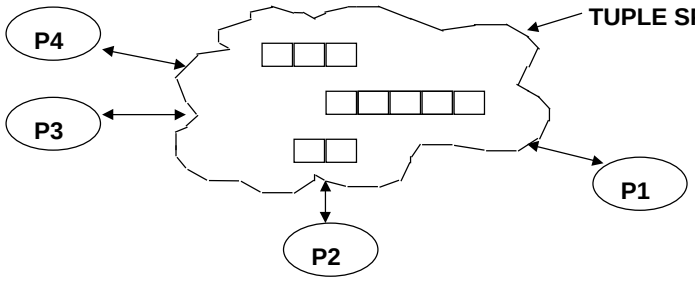
Linda - A Programming System

LINDA

All communication and synchronization is through "tuple space"



Tuple = ordered set of typed values where type name is placeholder for all values of a given type

Analogy - Synchronized operations on a relational database

Linda - A Programming System

Processes execute operations which act on specified tuples in tuple space

Tuple Space Operations - in, inp, out, rd, rdp, eval

Operations are defined as procedures invoked in c programs

out(5, "peter") - outputs a tuple (5, "peter") from a process to the tuple space

out(6, 7, 8) - places (6, 7, 8) in tuple space

in(5, "peter") - removes this tuple from the tuple space and instantiates it in the name space of the executing process. If no such tuple exists in the TS then the executing process BLOCKS.

inp(5, "peter") - = in(5, "peter") except that it will not block if a matching tuple is not present

Linda - A Programming System

in(i : integer, 7, 8) - removes a tuple matching the pattern (*, 7, 8) where "*" is any integer

example : let TS contain (3, 7, 8), (2, 7, 8) and (9, 7, 8).

in(i : integer, 7, 8) - non-deterministically return one of those three tuples

"OR" firing rule

i : integer is a "formal", a typed placeholder

7, 8 are actuals, objects with a value

out(i : integer) - puts in tuple space a tuple that will match in(j) requests for any value of j, in(6) or in(7).

In("a string",?f, ?l, "another string") - rremoves from tuple space a tuple with first element "a string", second and third elements with types matching f and l and a fourth element "another string."

Linda - A Programming System

rd(i : integer, "peter") - reads a matching tuple from TS but does not remove the tuple from tuple space. rd blocks if no match is found

rdp( ) - non-blocking version of rd.

eval("worker", worker(i)) - inserts in TS a tuple which has the values obtained by evaluating its arguments. Here worker(i) is a procedure which must be executed to create a result value for the tuple.

eval("e", 7, exp(7)) - Creates a process which evaluates to a tuple with the values (e,7,exp(7)) and leaves it in tuple space.

Rd("e",7,?value) - block until exp(7) is evaluated and the tuple created.

eval("Q", f(x,y)) - the values of f, x and y have the same values bound to them as in the invoking context..

Linda - A Programming System

Implementation of Useful Objects in Linda

Semaphores

V = out(“sem”), P = in(“sem”)

Bags

out(“task”, TaskDescription), in(“task”, ?NewTask)

Parallel Loop

```
for (<loop control>)
        eval(“this loop”, somefunction()); somefunction() => value
for (<loop control>)
        in(“this loop”, value);
```

Barrier

```
out(“barrier”, n);
in(“barrier”, ?val); out(“barrier”,val-1); rd(“barrier”, 0);
```


Linda - A Programming System

Bag of Indistinguishable Items

```
V(sem) = out(“sem”)
P(sem) = in(“sem”)

Out(“task”, TaskDescription)
in(“task”,? NewTask)
```

Name Accessed Structures

Barriers

```
out(barrier-37, n)

in(“barrier-37”, ? val)
out(“barrier-37”, val -1)
rd(“barrier-37”, 0)
```

Implementation of Useful Objects in Linda - Continued

Streams - dynamic list of values

View of Stream as it is built - Stream - head, tail, body

- **head is index of "next" element**
- **tail is index of last position**
- **body is each element of list**

View of Stream which is just read does not require head and tail entries.

Streams

in-streams, read-streams

read-stream - tuples with name, index and value items)

("stream", 1, val1)
("stream" , 2, val2)
("stream", 3, val3)

in/out-stream - add head and tail tuples

("stream", "tail", 14)
("stream, "head" , 14)

in("stream", "tail" ? index)
out("stream", "tail", index + 1)
out("stream", index, NewItem)

in(stream", "head", ? index)
out("stream", "head" index+1)
in("stream", index, ? element)

Linda - A Programming System

Implementation of Useful Objects in Linda - Continued

"Live" Streams - streams growing in tuple space.

```
for (i = 0,i < n)
        eval("stream", i, f(i));

for (i = 0, i < n)
        rd("stream", i, ?value);
```


Linda - A Programming System

Model of parallel computation

Units of computation - processes and argument evaluations
Name Model - Shared name at top category of taxomony
Name space - local + tuple space
Relationships - shared name dependencies
- m<-->n communication topology

Simulation of partitioned name space or message model

P1 : out("P2" message) implements message
P2 : in("P2" message) communication between P1 and P2

P1 : out(i : integer, message)
P2 : rd(2, message) message from P1 to all
P3 : rd(3, message) processes (assuming processes use integers as names)

Linda - A Programming System

```
#define LIMIT 1000
real main()
{
{Int count = 0, i, is prime(), ok;
for(i = 2; i <= LIMIT; ++i) eval("primes", i, is_prime(i));
 for(i = 2; i <= LIMIT; ++i) {
    rd("primes", i, ? ok);
    if (ok) ++count; }
 printf("%d.\n", count);}
is_prime(me)
       int      me;
{ int         i, limit, ok;
 double      sqrt();
 limit = sqrt((double) me) + 1;
 for (i = 2; i < limit; ++i) {
    rd("primes", i, ? ok);
    if (ok && (me%i == O)) return O; }
 return 1;}
```

Simple parallel program to compute primes up to LIMIT.

Linda - A Programming System

```
real main()
{
   eval("source", source());
   eval("sink", sink());}
source()
   { int i, out index=O;
   for (i = S; i < LIMIT; i += 2) out("seg", 3, out index++, i);
   out("seg", 3, out_index, 0);}
sink()
  { int in index--O, num, pipe seg(), prime=3, prime count=2;
   while(l) {
      in("seg", prime, in_index++, ? num);
      if (!num) break;
      if (num % prime) {
      ++prime count-
      if (num*num < LIMIT) {
      eval("pipe seg", pipe seg(prime, num, in_index));
      prime = num;
      in_index = O;}}}
```

```
    printf("count: %d.\n", prime count)

  pipe seg(prime, next, in index)
    {
    int num, out mdex=O;
    while(l) {
      in("seg", prime, in index++, ? num);
      if (!num) break;
      if (num % prime) out("seg", next, out_index++, num);
      }
    out("seg", next, out index, num);
  }
```


Matrix Multiplication

```
long       dim;
long       workers;
lmain(argc, argv)
int      argc;
char   **argv;{
      long          col[MAX], row[MAX];
      long          index, true_result;
      long          result[MAX], row_index, col_index;
      LINDA_BLOCK COL, RESULT, ROW;
      if (argc  != 3) {
         printf("Usage; %s <workers> <dim> \ n", *argv);
         exit(1);}
/*    LINDA_TRACE_ON;
      LINDA_LIST_ON;   */
      workers = atol(*++argv);
      dim = atol(*++argv);
      printf("matrix -- workers: %d, dim: %d.\ n", workers,
                                                    dim);
      start_timer( );
```

Matrix Multiplication (Continued)

```
/*start workers */
for (index = 0; index < workers; ++index) {
   eval("worker", worker( ));}
COL.data = col;
COL.size = dim;
ROW.data = row;
ROW.size = dim;
for (index = 0; index < dim; ++index)  {
   row[index] = 3;
   col[index] = 5;}
   for (index = 0; index < dim; ++index)  {
   out("row", index, ROW);
   out("col", index, COL);}
timer_split("done setting up");
out("task", 0);
```


Matrix Multiplication (Continued)

```
RESULT.data = result;
true_result = 15 * dim;
for (index = 0; index < dim; ++index)  {
   in("prod", ? &row_index, ? &RESULT);
   for (col_index = 0; col_index < dim; ++col_index)  {
       if (result[col_index]  != true_result)  {
           printf("got result(%ld, %ld) : %ld. \ n",
                   row_index, col_index, result);}}}
timer_split("all done");
print_times(  );

for(index=0, index<workers; ++index) in("worker,  ? int *);}
worker( ) {
long      col_index, dot, index, next_index, row_index
long      *cp, col[MAX], result[MAX], row[MAX], *rp;
LINDA_BLOCK COL, RESULT, ROW;
COL.data = col;
RESULT.data = result;
RESULT.size = dim;
ROW.data = row;
```

Matrix Multiplication
(Continued)

```
while(1)  {
   in("task",  ? &row_index);
   if (row_index < 0)  {
        out("task", -1);
        return;}
   next_index = row_index + 1;
   if (next_index < dim)
        out("task", next_index);
   else
        out("task", -1);
   rd("row", row_index,  ? &ROW);
   for (col_index = 0; col_index < dim; ++col_index)  {
        rd("col", col_index,  ? &COL);
        dot = 0;
        rp = row;
        cp = col;
        for (index = 0; index < dim; ++index, ++rp, ++cp) {
             dot += *rp * *cp;}
        result[col_index] = dot;}
   out("prod", row_index, RESULT);}}
```