Distributed Computing Paradigms

- Bag of Tasks
- Heartbeat Algorithms
- Pipelining

Bag of Tasks (also called Administrator/Workers)

- Basic idea
  - Server maintains a list (bag) of tasks to be performed
  - Clients send requests to server, receive a task, then may send back additional tasks to server
  - Implements recursive parallelism on a distributed-memory machine
- Main advantage
  - Automatic load balancing
    - When client needs work, gets some from server

Bag of Tasks Pseudocode

```
struct task {
   field1; field2; …; fieldN
}
process Manager {
   create initial task
   taskQ.enqueue(task)
   while (1) {
     in getTask(&task) such that taskQ.size > 0
       task = taskQ.dequeue( )
     [] putTask(task)
       taskQ.enqueue(task)
     [] result(val)
       incorporate val into final answer
     ni
   }
}
```

task is a reference parameter to getTask( )

Bag of Tasks Pseudocode

```
process Worker {
  while (1) {
    call getTask(&task)  // procedure call---blocks
    work on task
    if (this task does not need to be subdivided further)
      send result(val)   // val is whatever the "answer" is
    else {    // send task or tasks back into the bag
      send putTask(task1)  // send msg---doesn't block
      send putTask(task2)  // send msg---doesn't block
      ...
      send putTask(taskN) // send msg---doesn't block
    }
  }
}
```

Bag of Tasks Details

- Worker should keep a task for itself
  - When it finishes its task, it needs another
  - Relatively simple to restructure code on last two slides to do this
- Want to avoid too many tasks
  - Just as in recursive parallelism
- Termination is in deadlock
  - Some languages allow this (they detect deadlock)
  - Can modify to terminate normally
    - Typically uses one sentinel task per worker

Heartbeat Algorithms

- Useful for data parallel, iterative applications
- Basic outline:

  initialize local variables

  while not done

      send values to neighbors

      receive values from neighbors

      perform local computations

Example: Region Labeling

- Assume that we are given an image with a given number of lit pixels
- Goal: find sets of connected lit pixels; give each set a unique label

Region Labeling: Worker Outline

process worker [k = 0 to P-1] {

double pixel[N/P][N], label[N/P][N]

initialize pixel and label arrays in my strip (the label elements are zero if pixel off and unique if pixel on)

send/receive top/bottom rows of *pixel array* to neighbor

while not done

For this implementation, can optimize to only send top row, but this is not true generally and so not shown here.

send my top and bottom rows of *label array* to neighbor

receive my neighbor's top and bottom rows into my *label array*

update label array in my strip (see next slide)

send "yes" or "no" to coordinator, depending on if any elements of my label array changed

receive whether to continue from coordinator

} // end of worker

Need two rows of extra storage

Region Labeling: Updating Label Array

Updating label array is done as follows:

for each of my pixels

if pixel[i][j] == 1

North/East/West/South

for each (N/E/W/S) neighbor who has pixel[i][j] == 1

if neighbor's label[i][j] > my label[i][j]

replace my label[i][j] with neighbor's

Note: this algorithm takes time N^2 if there is a "snake"

Region Labeling: Coordinator

```
process coordinator {
  for each iteration
     collect change flag from each process (“yes” or “no”)
     if at least one process sends “yes”
        send “go on” to each process
     else
        send “done” to each process and exit
}   // end of coordinator
```

Can also be done via a global reduction (e.g., Allreduce in MPI)
--do not actually need a separate coordinator process

Region Labeling—Overlapping Communication and Computation

```
process worker [k = 0 to P-1] {
  …same as previous version of worker process…
  while not done
      send top/bottom rows of label to neighbor
      update label array based only on local data
      receive top/bottom rows of label from neighbor
      update label array based on the top and bottom rows
      if any label[i][j] changed, send "yes" to coordinator
      receive whether to continue from coordinator
}  // end of worker
```

Send must be nonblocking

(New parts in blue; unchanged in gray)

Pipelining

- Another parallel programming paradigm
- First must explain strip mining

An Example Nested Loop

- Original code

```
for i = 1 to n {
   for j = 1 to n {          (j loop is parallel)
      A[i][j] = f(A[i-1][j])
   }
}
```

Problem is that j loop is in many cases too fine-grain to parallelize effectively

An Example Nested Loop

- Could try loop interchange

```
for j = 1 to n {
    for i = 1 to n {
        A[i][j] = f(A[i-1][j])
    }
}
```

(Interchange loops; legal in this case)

Problem is that now with parallelizing j loop is one of efficiency
-- Traversal proceeds down the columns, which is inefficient (cache)

Strip Mining

- First step

```
for i = 1 to n {
    for k = 1 to n by blocksize {
        for j = k to k + blocksize - 1 {
            A[i][j] = f(A[i-1][j])
        }
    }
}
```

(Add an extra loop)

Adding extra loops is generally not a good idea! What's going on?

Strip Mining

- Second step

```
for k = 1 to n by blocksize {
    for i = 1 to n  {                    (Interchange first two loops)
        for j = k to k + blocksize - 1 {
            A[i][j] = f(A[i-1][j])
        }
    }
}
```

Still, we transformed two loops into three. Again…why is this a good idea?

Pipelining

```
for i = 1 to n
    for j = 1 to n
        A[i][j] = f(A[i][j-1])
for i = 1 to n
    for j = 1 to n
        A[i][j] = f(A[i-1][j])
```

How do we parallelize this program?

Options to Parallelize

- Transpose array between first and second loop nests (and between second and first loop nests, if both loops are nested within another loop)
  - Advantage: full parallelization in both loops
  - Disadvantage: transpose is slow (requires all-to-all)
- Sequentialize second loop nest
  - Bad idea: requires data movement and is sequential
- Pipeline second phase
  - Avoids transpose, but suffers pipeline latency and requires a larger number of messages

Pipelining Step 1: parallelize first loop

```
for i = start to end
    for j = 1 to n
        A[i][j] = f(A[i][j-1])
for i = 1 to n
    for j = 1 to n
        A[i][j] = f(A[i-1][j])
```

Standard parallelization of first loop

Pipelining Step 2: rewrite second loop nest via strip mining

```
for i = start to end
    for j = 1 to n
      A[i][j] = f(A[i][j-1])
for k = 1 to n by blocksize
  for i = 1 to n
    for j = k to k + blocksize - 1
      A[i][j] = f(A[i-1][j])
```

We transformed two loops into three via strip mining. Again, why is this a good idea?

Pipelining Step 3: partition middle loop in second loop nest

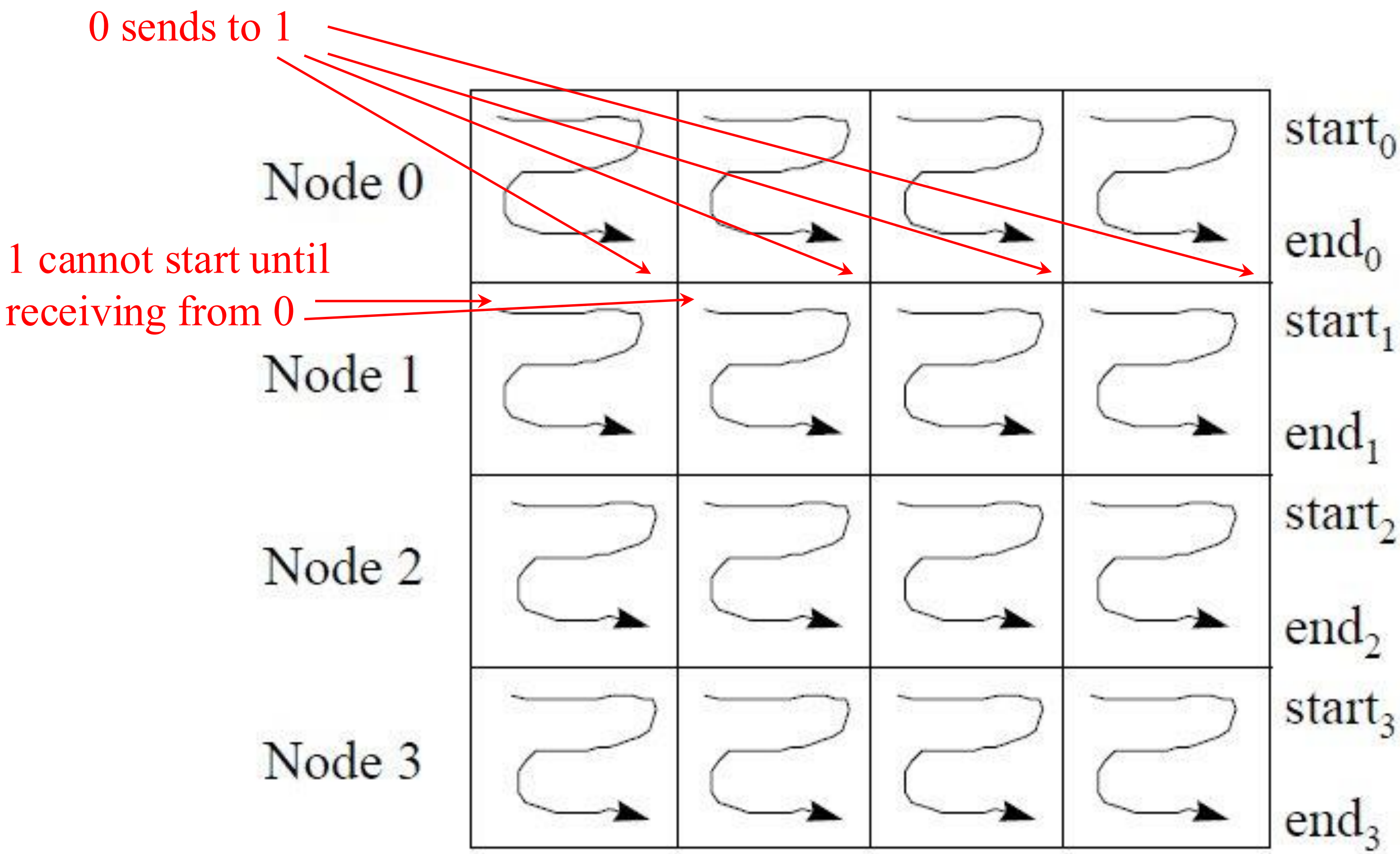
```
for i = start to end
    for j = 1 to n
      A[i][j] = f(A[i][j-1])
for k = 1 to n by blocksize
  for i = start to end
    for j = k to k + blocksize - 1
      A[i][j] = f(A[i-1][j])
```

Note the parallelization is not in the outermost loop

Pipelining Step 4: Insert Synchronization/Communication (code is for each process X, distributed memory, not a "boundary process" [not the first or the last])

```
for i = start to end
    for j = 1 to n
        A[i][j] = f(A[i][j-1])
for k = 1 to n by blocksize
    receive predecessor sub-row from process X-1
    for i = start to end
        for j = k to k + blocksize - 1
            A[i][j] = f(A[i-1][j])
    send bottom sub-row just computed to process X+1
```

Picture of pipelining



In parallel, Node 0 computes second block and Node 1 computes first block
In parallel, Node 0 computes third block, Node 1 second block, and Node 2 first block

Pipelining Step 4: Insert Synchronization/Communication (Code is for each thread X, shared memory, not a "boundary thread" [not the first or the last])

```
(Global) sem s[0:numThreads-1] = {0,0,0…,0}
for i = start to end
    for j = 1 to n
        A[i][j] = f(A[i][j-1])
  for k = 1 to n by blocksize
    P(s[X-1])
    for i = start to end
        for j = k to k + blocksize - 1
            A[i][j] = f(A[i-1][j])
    V(s[X]))
```