Four Steps in Developing a Parallel Program

1. Choose a Paradigm or Strategy
   How to organize
2. Choose a Programming Model
   How to structure
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 set 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.

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. $D=>\{d_i\}$
2. Execute function f 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.

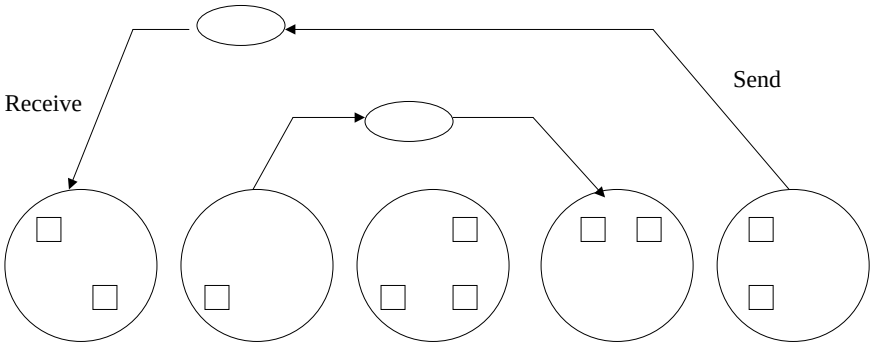
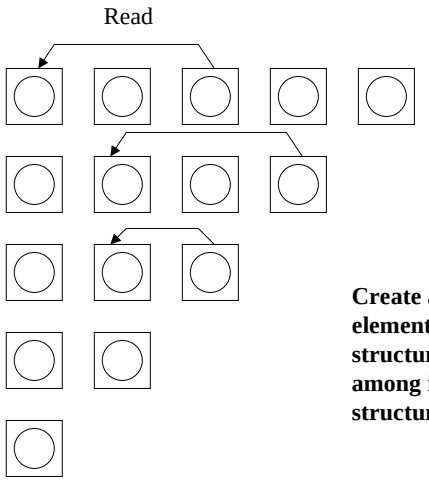


Figure 1: Message Passing



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

Figure 2. Live Data Structure

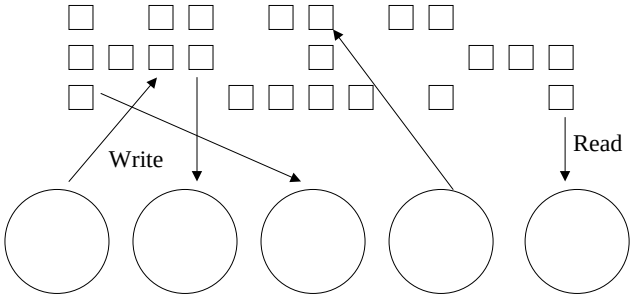


Figure 3. Distributed Data Structures

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

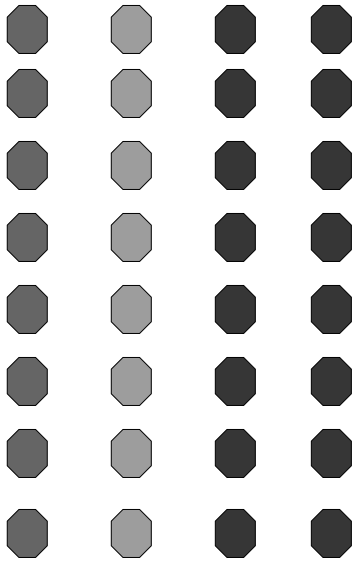
Result Parallelism - Live Data Structure

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.



Step 1 Step 2 Step 3

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. 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 particle 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.

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)


```
lmain ()
{
        int     i, 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" , i) ;
        }
}

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 ;
}
```

Prime Finder: Result Parallelism

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

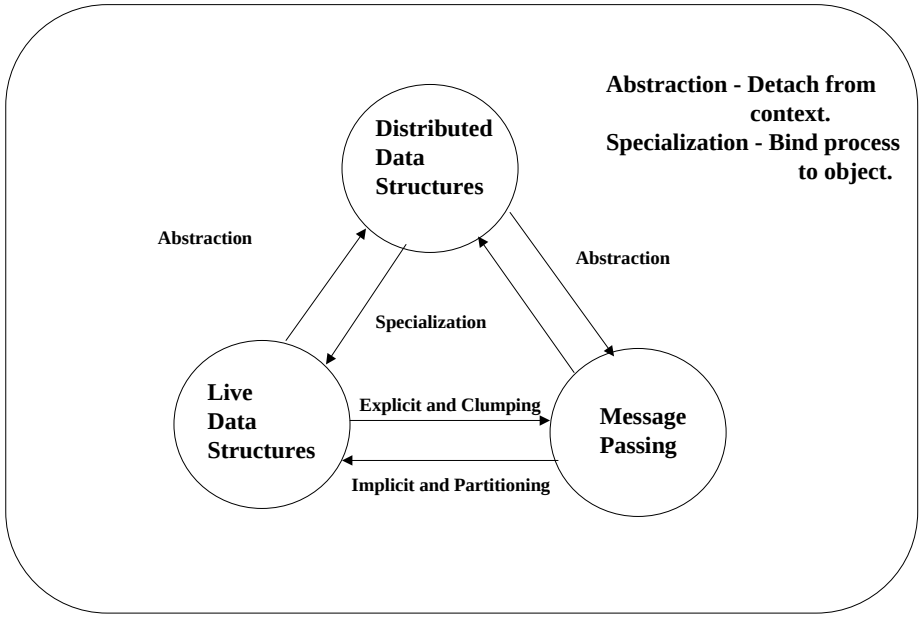
Creates as many processes as primes found.

Prime Finder: Specialist Parallelism

```
lmain()
{
        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, prime = 3, prime_count = 2 ;

        while(1) {
                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             prime, next, in_index ;
{
```



```
        int     num, out_index = 0 ;

        while(1) {
                in("seg", prime, in_index++, ? num) ;
                if (!num) {
                        out("seg", next, out_index, num) ;
                        return ;
                }
                if(num % prime) out("seg",next,out_index++,num);
        }
}
```

Prime Finder: Specialist Parallelism (Continued)

Notes on Carriero and Gelernter