

Lecture 13

Code Development Performance

Announcements

- A4 Due next Thursday 11/13
- Verification of login and use of Abe:
see the recent post of A4
www.cse.ucsd.edu/classes/fa08/cse160/HW/A4

Fixing odd/even sort

oesort -n 8 -t 2 *** Result has FAILED the verification test!

Input	Output
1531240967	219702300
219702300	857658880
857658880	1008028184
1008028184	1531240967
1705657730	15841591
837568134	837568134
15841591	1029050158
1029050158	1705657730

Testing

oesort -n 8 -t 2 -b “best case input”

*** Result has PASSED the verification test!

Input	Output
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7

Worst case input

oesort -n 8 -t 2 -w “reverse sorted list”

*** Result has FAILED the verification test!

Input	Output
7	3
6	4
5	5
4	6
3	0
2	1
1	2
0	7

Bounds of [0] are : 0, 3

Bounds of [1] are : 4, 7

Where is the bug?

```
int OE = lo % 2;
for (s = 0; s < MaxIter; s++) {
    barrier->synchronize();
    int done = Sweep(Keys, OE, lo, hi, NT, TID);  /* Odd phase */

    barrier->synchronize();
    done &= Sweep(Keys, 1-OE, lo, hi, NT, TID); /* Even phase */

    barrier->synchronize();
    int allDone = AllDone(done, TID);

    if (allDone){
        s++;
        break;
    }

} /* End of iteration loop. */
```

Inside Sweep

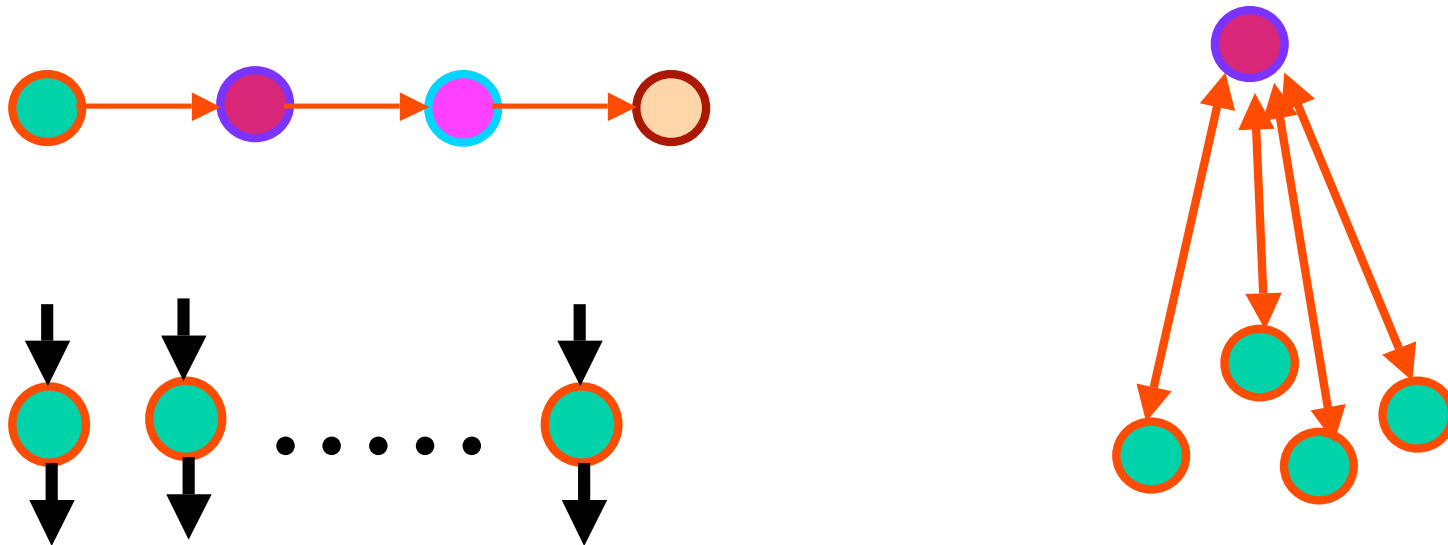
```
int Sweep(int *Keys, int OE, int lo, int hi, int NT, int TID){
    int Hi = hi;
    if (TID == NT-1)
        Hi--;
    int done = 1;
    for (int i = OE+lo; i <= Hi; i+=2) {
        if (Keys[i] > Keys[i+1]){
            int tmp = Keys[i];
            Keys[i] = Keys[i+1];
            Keys[i+1] = tmp;
            done = 0;
        }
    }
    return done;
}
```

Roadmap

- Threaded program design
 - Code organization and re-use
 - How to parallelize serial code (code reorganization)
 - How to design with parallelism in mind
 - Idioms - Design patterns
- Performance
- Troubleshooting and debugging
- OpenMP
 - Translation from OpenMP to pthreads
 - Performance

Paradigms

- Client/server
- Divide and Conquer - Data parallel
- Producer/Consumer (Pipelining) - function parallel



Benefits and Pitfalls of Multithreading

- Benefits
 - Harness parallelism to improve performance
 - Ability to multitask to realize concurrency, e.g. display
- Pitfalls
 - Program complexity
 - Partitioning, synchronization, serial sections, complicated parallel control flow
 - Data dependencies
 - Shared vs. local state (globals like errno)
 - Thread-safe code, re-entrant functions, protect the use of globals
 - New aspects of debugging
 - Race conditions
 - Deadlock

Case Study - odd/even sort

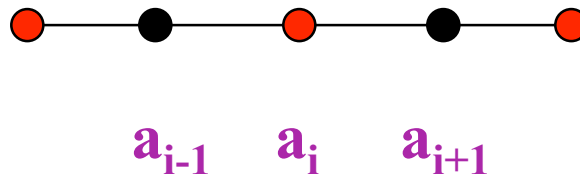
Compare and exchange sorts

- We cannot run bubble sort in parallel owing to the *loop carried dependence* in the inner loop
- The value of $a[j]$ computed in iteration j depends on the $a[i]$ computed in iterations $0, 1, \dots, j-1$

```
for i = N-1 to 1 by -1 do
  done = TRUE;
  for j = 0 to i-1 do
    done = Compare-exchange( $a[j]$  ,  $a[j+1]$ )
  end do
  if (done) break;
end do
```

Odd/Even sort

- If we re-order the comparisons we can parallelize the algorithm
 - number the points as even and odd
 - alternate between sorting the odd and even points
- This algorithm parallelizes since there are no loop carried dependences
- All the odd (even) points are decoupled



The algorithm

done = false;

for i = 0 **to** n-1 **do**

for j = 0 **to** n-1 **by** 2 **do** // Even

 done &= Compare-exchange(a[j] , a[j+1]);

end do

for j = 1 **to** n-1 **by** 2 **do** // Odd

 done &= Compare-exchange(a[j] , a[j+1]);

end do

if (done) **break**;

end do

Parallelizing the algorithm

- Splitting up the data
- Handling communication

Partitioning

- Each processor gets N/NT elements
- What if $N \% NT \neq 0$?
- Let $q = N \% NT$
- First q processors get $\lceil N/NT \rceil$ elements, others get $\lfloor N/NT \rfloor$ elements



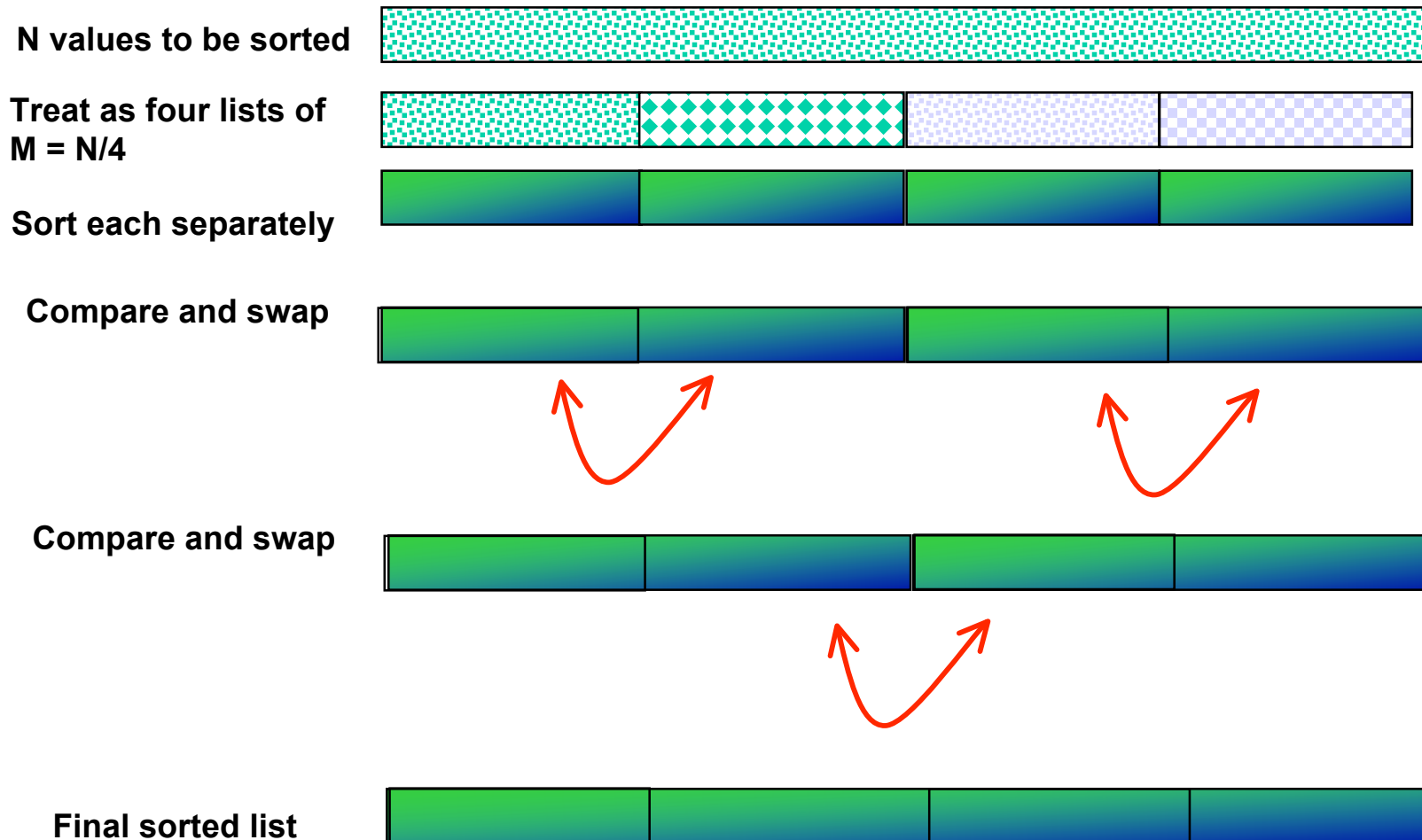
Odd Even Transposition sort

- While odd/even sort parallelizes, it has a long running time
- Only one data element at a time is swapped between neighboring processes, and each swaps value can take N/P steps to move across to the next neighboring process
- A more efficient algorithm moves blocks of data at each step
- Uses the odd/even ordering, but this time over block numbers (process IDs)
- The fundamental operation is a block *compare-swap*

The algorithm

- As a pre-processing step, each processor locally sorts its data using a fast serial algorithm like quicksort
- Processes exchange their data in odd-even pairs using *block compare and swap*
- Each process applies a local merge sort to extract the smallest (largest) N/P values, discards the rest
- What is the running time?

Odd-even merge sort in action



Block compare and swap



- Compare and swap
 - Each processor swaps data with its neighbor
-1 3 7 9 11 2 4 8 12 14
 - Sorts the merged list
-1 2 3 4 7 8 9 11 12 14
 - Processor 0 takes 5 smallest values: -1 2 3 4 7
 - Processor 1 takes 5 largest values: 8 9 11 12 14

The algorithm

done = false;

for i = 0 **to** P-1 **do**

for p = 0 **to** P-1 **by** 2 **do** **// Even**
 done &= Block-Compare-swap(A_p , A_{p+1});
 end do

for p = 1 **to** P-1 **by** 2 **do** **// Odd**
 done &= Block-Compare-swap(A_p , A_{p+1});
 end do
 if (done) **break**;

end do