Four Steps in Developing a Parallel Program

1. Choose a Paradigm or Strategy
   How to organize
2. Choose a Programming Model
   How to structure – Communication Model
3. Choose a Programming System
   How to represent
4. Choose an Execution Model
   How to execute

Carriero and Gelernter cover the first three with the coverage of the third topic restricted to Linda.

Carriero and Gelenter Approach

1. Choose a concept class or paradigm
   result,agenda or specialist
2. Write a simple program in that paradigm.
   (They use only Linda.)
3. Refine the program to make it efficient.
   (Transform among paradigms.)

1. Formulate the problem in an appropriate paradigm
2. Choose a programming method
3. Choose a programming language
4. Write a program
5. Measure the programs behavior
6. Transform to obtain efficient program.

Paradigms:

Result - Define the data structure which will hold the result, determine a computation which will generate the result from the initial state and compute in parallel

Data Parallel - Partition the initial state data structure) and define as sequence of functions which generate the final state from the initial state in an SPMD mode.

Agenda of Activities - Create a list of tasks and invoke generic workers to execute tasks from the list

Ensemble of Specialists - Create an ordered list of defined tasks and execute in sequence by specialist units of computations – Pipeline Parallelism

Result Model of Parallel Execution

Define the data structure which is the desired result of the computation and design parallel processes to construct the elements of the data structure in parallel.

Example: Compute the sum of two vectors, A,B construct a n-element vector where the ith element is the sum of the ith elements of A and B. Write a program which sums two elements of a vector, create a copy of this program for each element and apply in parallel.

Applies where there is a well-defined result.

But not to control programs which don't terminate.

(Contrast to Data Parallelism)

Agenda Model of Parallelism

The two elements are: a list of tasks to be performed and a set of workers who can perform some task.

Example - Database search for record with minimum value for some variable.

1. Put all records in a "bag."
2. Select a coordinator
3. Direct each worker to take records from the bag, examine them and send the smallest found to the coordinator

Broadly applicable, may use family of workers, may be communication limited. Dependence relations may cause complexity.

Specialist Model of Parallelism

The computation is defined by a logical network of tasks. Each task may require a specialized worker.

Examples:
- Physical System
- Pipelines
- N-Body problem partitioned by
  - spacial coordinates.
- Truck Routing
  - Process for routing within a state.
  - Generate route by passing control
    - of a given route to
    - the processes for each
    - new state encountered
    - in the route.

Data Parallel Model of Parallelism

1. Partition initial data structure. $D \Rightarrow \{d_i\}$
2. Define f_1, f_2 …..
3. Execute function each f_j in sequence on each d_i

Special form of Agenda Parallelism?

or

Special form of Result Parallelism

Speculative Parallelism

"Or" parallelism

Commonly used to formulate search problems

Breadth first search of trees for specific information.

Programming Methods

Message Passing - Explicit transfer of information across name space boundaries. Processes persist.

Live Data Structures - Associate processes with segments of a data structure. Processes are spawned, execute to create a result for their local transformation of the data structure and die leaving behind a transformed value.

Distributed Data Structures - Processes communicate through shared data objects. Processes are persistent and process multiple units of data. Sharing may involve communication.

Structure of parallel program is determined by number of processes.

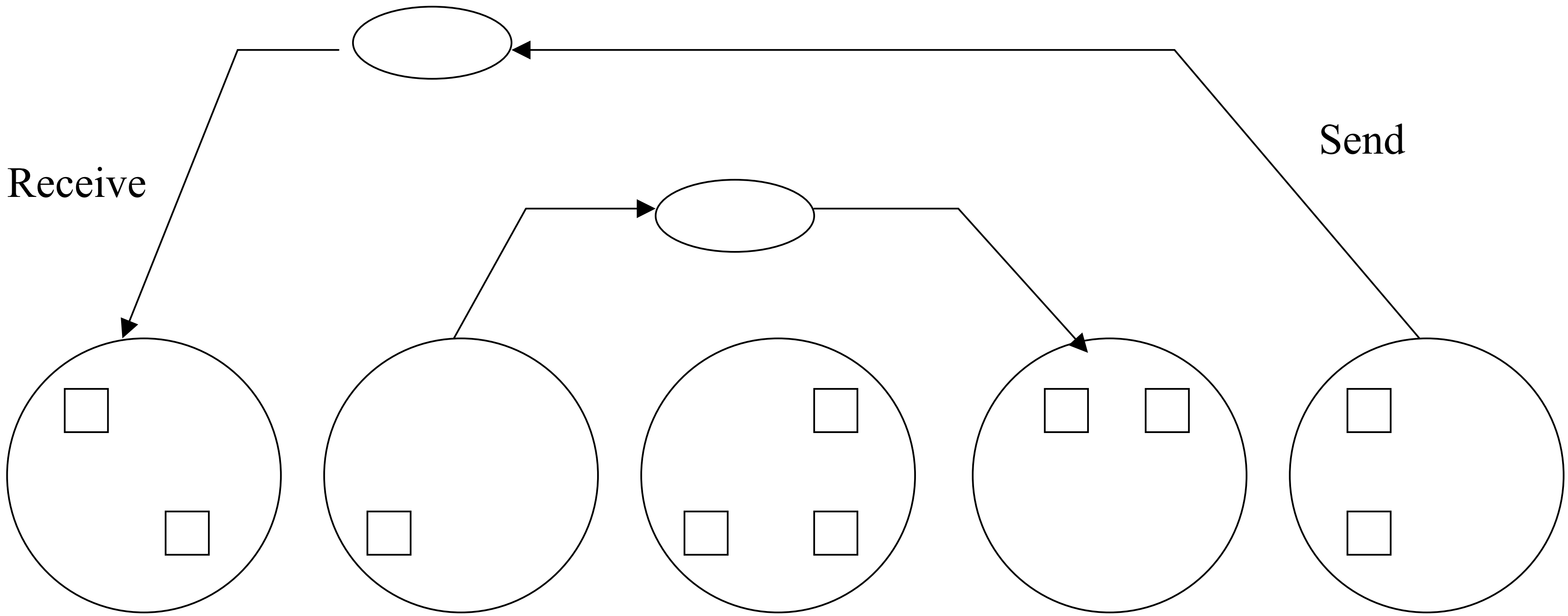
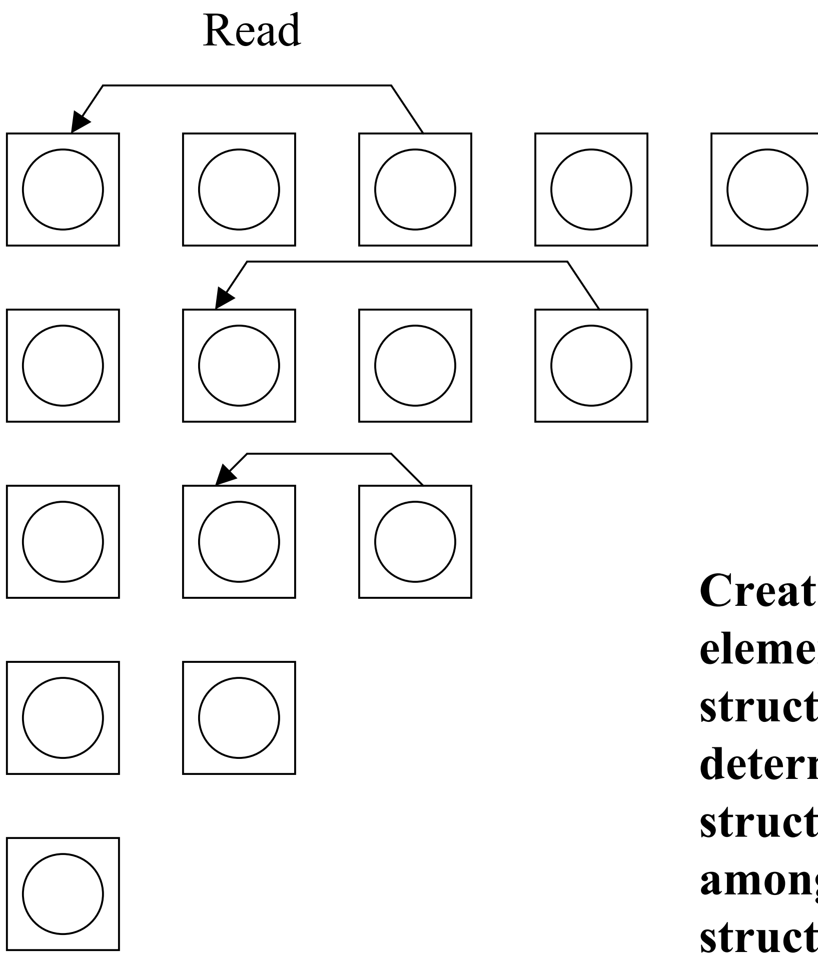


Figure 1: Message Passing



Create a process for each element of the initial data structure. Structure is determined by the data structure. Communication is among instances of data structures.

Figure 2. Live Data Structure

Formulation of Parallel Computations

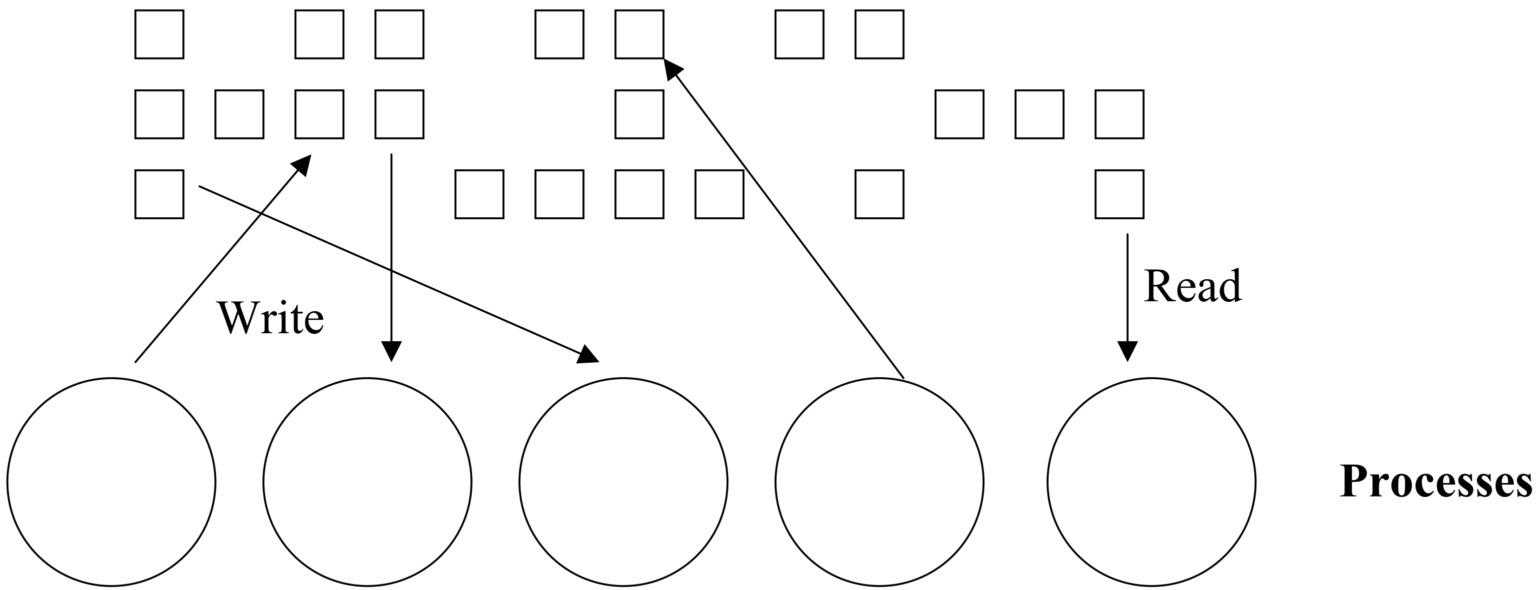


Figure 3. Distributed Data Structures

Each UC communicates with the other UCs by reading and writing shared instances of data structures.

Mappings from Paradigms to Programming Models

Specialist => Message Passing

Create processes for each task.

Network Routing - Pass off trucks from process to process

Result => Live Data Structures

Vector Sum - Associate a process with each element of the Sum Vector

Agenda => Distributed Data Structures

Data Base Search - Common data structure is “bag” of tasks.

N-Body Problem as Example

Result Parallelism - Live Data Structure

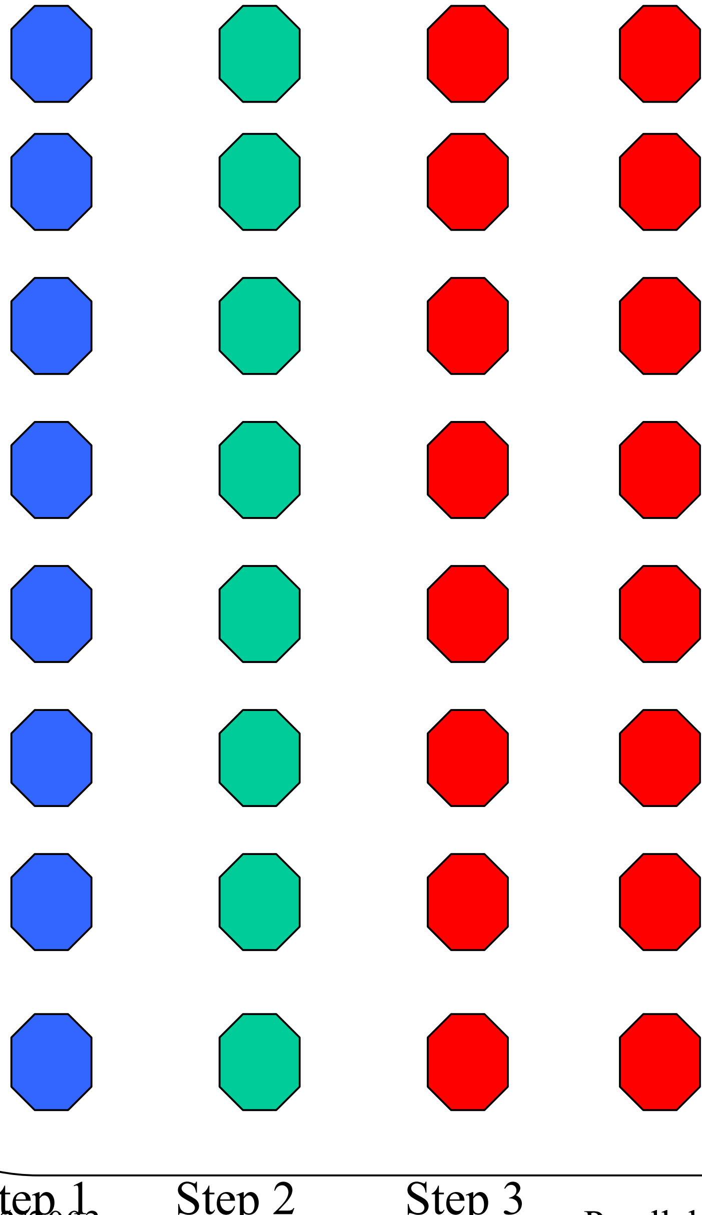
Result - A matrix M(i,j) such that M(i,j) is the position of the ith particle after the jth step of particle motion with column 0 of M containing the initial positions.

Define position(i,j,M(i,j-1)) to be a function which generates M(i,j) from M(i,j-1).

Invoke position(i,j,M(i,j-1) on each element of row i at each time step.

Parallelism of degree N where N is number of particles.

Formulation of Parallel Computations



Each hexagon holds a particle and a copy of M(i,j). Blue indicates the computation is complete, green that the computation is ongoing and red that the computation is waiting for the next column to the left to be completed. Position(i,j,M(i,j)) fetches the entries in the left column or accesses them.

N-Body Problem as Example

Agenda Parallelism - Distributed Data Structure - "Bag of Tasks"

1. Create a task description for each particle which gives its current state.
2. Create k processes which execute the function "compute next position" and leave a new task description for the next time step in the "bag."
3. The k processes will compute all the new positions and leave the task descriptions for advancing from the new position in the bag.
4. The jth time step can begin begin when all the particles have been advanced to their position in (j-1)st time step.

Note: Each worker must have access to positions of all particles at last time step.

N-Body Problem as Example

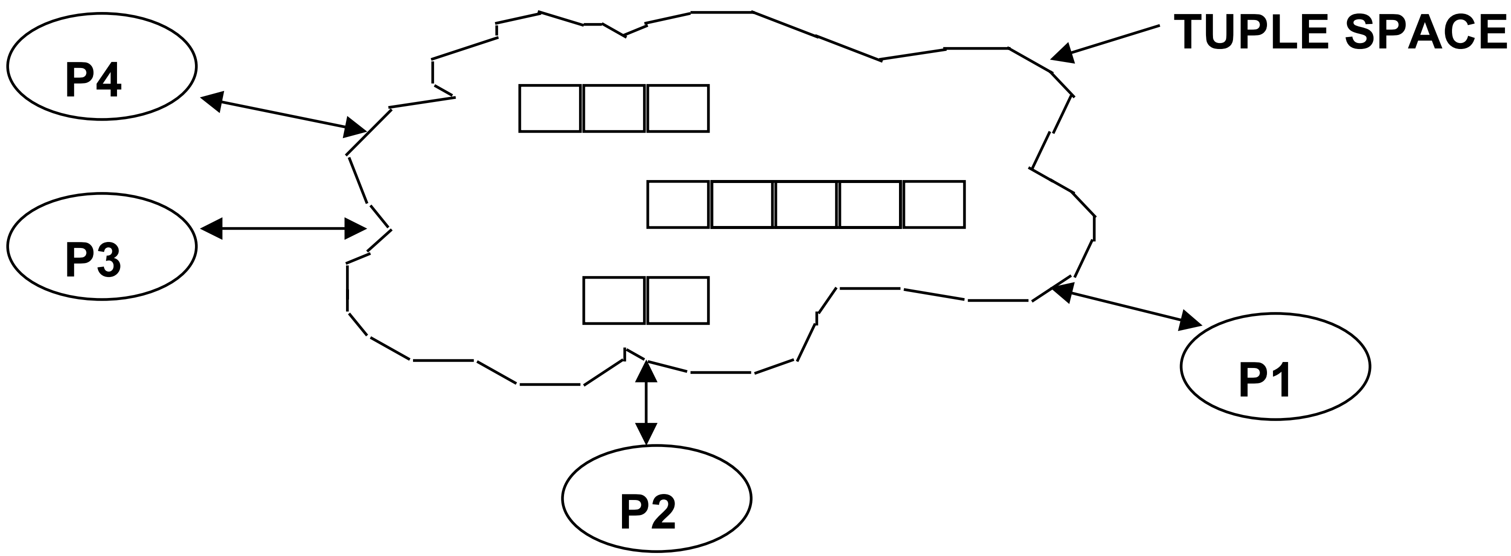
Specialist Parallelism - Message Passing among Independent Processes

1. Create a process for advancing the position of each individual particle.
2. After each time step each process exchanges location information with all other processes.
3. Each process computes the position of its particle for the next time step and initiates data exchange in preparation for the next time step.

Note - Each process must still have position of each charge at the last completed time step.

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.

P1, P2 …. are processes.

Analogy - Synchronized operations on a relational database

Processes execute operations which act on specified tuples in tuple space

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

Tuple space 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 th executing the process will not block if a matching tuple is not present

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.”

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..

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);
```

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)

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);
```

Model of parallel computation

Units of computation - processes and argument evaluations
Name Model - Shared name at top category of taxonomy
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)

To use the sieve of Eratosthenes to find the prime numbers up to 100, make a chart of the first one hundred whole numbers (1-100):

```
  1   2   3   4   5   6   7   8   9  10
 11  12  13  14  15  16  17  18  19  20
 21  22  23  24  25  26  27  28  29  30
 31  32  33  34  35  36  37  38  39  40
 41  42  43  44  45  46  47  48  49  50
 51  52  53  54  55  56  57  58  59  60
 61  62  63  64  65  66  67  68  69  70
 71  72  73  74  75  76  77  78  79  80
 81  82  83  84  85  86  87  88  89  90
 91  92  93  94  95  96  97  98  99 100
```

Cross out 1, because it is not prime.

Circle 2, because it is the smallest positive even prime. Now cross out every multiple of 2; in other words, cross out every second number.

Circle 3, the next prime. Then cross out all of the multiples of 3; in other words, every third number. Some, like 6, may have already been crossed out because they are multiples of 2.

Circle the next open number, 5. Now cross out all of the multiples of 5, or every 5th number.

Continue doing this until all the numbers through 100 have either been circled or crossed out. You have just circled all the prime numbers from 1 to 100!

Prime Finder Example Programs

Result Model - Live Data Structure Program
Many processes - elegant but not efficient

Agenda Model - Shared Data Structure Program
Ugly but efficient

Specialist Model - Message Passing Program with Streams.
Simple but not efficient.

Structure of Result Parallel Sieve - Live Data Structure Approach

Main Program

- **Create a tuple with a worker to determine the primeness of each integer in a range of integers .**
- **Wait for all workers to complete**
- **Print out list of integers**

Worker Program for k

- **read in primes for all numbers up to Sqrt(k)**
- **Check primeness of k for all primes up to Sqrt(k).**
- **Leave tuple with true or false for primeness of k.**

Parallelism

- **n processes where n is the range of integers.**
- **Amount of parallelism - about nlog(n)**

```
#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) 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 == 0)) return 0; }
 return 1;}
```

Structure of Agenda Parallel Program - Message Passing - Master/Workers

Master Program

- **Create p workers**
- **initialize data structures**
- **Create first task of finding primes among the first subrange of the range of integers to be sieved.**
- **Build "Distributed Table' of primes found.**
- **Print count of primes found.**

Worker Program

- **Read in task specifications for range of integers to be sieved.**
- **Output task specification for the next subrange of integers to be sieved.**
- **Read in list of primes found by predecessors.**
- **Determine primeness of each integer in assigned subrange by moding with all previously found primes.**
- **On completion send new primes found in subrange to master for addition to list of primes found.**

Parallelism

Number of processes - p: determined by number of processors available

Amount of Parallelism - roughly p log(p).

Prime Finder (master): Agenda Parallelism

```
#include "linda.h"

#define GRAIN 2000
#define LIMIT 1000000
#define NUM_INIT_PRIME 15

long primes[LIMIT/10+1] =
        {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47};
long p2[LIMIT/10+1] =
        {4,9,25,49,121,169,289,361,529,841,961,1369,1681,
                        1849,2209} ;

lmain(argc, argv)
                int argc ;
                char *argv [] ;
{
        int eot, first_num, i, num, num_primes, num_workers ;
        long new_primes[GRAIN], np2 ;

        num_workers = atoi(argc[1]) ;
        for (i = 0; i < num_workers ; ++1)
                eval("worker", worker()) ;

        num_primes = NUM_INIT_PRIME ;
        first_num = primes[num_primes-1] + 2 ;

        out("next task", first_num) ;

        eot = 0 ; /* becomes 1 at "end of table" – i.e., table
                                complete */
        for (num = first_num; num < LIMIT ; num += GRAIN) {
                in("result", num, ? new_primes; size) ;
```

```
                for (i = 0, i < size, ++1, ++num_primes) {
                        primes[num_primes] = new_primes[i] ;

                        if (!eot) {
                                        np2 = new_primes[i]*new_primes[i] ;
                                        if (np2) > LIMIT) {
                                                eot = 1 ;
                                                np2 = -1 ;
                                        }
                        out("primes", num_primes,new_primes[i],np2);
                        }
                }
        }
        /*" ? int" means "match any int; throw out the value"*/
        for (i = 0, i < num_workers; ::i) in("worker", 7 int) ;

        printf("%d: %d\n", num_primes,primes[num_primes-1]);
}
```

Prime Finder (master): Agenda Parallelism (Continued)

Prime Finder (worker): Agenda Parallelism

```
worker ()
{
        long count, eot, i, limit, num, num_primes, ok, start ;
        long my_primes[GRAIN] ;

        num_primes = NUM_INIT_PRIME ;

        eot = 0 ;
        while(1) {
                in("next task", ? num) ;
                if (num == -1) {
                        out("next task", -1) ;
                        return ;
                }
                limit = num + GRAIN ;
                out("next task", (limit > LIMIT ? –1 : limit) ;
                if (limit < LIMIT) limit = LIMIT ;

                start = num ;
                for (count = 0, num < limit ; num += 2) {
                        while (!eot && num > p2[num_primes-1]) {
                                rd ("primes", num_primes, ? primes
                                        [num_primes] , ? p2[num_primes]) ;
                                if (p2[num_primes] < 0)
                                        eot = 1 ;
                                else
                                        ++num_primes ;
                        }
                        for (i = 1, ok = 1 ; i < num_primes ; ++i) {
                                if (!(num%primes[i])) {
                                        ok = 0 ;
                                        break ;
                                }
                                if (num < p2[i] break ;
```

```
                }
                if (ok) {
                        my_primes[count] = num ;
                        ++count ;
                }
        }
        /* Send the control process any primes found. */
        out("result", start, my_primes ; count) ;
    }
}
```

Prime Finder (worker): Agenda Parallelism (Continued)

Specialist Parallelism - Dynamic Network of Processes

1. Set up sequence of processes which determine if the entries in an increasing stream of integers are prime relative to a given prime..
2. Each time a prime is found create a new process to continue sieving with the prime which discovered the new prime.
3. Continue the original process sieve with the newly discovered prime.

Parallelism

Creates as many processes as primes found.

Formulation of Parallel Computations

```
real main()
{
  eval("source", source());
  eval("sink", sink());}
source()
  { int i, out index=0;
  for (i = 5; i < LIMIT; i += 2) out("seg", 3, out index++, i);
  out("seg", 3, out_index, 0);}
sink()
 { int in index=0, 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 = 0;}}}
```

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

pipe seg(prime, next, in index)
  {
  int num, out mdex=0;
  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)

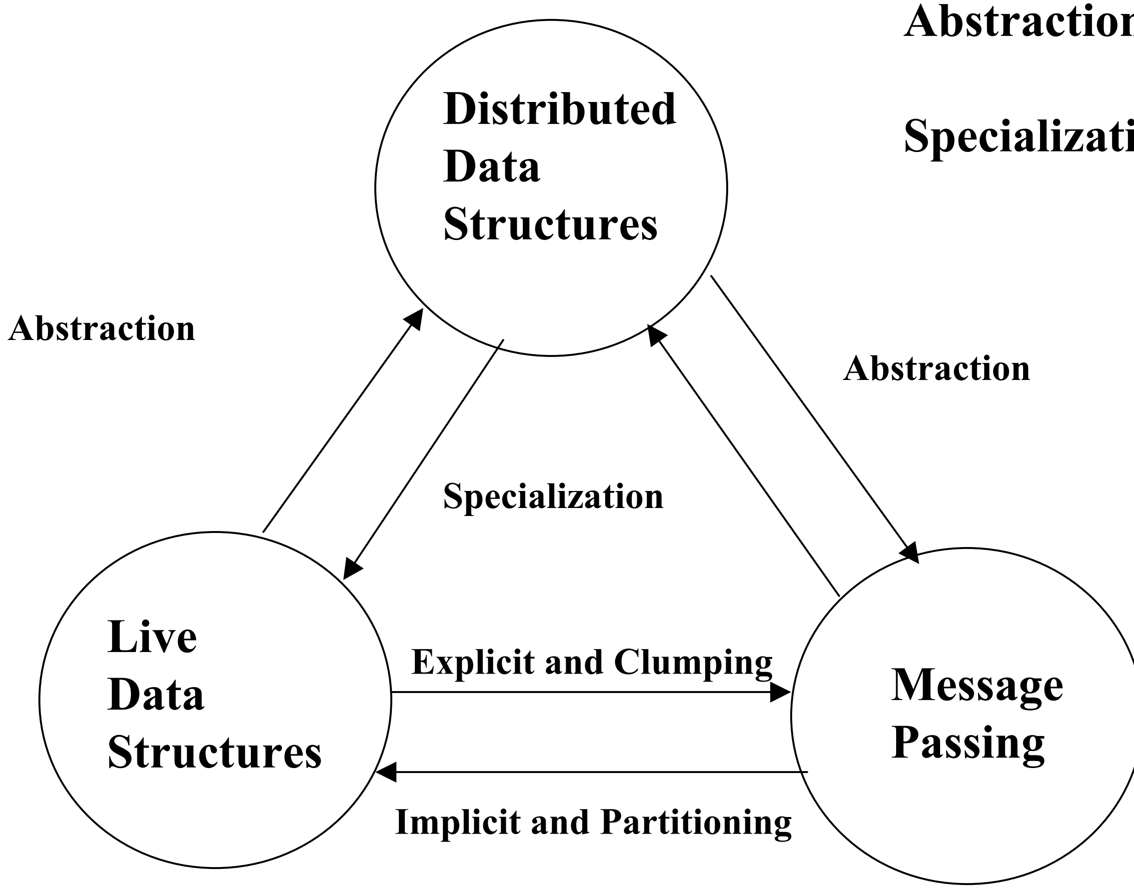
```
RESULT.data = result;
true_result = 15 * dim;
for (index = 0; index < dim; ++index)  {
   in("prod", ? &row_index, ? &RESULT);
   for (col_index = 0; col_index < dim; ++col_index)  {
       if (result[col_index]  != true_result)  {
           printf("got result(%ld, %ld) : %ld. \ n",
           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

Formulation of Parallel Computations



Abstraction - Detach from context.
Specialization - Bind process to object.