

# Parallel Processing

The following material is based on lectures given by

**Jeremy R. Johnson at Drexel University**

*Original lecture materials are available at:*

<http://www.cs.drexel.edu/~jjohnson/wi01/cs730.html#lectures>

These lectures were derived from material in:

**HOW TO WRITE PARALLEL PROGRAMS**

**A FIRST COURSE**

By Nicholas Carriero and David Gelernter

Published by MIT Press (3rd Printing, 1992)

*An electronic copy can be downloaded from the following site:*

<http://www.lindaspaces.com/book/index.html>

# Parallel Processing

## Lecture 1: Introduction to Parallel Programming with Linda

# Introduction

- **Objective: To introduce a methodology for designing and implementing parallel programs. To illustrate the Linda coordination language for implementing and running parallel programs.**
- **Topics**
  - **Basic Paradigms of Parallelism**
    - result parallelism
    - specialist parallelism
    - agenda parallelism
  - **Methods for Implementing the Paradigms**
    - live data structures
    - message passing
    - distributed data structures
  - **Linda Coordination Language**
  - **An Example**

# Goal of Parallelism

- **To run large and difficult programs fast.**

# Basic Idea

- **One way to solve a problem fast is to break the problem into pieces, and arrange for all of the pieces to be solved simultaneously.**
- **The more pieces, the faster the job goes - upto a point where the pieces become too small to make the effort of breaking-up and distributing worth the bother.**
- **A “parallel program” is a program that uses the breaking up and handing-out approach to solve large or difficult problems.**

# Coordination

- We use the term *coordination* to refer to the process of building programs by gluing together active pieces.
- Each *active piece* is a process, task, thread, or any locus of execution independent of the rest.
- To *glue active pieces together* means to gather them into an ensemble in such a way that we can regard the ensemble itself as the program. The glued pieces are working on the same problem.
- The *glue* must allow these independent activities to communicate and to synchronize with each other exactly as they need to. A coordination language provides this kind of glue.

# Paradigms

- **Result Parallelism**
  - focuses on the shape of the finished product
  - Break the result into components, and assign processes to work on each part of the result
- **Specialist Parallelism**
  - focuses on the make-up of the work crew
  - Collect a group of specialists and assign different parts of the problem to the appropriate specialist
- **Agenda Parallelism**
  - focuses on the list of tasks to be performed
  - Break the problem into an agenda of tasks and assign workers to execute the tasks

# Application of Paradigms to Programming

- **Result Parallelism**
  - Plan a parallel application around the data structures yielded as the ultimate result; we get parallelism by computing all elements of the result simultaneously
- **Specialist Parallelism**
  - We can plan an application around an ensemble of specialists connected in a logical network of some kind. Parallelism results from all nodes of the logical network (all the specialists) being active simultaneously.
- **Agenda Parallelism**
  - We can plan an application around a particular agenda of tasks, and then assign many workers to execute the tasks.
  - Master-slave programs

# Programming Methods

- **Live Data Structures**
  - Build a program in the shape of the data structure that will ultimately be yielded as the result. Each element of this data structure is implicitly a separate process.
  - To communicate, these implicit processes don't exchange messages, they simply refer to each other as elements of some data structure.
- **Message Passing**
  - Create many concurrent processes and enclose every data structure within some process; processes communicate by exchanging messages
  - In order to communicate, processes must send data objects from one local space to another (use explicit send and receive operations)
- **Distributed Data Structures**
  - Many processes share direct access to many data objects or structures
  - Processes communicate and coordinate by leaving data in shared objects

# An Example: N-Body Problem

- **Consider a naive n-body simulator: on each iteration of the simulation we calculate the prevailing forces between each body and all the rest, and update each body's position accordingly.**
- **Assume n bodies and q iterations. Let  $M[i,j]$  contain the position of the i-th body after the j-th iteration**
- **Result Parallelism: Create a live data structure for M, and a function  $\text{position}(i,j)$  that computes the position of body i after the j-th iteration. This function will need to refer to elements of M corresponding the the (j-1)-st iteration.**

# **An Example: N-Body Problem**

- **Agenda Parallelism:** At each iteration, workers repeatedly pull a task out of a distributed bag and compute the corresponding body's new position, referring to a distributed table for information on the previous position of each body. After each computation, a worker might update the table (without erasing information on the previous positions, which may still be needed), or might send newly-computed data to a master process, which updates the table in a single sweep at the end of each iteration.

# An Example: N-Body Problem

- **Specialist Parallelism:** Create one process for each body. On each iteration, the process (specialist) associated with the  $i$ -th body updates its position. It must get previous position information from each other process via message passing. Similarly, it must send its previous position to each other process so that they can update their positions.

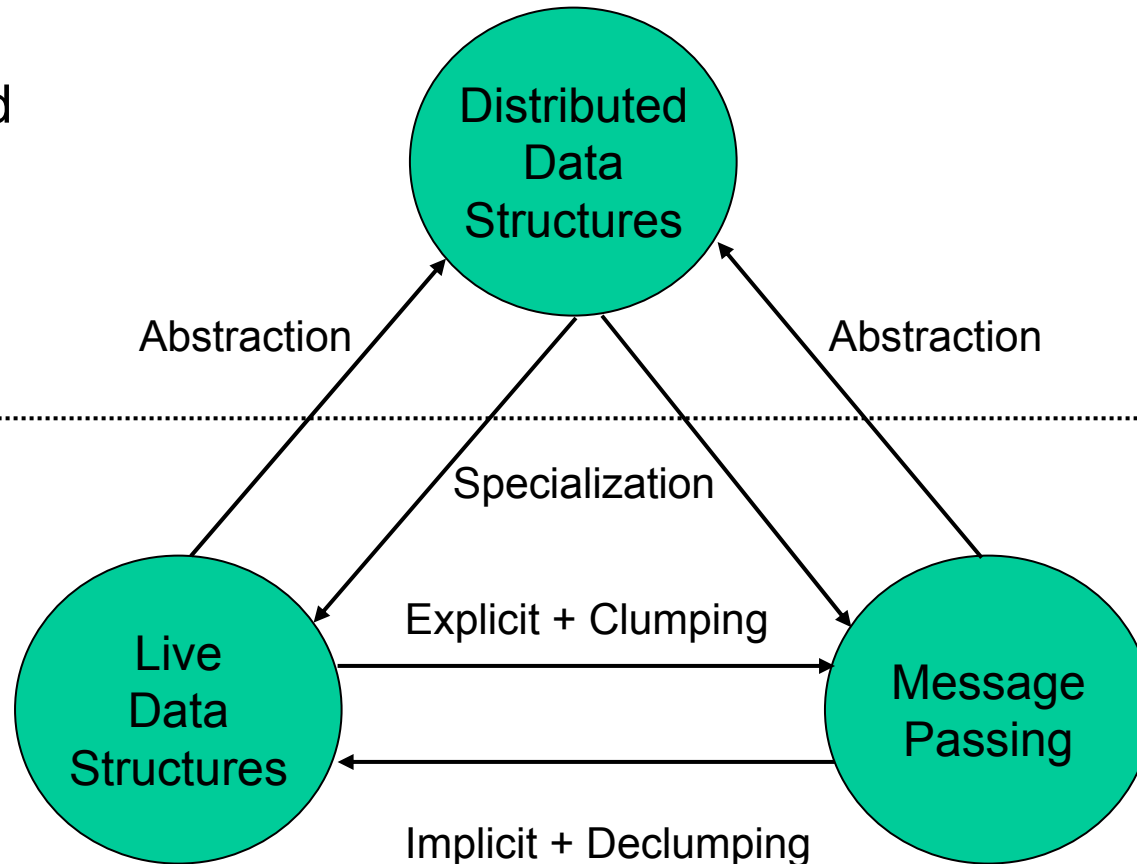
# Methodology

- **To write a parallel program, (1) choose the paradigm that is most natural for the problem, (2) write a program using the method most natural for that paradigm, and (3) if the resulting program isn't acceptably efficient, transform it methodically into a more efficient version by switching from a more natural method to a more efficient one.**

# Program Transformations

Delocalized  
Data  
Objects

Captive  
Data  
Objects



# Transformations for Efficiency

- **Start with result parallelism**
  - many processes
  - fine grained
  - May have too many processes or granularity too small (too little computation to compensate for overhead)
- **Abstract to distributed data structure**
  - each process fills in many elements rather than one process becoming a single element
  - can match the number of processes to environment
- **Specialize to reduce overhead of distributed data structure**
  - clump data elements and localize access to process
  - use explicit message passing to communicate chunks of data
- **Program gets more efficient but also more complicated**

# An Example: N-Body Problem

- **Start with live data structure version**
  - $n \cdot q$  processes
- **Abstract by putting bands of the M matrix into a distributed data structure**
  - number of processes under programmers control
  - lower process management overhead
  - higher granularity
- **Specialize to a message passing program**
  - each band in the distributed data structure is stored in a separate process
  - explicit message passing is now needed for each iteration
  - Eliminate overhead of referring to shared distributed data structure
  - Cost is a more complicated program

# Linda

- **To create parallel programs you must be able to create and coordinate multiple execution threads. Linda is a model of process creation and coordination that is orthogonal to the base language.**
- **Linda is a memory model. Linda memory consists of a collection of logical tuples called *tuplespace***
  - *live tuples* are under active evaluation
  - *data tuples* are passive
- **Live tuples coordinate by generating, reading, and consuming tuples**

# C-Linda

- **Linda is a model, not a tool. A model represents a particular way of thinking about problems.**
- **C-Linda is an instantiation of the Linda model, where the base language is C. Additional operations have been added to support Linda's memory model and process creation and coordination.**
- **See appendix A of Carriero and Gelernter for a summary of C-linda**

# Linda Tuples

- **A *tuple* is a series of typed values**
  - (0,1)
  - (“a string”, 15.01, 17, x)
- **An *anti-tuple* (pattern) is a series of typed fields; some are values (actuals) and some are place holders (formals)**
  - (“a string”, ? f, ? i, y)

# Tuple Operations

- **out(t);**
  - causes the tuple t to be added to tuple space
- **in(s);**
  - causes some tuple t that matches the anti-tuple s to be withdrawn from tuple space.
  - Once a matching tuple t has been found, the actuals in t are assigned to the formals in s.
  - If no matching tuple is found the process suspends until one is available.
  - If multiple tuples match s, then one is chosen arbitrarily.
- **rd(s);**
  - same as in(s), except the matching tuple t remains in tuplespace
- **eval(t);**
  - same as out(t), except t is evaluated after rather than before it is entered in tuple space.
  - Eval implicitly creates one new process to evaluate all fields of t.
  - After all fields have been evaluated, t becomes an ordinary tuple

# Example Tuple operations

- **out("a string", 15.01, 17, x)**
- **out(0,1)**
- **in("a string", ? f, ? i, y)**
- **rd("a string", ? f, ? i, y)**
- **eval("e", 7, exp(7))**
- **rd("e", 7, ? Value)**

# Distributed Data Structures

- **A tuple exists independently of the process that created it, and in fact many tuples may exist independently of many creators, and may collectively form a data structure in tuple space.**
- **Such a data structure is distributed over tuple space**
- **It's convenient to build data structures out of tuples because tuples are referenced associatively somewhat like the tuples in a relational database.**

# Data Structures

- **Structures whose elements are identical or indistinguishable**
  - set of identical elements
  - Not seen in sequential programming
  - used for synchronization
- **Structures whose elements are distinguished by name**
  - records
  - objects
  - sets and multisets
  - associative memories
- **Structures whose elements are distinguished by position**
  - random access: arrays
  - accessed under some ordering: lists, trees, graphs

# Structures with Identical Elements

- **Semaphores**
  - A counting semaphore is a collection of identical elements
  - Initialize to n by executing n out(“sem”) operations
  - V operation is out(“sem”)
  - P operation is in(“sem”)
- **Bag**
  - collection of related, indistinguishable, elements
  - add an element
  - withdraw an element
  
  - Replicated worker program depends on a bag of tasks
    - out(“task”, TaskDescription)
    - in(“task”, ? NewTask)

# Parallel Loop

```
for ( <loop control> )  
  <something>
```

Suppose the function `something()` executes one iteration of the loop body and returns 1.

```
for (<loop control>)  
  eval("this loop", something(<iteration specific arg>));  
for (<loop control>)  
  in("this loop", 1)
```

# Name Accessed Structures

- **Each element of a record can be stored by a tuple**
  - (name, value)
- **To read such a “record field”**
  - rd(name, ? value)
- **To update a “record field”**
  - in(name, ? old)
  - out(name, new)
- **Any process trying to read a distributed record field while it is being updated will block until the update is complete and the tuple is reinstated**

# Barrier Synchronization

- **Each process within some group must wait at a barrier until all processes in the group have reached the barrier, then they can proceed.**
- **A barrier with n processes is initialized with**
  - `out("barrier", n)`
- **Each process reaching the barrier executes**
  - `in("barrier", ? val)`
  - `out("barrier", val - 1)`
  - `rd("barrier", 0)`

# Position Accessed Structures

- **Distributed Array**
  - (Array Name, index fields, value)
  - (“V”, 14, 123.5)
  - (“A”, 12, 18, 5, 123.5)
- **Matrix Multiplication:  $C = A * B$** 
  - (“A”, 1, 1, <first block of A>)
  - (“A”, 1, 2, <second block of A>)
  - ...
- **Workers step through tasks to compute the (i,j) block of C**  
**for (next = 0; next < ColBlocks, next++)**  
**rd(“A”, i, next, ?RowBand[next])**

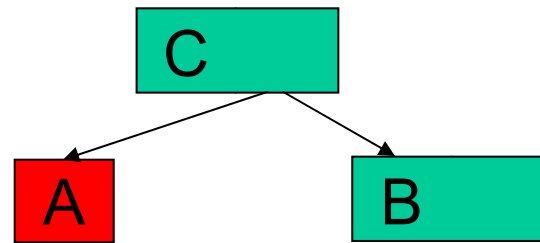
**Similarly read j-th ColBand of B, then produce (i,j) block of C**  
**out(“C”, i, j, Product)**

# Distributed Table

- **Consider a program to compute all primes between 1 and  $n$  which constructs a table of primes**
- **(“primes”, 1, 2)**
- **(“primes”, 2, 3)**
- **(“primes”, 3, 5)**
- **Reading past the end of the table will block until the entry is generated. Suppose a process needs the first  $k$  primes and only  $j < k$  have been generated, then the following blocks**
  - **rd(“primes”,  $j+1$ , ? val)**

# Ordered or Linked Data Structures

- Instead of linking by address, we link by logical name
- A list cell linking A and B
  - The cons cell could be represented by the tuple: ("C", "cons", ["A", "B"])
  - If the cell "A" is an atom we might represent it by the tuple: ("A", "atom", value)



# Streams

- **Ordered sequence of elements to which arbitrary many processes may append**
- **Streams come in two flavors**
  - **in-stream**
    - at any time each of arbitrarily many processes may remove the head element
    - If many processes try to simultaneously remove an element at the stream's head access is serialized arbitrarily at runtime
    - A process that tries to remove from an empty stream blocks
  - **read-stream**
    - Arbitrarily many processes read the stream simultaneously
    - Each reading process reads the stream's first element, then its second and so on...
    - Reading processes block at the end of the stream

# Implementing Streams in Linda

- **Sequence of elements represented by a series of tuples:**
  - (“stream”, 1, val1)
  - (“stream”, 2, val2)
  - ...
- **Index of the last element is kept in a tail-tuple**
  - (“stream”, “tail”, 14)
- **To append**
  - in(“stream”, “tail”, ?index)
  - out(“stream”, “tail”, index+1)
  - out(“stream”, index, NewElement)

# Implementing Streams in Linda

- **An in-stream needs a head tuple to store the index of the head value (next value to be removed)**
- **To remove the head tuple:**
  - `in("stream", "head", ? index);`
  - `out("stream", "head", index+1);`
  - `in("stream", index, ? Element);`
- **When the stream is empty, blocked processes will continue in the order in which they blocked**
- **A read stream dispenses with the head tuple. Each process maintains its own local index**
- **To read each element of the stream**
  - `index = 1;`
  - `<loop> {`
  - `rd("stream", index++, ? Element);`
  - `...`
  - `}`

# More Streams

- **When an in-stream is consumed by only one process, then we can dispense with the head tuple**
- **When a single process appends to a stream, we can dispense with the tail tuple**
- **Streams we have considered are**
  - multi-source, multi-sink; many processes add and remove elements
- **Specializations**
  - multi-source, single-sink; many workers generate data which is consumed by a master process
  - single-source, multi-sink; master produces sequence of tasks for many workers

# Message Passing and Live Data Structures

- **Message Passing**
  - use `eval` to create one process per node in the logical network
  - Communicate through message streams
  - In tightly synchronized message passing protocols (CSP, occam), communicate through single tuples rather than distributed data structures
- **Live data structures**
  - simply use `eval` instead of `out` to create data structure
  - use `eval` to create one process for each element of the live data structure
  - use `rd` or `in` to refer to elements in such a data structure
  - If element is still under active computation, access blocks

# Example: Stream of Processes

- **Execute a sequence of**
  - `eval("live stream", i, f(i));`
- **This creates**
  - ("live stream", 1, <computation of f(1)>)
  - ("live stream", 2, <computation of f(2)>)
  - ("live stream", 3, <computation of f(3)>)
- **Access to a live tuple blocks until computation completes and it becomes passive**
  - `rd("live stream",1, ? x)`
- **blocks until f(1) completes, whereupon it finds the tuple it is looking for and continues**

# **Parallel Processing**

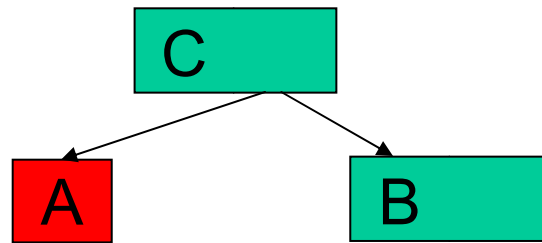
## **Lecture 2: Designing and Implementing Parallel Programs with Linda\***

# Introduction

- **Objective: To implement result, agenda, and specialist parallel algorithms using Linda. To further study the use of Linda as a coordination language. To develop techniques for debugging and measuring performance of Linda programs. To modify Linda programs to obtain greater efficiency.**
- **Topics**
  - **Exercises in coordination of programs using Linda**
    - Bounded Buffer
    - Readers/Writers
  - **Debugging parallel programs**
    - Non-determinism
    - Deadlock
    - Tuplescope
  - **Performance of parallel programs**
    - Speedup, efficiency, and Amdahl's Law
    - Load balancing and work starvation
    - Granularity and communication overhead
  - **An example: finding all primes between 1 and n**

# Ordered or Linked Data Structures

- Instead of linking by address, we link by logical name
- A list cell linking A and B



- The cons cell could be represented by the tuple:  
(“C”, “cons”, [“A”, “B”])
- If the cell “A” is an atom we might represent it by the tuple:  
(“A”, “atom”, value)

# Unbounded Buffer

- **Problem: Two sets of processors called producers and consumers share a common buffer.**
  - Producers put items into the buffer
  - Consumers remove items from the buffer
  - If the buffer is empty consumers must block
- **Solution using streams**
  - Use a multi-source, multi-sink stream
  - This preserves order in which elements are produced
  - If producers are sufficiently ahead of the consumers the amount of items in the buffer may become arbitrarily large

# Implementing Streams in Linda

- **Sequence of elements represented by a series of tuples:**
  - (“stream”, 1, val1)
  - (“stream”, 2, val2)
  - ...
- **Index of the last element is kept in a tail-tuple**
  - (“stream”, “tail”, 14)
- **To append**
  - in(“stream”, “tail”, ?index)
  - out(“stream”, “tail”, index+1)
  - out(“stream”, index, NewElement)

# Implementing Streams in Linda

- **An in-stream needs a head tuple to store the index of the head value (next value to be removed)**
- **To remove the head tuple:**
  - `in("stream", "head", ? index);`
  - `out("stream", "head", index+1);`
  - `in("stream", index, ? Element);`
- **When the stream is empty, blocked processes will continue in the order in which they blocked**

# Bounded Buffer

- **Same as unbounded buffer problem except the size of the buffer is fixed**
- **In this case if the buffer is full producers must block**
- **The buffer can be implemented using a bag or distributed table or stream where the tail and head can differ by no more than the buffer size.**
- **Two approaches for coordination are provided**
  - **use a tuple to count the number of elements in the buffer**
  - **use semaphores to count the number of empty and full slots in the buffer**

# Solution using a Counter

```
int producer(int id)
```

```
{
```

```
    int n;
```

```
    while (TRUE) {
```

```
        in("count",?n);
```

```
        if (n < BUF_SIZE) {
```

```
            out("count",n+1);
```

```
            out("buffer",id);
```

```
        }
```

```
        else
```

```
            out("count",n);
```

```
        }
```

```
    return 1;
```

```
}
```

```
int consumer(void)
```

```
{
```

```
    int n;
```

```
    int item;
```

```
    while (TRUE) {
```

```
        in("count",?n);
```

```
        if (n > 0) {
```

```
            out("count",n-1);
```

```
            in("buffer",?item);
```

```
            printf("Consumed item from Producer %d\n",item);
```

```
        }
```

```
        else
```

```
            out("count",n);
```

```
        }
```

```
    return 1;
```

```
}
```

# Solution using Semaphores

```
int producer(int id)
{
    while (TRUE) {
        in("empty");
        out("buffer",id);
        out("full");
    }
    return 1;
}

int consumer(int id)
{
    while (TRUE) {
        in("full");
        in("buffer",?item);
        printf("Consumer %d consumed item
            %d\n",id,item);
        out("empty");
    }
    return 1;
}
```

# Readers/Writers Problem

- **Many processes share a distributed data structure. Processes are allowed direct access to the data structure but only after they have been given permission.**
- **Many readers or a single writer may have access to the data structure but not both.**
- **A constant stream of read requests may not be allowed to postpone a write request and similarly a constant stream of write requests may not be allowed to postpone a read request.**

# **A Solution to the Readers/Writers Problem**

- **Use a queue of requests - requests are handled in the order they are made. This handles starvation issues.**
- **If the queue's head is a read request, the request is permitted as soon as there are no writers. If the head is a write request, the request is permitted as soon as there are no readers or writers. When the request is granted it is removed from the head, reads/writes, and notifies the system when it is done.**
- **Readers/writers determine on their own when it is permissible to proceed**

# Readers/Writers Solution

- **Instead of creating a distributed queue use 4 shared variables**
  - **rw-tail** - requests added here
  - **rw-head** - requests serviced here
  - **active-readers**
  - **active-writers**

```
startread();  
<read>  
stopread();
```

```
startread()  
{  
    rd("rw-head",incr("rw-tail"));  
    rd("active-writers",0);  
    incr("active-readers");  
    incr("rw-head");  
}
```

# Debugging Linda Programs

- In addition to normal sequential debugging we need to debug the coordination aspects of parallel programs.
- It is helpful to have a tool that allows us to keep track of the coordination state of a parallel program and provides access to sequential debuggers that can be attached to individual processes.
- Linda “Tuplescope” is such a coordination framework debugger. Independent of tuplescope the parallel programmer must think about the coordination that goes on in a parallel program and should design to keep this coordination as simple as possible.

# Logical Issues in Debugging Parallel Programs

- **Deadlock**
  - Suppose we have a state where process A is waiting for a tuple from process B and process B is waiting for a tuple from process A. The program is deadlocked. No further progress can be made.
  - More complex cases can occur where an arbitrarily long cycle of processes and resources exist. Each process in the cycle owns the previous tuple (only process that can generate it) and wants the next one.
  - Can not distinguish from a really slow program or one that has died for other reasons.
  - Can occur in Linda programs when there is no matching tuples for an in
  - Can use tuplescope to detect deadlock

# Logical Issues in Debugging Parallel Programs

- **Non-Determinism**
  - refers to those aspects of a program's behavior that can not be predicted from the source program.
- **Can occur due to the semantics of Linda**
  - don't know which tuple will be returned by an in operation if there is more than one matching tuple
  - If many processes are blocked on similar in statements and an out is produced that matches, some blocked process will get the tuple, but we do not know which one
- **Can occur due the execution model**
  - if tow processes execute asynchronously, we do not know which will finish sooner. The execution order can change from run to run.
- **Make sure the correctness of your program does not depend on a particular order of events when non-determinism can occur.**

# Tuplescope

- **Provides a window on the contents of tuplespace (organized into panes representing disjoint spheres of activity - i.e. different classes of tuples.**
- **As the computation proceeds, tuples appear and disappear, and processes move from pane to pane as their foci changes.**
- **Each of the objects represented can be studied in greater detail by zooming in: contents of data tuples become visible, operations performed by process tuples become visible and on closer scrutiny a sequential debugger becomes available.**

# Snapshot of Tuplescope

- **A data tuple is represented by a round icon**
  - clicking on such an icon reveals its fields
- **A live tuple by a square-ish icon**
  - an arrow pointed upward indicates that the last linda operation was an in
  - an arrow pointed downward indicates that the last linda operation was in out
  - a diamond indicates that the process is blocked
  - clicking on a live tuple icon reveals the source of the last linda operation performed

# **Finding all Primes between 1 and n**

- **This problem will be used to illustrate the basic paradigms for parallelism discussed in the first lecture and to further illustrate parallel programming using Linda.**
- **First Approach: A number is prime if it is not divisible by any of the previous primes less than or equal to its square root. We will use this to obtain a result parallel program, and then we will transform the result parallel program into a more efficient agenda parallel program.**
- **Second approach: Sieve of Eratosthenes - pass a stream of integers through a series of sieves. First remove all multiples of two, and then multiples of three, then five, and so on. An integer that emerges from that last sieve is a prime. This idea will be used to develop a specialist parallel program.**

# Result Parallel Program

- **Build a live data structure which is a vector of integers from 2 to n each executing a function called `is_prime()`.**
  - `for (i=2; i < LIMIT; ++i) {`
  - `eval("primes", i, is_prime(i));`
  - `}`
- **`is_prime()` must read previous elements of this data structure**
  - `rd("primes", j, ? ok);`
- **The main program traverses the resulting distributed vector and counts the number of primes.**
- **Once we know whether  $k$  is prime or not, we can determine in parallel the primality of all numbers between  $k+1$  and  $k^2$ .**

# Agenda Parallel Program

- **The previous program is highly inefficient due to the large number of processes created and the small granularity. We will transform it into an agenda parallel program in order to obtain greater efficiency.**
  - **Instead of using a live vector, create a passive vector, and create worker processes. Each worker will choose some block of vector elements and will fill in the entire block**
  - **Use (“next task”, start) to allocate the task of computing primes between start and start + GRAIN. GRAIN is programmer defined and is a granularity knob for this application**
  - **Use a distributed table of primes instead of a bit vector indicating whether the nth number is prime.**
  - **Need to know how many primes have been computed. Use a master process to receive batches of primes and to build prime table (“primes”, i, <ith prime>, <ith prime squared>)**
  - **Worker processes build local copies of the prime table, referring to the global table only when the local table needs extending (this is essentially a prime cache)**

# Specialist Parallel Program

- **Program organized as an expanding pipeline**
  - The first pipe segment, called source, produces a stream of integers is passed through the pipeline.
  - each segment of the pipeline is a specialist that sieves multiples of a particular prime. The first segment removes multiples of 2, the second segment removes multiples of 3, the third removes multiples of 5 and so on.
  - The last segment, called sink, of the pipeline removes multiples of the largest prime so far.
  - When an integer emerges from the end of the pipe it is determined to be a prime and a new segment at the end of the pipe corresponding to the newly discovered prime is added. eval is used to create new segments.
  - Initially the pipe has only two segments.
  - A single-source single-sink in-stream is used to communicate between stages of the pipeline: (“seg”, <dest>, <stream index>, <int>)
  - Upon termination, signaled, by sending 0 through the pipe, each segment yields its prime (i-th segment contains the i-th prime).

# Performance of the Specialist Program

- The previous program allowed simultaneous checking of all primes between  $k+1$  and  $k^2$  for each new prime  $k$ .
- In this version primes are discovered one at a time.
- Parallelism is obtained from the pipelining of “prime checking”

# Parallel Processing

## Lecture 3: Performance Analysis and Optimization of Linda Programs\*

# Introduction

- **Objective: To develop techniques for measuring performance of Linda programs. To modify Linda programs to obtain greater efficiency. To provide a more extensive example parallel problem solving and programming and to illustrate the steps required to obtain an efficient parallel program.**
- **Topics**
  - **Performance of parallel programs**
    - **Speedup, efficiency, and Amdahl's Law**
    - **Load balancing and work starvation**
    - **Granularity and communication overhead**
    - **Measuring the performance of Linda programs**
  - **A case study (database and matching problem)**
    - **An agenda parallel program for searching a database**
    - **Improving load balance**
    - **A result parallel program for pattern matching**
    - **A result to agenda transformation**
    - **Granularity control and scheduling**
    - **Hybrid search**

# Performance

- **Once a parallel program has been written and debugged it is incumbent to investigate its performance**
- **If it doesn't run faster as more processors are added it is a failure: A "performance bug" that makes a program run too slowly is every bit a bug as a logic bug.**
- **A parallel program that doesn't run fast is, in most cases, is as useful as a parallel program that doesn't run at all.**

# Speedup and Efficiency

- **Speedup - the ratio of sequential time to parallel time**
  - **What do we mean by sequential time?**
    - Same program running on a single processor
    - Fastest sequential program (we want absolute speedup)
  - **Do we compare the same program?**
    - the fastest sequential program may not parallelize well
  - **Do we use the same problem size?**
    - A large program intended to run on a parallel machine may not fit on a sequential machine (memory limitations).
    - A large program may run slow sequentially due to poor cache performance while the parallel program accesses less memory per node and consequently has better performance
    - This may lead to superlinear speedup.
  - **Should we scale sequential performance?**
    - Hard to model performance (especially as problem size increases)
- **Efficiency - the ratio of Speedup to the number of processors**

# Modeling Parallel Performance

- **Time with k processors modeled by**
  - $a/k + b$ , where a is the time for the parallel part of the program and b is the time for the sequential part of the program.
  - Both a and b likely depend on k and problem size
  - In general  $t_{seq} < a + b$  due to parallel overhead (some overhead can be parallelized and occurs in a, while some cannot and occurs in b)
- **When the number of processors is large, speedup is the main concern**
  - Speedup is limited by  $t_{seq}/b \Rightarrow$  must reduce b
  - E.G. parallelize a larger part of the program (parallel I/O)
- **When processors are limited efficiency becomes a concern**
  - Assume  $a/k \gg b$ , then efficiency is limited by  $t_{seq}/a \Rightarrow$  must reduce a (parallel overhead)
  - E.G. reduce overhead of task acquisition, communication overhead
- **Note that a may scale faster than b with problem size**

# Load Balancing and Work Starvation

- **The previous model may not be accurate. In particular when work is not balanced amongst processors**
  - A specialist program with 4 specialists can not attain greater speedup than 4 independent of the number of processors
  - An agenda based parallel program with 36 tasks of equal complexity will finish in the same time on 12 processors as it will on 17. This is an example of *work starvation*.
  - Tasks may vary greatly in size. In the worst case one processor executes one big task while the other processors are idle. To protect against this, order tasks appropriately.
- **Can use different scheduling strategies (static or dynamic) - distribution of parallel components - to obtain a better load balance**
- **Master/worker programs tend to be naturally load balancing**
  - workers grab work as needed

# Granularity

- **If parallel tasks are too small, then the overhead of creating, distributing, and communicating amongst them takes more time than is gained by performing the tasks in parallel.**
- **A message passing program does not perform well if each specialist does not do “enough” between message exchange operations.**
- **A live data structure program won’t perform well if each live data element does “too little” computer**
- **A master-worker program won’t perform well if the task size is “too small”**
- **There is a cross over point where granularity - task size - is too small to compensate for the gains of parallelism. This point must be found in order to obtain an efficient parallel program.**
  - **Depends on communication cost.**
  - **Task size should be programmer controllable - granularity knob**

# Measuring Performance

- Initialize timing facilities with `start_timer()`
- Time events using `timer_split(string)`
- Record times of events using `print_times()`
  
- Example

```
start_timer();  
timer_split("begin computation");  
  <computation>  
timer_split("end computation");  
print_times();
```

# A Case Study

- **Database search with complex search criterion**
  - Match a target DNA sequence with a database of DNA sequences to find the sequence the new one most resembles
- **Perform the search in parallel using a natural agenda parallel program**
  - A sequence of modifications will be applied in order to obtain a more efficient program.
  - The improvements deal with flow control, load balance, and contention
- **Use a natural result parallel program to determine similarity**
  - Transform to an agenda parallel program to control granularity and improve load balance
- **Hybrid search algorithm**

# Databases: Starting with Agenda

- **Main themes:**
  - Load balance is crucial to good performance. The master worker structure is a strong basis for building parallel applications with good load balance characteristics
- **Special Considerations**
  - Watermark techniques can meter the flow of data from a large database to a parallel application
  - Large variance in task size calls for special handling in the interest of good load balance. An ordered task stream is one possible approach.
  - Communication through a single source or sink can produce contention. Task allocation and post processing of worker results can be done in parallel with reduced interaction with the master in order to reduce contention.
  - Parallel I/O can improve performance for large database programs, though we can not count on it.

# Sequential Starting Point

```
While (TRUE) {  
  done = get_next_seq(seq_info);  
  if (done) break;  
  result = compare(target_info, seq_info);  
  update_best(result);  
}  
  
output_best();
```

# First Parallel Version (Worker)

```
char dbe[MAX + HEADER], target[MAX];           /* database entry contains header & DNA sequence */
char *dbs = dbe+HEADER;
/* Work space for a vertical slice of the similarity matrix. */
ENTRY_TYPE col_0[MAX+2], col_1[MAX+2], *cols[2], col_0, col_1];
compare() {
    SIDE_TYPE left_side, top_side;
    rd("target", ? target:t_length);
    left_side_seg_start = target; left_side_seg_end = target + t_length;
    top_side_seg_start = dbs;
    while (1) {                                /* get and process tasks until poison pill */
        in("task", ? db_id, ? dbs:d_length);
        if (!d_length) break;                 /* if poison exit */
        for (i=0; i <= t_length+1; i++) cols[0][i] = cols[1][i] = ZERO_ENTRY; /* zero out column buffer */
        top_side_seg_end = dbs + d_length; max = 0;
        similarity(&top_side, &left_side, cols, 0, &max);
        out("result", db_id, max);
    }
    out("worker done");
}
```

## Second Version

- **Tasks can be created much faster than they are processed. Since the database is large, the tasks may not fit in tuplespace.**
- **This is not a problem in the sequential program.**
- **Could be solved with parallel I/O, but this may not be available.**
- **Use a “high watermark/low watermark” approach to control the amount of sequence data in tuplespace.**
  - **Maintain the number of sequences between an upper and lower limit**
  - **The upper limit ensures that we don’t flood tuplespace**
  - **The lower limit ensures that workers have tasks**
  - **This introduces extra synchronization. To limit the overhead, we only keep an approximate count of outstanding sequences  $\geq$  true number**

# Second Parallel Version (Master)

```
char dbe[MAX + HEADER], target[MAX];           /* database entry contains header & DNA sequence */
char *dbs = dbe+HEADER;
real_main(int argc, char *argv) {
    t_length = get_target(argv[1], target);      /* get argument info. */
    open_db(argv[2]); num_workers = atoi(argv[3]); lower_limit = atoi(argv[4]); upper_limit = atoi(argv[5]);
    for (i=0; i < num_workers; i++) eval("worker", compare()); /* set up */
    out("target", target:t_length); real_max = 0; tasks = 0;
    while(d_length = get_seq(dbe)) {            /* loop putting sequences into tuples. */
        out("task", get_db_id(dbe), dbs:d_length);
        if (++tasks > upper_limit)             /* too many tasks get some results. */
            do {
                in("result", ? Db_id, ? Max);
                if (real_max < max) { real_max = max; real_max_id = db_id; }
            } while (--tasks > lower_limit);
    }
    close_db();
    while (tasks--) {                            /* get and process results */
        in("result", ? db_id, ? max);
        if (real_max < max) { real_max = max; real_max_id = db_id; }
    }
    for (i=0; i < num_workers; i++) out("task", 0, "":0); /* poison tasks */
    print_max(real_max_id, real_max)
}
```

# Quiz One

- **Explain how to keep an exact count**
- **Sketch a solution that works in this way: at any given time, tuplespace holds at most  $n$  sequences, where  $n$  is the number of workers.**

# Improving Load Balance

- **Potential for a wide variance of task size**
  - Suppose we have lots of small tasks and one very big task. Further suppose that the big task is allocated to the last worker. While this worker plugs away on the long task, every other worker sits idle.
  - Example:
    - $10^6$  bases in database
    - longest single sequence has  $10^4$  bases
    - Rest of the sequences have approximately 100 bases
    - 20 workers
    - Assume time is proportional to the number of bases processed
    - One worker (assigned the big task) processes 60,000 bases
    - The remaining workers process 50,000 bases
    - Speedup =  $10^6/60000 \approx 16.7$  as compared to the ideal 20
- **Solution**
  - Order tasks by size using a single-source multi-sink stream
  - Here is a case where efficiency is obtained by adding synchronization

# Reducing Contention

- **With many workers, access to the shared index tuple may lead to contention**
- **Task assignment overhead**
  - Assume 10 workers and 100 time units for tasks with 1 time unit per index tuple access
  - First worker gets first task at time 1 and second task at time 101, second worker gets first task at time 2 and second task at time 102,...
  - 1% task-assignment overhead
  - With two hundred workers the first round of work assignment does not complete until time 200.
  - On average, half the workers will be idle waiting for a task tuple
  - Drop off in performance seen at 100 workers
- **Solution**
  - Use multiple task tuples (sets of workers access different tuples)

# Parallel Result Computation

- **Currently the master process is responsible for all result computation (i.e. update of max). Consequently this part of the program is sequentialized.**
- **Solution**
  - **Have individual workers update a local max**
  - **Only when workers are finished do they send their result to the master**
  - **At this time the master has to process results sequentially, but there is only one result per worker rather than one result per task.**

# Final Parallel Version (Master)

```
char dbe[MAX + HEADER], target[MAX];           /* database entry contains header & DNA sequence */
char *dbs = dbe+HEADER;
real_main(int argc, char *argv) {
    t_length = get_target(argv[1], target);     /* get argument info. */
    open_db(argv[2]); num_workers = atoi(argv[3]); lower_limit = atoi(argv[4]); upper_limit = atoi(argv[5]);
    for (i=0; i < num_workers; i++) eval("worker", compare()); /* set up */
    out("target", target:t_length); out("index", 1); tasks = 0; task_id = 0;
    while(d_length = get_seq(dbe)) {           /* loop putting sequences into tuples. */
        out("task", ++task_id, get_db_id(dbe), dbs:d_length);
        if (++tasks > upper_limit)           /* too many tasks get some results. */
            do { in("task done"); } while (--tasks > lower_limit);
    }
    for (i=0; i < num_workers; i++) out("task", ++task_id, 0, "":0); /* poison tasks */
    close_db();
    while (tasks--) in("task done");          /* clean up */
    real_max = 0;                             /* get and process results */
    for (l=0; l < num_workers; ++l) {
        in("worker done", ? db_id, ? max);
        if (real_max < max) { real_max = max; real_max_id = db_id; }
    }
    print_max(real_max_id, real_max)
}
```

# Final Parallel Version (Worker)

```
char dbe[MAX + HEADER], target[MAX];           /* database entry contains header & DNA sequence */
char *dbs = dbe+HEADER;
/* Work space for a vertical slice of the similarity matrix. */
ENTRY_TYPE col_0[MAX+2], col_1[MAX+2], *cols[2], col_0, col_1];
compare() {
    SIDE_TYPE left_side, top_side;
    rd("target", ? target:t_length);
    left_side_seg_start = target; left_side_seg_end = target + t_length;
    top_side_seg_start = dbs;
    local_max = 0;
    while (1) {                                /* get and process tasks until poison pill */
        in("index". ? Task_id); out("index", task_id + 1);
        in("task", task_id, ? db_id, ? dbs:d_length);
        if (!d_length) break;                 /* if poison task, dump local max and exit */
        for (i=0; i <= t_length+1; i++) cols[0][i] = cols[1][i] = ZERO_ENTRY; /* zero out column buffer */
        top_side_seg_end = dbs + d_length; max = 0;
        similarity(&top_side, &left_side, cols, 0, &max);
        out("task done");
        if (max > local_max) { local_max = max; local_max_id = db_id; }
    }
    out("worker done", local_max_id, local_max);
}
```

# Performance Analysis

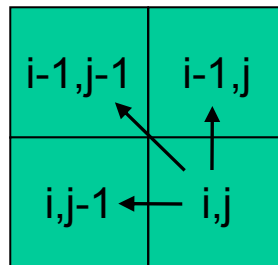
- **Sequential code has two main components**
  - I/O (linear function of length of the database)
  - Comparisons (proportional to product of lengths of target and database)
- **Parallel code has same I/O cost, but it is overlapped with computation. Likewise each comparison takes the same amount of time, but many comparisons are done in parallel.**
- **Parallel overhead (D = # tasks, K = # workers)**
  - D + K task outs, D + K index in/out's, D + K task ins, K result in/out's
  - $T_{\text{Synch}}$  proportional to D (assume  $D > K$ )
- **Parallel time**
  - $\max(t_{\text{IO}}, t_{\text{Seq}}/K + t_{\text{TO}} * (D/K), t_{\text{Synch}} * D)$
  - IO = input/output, TO = parallelizable task overhead, Seq = sequential computation, Synch = non-parallelizable task overhead,

# Parallel Comparison using Matrices Starting with Result

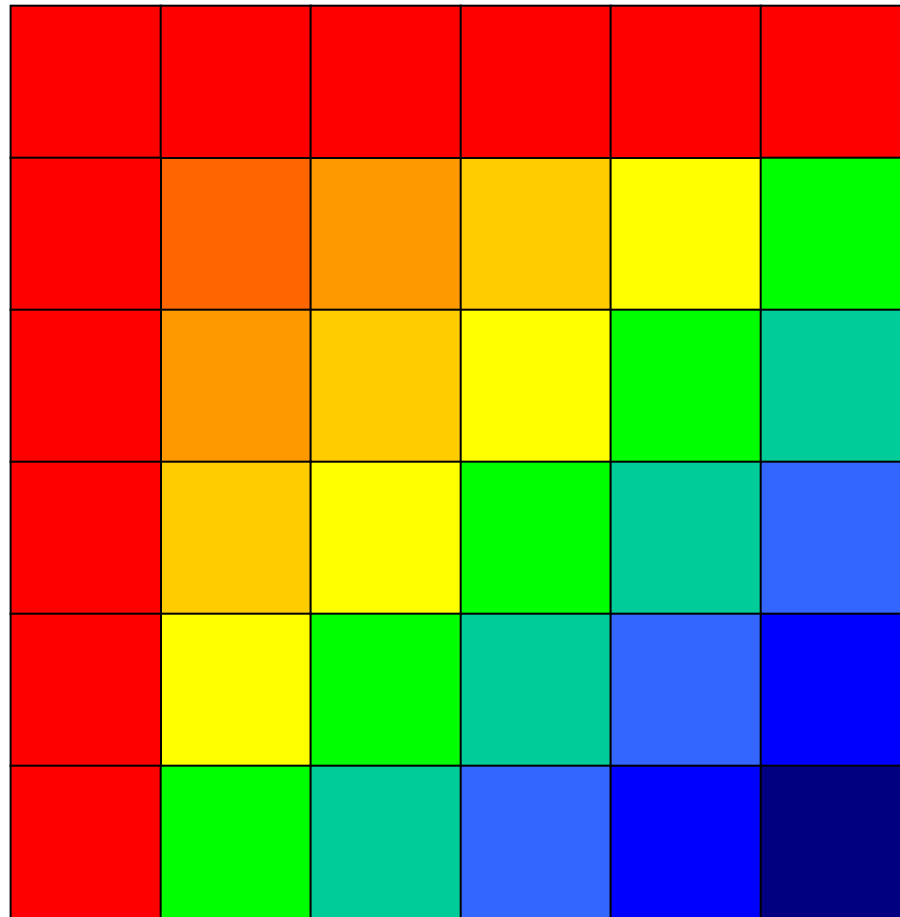
- **Main themes:**
  - Load balance is again crucial to good performance. The need for good load balance motivates our transformation from a result to an agenda parallel strategy.
  - Granularity control is the other crucial issue. Here, we use the size of a matrix sub-block as a granularity knob.
- **Special Considerations**
  - Matrix sub-blocking is a powerful technique for building efficient matrix computations
  - Logical inter-dependencies among sub-computations need to be understood and taken into account in building efficient programs. Here, we focus on an application with “wavefront” type dependencies.

# Comparison Problem

- **String comparison algorithm for DNA sequences**
- **Involves filling in a matrix, where each matrix element depends either on the input or a previously computed result**
  - Given a function  $h(x,y)$ , we need to compute a matrix  $H$  such that  $H[i,j] = h(i,j)$
  - $h(i,j)$  depends on  $h(i-1,j)$ ,  $h(i,j-1)$ ,  $h(i-1,j-1)$
  - Initial values  $h(0,j)$  and  $h(i,0)$  for all  $i$  and  $j$  are given.
  - In our problem the initial values are the two strings we want to compare



# Wavefront Computation



Parallel Processing

# Result Parallel Program

```
typedef struct entry {
    int d, max, p, q;
} ENTRY_TYPE;
ENTRY_TYPE zero_entry = {0, 0, 0, 0};
#define ALPHA 4
#define BETA 1
char side_seq[MAX], top_seq[MAX];

real_main(int argc, char *argv) {
    ENTRY_TYPE compare(), max_entry;
    int i, j, side_len, top_len;

    side_len = get_target(argv[1], side_seq);
    top_len = get_target(argv[2], top_seq);
    for (i = 0; i < side_len; ++i)
        for (j=0; j < top_len; ++j)
            eval("H", i, j, compare(i,j,side_seq[i], top_seq[j]));
    in("H", side_len - 1, top_len - 1, ? max_entry);
    printf("max: %d", max_entry.max);
}
```

# Comparison Function

```
ENTRY_TYPE compare(int i, int j, ENTRY_TYPE b_i, ENTRY_TYPE b_j) {
    ENTRY_TYPE d, p, q, me;
    int t;

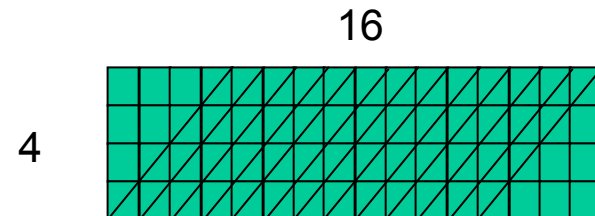
    d = p = q = zero_entry;
    if (i) rd("H", i-1, j, ?q);
    if (j) rd("H", i, j-1, ?p);
    if (i && j) rd("H", i-1, j-1, ? d);
    me.d = d.d + match_weights[b_i & 0xf][b_j & 0xf]; if (me.d < 0) ,me.d = 0;
    me.p = pe.d - ALPHA; t = p.p - BETA; if (me.p < t) ,me.p = t;
    me.q = q.d - ALPHA; t = q.q - BETA; if (me.q < t) me.q = t;
    me.d = q.d - ALPHA; t = q.d - BETA; if (me.d < t) me.d = t;
    if (me.p > me.d) me.d = me.p;
    if (me.q > me.d) me.d = me.q;
    me.max = me.d;
    if (d.max > me.max) me.max = d.max;
    if (p.max > me.max) me.max = p.max;
    if (q.max > me.max) me.max = q.max;
    return me;
}
```

# Result $\Rightarrow$ Agenda Transformation

- **Shortcomings of the result parallel program**
  - High communication to computation ratio
  - Granularity too small - 3 rd's for a small computation
  - Poor load balance due to dependencies - to compare two length  $n$  sequences on  $n$  processors the best speedup is approximately  $n/2 \Rightarrow$  50% efficiency (in practice may be less due to additional overhead)
  - Startup and shutdown costs (it is not until step  $K$  that there are  $K$  parallel tasks - the same phenomenon occurs as the computation winds down)
- **Solution**
  - sub-block computation - a block of size  $n \times n$  depends on  $n$  elements from the left block,  $n$  elements from the upper block and 1 element from the upper-left block ( $n^2$  computations and  $2n+1$  communications)
  - A result parallel program is inefficient due to the number of processes created. If the underlying system handles process creation and load balance well, this is ok, but it is safer to abstract and use an agenda program where sub-blocks form tasks that are allocated to a worker pool

# Sub-Block Shape

- **Efficiency can be controlled by changing the shape of subblocks (aspect ratio)**
- **Assume a sequences of size m and n with  $m < n$** 
  - **Parallel time with m workers =  $(m-1) + (n - (m-1)) + (m-1) = 2m+(n-m) - 1 \approx m + n$  for  $m,n \gg 1$ .**
  - **Speedup =  $S = t_{seq}/t_{par} = mn/(m+n)$ , if  $n \gg m$ ,  $S \approx m$ .**
  - **Let  $\alpha = n/m$  (aspect ratio) and suppose there are m workers**
  - **$S = (\alpha/(1+ \alpha)) * m$ , with  $\alpha = 10$ , efficiency  $\approx 90\%$**
  - **Choose sub-block size high enough to use all workers and wide enough to control startup and shutdown cost**
  - **To use W workers at 90% efficiency**
  - **sub-block size =  $m/W \times n/(10W)$**



# Task Scheduling

- **Could begin with a single enabled task (upper-left) where workers generate tasks dynamically as they become enabled.**
- **Alternatively create a worker for each band of the block matrix.**
  - **Reduces task allocation overhead**
  - **Reduces communication**
  - **As a task completes, its right and bottom edges need to be communicated. The right edge remains with the current worker and only the bottom needs to be put in tuple space**
  - **As soon as the first task for the worker in the first band is completed the second worker may start and so on.**
  - **To improve load balance any extra rows are distributed evenly amongst workers by increasing block width by one.**

# Agenda Parallel Program (Master)

```
char side_seq[MAX], top_seq[MAX];

real_main(int argc, char *argv) {
    side_len = get_target(argv[1], side_seq);
    top_len = get_target(argv[2], top_seq);
    num_workers = atoi(argv[3]); aspect_ratio = atoi(argv[4]);
    for (i=0; i < num_workers; ++i) eval("worker", compare());
    out("top sequence", top_seq:top_len);
    height = side_len/num_workers; left_over = side_len - (height*num_workers); ++height;
    for (i=0; sp = side_seq; i < num_workers; ++i, sp += height) {
        if (i == left_over) --height;
        out("task", i, num_workers, aspect_ratio, sp:height);
    }
    real_max = 0;
    for (i=0; i < num_workers; ++i) {
        in("result", ? max); if (max > real_max) real_max = max;
    }
    printf("max: %d", max_entry.max);
}
```

# Agenda Parallel Program (Worker)

```
char side_seq[MAX], top_seq[MAX];
ENTRY_TYPE col_0[MAX+2], col_1[MAX+2]. *cols={cp;_0, col_1};
ENTRY_TYPE top_edge[MAX];
compare() {
    SIDE_TYPE left_side, top_side;
    rd("top sequence", ?top_seq:top_length); top_side.seq_start = top_side.seq_start;
    in("task", ? id, ? num_workers, ? aspect_ratio, ?side_seq:height);
    left_side.seq_start = side_seq; left_side.seq_end = left_side.seq_start + height;
    for (i=0; i <= height+1; ++i) cols[0][i] = cols[1][i] = ZERO_ENTRY; /* zero out column buffers */
    max = 0; blocks = aspect_ratio * num_workers; width = top_len/blocks; left_over = top_len - (width*blocks); ++width;
    /* loop across top sequence, stride is width of a sub-block */
    for (block_id = 0; block_id < blocks; ++block_id, top_side.seq_start += width) {
        if (block_id == left_over) --width;
        top_side.seq_end = top_side.seq_start + width;
        if (id) in("top edge", id, block_id, ? Top_edge:);
        else for (i=0; i < width; ++i) top_edge[i] = ZERO_ENTRY;
        similarity(&top_side, &left_side, cols, top_edge, &max);
        if (id+1) < num_workers) out("top edge", id+1, block_id, top_edge:width); /* send bottom edge (in reality overwritten top edge) */
    }
    out("result", max);
}
```

# Hybrid Search

- **Comparison can be sped up using the parallelized comparison**
- **If many comparisons are done, we can overlap the previous shutdown phase with the startup of the next**
  - **processors that would normally be idle during the last few comparisons of one sub-block were being performed could be used with the first few comparisons of the next sub-block.**
  - **As a result, we pay the startup and shutdown costs once over the whole database**
- **We can combine the parallel search with parallelized search.**
  - **Prefer performing comparisons in parallel to a parallelized search (less overhead) unless needed (i.e. very large individual task).**
  - **Make block size sufficiently large so that we only pay overhead when needed**