**The Linda Parallel Programming Paradigm**

Linda® is a programming framework of language-independent primitives enabling communication between, and synchronization of, parallel processes. Linda was first proposed by Prof. David Gelernter and Nick Carriero of Yale University in 1983.

Or, a single sentence definition from Nadkarni’s unpublished book:

“Linda is a machine-independent, language-independent model of parallel computing that adds a small number of keywords and concepts to an existing sequential language such as C or FORTRAN to give it parallel capability.”

The Linda primitives are typically injected into the syntax of existing, sequential programming languages.

To date, Linda primitives have successfully be added to compilers for C, Pascal, Modula II, Prolog, variants of LISP (Scheme), Standard ML, Postscript, FORTRAN and Java resulting in new parallel programming languages.

Most recently, Linda has “reappeared” as a class library in JavaSpaces.

**Linda’s Data Tuples**

Tuples are finite collections of strictly typed data objects.

Tuples have an *arity* which reflects the number of their data fields and a *type-signature* which reflects the ordered types of these fields.

For example, the tuple

\[(3, \text{true}, 4.3)\]

has an arity of 3 (three fields) and a type-signature of

\[\text{integer X Boolean X real}\]

The above tuple is *type-equivalent* to

\[(8, \text{false}, 0.0)\] but not \[(8.0, \text{false})\]

Every field of a tuple has an associated datatype; these data types are drawn from the host language (Linda does not, itself, define datatypes).

† This work has been supported by The University of Western Australia’s Special Research Grant 04/15/087/244.

® Linda is registered trademark of Scientific Computing Associates, Inc (SCA), New Haven, CT 06510-7010.

CS304 – The Linda Parallel Programming Paradigm  
McDONALD, CS (chris@cse.uwa.edu.au), page 1

CS304 – The Linda Parallel Programming Paradigm  
McDONALD, CS (chris@cse.uwa.edu.au), page 2
Actual and Formal Tuple Fields

Each field of a tuple can be an actual or a formal.

An actual field has both a type and a value. The type of an actual field is determined by the type of the expression providing its value.

For example, if `str1` is a variable of type `string` containing "hello", and `x` is a variable of type `integer` containing the value 24, then the tuple

\[(str1, x)\]

has two actual fields, the first being of type `string` with value "hello", and the second being of type `integer` with value 24.

A formal field has a type but no value, and can be thought of as a sort of place-holder. The type of a formal field is provided by the type of a variable in the host language.

A formal field is syntactically specified by a variable name preceded by a question mark, `?`. The tuple

\[(str1, ?x)\]

has an actual field and a formal field, the actual field has value `str1("hello")` and type `str1`, while the formal field has the type of variable `x` i.e. an integer.

Linda’s Tuple-Spaces and Partitions

Linda permits cooperation between parallel processes by controlling access to a shared data structure called a tuple-space.

A tuple-space is a non-ordered, possibly replicated, collection of tuples. Any number of identical tuples may exist in tuple-space; that is, a tuple-space is a bag and not a set.

Multiple tuple-spaces, each holding tuples of the same type-signature (and hence arity) are frequently termed partitions.
Communicating Through Linda’s Tuple Spaces

Processes communicate through tuple-spaces by depositing and removing tuples. The tuple-space is thus a shared data object. All processes having access to the same space have (potential) access to all tuples in it.

Manipulation of the tuple-space is only possible using a very small set of Linda primitives. Each primitive is applied to a single tuple.

Within most implementations of the Linda paradigm, each primitive appears syntactically as a procedure or function call with a variable number of parameters.

The out Primitive

The out primitive deposits a tuple into tuple-space. Using the above values of str1 and x, the operation

\[ \text{out}(\text{str1}, x) \]

deposits a tuple with values \(\text{"hello"}, 24\) into tuple-space. Formal fields are (generally) not permitted in out operations.

Tuple Matching with the in and rd Primitives

The in primitive removes a tuple from tuple-space. in uses its argument as a template or anti-tuple to specify the required tuple.

A tuple, P, in tuple-space is said to match a tuple, Q, iff

- P and Q have the same number of fields.
- Each tuple field \(p_i\) matches the corresponding tuple field \(q_i\).

Four cases must be considered in matching \(p_i\) against \(q_i\)

- \(p_i\) actual, \(q_i\) actual \(p_i\) must have the same type and value as \(q_i\).
- \(p_i\) actual, \(q_i\) formal \(p_i\) must have the same type as \(q_i\).
- \(p_i\) formal, \(q_i\) actual \(p_i\) must have the same type as \(q_i\).
- \(p_i\) formal, \(q_i\) formal no match.

E.g.: given the following C declarations:

\[
\begin{align*}
\text{int } & x; \\
\text{float } & f; \\
\text{char } & \ast str;
\end{align*}
\]

the tuple \((24, \text{"hello"}, 5.7)\) would be matched by

\[
\begin{align*}
in(?x, \text{"hello"}, 5.7) & \to x \text{ updated} \\
in(24, ?str, ?f) & \to str, f \text{ updated} \\
in(?x, ?str, ?f) & \to x, str, f \text{ updated}
\end{align*}
\]
Tuple Matching with \texttt{in} and \texttt{rd}, continued

For example, if \texttt{z} is another integer variable, the request

\begin{verbatim}
   in("hello", ?z)
\end{verbatim}

could remove the previously deposited tuple from tuple-space. In addition to withdrawing the tuple, the value 24 would be assigned to the variable \texttt{z}.

This request could be made by the original depositing process or any other process with access to the tuple-space.

The \texttt{rd} primitive, pronounced \textit{read}, is similar to \texttt{in}, but leaves the matching tuple in tuple-space. It is used for its binding and synchronization side-effects.

Synchronization Using Tuples

If more than one tuple matches a template, one is chosen \textit{non-deterministically}. If no matching tuple can be found in tuple-space, the \texttt{in}-ing or \texttt{rd}-ing process will \textit{block} awaiting a matching tuple to be deposited.

These blocking semantics enable processes to \textit{synchronize} both their execution and data accesses.

The Unwelcome \texttt{inp} and \texttt{rdp} Predicates

Two related primitives are the \texttt{inp} and \texttt{rdp} predicates.

These perform tasks equivalent to \texttt{in} and \texttt{rd} but are \textit{non-blocking}.

Instead they return a Boolean value which indicates the success of the operation. The operation is deemed \textit{successful} if the required tuple is in tuple-space \textit{at the time of search}.

If the operation is successful, actuals are assigned to formals as before.

\begin{verbatim}
   if( rdp( "factorial", 100, ?result ) == TRUE ) {
      ... the tuple exists ...
   }
\end{verbatim}

Recent research argues strongly \textit{against} the use of these pathological predicates (particularly in distributed tuple spaces).

Implementations of \texttt{inp} and \texttt{rdp} typically result in the \textit{locking} of the tuple-space while the search is undertaken, or introduce \textit{temporal semantics} into the Linda model.
Linda’s eval Primitive for Process Creation

Linda supports process creation using the final primitive, eval.

The eval statement is syntactically similar to out except that a new process is created to evaluate each of the fields in the tuple (in theory).

When the evaluation has terminated, the tuple becomes an ordinary (a data) tuple in tuple-space.

For example:

```c
eval( 2, 3, a_time_intensive_calculation() );
```

As stated in Nadkarni’s book, examples typically cited in the Linda literature are often unrepresentative of the purpose of eval.

```c
for(n=1 ; n<=MAXIMUM ; n++)
  eval( "SQRT", n, sqrt(double)n );
```

When all fields of each eval have completed (are evaluated) all fields combine to form traditional data tuples in tuple space:

- ("SQRT", 1, 1.0)
- ("SQRT", 2, 1.4142136)
- ("SQRT", 3, 1.7320508)
- ("SQRT", 4, 2.0)
  .......

**NB:** In such an example, the cost of process creation far outweighs the cost of the calculation itself, and is thus not realistic.

Linda’s eval Primitive, continued

There is a significant controversy about the semantics (and implementation) of eval in Linda-based languages.

At present, it appears that the current semantics of eval are defined by Yale University’s latest implementation of C-Linda.

From Nick Carriero’s thesis (Dec 1987):

"There are significant unanswered questions about eval. Some of the most important concern variable bindings in [a process]. Can [a process] have unbound variables? If it can, when will these variables be bound? At the time eval executes or at the time [the process] executes?"

From Jerry Leichter’s thesis (Jul 1989):

"However, producing an exact specification for [eval] that is both natural within the C environment, and implementable with reasonable efficiency, is difficult."

... "In no case is any information passed back from the execution environment to the creating environment, except by explicit tuple space operations."
Simple Communicating Processes in C-Linda

Consider the simple requirement that one process passes an integer to another process, which then “replies” with another integer.

```c
void write_then_read()
{
    int x;

    out("first", 42);
    in("second", ?x);
    printf("write_then_read() received %d\n", x);
}

void read_then_write()
{
    int x;

    in("first", ?x);
    printf("read_then_write() received %d\n", x);
    out("second", 24);
}

void main()
{
    eval(write_then_read());
    eval(read_then_write());
    exit(0);
}
```

The Dining Philosophers’ Problem

The ubiquitous Dining Philosophers’ problem demonstrates constrained access to a shared resource:

```c
#define NUMBER 5

void philosopher(int i)
{
    for(;;)
    {
        think();
        in("room ticket");
        in("chopstick", i);
        in("chopstick", (i+1)%NUMBER);
        eat();
        out("chopstick", i);
        out("chopstick", (i+1)%NUMBER);
        out("room ticket");
    }
}

void main()
{
    int i;

    for(i=0; i<NUMBER; i++)
    {
        out("chopstick", i);
        eval(philosopher(i));
        if(i < NUMBER-1)
            out("room ticket");
    }
}
```
Distributed Data Structures with Linda

A distributed data structure is one that can be manipulated simultaneously by several concurrently executing processes.

Although the natural complement of parallel program structures, distributed data structure are not supported in most parallel programming languages.

Consider the maintenance of a queue structure in Linda:

Adding to the queue:

```
in ("widgets", "tail", ?i);
out ("widgets", "tail", i+1);
out ("widgets", i+1, value);
```

Removing from the queue:

```
in ("widgets", "head", ?i);
in ("widgets", "head", i);
out ("widgets", "head", i+1);
eval (producer()); /* or any number of these */
eval (consumer()); /* or any number of these */
```

The Producer-Consumer Problem in C-Linda

```c
void producer()
{
    int index;

    for(;;)
    {
        in ("widgets", "tail", ?index);
        out ("widgets", "tail", index+1);
        out ("widgets", index+1, next_value(index));
    }
}

void consumer()
{
    int index, value;

    for(;;)
    {
        in ("widgets", "head", ?index);
        out ("widgets", "head", index+1);
        in ("widgets", index, ?value);
        consume_widget(value);
    }
}

void main()
{
    out ("widgets", "head", 1);
    out ("widgets", "tail", 1);
    eval (producer()); /* any number of these */
eval (consumer()); /* or any number of these */
}
Designing a Linda-based Language

Linda-based systems present great opportunities for experimenting with concurrency issues, concurrent language design, parallel algorithms and network/cluster computing implementation.

Some important questions must be answered in designing your own Linda system:

- Should we add Linda operators to an existing sequential language, or design a new language?

- What granularity of parallelism should we implement? Should eval mirror operating system facilities for process creation, should lightweight processes be employed, or should heavyweight processes be invoked on a local area network (or a mixture)?

- How should the tuple spaces be distributed, should they reside on a single processor, or be scattered over a local area network?

We shall discuss the design and implementation of our Joyce/Linda concurrent language which has been used successfully in our third year undergraduate Concurrent Programming course and supported with a number of fourth year (Honours) projects at The University of Western Australia.

The Joyce Concurrent Programming Language

Joyce (per-Brinch Hansen 1987) is a concurrent programming language syntactically similar to Pascal, which adds message passing primitives similar to those found in Hoare’s CSP.

Joyce provides the base datatypes of integer, char, Boolean, real, user-defined enumerated types and records and arrays of these types.

The unit of execution in a Joyce program is the agent, which defines concurrently executing processes. An agent is syntactically similar to a Pascal procedure, but exhibits the semantics that all instantiations result in the concurrent execution of caller and callee.

A process is created dynamically by invoking its agent, and it then runs in parallel with its creator.

```
agent main;
const MAX = 100;

agent concurrent(n: integer);
begin
  { ... perform some calculation ... }
end;

begin
  for i := 1 to MAX do
    concurrent(i);
  { ... perform some calculation ... }
end;
```
The Joyce/Linda Programming Language

The Joyce/Linda programming language is derived from per Brinch-Hansen’s Joyce programming language and Gelemer’s Linda programming paradigm.

Variables may only be declared *locally* to agents, that is lexical restrictions do not permit the sharing of variables. Parameters to agents may only be passed by value, again denying a concurrent agent access to any but its own variables.

---

A Parallel Sieve of Eratosthenes

Consider the traditional (sequential) solution to finding prime numbers. We use a sieve to remove multiples of certain values (which are known to be prime). A simple parallel implementation could use a separate process to perform the filtering of each prime number’s multiples.

---

![Diagram of tuple-space partition](image-url)
A Parallel Sieve of Eratosthenes, continued

agent primes;
const MAXPRIME = 2000; /* all primes up to MAXPRIME */

agent filter(prime : integer);
    /* filters out multiples of prime */

var try, nextprime, myseqno, nextseqno : integer; /* initialized to zero */

begin /* filter */
    printf(" %d",prime);
    while true do begin
        in(prime,myseqno,?try);
        myseqno := myseqno+1;
        if (try mod prime <> 0) then begin
            if nextprime = 0 then begin
                nextprime := try;filter(nextprime);
                end
            else begin
                out(nextprime,nextseqno,try);
                nextseqno := nextseqno+1
                end
            end
        end
    end /* filter */;

begin /* primes */
    filter(2);
    for i := 3 to MAXPRIME do
        out(2,i-3,i);
    end /* primes */;

---

Early Uni-processor Joyce/Linda (Pre-)Compilers

Two similar pre-compilers for the Joyce/Linda language were initially developed under Solaris and lightweight process libraries.

- The first design, developed by James Pinakis, used the UNIX compiler writing tools `lex(1)` and `yacc(1)`, the standard C compiler `cc(1)` and the GNU C compiler `gcc(1)`.

- The implementation currently used in our 3rd year teaching, written by Chris McDonald, employs a hand-written recursive descent parser, the GNU C compiler `gcc(1)` and the Solaris lightweight process library. This pre-compiler is about 6800 lines of C compiling to 80k.

The latter compiler produces ANSI-C code (typically 6 to 10 times as many lines as the original Joyce/Linda program) which is then compiled and linked with a lightweight process library.

**NB:** All required concurrency is implicitly defined in the semantics of Joyce/Linda and the compilers need not infer any concurrency nor vectorize statements. The granularity of the parallelism is the agent – neither the statement, nor the expression.
Compilation for Lightweight Processes under Solaris

The Lightweight Process Runtime Structure

Consider how Joyce/Linda’s agents and tuple-space partitions are supported, firstly, in a single processor implementation.

The Joyce/Linda compiler has determined that three independent tuple-space partitions are required. Static analysis alone cannot (in general) determine how many agents are required.
Architecture Independent Compile-time Analysis

The challenge in creating efficient Linda systems lies in:

- minimizing the run-time matching, and
- minimizing the interprocess communication.

Linda-based programs present an opportunity for analysis by their compiler in which tuple-space operations are transformed into simpler operations requiring little or no matching at run-time.

For example:

- different type-signatures result in physically disjoint tuple-spaces (partitions).
- constant fields in a type-signature result in these fields being ignored in matching.
- “always actual” (never formal) fields form keys in a distributed hashing scheme.
- always constant tuples are never stored – a simple count implements a semaphore.
- tuple accesses indicating indexing of a dense array of determinable size, result in the tuple-space being implemented as a true array.

A Distributed Binary Tree Example

```
agent treesort;
const NULL = 0, ROOT = 1;
agent add(newvalue, newindex:integer);
var index, value, left, right : integer;
done : Boolean;
begin
  if newindex = ROOT then /* create new tree */
    out("tree", newvalue, ROOT, NULL, NULL)
  else begin
    done := false; index := ROOT;
    repeat
      in("tree", ?value, index, ?left, ?right);
      if newvalue < value then
        if left = NULL then begin /* add to left subtree */
          out("tree", value, index, newindex, right);
          out("tree", newvalue, newindex, NULL, NULL);
          done := true
        end
        else begin /* walk left */
          out("tree", value, index, left, right);
          index := left
        end
      else /* add to right subtree */
        .......
        until done
    end;
  end;
out("done")
end;
begin
for i := 1 to MAXNODES do
  add(random,i);
for i := 1 to MAXNODES do
  in("done");
end;
```
Compile-time Analysis, continued

As an example, consider the compile-time analysis of the Joyce/Linda program building a distributed tree:

```
prompt> jlc -DMAXNODES=1000 -a treesort.jl
partition ("tree",integer,integer,integer,integer):
  1 in, 0 rds, 7 outs
  only accessed by agent 'add'
  field 0 is always "tree"
  field 1 is always a formal
  field 2 is always an actual
  field 3 is always a formal
  field 4 is always a formal
  actuals (a*a**) formals (*f*ff) const (c****)

partition ("done"):
  1 in, 0 rds, 1 out
  field 0 is always "done"
  actuals (a) formals (f) const (c)
  all fields are constant => a semaphore
```

Code Generation for Tuple Types

The results of the tuple analysis directs the code generation for tuple declaration and transmission.

```c
/* partition ("tree",integer,integer,integer,integer) */
typedef struct {
  struct {
    /* f0 is always "tree" */
    long f2;
  } aa;
  long f1;
  long f3;
  long f4;
} TT_0;

static int MATCH_0(TT_0 *t1, TT_0 *t2, BITMAP B)
{
  /* f0 is always "tree" */
  /* f1 is always a formal */
  if((B&(1<< 2)) && (t1->aa.f2!=t2->aa.f2)) return(FALSE);
  /* f3 is always a formal */
  /* f4 is always a formal */
  return(TRUE);
}

static void PRINT_0(char *pre, TT_0 *t, BITMAP B, char *post)
{
  PRINTF("%s("tree","",pre);
  if(B&(1<<1)) PRINTF(",%d", t->f1);
  else PRINTF(",%d", t->f1);
  PRINTF(",%d", t->f2);
  if(B&(1<<3)) PRINTF(",%d", t->f3);
  else PRINTF(",%d", t->f3);
  if(B&(1<<4)) PRINTF(",%d", t->f4);
  else PRINTF(",%d", t->f4);
  PRINTF("),%s",post);
}```
Isolating Architecture Specific Code Generation

Joyce/Linda currently provides facility to define new target software (MIMD) architectures.

A command-line argument declares the required architecture, and an architecture specification is dynamically loaded into the compiler.

Each architectural description defines:

- Degree of process memory sharing,
- Whether asynchronous communication is supported,
- Maximum stack sizes of each Joyce/Linda agent,
- Location of formals, locals and tuples – stack or heap,
- Code generation routines for –
  
  agent declaration and definition,
  
  agent invocation,
  
  tuple initialization and transmission, and
  
  final compilation and linking.

These attributes control both the tuple space analysis and code generation.

Architecture-specific code generation “calls back” to generic routines such as statement and expression generation.
The Fujitsu AP1000 Architecture

The Fujitsu AP1000 employs between 16 and 1024 identical SPARC ver.8 processors (cells), each with its own local memory. Processors are connected by three high-speed communication buses.

- The S-net and B-net provide 50MBps throughput.
- The T-net provides 40MBps throughput, with 160ns latency.
- The VME-based CAP-host link provides only 6MBps throughput.

The AP1000 at the Australian National University consists of 128 SPARC 1+ cells, each with 16Mb of memory and 128KB of cache. An AP3000 at Imperial College, London, consists of 80 UltraSPARC processors and 12.5GB of memory.

Compilation of Joyce/Linda on the AP1000

Each cell executes its own minimal operating system kernel and communicates with other cells using either broadcast or point-to-point messaging. There is no virtual memory nor pre-emptive multitasking.

The naive placement of agents and tuples on AP1000 cells is as follows:

- Each cell executes the same (statically linked) program.
- Cells act as either the agent pool (cell 0), tuple partition managers, or worker cells.
- An idle worker cell requests work from the agent pool.
- A cell invoking a new agent informs the agent pool of the agent’s address and arguments.
- The agent’s address and arguments are sent to idle cells.
- Terminates occurs when all worker cells are idle in the agent pool; otherwise we have deadlock.
Joyce/Linda and MPI on the Fujitsu AP1000

A recent addition to the AP1000 software architecture has been an implementation of the Message Passing Interface (MPI-1, April 1994) standard.

Key MPI features:

- Library bindings for both C and Fortran-77,
- Communication modes: point-to-point, buffered, synchronous,
- Message tagging and communication domains,
- Hierarchical derived datatypes,
- Heterogeneous data and network support, and
- Robust error handling, including exceptions.

The implementation of the Joyce/Linda code generator for MPI:

- The host process acting as MPI_Comm_rank = 1,
- The single MPI_COMM_WORLD communicator for all messages,
- MPI_TAGS for Linda operations, agent instantiation, and MPI_cell ←→ MPI_host communication,
- The single MPI_BYTE data representation because of the AP1000’s homogeneous architecture.

The addition of an MPI architectural specification for Joyce/Linda took about 4 hours.

Post-Mortem Analysis of Joyce/Linda

The existing Joyce/Linda execution viewer is still supported with the AP1000 implementation, though some additional synchronization is required.

Post-mortem studies of Joyce/Linda programs are in their early stages:
Using Joyce/Linda in Teaching Concurrency

A number of features have been added to our Joyce/Linda environment for its use as a language for teaching concurrent programming:

- Options have been added to the compiler to vary some attributes of execution, such as the size of each agent’s runtime stack and the pre-emptive timeslice of each agent.

- Facility has been added to view the temporary C code produced by the Joyce/Linda compiler.

- Both static and dynamic analysis of tuple-space usage is available.

- An annotated trace of tuple-space activity may be requested. Here, the “normal” output of a Dining Philosophers’ solution is indented:

```plaintext
in("room ticket") matches:
  ("room ticket")
    philosopher 2 eating
in("chopstick",3) blocked
in("room ticket") blocked
in("chopstick",2) blocked
in("chopstick",0) matches:
  ("chopstick",0)
out("chopstick",2)
in("chopstick",2) unblocked
philosopher 1 eating
in("chopstick",1) blocked
out("chopstick",3)
```

Viewing Joyce/Linda Programs under Execution

Our Joyce/Linda compiler optionally generates code to interact with a specially written Joyce/Linda viewer of programs running under X-windows.

Using this viewer, `jlv`:

- Each agent is represented in a scrollable window as an icon which persists until the agent terminates.

- Agent icons may be selected with the mouse to view an agent’s source code under execution.

- Tuple-space partitions are shown as selectable, pop-up, scrollable windows containing different shaped icons representing `out`-ed tuples and `in` and `rd` templates (requests).

- Selection of a tuple’s icon displays the contents of that tuple.

- The speed of execution of the Joyce/Linda program can be controlled or the whole program single-stepped.

- Breakpoints may be established in any agent’s source code.

This Joyce/Linda viewer only annotates a program’s execution. There is, as yet, no facility to modify an agent’s local variables or to add or remove tuples from the tuple-space.

Single-stepping large programs has proved tedious and we will next add `template-matching` breakpoints to the tuple-space animation.
Parallel Mergesort under Execution

Invoked as:

```
prompt> jlc -jlv mergesort.jl
prompt> jlv mergesort
```

Acknowledgements

Thanks go to all Honours and postgraduate students who have worked on (and still are working on) various aspects of the Linda model:

Voon-Li Chung (PhD 2003), Timothy Duncan (Hons 1991), Andrew Forth (Hons 1992), Kamran Kazemi (PhD 2000), Nicholas Kornweibel (Hons 1991), Ranadeva Peries (MSc 1998), James Pinakis (PhD 1993), Geoff Sutcliffe (PhD student), and Brett Tyson (GradDip 1989).

Some Linda Reading Materials


