: operations that generate, extract, and examine tuples.

Generation operations (OUT and EVAL)

The OUT operation inserts a passive tuple into tuple space. Each field of the operation is evaluated by the process issuing the operation, and the resulting values are collected and deposited into tuple space.

Parallelism is explicitly specified by using the EVAL operation to insert an active tuple into tuple space. New threads of control are created to evaluate each of the fields of an active tuple. However, in response to the high cost of process creation, most distributed tuple space implementations limit new threads of control to only the function-valued fields of an active tuple. For example, upon receiving the active tuple ⟨`5,x+y,foo(),x+foo()`⟩, a typical distributed tuple space creates only one new process to evaluate the third field. The fourth field contains a function call,

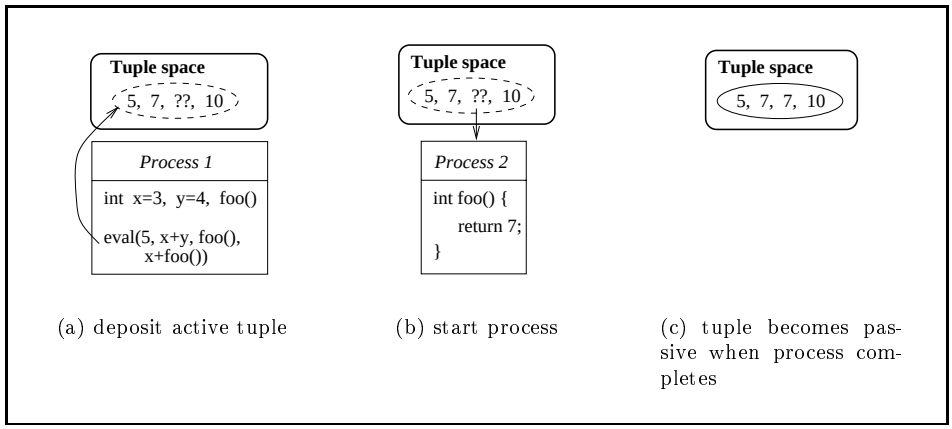


Figure 1.1 Snapshots of EVAL operation in action.

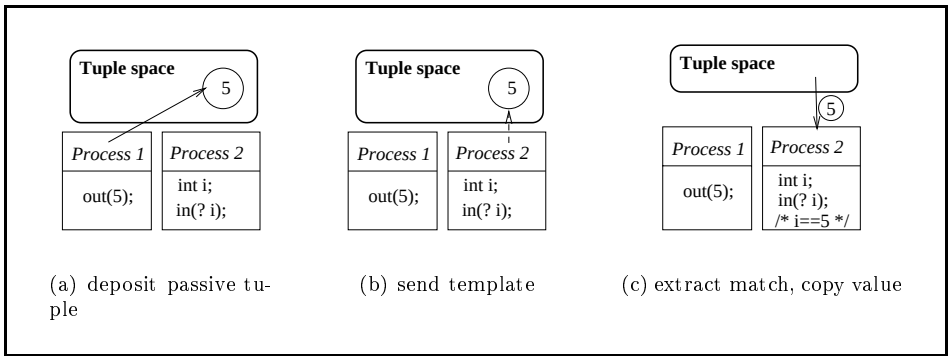
but the field is determined to not be function-valued since the call is a part of an expression; therefore, a new process is not created to evaluate the fourth field. An active tuple quiesces into a passive one upon completion of the evaluation of each of its fields. Figure 1.1 diagrams a possible behavior of an EVAL operation. The active tuple, indicated by the dashed ellipse, is not available for matching via an extraction or examination operation. The resultant passive tuple of Figure 1.1(c) is available for matching.

Both the OUT and EVAL are non-blocking asynchronous operations, meaning that these generation operations immediately return control back to their issuing process. Specifically, a tuple generated by a process does not have to be consumed before the process can continue.

Extraction operations (IN and INP)

The IN operation extracts a tuple from tuple space and copies the values of the tuple's actual fields into corresponding formal fields of the IN operation. Technically, an extraction operation generates a description of the tuple it wants to extract. This description of the desired tuple is called a template (or an anti-tuple). Data is sent from one process to another by having the sender issue an OUT operation while the receiver issues an IN operation that has a formal field of the same type in the same position. Figure 1.2 shows a possible behavior of two processes issuing IN and OUT operations.

The IN is a blocking, synchronous operation. Unlike the generation operations, the process issuing an IN operation does not resume until it receives an extracted tuple. The INP operation behaves like an IN except rather than blocking when no tuple is currently available, a boolean false value is returned, indicating that no tuple was removed and no copying into formal fields was performed.



(a) deposit passive tuple (b) send template (c) extract match, copy value

Figure 1.2 Snapshots of IN and OUT operations in action.

Examination operations (RD and RDP)

The RD operation is also a blocking and synchronous operation like the IN. The only difference is that a tuple is not removed from tuple space in response to a RD operation. However, data values are copied into formal fields of the RD. The RDP operation is a predicate form of the RD.

1.2.2 Example Linda Program

Figure 1.3 shows a Linda program to perform matrix multiplication. Each element of the result data structure is computed by a separate process. This example intends to demonstrate the tuple space operations on a well-known problem, but is not a particularly efficient solution, especially on a cluster. The emphmain() procedure creates N^2 active tuples. Each process executes the *worker()* function but with different input arguments, which tell the process the element of the result matrix to compute. The inner product computation requires a specific row and column from the input matrices. Thus, the *main()* procedure deposits each row of **A** as a tuple and each column of **B** as a tuple after a transposition of **B** to accommodate the row-major ordering. Since each row and column is used in many computations, the worker processes read their required data, then return the inner product. This return value is placed into the active tuple, making it passive. The *main()* process extracts the resultant passive tuples, which may come in any order, thus requiring the row and column identification in this tuple.

1.2.3 Associative Memory Analysis

An important characteristic of the Linda generative communication model is the associativity of tuple space, which necessitates the comparison of tuples and templates. Simplistic implementation approaches could, in the worst case, require a number of comparisons equal to the number of tuples generated at run-time. Carriero assuaged early Linda critics of this fear by developing compiler analysis to

```
main()  {
  int A[N][N], B[N][N], C[N][N];
  int r,c,e,i,worker(int,int);

  for (r=0; r < N; r++)
    for (c=0; c < N; c++)
      eval("C", r, c, worker(r,c));

  init(A, B);
  transpose(B);
  for (i=0; i < N; i++) {
    out("row of A", i, A[i]);
    out("col of B", i, B[i]); }

  for (i=0; i < N*N; i++) {
    in("C", ?r, ?c, ?e);
    C[r][c] = e; }
}

int worker(int myrow, int mycol)  {
  int i, value=0, row[N], col[N];

  rd("row of A", myrow, ?row);
  rd("col of B", mycol, ?col);
  for (i=0; i < N; i++)
    value += row[i]*col[i];

  return value;
}
```

Figure 1.3 Example Linda program performing matrix multiplication.

partition a program's tuple space operations into disjoint sets, thus limiting the associative search to a single partition rather than all of tuple space, and significantly decreasing the impact of the associative memory [5].

There are two cases in which tuple space must attempt to find a match between a tuple and a template. In the case where a tuple has been generated by an OUT or EVAL operation, the tuple space system must decide if this newly arriving tuple into tuple space has already been requested by a user process; that is, a process is blocked on an IN or RD operation awaiting this tuple. If no request for this tuple has been made, the tuple is stored. In the case where a template has been generated by an IN or RD or INP or RDP operation, the tuple space system must decide if a stored tuple matches this request. In both cases, the tuple space system must answer the question, *Does tuple X match the tuple description of template Y?*, for some tuple X and template Y. A tuple and a template are said to match if the tuple and template (1) agree on the number of fields, (2) agree on the types of corresponding fields, (3) agree on the value of corresponding actual fields, and (4) have no corresponding formal fields.

1.3 Compilation Environment

1.3.1 Basic Translation

Because Linda extends a base computation language, a Linda compiler is a source-to-source translator. A basic Linda compiler for C, for example, accepts a C-Linda program as input, performs tuple space partitioning to increase the efficiency of the

associative search, maps the tuple space operations to a Linda run-time library, and outputs a C program. This C program is then compiled with a native C compiler to generate an executable image, which is then executed in a distributed Linda run-time environment.

1.3.2 Optimizing Compilers

Several methods for increasing the efficiency of distributed tuple space programs have been developed. Some approaches involve the run-time system dynamically managing its resources and gathering statistical information that triggers alternative processing[13], [3]. Other approaches to increasing tuple space efficiency involve collaboration between the compiler and run-time system [12], [3]; however, the analyses and transformations are very localized. Carriero and Gelernter outline several ideas for achieving increased performance through more global compiler analysis [4]. They describe commonly used patterns of tuple space operations which can be improved, and speculate about information that an optimizing compiler would need to perform analysis.

Fenwick and Pollock [7], [9] have realized some of these visions by designing and building the Deli optimizing C-Linda compiler. The optimizing compiler performs global and interprocess data flow analyses and uses the results from these analyses to identify the safety and profitability of several compile-time transformations for improving run-time tuple space communication. This section describes the intermediate program representation, data flow analysis framework, and several of the optimizing analyses and transformations in the Deli optimizing C-Linda compiler.

Program Representation

All of the information available in tuple space operations is needed in order to perform compile-time analysis to identify opportunities for tuple space communication optimizations. Unfortunately, most tuple space programming systems lose this high level information in much the same way as high level array access information is lost in low level intermediate representations. This information is lost because tuple space operations are typically mapped to calls to a message passing library. To retain the necessary high level information in the Deli optimizing C-Linda compiler, compile-time analysis is performed on a high level intermediate program representation. Standard data flow analyses remain feasible, while tuple space communication analysis is also effectively supported.

In particular, the high level representation developed for the Stanford University Intermediate Format (SUIF) shared memory parallelizing compiler [17] has been extended to support the additional tuple space operation syntax and to include information about tuple space operations. Each tuple space operation outwardly appears like a procedure call, but is further annotated with high level information, including its tuple space partition, an indication of whether each field is formal or actual, and other useful information. Processes are identified by examining the EVAL tuple space operations, which contain the name of the procedure that will

be executed in parallel. The definitions of the named procedures are annotated to indicate that they are process entry points. Tuple space communications are modeled by constructing directed edges from tuple space generation operations to extraction/examination operations.

The entire program is a forest of process intermediate representations, where each process is a collection of procedures. The current version of the Deli optimizing C-Linda compiler builds an interprocedural flow graph similar to [6] for each process. Each procedure is represented in the form of the SUIF intermediate representation, and it is not necessary to have all procedure representations in memory at once. Interprocedural execution paths are not explicitly represented by edges in the intermediate representation, and similarly there are no edges from process invocation sites to process entry points. The communication edges connecting these process representations create the Linda intermediate representation.

Figure 1.4 depicts an example representation of a program with two processes

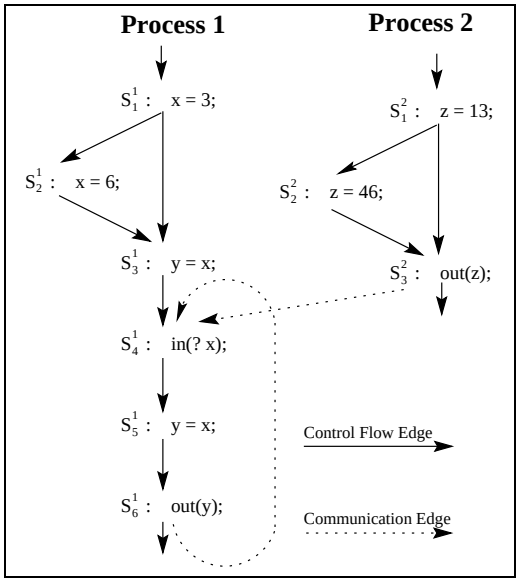


Figure 1.4 Example program representation.

that are connected by tuple space edges. In this example, there is only one tuple space partition, so each OUT is connected to every IN operation.

Data Flow Analysis

Data flow analysis plays a key role in ambitious optimizing compilers for sequential programming languages, but current compilers for parallel programs with user-specified parallelism restrict the scope of this analysis due to the complex issues

involved in analyzing shared memory programs. In contrast, tuple space parallel programs are quite amenable to data flow analysis.

Intraprocess Data Flow Analysis

Tuple space processes share access to a logically global data space, but address spaces of the tuple space processes are distinct. Intuitively, a memory location in one process can not be accessed by another process. Therefore, only the actions of process i itself need to be examined in order to analyze how definitions and uses of variables accessible to process i flow within process i. While the concept of "last value written" is not well defined in other shared memory systems, it can be conservatively determined for the local variables of tuple space processes by analyzing only the process in which the variable is declared. As such, it is not difficult to understand that standard data flow analysis *within a process* remains feasible in the context of tuple space parallel programming, unlike other shared memory parallel programming systems [15].

The precision of the standard data flow analyses for tuple space programs can be improved by maintaining and using the high level information of tuple space operations. In particular, a straightforward modification of the computation of GEN and KILL examines the statement to determine whether it is a tuple space operation. If so, then formal fields become *un*ambiguous definitions of the associated variable, and actual fields are not considered to be definitions at all.

Interprocess Reaching Definitions

In a sequential program, a definition of a variable is said to reach a program point p if there is a definition-free path in the control flow graph from that definition to p. In addition to classic optimizations for individual processes, interprocess reaching definitions information is useful for improving the tuple space partitioning and tuple space communication operations[9].

The data flow system for computing reaching definitions across tuple space process boundaries extends the sequential reaching definitions data flow analysis. The GEN and KILL data flow sets are computed as described in the intraprocess data flow analysis. The IN set is unchanged and represents the definitions reaching a point p as the union of the definitions leaving all its control flow predecessors. The OUT data flow set is the most significantly changed. In addition to definitions generated and those coming in that are not killed, definitions coming in from tuple space communication edges are included. For example, in addition to the definition of y generated at S_3^1, the OUT set for statement S_4^1 of Figure 1.4 would also include the definitions at statements S_5^1, S_1^2, S_2^2 that are present due to the two communication edges. A slight technical consideration is necessary to accommodate mapping a definition of a variable in one process context to another process context. For example, the definition of z at S_1^2 becomes a definition of x at S_4^1. This accommodation is also required in interprocedural reaching definitions.

In [7], the interprocess reaching definitions problem is shown to be monotone.

To compute the reaching definitions using these equations, the standard $\mathcal{O}(N^2)$ algorithm is run to iterate over the program nodes, N, until a fixed point is reached.

Tuple Space Edge Elimination

More precise interprocess data flow information is possible if the number of tuple space edges connecting tuple space generation operations to extraction/examination operations can be reduced. This reduction can be achieved by propagating control flow information. In the example in Figure 1.4, the edge connecting the OUT operation in S_6^1 to the IN operation in S_4^1 can be eliminated because there is no control flow path from S_6^1 to S_4^1. That is, it is not possible for a tuple generated by the OUT in S_6^1 to be consumed by the IN in S_4^1.

In [9], Fenwick and Pollock present an algorithm for eliminating tuple space edges based on computing and using NREACH information, where NREACH(n) is the set of extraction/examination operations that node n cannot reach. A tuple space edge from x to y can be safely eliminated if $y \in$ NREACH(x).

Analysis and Transformation

Carriero and Gelernter [4] describe several classes of tuple usage that are commonly found in tuple space programs and can be transformed into more efficient communications. These tuple usage classes include: shared variable tuples, distributed queues, and message tuples. Being able to distinguish a tuple as belonging to one of these tuple usage classes enables detection of the safety and profitability of communication improving transformations. These tuple usage patterns, a method for automatic identification of the pattern, and transformation for increased performance are described in this section.

Shared Variable Tuples

Shared data is common in tuple space parallel programs. In a shared memory context, a shared location contains only a single value, and processes must synchronize among themselves to ensure exclusive access to the shared location. In a tuple space context, a partition that never contains more than a single tuple is said to contain a shared variable tuple. Removal of a shared variable tuple by a process implicitly synchronizes access to the tuple. Inserting the shared variable tuple into the empty partition makes the value available for other processes. Figure 1.5 depicts shared variable tuple space operations.

```
IN("shared variable", ?value);
newvalue = compute(value);
OUT("shared variable", newvalue);
```

Figure 1.5 Shared variable tuple space operations.

Finding a tuple space partition that never contains more than one tuple requires determining the number of tuples, the tuple count, that may be present in the partition during program execution. In [7], Fenwick presents a data flow analysis framework that answers the question: For each tuple space partition, may there ever be more than one tuple in that partition? This data flow analysis is the crucial component in the analysis to automatically identify shared variable tuples.

The in/out collapse optimization [3] can be applied to many shared variable tuples. This transformation collapses an IN and a subsequent OUT into a single operation. This reduces underlying communication and the time spent allocating storage for the tuple. In addition, the basic synchronization tuple is a specialization of the shared variable tuple, and can be optimized by identifying the synchronization and replacing it with a more efficient method native to the host architecture.

Distributed Queues in Tuple Space

In a comparative study of several parallel programming languages, Bal observed that the concept of distributed data structures is an important contribution of the tuple space communication model [2]. Distributing a queue essentially involves distributing the individual data elements across the memories of the cluster. A distributed queue used by a group of processes requires additional shared variable tuples to coordinate access to the front and rear of the queue. Thus, a distributed queue in tuple space is a multipartition data structure.

Figure 1.6 shows operations used to remove an item from a queue. The "front" shared variable tuple coordinates removing data from the distributed queue among multiple processes, but at the cost of efficiency. As Figure 1.6(a) shows, five network accesses may be necessary. Figure 1.6(b) illustrates an improvement to this default tuple space handling of distributed queues, which Wilson terms triangular messaging [18]. Triangular messaging does not eliminate any of the required messages, but rather changes when and who initiates messages. At best, the triangular messaging scheme results in the user process experiencing no delay at the second IN operation because the tuple space manager has already sent the tuple.

Wilson did not specify how or when the improvement can be applied safely. In [7], Fenwick presents a compiler analysis to detect and transform the set of tuple space operations acting on a distributed queue. The analysis links the value of the shared variable tuple to the IN operation that uses this value as a position in the queue. After detecting distributed queue operations, the compiler transforms the program so that triangular messaging is performed during program execution. This involves augmenting the shared variable tuple request with information so that the tuple space run-time system can send the template for the queue data item, and ensuring that the user process does not send this template. In addition, triangular messaging requires that the run-time system support the sending of a template on behalf of a user process.

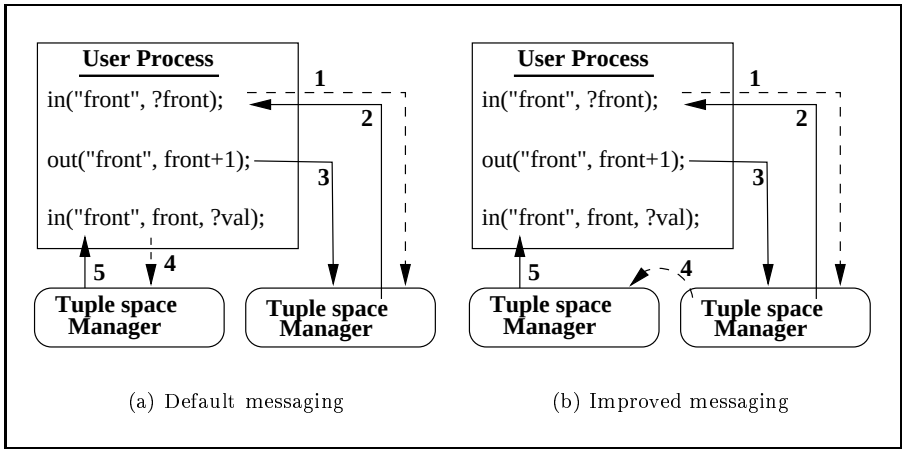


Figure 1.6 Messages required to access a distributed queue.

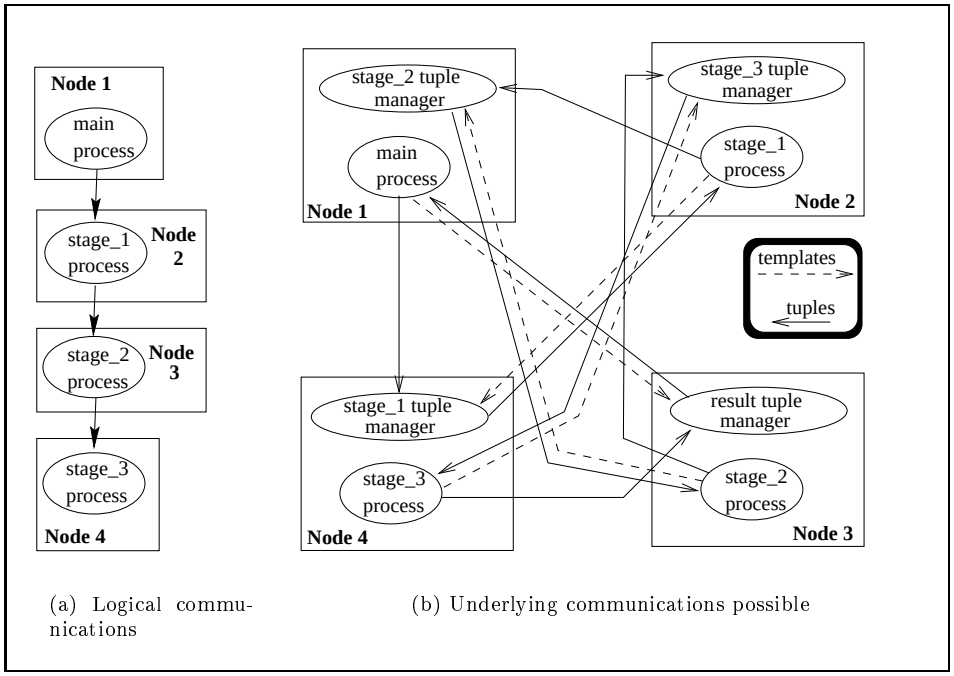


Figure 1.7 Communications of using a tuple space message tuple.

Tuples as Messages

A tuple space process needing to communicate data to a specific process may suffer an unwanted latency. Figure 1.7(a) shows a logical communication of a pipelined calculation that sends a message from *main* to *stage_1*, which then sends a message to *stage_2*, etc. However, the uncoupling property of tuple space described in Section 1.2 means that the messages will probably be stored at cluster nodes that are not executing the receiving process. This scenario is depicted in Figure 1.7(b), illustrating the source of the unwanted latency in sending a directed communication through tuple space.

In [7], Fenwick presents an analysis that is able to detect some tuples that are received by a single process invocation, and thus are messages. The identification of a message tuple requires the identification of a tuple that is intended to be received by a single, specific target process. In the intermediate representation of a tuple space program, this is characterized by a generation operation that has all of its communication successor nodes residing in the same process. Additionally, this process must be invoked only once.

The compiler transforms the program to invoke a dynamic transformation of the tuple space run-time system ensuring that the message tuple will be stored at the cluster node executing the process that receives the message tuple. The Deli tuple space implementation allows the process receiving the message tuple to run anywhere in the cluster. Once a node has been selected to execute this process, the run-time system reassigns the original node for the message tuple to this node, and lazily informs the other nodes of the reassignment.

Footprinting

In [12], Landry and Arthur describe the tuple space operations footprinting optimization that divides each IN and RD operation into two suboperations that send the template and receive the tuple. Efficiency is improved by allowing the movement of noninterfering computation between the suboperations, thus, overlapping computation with communication.

1.4 Run-time Environment

The run-time environment of a tuple space system is composed of several subsystems. The cluster execution environment is the subsystem enabling the compiled tuple space application to execute in a cluster environment. This environment must unite distinct cluster nodes into a logically parallel machine. The tuple space data subsystem implements the storing, searching, and matching of run-time tuples and templates. This subsystem must determine which node of the cluster has the requested data and efficiently access this data on that node. The process execution subsystem is responsible for concurrently executing the user-specified processes. This subsystem must find a node in the cluster to execute a user process. The interface subsystem is a run-time library that connects the application to the other

subsystems. To maximize the opportunities for compiler optimization, these subsystems are bundled together with the application program.

Figure 1.8 illustrates the relationships between these subsytems. The application interfaces to the data subsystem through its uses of the OUT, IN, INP, RD, and RDP operations. The EVAL operation interfaces to the process execution and data subsystems. The application does not directly interface to the cluster execution environment; rather, the compiler imperceptibly inserts this code into the application. In Figure 1.8 the cluster execution environment is using cluster nodes 2–6 for the current execution of the application. Another run of the application may result in the cluster execution environment selecting a different subset of nodes. The major implementation issues for a distributed tuple space are described in this section.

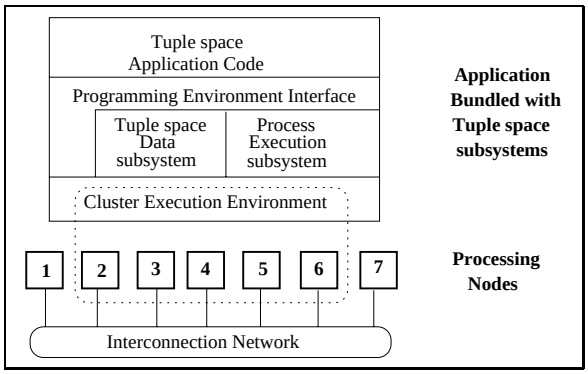


Figure 1.8 Overview of tuple space subsystem relationships.

1.4.1 Processor Location of Data

Supporting the sharing of data necessitates that processes be able to locate and retrieve the data they require. In a cluster environment, this means that processes must determine the node (processor) that is holding the data. In the tuple space programming paradigm, processes may need to determine the node that will hold the data in the event that the data has not yet been produced. There are two primary approaches to resolving this question: centralizing the data in the cluster, or distributing it in a controlled fashion.

Centralized Data

A logical choice for implementing tuple space is to use a client/server approach. Using this strategy, clients are the concurrent processes executing on nodes throughout the cluster, and the server manages all the tuples and templates. Several tuple space implementations follow this approach of centralizing all the tuple space data on a single server node. Consequently, a client process can easily locate the node holding

the data it requires; it is the preselected server node. However, centralizing the data serializes data access and may become a communication bottleneck.

Distributed Data

Distributing the tasks of tuple storage and management over some subset, possibly all, of the nodes of the system increases parallelism as data requests can be concurrently handled by different nodes. However, locating a tuple becomes more difficult. There are two primary approaches to distributing tuples and templates throughout the nodes of the cluster: hash-based and operation-based tuple distribution.

A hash-based distribution method deterministically maps tuples and templates to a specific node using a hash function, which requires every tuple and template to contain an input to the hash function. Some tuple space implementations require a dedicated key field in all tuples and templates [5]. Others use a combination of the tuple space partition information and the values of actual fields [3]. A thorough investigation of the effectiveness of hash functions of this sort was performed by Bjornson [3]. Hashing is one of the best methods for efficient tuple distribution, and it does not assume any particular communication topology (i.e., broadcast/multicast).

An operation-based tuple distribution method uses the tuple space operation itself to determine the processor responsible for the tuple or template. One possibility is for tuples to be stored locally on the node executing the OUT or EVALoperation. [2] Then, templates, which are generated by the IN, RD, INP, and RDP operations, are sent to all nodes. Bjornson termed this scheme *negative broadcast*, and it is used in Leichter's implementation [13]. A second operation-based possibility, called *positive broadcast*, is the inverse of negative broadcast. In this case, it is the tuples that are broadcast to all nodes, and templates are sent locally [5]. A hybrid of these two methods multicasts tuples and templates to a subset of nodes. The intersection of these subsets must not be null; otherwise, templates can not find matching tuples. Restricting the intersection to a single node simplifies matching of tuples and templates. Methods using broadcast or multicast require a coherence protocol, which is discussed later.

Whichever tuple distribution method is selected, it should evenly distribute the load of managing the tuples since this requires CPU time and memory. The tuple distribution method should also strive to have tuples found locally, thus reducing network access and tuple space operation latency. Unfortunately, achieving one of these goals often conflicts with the achievement of the other goal.

It is uncertain which technique is superior, so attempts to safely hybrid these methods are justified. One such technique is sophisticated compiler analysis to estimate the tuple space access patterns of processes[7]; another has the tuple space implementation itself gather run-time statistics[3]. Both techniques attempt to circumvent the underlying distribution scheme when beneficial.

[2] Tuple in this context refers to the resultant passive tuple generated by an EVAL and not any tuples created by the processes spawned by the EVAL.

1.4.2 Data Structures for Efficient Data Access

After the node managing a desired tuple is identified, the tuple must be retrieved from that processor's memory. There are several efficient data structure paradigms for tuple storage (hence, tuple access): trees, hash tables, queues, counters, and lists.

The tuple space implementor can use a single data structure for all of tuple space, or use different paradigms for different tuple space partitions. Both hash tables and trees require a key in each tuple (template) so that its location in the hash table (i.e., its location within the memory of this node) can be determined. The ordering properties of trees are an unnecessary overhead; hence, trees are not generally used. Recall the operations from Section 1.3.2 and Figure 1.6 that act on the data elements of the distributed queue. The second field in all those operations was always an actual; thus, this partition could use a hash table with this field serving as the key. If a hash table key consists of every actual field in the tuple, then access is $\mathcal{O}(1)$. However, most implementations use a subset of the tuple's actual fields for the key. In this case, different tuples may have the same hash value thus necessitating a search of the colliding tuples.

The worst case for representing tuple space is as a list. Every tuple in the list must be examined to determine that no match exists in the list. Partitioning tuple space is an improvement as it divides a single list into smaller sublists, only one of which needs to be searched, but the potential for expensive tuple access remains. Sophisticated compilers may perform analysis revealing that a tuple space partition never requires run-time matching; that is, any tuple may satisfy any template. For example, a partition containing the operations OUT`("data",value)` and IN`("data",?item)` meets these restrictions regardless of when, where, or how often the operations occur in the application. In this case, the list can be viewed as a queue, and the access becomes constant. Compiler analysis can also determine that not only is no run-time matching required, but also there is no data copying necessary. In this situation, no data needs to be stored, and a simple counter provides efficiency. Example operations qualifying as a counter are OUT`("lock")` and IN`("lock")`.

The Deli and Bjornson tuple space implementations represent tuple space as a table of partitions [7], [3]. Each partition can be any of the available data structures: a hash table, a queue, a counter, or a list. In the case of queues and lists, the partition data structure maintains two chains, one for unmatched tuples and another for unsatisfied templates. In the case of counters, the partition data structure maintains a single counter field for tuples and a chain for templates. If the partition data structure is a hash table, then each element of the hash table contains chains for tuples and templates.

Using appropriate data structures can allow a template to find a matching tuple efficiently, often in constant time. However, the converse may not be true. This is because an arriving template is satisfied by a single tuple, which can be found in constant time with efficient data structures. In contrast, an arriving tuple can satisfy

multiple templates. Consider tuple space containing several RD and IN templates when a matching tuple arrives. The tuple can satisfy at most one of the IN templates and a subset, possibly empty, of the RD templates. This requires special attention by the tuple space implementor.

1.4.3 Data Transfer Protocol

There are a number of protocols that a tuple space implementor must support. In tuple space as a whole, a coherence protocol is necessary for any replicated tuples or templates. This is similar to the problem of keeping caches coherent in a multiprocessor, and the solutions are likewise similar. The physical transmission of tuples and templates requires a low level transport protocol providing reliability, retransmissions, reassembly, etc. In response to requests for tuples, a protocol is needed to ensure that multiple matching tuples are properly managed.

This tuple transfer protocol controls the movement of tuples and templates among nodes of the machine. A template travels between nodes to find a matching tuple. A tuple travels between nodes to find a matching template and then onward to the node issuing the tuple. The tuple transfer protocol must that ensure the atomicity of the tuple space operations while simultaneously attempting to minimize unnecessary movement. The tuple distribution method and the cluster's communication architecture have effects on appropriate tuple transfer protocols. In [8], Fenwick and Pollock explore the issues faced in selecting a tuple transfer protocol. The tuple space implementor must also realize that the protocol may work well for some tuple access patterns, and poorly (even terribly) for others. Again, sophisticated compiler analysis for estimating tuple space access patterns and/or run-time statistics gathering of actual access patterns may be able to allow the selection of alternative protocols. Because each application has its own tuple space access patterns, Shekhar and Srikant suggest that the tuple space implementation itself should be reconfigurable for each application so as to minimize inefficiencies [16].

1.4.4 Process Creation

The EVAL operation is the tuple space programmer's vehicle for explicitly expressing parallelism through the creation of a new process. There are several issues requiring attention, including the semantic difficulties of the EVAL operation, deciding upon a node to execute a new process, and establishing communication between the new process and other nodes.

Eval Semantics

Unfortunately, the semantics of the EVAL operation are not well-specified. Specifically, the tuple space model does not define whether the new process shares global variables with its parent, or if a process can modify parameters passed by reference [16], [18]. In the absence of a clear semantic definition, implementations generally

seem to let the machine's available process creation primitives dictate the definition of the EVAL semantics. In a cluster environment, the most restrictive semantic interpretation is most feasible; that is, a new process only has access to the *r*-values of explicitly passed parameters.

Eval Implementation Alternatives

While several possibilities exist [8], a general machine-independent approach that is particularly well-suited for clusters is to use an *eval server* [3], [7]. An eval server runs on every node and monitors tuple space for active tuples created by an EVAL operation. In the Deli tuple space implementation, a single active tuple is actually a set of passive tuples created by the compiler when translating an EVAL operation. One of these tuples describes the overall EVAL operation and holds the known values of all the fields of the resultant tuple. The other tuples describe the processes created. For example, EVAL`(matmul(),eigenvals())` is an EVAL operation with an active tuple consisting of three compiler-created tuples; one for the EVAL operation, one to describe the *matmul()* process, and another to describe the *eigenvals()* process. A unique tag associates these tuples as belonging to the same set. The eval server requests tuples describing an overall EVAL operation. This gives it the tag for a specific EVAL operation. Then this tag is used to request one of the associated tuples that describes a process. In turn, this tuple identifies a specific process which is then executed by the eval server. The result of the process execution is placed back into the overall EVAL operation tuple, and the eval server begins anew.

1.4.5 Cluster Execution Environment

The execution environment is the tuple space subsystem that facilitates, transparently to the user, both the utilization of available cluster nodes and the preparation of these nodes for participation in the execution of the user's tuple space parallel program. The Deli tuple space implementation refers to these operations collectively as *bootstrapping* the execution environment [7]. When bootstrapping is complete, the execution environment is poised to commence execution of the user's tuple space parallel program. The bootstrapping process in Deli is described here as an example.

The user initiates the bootstrapping sequence by invoking Deli directly at one node, called the host node. The invocation of Deli specifies the tuple space executable program to run and the number of additional cluster nodes desired. A local interprocess communication channel is established for the data and process subsystems that will run on the host node. Next, an external communication port is obtained for communication with other cluster nodes. A communication subsequence then commences with other nodes in the cluster. This subsequence results in each node of the cluster that is participating in the execution obtaining a communication port that can be used by the other nodes. All participating nodes are informed of the communication ports of the other nodes. Determination of which other cluster

nodes to include in the execution environment can vary. A list of available nodes can be maintained either statically within the tuple space implementation or read from an external file. The nodes in this list can be selected sequentially or randomly. A nice feature allows a selected node to refuse participation if it is already busy.

After the appropriate number of nodes are included, each node is responsible for instantiating its data and process subsystems. These subsystems have access to the communication ports of all the nodes participating in the application execution. The process subsystem on the original host node begins execution of the user's *main()* routine. The process subsystem on all the other nodes acts as an eval server ready to execute an application-specified process.

1.4.6 Run-time Optimizations

In [3], Bjornson describes several run-time optimizations incorporated into his hash-based tuple space implementation. Inspection of tuple space programs showed that some shared data is only read with the RD operation; thus, these tuples can be safely replicated at each node. Bjornson's tuple space continually monitors itself at runtime, gathering statistics about application communications. If tuple space determines that the tuples of a partition are predominantly being sent to a particular node, then the task of managing these tuples will be dynamically reassigned to the node using the tuples. Lastly, in spite of distributing tuple space data throughout the cluster, it remains possible for a single node to be overwhelmed with data. In this case, Bjornson's tuple space further subdivides the data manager responsibilities at runtime. The application processes are informed lazily of the subdivision and may then direct data to an alternate node. Bjornson also describes the additional protocols necessary for this technique.

1.5 Extensions

The basic tuple space system described thus far can be extended in a number of interesting ways. Extensions accommodating heterogeneity, multiple tuple spaces, persistence, and fault tolerance are discussed here.

Clusters are increasingly being populated with nodes of different capabilities and architectures. Cluster-based systems such as tuple space should accommodate this heterogeneity. Allowing heterogeneous nodes to participate in the distributed, parallel execution of a single application presents several additional implementation issues including data formats (big/little endian, word size, etc.), and binary incompatibility of nodes (e.g., binary image for an Alpha will not execute on an Intel). A typical solution to the data format problem is the use of the XDR extended data representation. Binary incompatibility requires multiple versions of the application executable.

The model of tuple space described in this chapter uses a single tuple space equally accessible by any operation in any process of the application. Some researchers are exploring ways for an application to use multiple tuple spaces; thus,

treating a tuple space as a fundamental *object* of the model. Multiple tuple spaces allow for several interesting possibilities: the coordination of distinct applications; levels of security, or access permissions, for an individual tuple space; and new distributed data abstractions.

Another extension has the tuple space memory persist beyond the lifetime of an application, thus decoupling invocations of an application over time. Some communication optimizations may not be possible in a persistent tuple space setting since the tuple space data manager cannot be bundled with the application. There are several open issues including how an application connects to a persistent tuple space and how the tuple space is initially created.

Fault tolerant tuple space systems prevent the loss of entire computations through transactions and checkpoints [1], [11]. Failures can occur for a variety of reasons, both hardware- and software-related. Fault tolerance is especially difficult in a cluster environment since defining a consistent global state across multiple, asynchronous nodes is problematic. Supporting fault tolerance for tuple space is facilitated by the characteristics of tuple space itself. Only a few operations require extension for transactions, and the uncoupled communication and synchronization of tuple space simplifies the recreation of processes during recovery.

1.6 Conclusions

The associative tuple space access and uncoupled communication of tuple space parallel programs are the key to the power and flexibility of this model, but also lie at the heart of the compiler and run-time system implementation challenges, especially in a cluster environment. Compile-time analysis can structure tuple space to significantly reduce the time to find data in tuple space. Run-time strategies counteract some communication inefficiencies. However, gains in performance in a cluster environment can also be made through sophisticated optimizing compiler techniques and corresponding run-time system modifications. The viability of classical intraprocess data flow analyses of tuple space parallel programs has been demonstrated, and a technique for interprocess data flow analysis via analysis of tuple space operations has been developed. Analyses for identifying common tuple usage patterns at compile time and code transformations for improving the underlying message communication in a distributed memory architecture have led to good performance gains at runtime[7]. The static analysis can be performed on a traditionally-based program representation, which also enables standard, sequential analysis of the parallel programs.

1.7 Bibliography

[1] David E. Bakken and Richard D. Schlichting. Supporting Fault-tolerant Parallel Programming in Linda. *IEEE Transactions on Parallel and Distributed Systems*, vol 6(3), pages 287–302, March 1995.

[2] Henri E. Bal. A Comparative Study of Five Parallel Programming Languages. *Distributed Open Systems*, F. Brazier and D. Johansen (Eds.), pages 134–151, IEEE Computer Society Press, 1994.

[3] Robert D. Bjornson. Linda on Distributed Memory Multiprocessors. *PhD Thesis*. Yale University, November 1992.

[4] Nicholas Carriero and David Gelernter. A Foundation for Advanced Compile-time Analysis of Linda Programs. *Languages and Compilers for Parallel Computing*, pages 389–404, Springer-Verlag, 1992.

[5] Nicholas John Carriero, Jr. Implementation of Tuple Space Machines. *PhD Thesis*. Yale University, December 1987.

[6] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven Computation of Interprocedural Data Flow. *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*, pages 37–48, January 1995.

[7] James B. Fenwick, Jr. Compiler Analysis and Optimization of Tuple Space Programs for Distributed-memory Systems. *PhD Thesis*. University of Delaware, August 1998.

[8] James B. Fenwick, Jr. and Lori L. Pollock. Issues and Experiences in Implementing a Distributed Tuple Space. *Software–Practice and Experience*, vol 27(10), pages 1199–1232, October 1997.

[9] James B. Fenwick, Jr. and Lori L. Pollock. Data Flow Analysis Across Tuple Space Process Boundaries. *Proceedings of the International Conference on Computer Languages*, pages 272–281, May 1998.

[10] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, vol 7(1), pages 80–112, January 1985.

[11] Karpjoo Jeong and Dennis Shasha. PLinda 2.0: A Transaction/Checkpointing Approach to Fault-tolerant Linda. *Proceedings of the 13th IEEE Symposium on Reliable Distributed Systems*, pages 96–105, 1994.

[12] Kenneth Landry and James D. Arthur. Achieving Asynchronous Speedup While Preserving Synchronous Semantics: An Implementation of Instructional Footprinting in Linda. *The 1994 International Conference on Computer Languages*, pages 55–63, May 1994.

[13] Jerrold Sol Leichter. Shared Tuple Memories, Shared Memories, Buses and LAN's – Linda Implementations Across the Spectrum of Connectivity. *PhD Thesis*. Yale University, July 1989.

[14] Timothy G. Mattson. The Efficiency of Linda for General Purpose Scientific Programming. *Scientific Programming*, vol 3(1), pages 61–71, 1994.

[15] Samuel P. Midkiff and David A. Padua. Issues in the Optimization of Parallel Programs. *Proceedings of the International Conference on Parallel Programming*, vol II, pages 105–113, 1990.

[16] K.H. Shekhar and Y.N. Srikant. Linda Sub-system on Transputers. *Computer Languages*, vol 18(2), pages 125–136, 1993.

[17] Stanford SUIF Compiler Group. *The SUIF Parallelizing Compiler Guide Version 1.0.* Stanford University, 1994.

[18] Gregory V. Wilson. *Practical Parallel Programming.* The MIT Press, 1995.

Index