ID1217 Concurrent Programming
Lecture 5

ROYAL INSTITUTE
OF TECHNOLOGY

Barriers. Flags.
Data Parallel Algorithms

Vladimir Vlassov

KTH/EECS

ROYAL INSTITUTE OF TECHNOLOGY

Outline

- Barriers
  - Counter barrier
  - Coordinator barrier
    - Point-to-point flag synchronization
  - Combining tree barrier
  - Symmetric barriers
- Data parallel algorithms. Examples:
  - The heat problem (Computation on a grid of points)
- Parallel computing with a Bag of Tasks

ROYAL INSTITUTE OF TECHNOLOGY

Need for Barriers

- In iterative computations, often inner loops (e.g. over data) can be parallelized, whereas an outer loop (e.g. over time steps) cannot.
  - True dependencies between computation steps or stages
- Need for barriers:
  - The next computation stage needs results computed by several processes on the previous stage(s)
    - The next stage can start only after all processes complete the previous stage.
  - Barrier synchronization among processes is required at the end of computation stages.

Barrier Synchronization

- A ***barrier*** is a point that all processes must reach before any proceed

```
boolean done = false;
process P[i = 0 to n - 1] {
  while (!done)  {
     code to implement task i;
     Barrier(i, n); // Wait for all tasks to complete
  }
}
```

- Barriers
  - Centralized: Counter barrier, Coordinator barrier
  - Decentralized: Combining tree barrier, Symmetric barriers
- Barriers can be implemented using:
  - Locks, flags, counters – busy-waiting at a barrier
  - Locks and condition variables – blocking at a barrier
  - Semaphores – blocking at a barrier

ROYAL INSTITUTE OF TECHNOLOGY

Counter Barrier

- Uses a shared counter to count processes arrived at the barrier

```
int count = 0;
Barrier (int n) {
   < count++; >          /* arrival */
   while (count < n); /* wait for others */
}
```

- Lock or atomic Fetch&Increment can be used for atomic increment of the counter
- How to reuse for next rounds?
  - Sense reversal technique (on the next slide)

ROYAL INSTITUTE OF TECHNOLOGY

Reusable Counter Barrier with Sense Reversal

- Wait for a binary flag to take different value consecutive times
- Toggle this value only when all processes reach the instance of the barrier

```
int count = 0;                          /* shared counter */
boolean go = false;                     /* shared flag to go */
Barrier (int n, int id) {
  static bool localSense[1:n];          /* sense is private */
  int c;                                /* local copy of the counter */
  localSense[id] = !localSense[id];          /* toggle private sense */
  <count++; c = count;>                      /* count arrivals */
  if (c == n) {                         /* last to arrive */
          count = 0;                    /* reset for next barrier */
          go = localSense[id];               /* release waiters */
  } else
    while (go != localSense[id]) continue; /* busy wait for release */
}
```

ROYAL INSTITUTE OF TECHNOLOGY

Evaluation of the Counter Barrier

- Storage cost is low: one counter (with a lock) and one flag
- Fairness
  - Same processor should not always be last to exit barrier
  - No such bias in centralized
- The counter barrier is efficient if the machine has
  - An atomic fetch&op instruction
  - Coherent caches (in a multiprocessor) – to spin on the go flag
- The main problem: memory contention
  - All processes increment counter
  - In the worst case, n-1 processes spin on a cached copy of the go flag, but
  - All n-1 read flag when it's written by the very last process arrived at the barrier

ROYAL INSTITUTE OF TECHNOLOGY

Point-to-Point Flag Synchronization

- **Flag synchronization** delays a proc until a binary flag is set (or cleared) by another proc.
  - A point-to-point signaling mechanism used as condition synchronization (to wait for a condition, to signal the condition).

ROYAL INSTITUTE OF TECHNOLOGY

Flag Synchronization (cont'd)

- Flag reuse rules:
  - The process that waits for a flag to be set (to 1) is the one that should reset it (to 0).
  - The flag should not be set (to 1) until it is known that it is reset (to 0), i.e. to set the flag (to 1), the process waits until it's reset (to 0).
- Signaling process setting the flag (sends the signal):

```
while (flag) continue; flag = 1;
```

- Signaled process clearing the flag (receives the signal):

```
while (!flag) continue; flag = 0;
```

ROYAL INSTITUTE OF TECHNOLOGY

Use of Flags for Barriers

- Idea:
  - To signal arrival, use ***an array of arrival flags*** (one flag per process) instead of the shared counter;
    - Each proc set its arrival flag when it arrives at the barrier.
  - To release, use ***an array of release flags*** (one flag per process) instead of one release flag.
    - Each proc waits for its release flag to be set to exit the barrier.
  - To avoid false sharing, store flags in different memory blocks.
  - A coordinator proc can be used to control the barrier.
- This approach, allows to avoid contention for the centralized counter and the release flag.

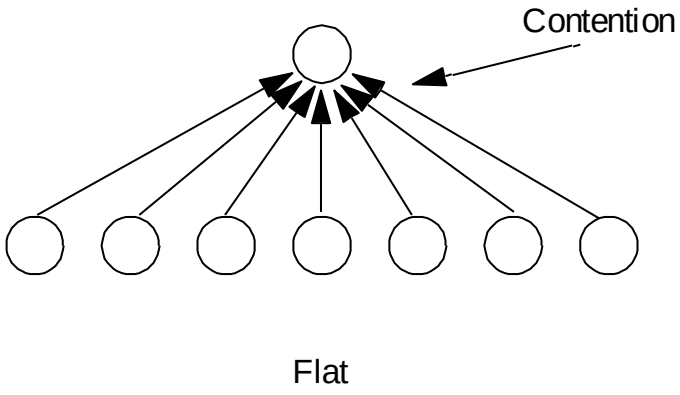
Coordinator Barrier

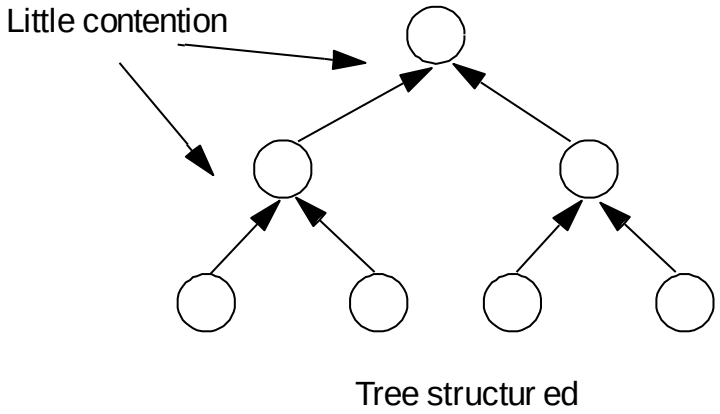
- The Coordinator process waits for all arrival flags to be set, then sets continue flags to release processes at the barrier

```
int arrive[1:n] = ([n] 0),  continue[1:n] = ([n] 0);
process Worker[i = 1 to n] {
  while (true) {
    code to implement task i;
    arrive[i] = 1;
    ⟨await (continue[i] == 1);⟩
    continue[i] = 0;
  }
}
process Coordinator {
  while (true) {
    for [i = 1 to n] {
      ⟨await (arrive[i] == 1);⟩
      arrive[i] = 0;
      }
    for [i = 1 to n] continue[i] = 1;
  }
}
```

Shortcomings of the Coordinator Barrier

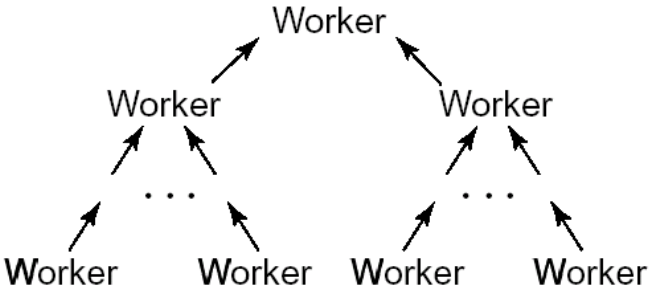
- Extra process (on its own processor), 2n flags
- Pure scalability: waiting and releasing time proportional to the number of processes in the barrier O(n)
- Enhancement: do not use Coordinator, distribute its work among processes (Workers)
  - Combining tree barrier
  - Symmetric barriers





Combining Tree Barrier

- ***Combining tree*** is a tree of processes where each proc combines results of its children (if any), and passes to its parent (if any).
- ***Combining tree barrier***
  - Arrival signals (flags) are sent up the tree (i.e. from leaves up to the root), continue signals are sent down the tree (i.e. from the root to all the nodes)
- A worker node at the barrier
  - Waits for signals from children (if any),
  - Signals parent (if any) that it too has arrived,
  - Waits for a continue signal from parent (if any),
  - Signals its children (if any) to continue
  - Exits the barrier.



Code of Combining (Binary) Tree Barrier

leaf node `L:`

```
arrive[L] = 1;
⟨await (continue[L] == 1);⟩
continue[L] = 0;
```

interior node `I:`

```
⟨await (arrive[left] == 1);⟩
arrive[left] = 0;
⟨await (arrive[right] == 1);⟩
arrive[right] = 0;
arrive[I] = 1;
⟨await (continue[I] == 1);⟩
continue[I] = 0;
continue[left] = 1; continue[right] = 1;
```

root node `R:`

```
⟨await (arrive[left] == 1);⟩
arrive[left] = 0;
⟨await (arrive[right] == 1);⟩
arrive[right] = 0;
continue[left] = 1; continue[right] = 1;
```

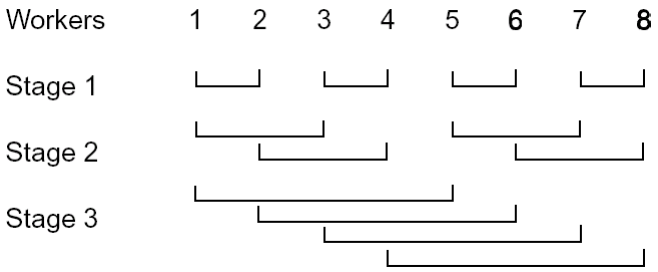
ROYAL INSTITUTE OF TECHNOLOGY

Evaluation of the Combining Tree Barrier

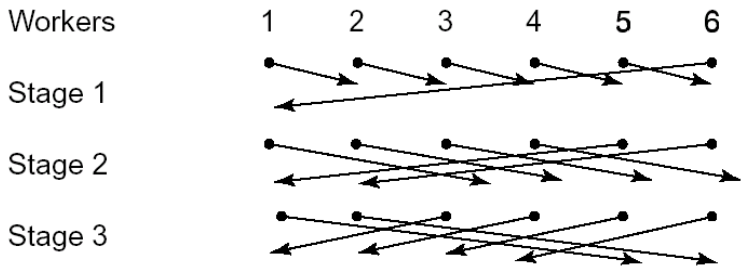
- $2n$ flags – the same number of flags as in the Coordinator barrier
- Latency is $O(\log n)$
- Shortcomings:
  - Too many signals – not good for bus-based SMP
  - Asymmetric and harder to program
- Design options:
  - Use one continue flag for all waiters.
    - Set by the Worker at the root of the tree
    - Alternate sense of the flag for reuse
  - Use two continue flags and alternate them for reuse

Symmetric Barriers

- Idea:
  - Construct an *n*-process barrier using **2-process barriers** passed in several rounds.
  - Change pairs in 2-proc barriers in each round so that after **log *n* rounds** all processes are synchronized, i.e. that are passed the barrier.
  - At the ***n*-process barrier**, each proc should synchronize in total with log *n* others.
- Symmetric barriers differ in their schemes of pairing



Butterfly barrier



Dissemination barrier

ROYAL INSTITUTE OF TECHNOLOGY

A Two-Process Barrier

- A proc signals its arrival by setting a flag, and then waits for another flag to be set (another proc to arrive)

```
// butterfly
// Worker[i]:
 …
 < await ( ar[i] == 0 ) >
 ar[i] = 1;
 < await ( ar[j] == 1 ) >
 ar[j] = 0;
 …
```

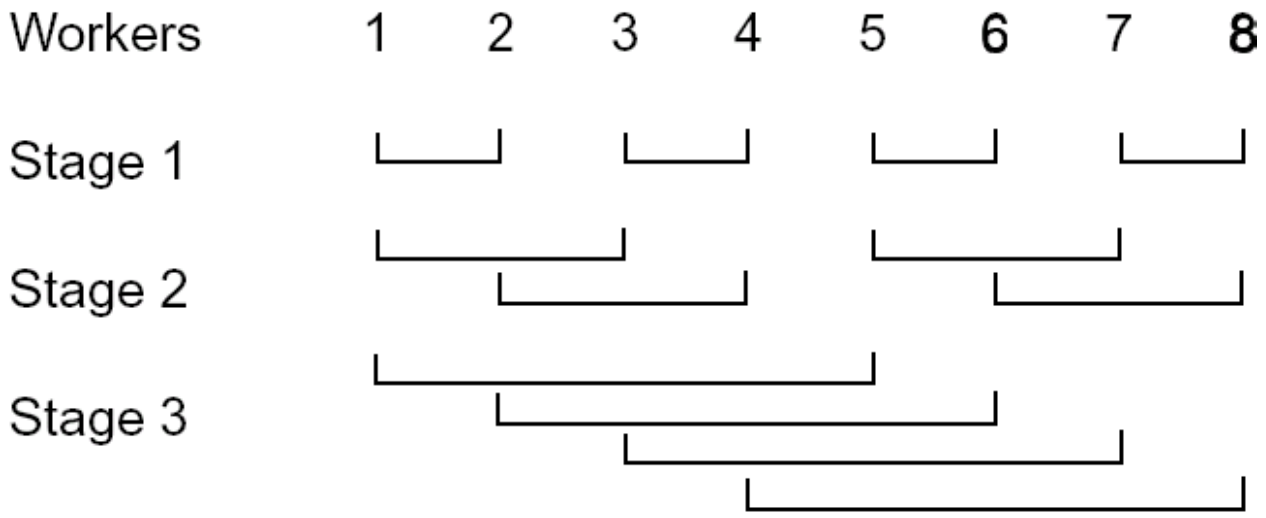
OR

```
// dissemination
// Worker[i]:
 …
 < await ( ar[j] == 0 ) >
 ar[j] = 1;
 < await ( ar[i] == 1 ) >
 ar[i] = 0;
 …
```

- The first await is to be sure that the first flag have been seen and cleared in previous round. The last line clears the second flag for the next round

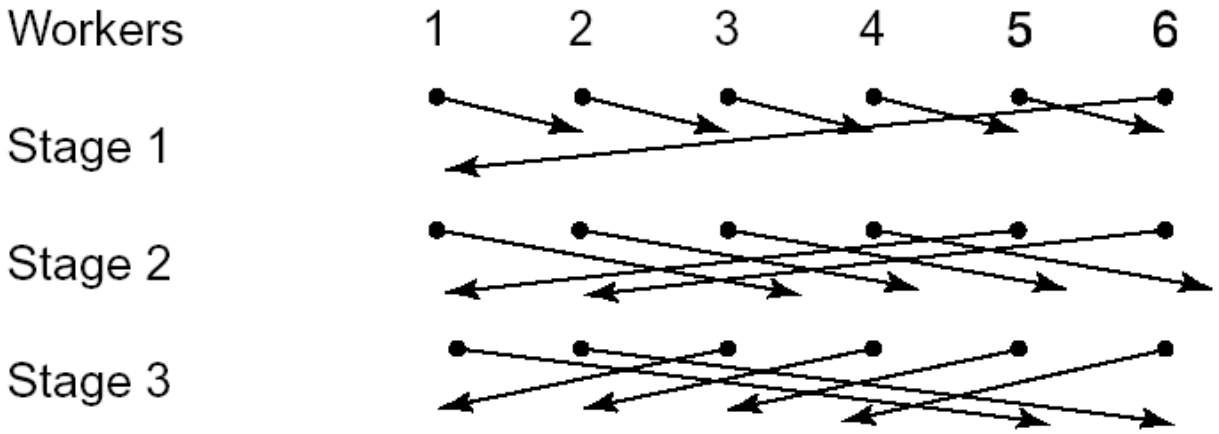
Butterfly Barrier

- Suitable if the number of processes *n* is power of 2
- $\log_2 n$ rounds of pair-wise barriers
  - In round *k*, a process *i* synchronizes with proc *j* at distance 2^{k-1} away
  - In total, each proc "barriers" with log *n* others



Dissemination Barrier

- Suitable for any number of processes n
- $\lceil \log_2 n \rceil$ rounds
  - In round k, a proc i signals to the node at $(i+2^{k-1}) \bmod n$
  - A process disseminates notice of its arrival: it sets the flag of a proc on its right and waits for its own flag to be set, then clears it
  - In total, each proc signals $\lceil \log n \rceil$ others:



Race Conditions in Symmetric Barriers

- The race conditions result from using multiple instances of the basic two-process barrier, and may cause deadlock.
- For example, consider the **Butterfly barrier**:
  - Assume, proc 3 has synchronized with proc 4 in round 1, and proceeds to synchronize with proc 1 in round 2;
  - Proc 1 is in round 1: it sets **`ar[1]`** (for proc 2 which is late) and waits for **`ar[2]`** to be set.
  - Proc 3 clears **`ar[1]`** which was set by proc 1 for proc 2 in the 1st round, and proceeds to the next round with proc 7.

ROYAL INSTITUTE OF TECHNOLOGY

Avoiding Race Conditions

1) Use different set of flags in each round of the *n*-process barrier: log*n* by *n* flags
2) Better yet, use counters instead of binary flags: each counter records the number of the current round executed
   - Use increment (decrement) instead of set (clear)
   - Wait for the value of **`ar[j]`** to be at least as large as **`ar[i]`**

```
// Worker[i]:
 …
 ar[i]++;
 while(ar[j]<ar[i]); skip;
 …
```

ROYAL INSTITUTE OF TECHNOLOGY

Data Parallel Algorithms

- In data parallel algorithms several proc execute the same code and work on different parts of shared data.
- **The idea of domain decomposition ("divide-and-conquer")** for iterative or recursive parallelism
- Barriers – to synchronize execution phases
- Locks – to access shared variables, e.g. partition boundaries, global totals
- Examples: Quick sort; Sum, max, min of matrix elements; Computations on a grid of points (PDE solvers), etc.

ROYAL INSTITUTE OF TECHNOLOGY

Data Parallel Algorithms (cont'd)

- Synchronous execution on a SIMD (Single-Instruction-Multiple-Data) multiprocessor
  - Every processor does the same thing at the same time
- Asynchronous execution on a MIMD (Multiple-Instructions-Multiple-Data) multiprocessor
  - Divide up data and use the **SPMD (Single-Program-Multiple-Data)** model.
    - All processes are given the same program but different data
    - A process knowing the data domain size, the number of processes and its number, can determine its data partition
  - Software barriers to synchronize phases;
  - Granularity of parallelism: Run time between barriers is typically (should be) much longer than the barrier latency.

Example: Computations on a Grid of Points

- Iterative computations on a grid of points

```
initialize the matrix;
while (not yet terminated) {
        compute a new value for each point;
        check for termination;
}
```

  - A new value in a point depends on values in its neighboring points, e.g.

```
y[i,j] = f(y[i,j-1], y[i,j+1], y[i-1,j], y[i+1,j])
```

  - The computation terminates
    - either after a fixed number of iterations,
    - or when all new values are within some tolerance ε of previous values.
- Examples:
  - Image processing: Find regions of neighboring pixels with the same intensity
  - PDE solvers, e.g. Jakobi iterations

ROYAL INSTITUTE OF TECHNOLOGY

Example (cont'd): Laplace Equation

- The Laplace equation with Dirichlet boundary conditions:

$$\partial^2\Phi/\partial^2 x + \partial^2\Phi/\partial^2 y = 0$$

  - Here Φ is unknown potential such as stress or temperature (heat)
  - Values on boundaries are constant
- The task is to compute an approximation of Φ in interior points in a steady state or after a given number of iterations (time steps) using some numerical method
  - Model the surface as a 2D grid of points.
  - Parallelization – the idea of domain decomposition: divide the grid into blocks or strips of points; assign a worker to each data partition.

Example (cont'd)
Jakobi Method for the Laplace Equation

- A value of a point on each iteration is computed as

$$\Phi^{(k)}_{i,j} = (\Phi^{(k-1)}_{i-1,j} + \Phi^{(k-1)}_{i+1,j} + \Phi^{(k-1)}_{i,j-1} + \Phi^{(k-1)}_{i,j+1}) \ / \ 4$$

  - the average of the previous values of its four closest neighbors.
- Fine-grained program: one proc is assigned to each point

```
real grid[n+1,n+1], newgrid[n+1,n+1];
bool converged = false;
process Grid[i = 1 to n, j = 1 to n] {
  while (not converged) {
    newgrid[i,j] = (grid[i-1,j] + grid[i+1,j] +
                    grid[i,j-1] + grid[i,j+1]) / 4;
    check for convergence as described in the text;
    barrier(i);
    grid[i,j] = newgrid[i,j];
    barrier(i);
  }
}
```

Bag of Tasks

- Many iterative problems can be solved by “divide and conquer“ (domain decomposition)
  - Divide data into partitions (blocks, strips); assign a process to each
- Another technique is **Bag of tasks**
  - Processes share a bag of tasks protected with a lock:

```
boolean done = false;
shared bag of tasks;
process Worker[w = 1 to P] {
  while (bag of tasks is not empty or !done) {
        <get a task from the bag;>
        execute the task;
        possibly generate new task(s) and
        <put it to the bag;>
  }
}
```

ROYAL INSTITUTE OF TECHNOLOGY

Bag of Tasks (cont'd)

- Useful for a fixed number of processes ("**work farm**")
- Provides **good load balancing** pretty much automatically
  - also easy to make tasks larger or smaller
- The bag of tasks is shared and must be accessed with mutual exclusion.
  - Use a lock to protect the bag.
- **How to detect termination?**
  - When bag is empty and all tasks are done
  - all tasks are done when all workers are waiting to get a new task
- Examples: (1) matrix multiplication, (2) adaptive quadrature

Example 1: Matrix Multiplication

- Shared bag: **int** `nextRow;`
- Get a task:
  `<row = nextRow++;>`
  - Can be implemented using atomic fetch&increment
  - The bag is empty when `row` is at least `n`
- Termination: the entire result matrix is computed after all workers break their loops
- Maintain a counter barrier after the loop and print the result:

`<done++;>`

`if (done == P)` **print matrix** `C;`

```
int nextRow = 0; # the bag of tasks
double a[n,n], b[n,n], c[n,n];

process Worker[w = 1 to P] {
  int row;
  double sum;  # for inner products
  while (true) {
    # get a task
    ⟨ row = nextRow; nextRow++; ⟩
    if (row >= n)
      break;
    compute inner products for c[row,*];
  }
}
```

ROYAL INSTITUTE OF TECHNOLOGY

Example 2: Parallel Adaptive Quadrature Using Bag of Tasks

- Shared bag: a list of records
  `{a, b, f(a), f(b), area}`
  - Get a task: extract a record from the bag
  - Put a task: add a new record to the bag whenever you would have done recursion in the parallel recursive program
- For efficiency, would want to do "pruning" when there are enough tasks
  - For example, keep track of how many there are, and stop generating tasks when there are enough. Instead, just use the basic recursive algorithm at this point (do not produce tasks, do yourself).
  - What is enough tasks: heuristic is 2-3 times the number of workers
  - This spreads out the load and should be pretty balanced

ROYAL INSTITUTE OF TECHNOLOGY

Exercise

- Assume there are n worker processes, numbered from 1 to n. Also assume that our machine has an atomic increment instruction. Consider the following code for a *reusable* n-process barrier:

```
int count = 0, go = 0;

code executed by Worker[1]:
        < await count == n-1>;
        count = 0;
        go = 1;

code executed by Worker[2:n]:
        <count++>
        <await go == 1>
```

- (a) What is wrong with the above code?
- (b) Fix the code so that it works. Do not use any more shared variables, but you may introduce local variables.

Solution

(a) The binary flag `go` is never reset to 0 to reuse the barrier.

(b) Use sense reversal: wait for the flag to take different value consecutive times. Introduce a local variable `local_sense` in each process to toggle sense of the flag.

```
int count = 0, go = 0;
process Worker[1]:
    int local_sense = 0; /* initial value */
    ...
    // barrier code executed by Worker[1]:
    local_sense = 1 - local_sense; /* toggle local sense */
    < await count == n-1>;
    count = 0;
    go = local_sence;
    ...
process Worker[2:n]:
    int local_sense = 0; /* initial value */
    ...
    //barrier code executed by Worker[2:n]:
    local_sense = 1 - local_sense; /* toggle local sense */
    <count++>
    <await go == local_sense>
```