Sorting and Searching (in parallel)

- The always changing Timeline again
- Sorting and searching

Sorting

- Fundamental operation: Sorting makes faster searching
- Knuth's categorization of Sorting (volume 3)
  - Insertion
  - Exchange
  - Selection
  - Merging
  - Distribution
  - Networks

Searching

- Fundamental operation: Searching is everywhere
- Knuth’s categorization of Searching (volume 3)
  - Sequential
  - Searching ordered table
  - Binary tree searching
  - Balanced trees
  - Digital searching (by digits)
  - Hashing

Finding the kth smallest element (selection)

- Common case: Finding the median
  - Fundamental building block for various algorithms (example: Markov clustering)
- Quickselect (Hoare): Like quicksort, but recurses only one direction. Average time O(N), worst case $O(N^2)$
- Median of the Medians (Blum, Floyd, Pratt, Rivest, Tarjan):
  - Based on quickselect, but guarantees worst-case linear time. Ties in nicely to the parallel algorithm

Quickselect: select

```
function select(list, left, right, k) is
    if left == right then   // If the list contains only one element,
        return list[left]  // return that element
    pivotIndex  := partition(list, left, right, pivotIndex)
    // The pivot is in its final sorted position
    if k == pivotIndex then
        return list[k]
    else if k < pivotIndex then
        return select(list, left, pivotIndex − 1, k)
    else
        return select(list, pivotIndex + 1, right, k)
```

Quickselect: partition

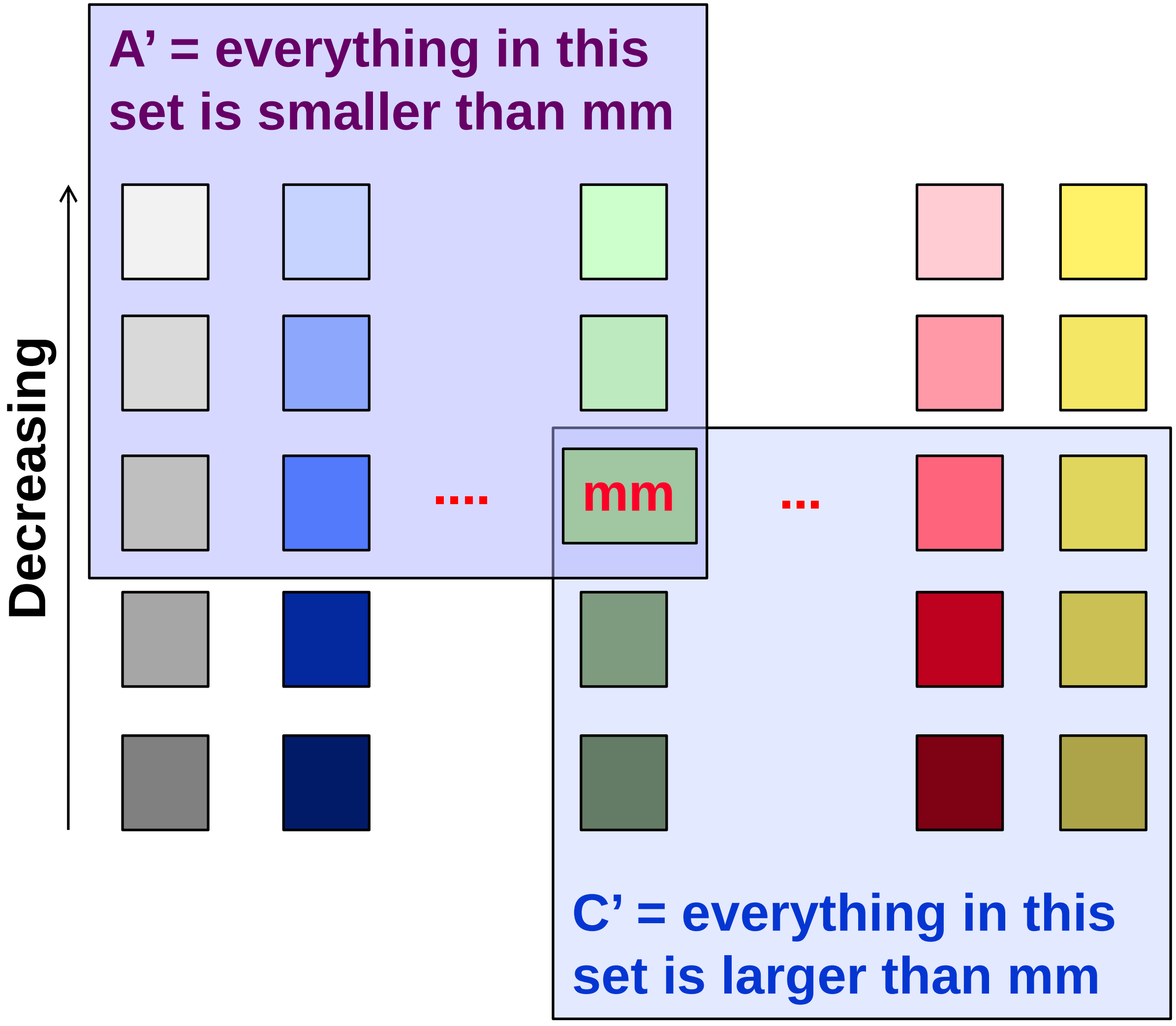
```
function partition(list, left, right, pivotIndex) is
    pivotValue := list[pivotIndex]
    swap list[pivotIndex] and list[right]  // Move pivot to end
    storeIndex := left
    for i from left to right − 1 do
        if list[i] < pivotValue then
            swap list[storeIndex] and list[i]
            increment storeIndex
    swap list[right] and list[storeIndex]  // Move pivot to its final place
    return storeIndex
```

Median of Medians

- Divide list into ~N/5 sub-lists of each (at least) 5 elements
- Sort each sub-list and find median
- Continue *recursively* on the medians list

Median of Medians

- Base case: the recursive call will find the "median"



We actually don't know the total order between these N/5 medians, but it doesn't matter...

Median of Medians algorithm

- Partition the whole list into three district sets: A, B, C (mm is the median of the median)
- A = set of elements smaller than mm (A’ is subset of A)
- B = set of elements equal to mm
- C = set of elements larger than mm (C’ is subset of C)

Outline

```
SELECT(S, k) // find kth smallest in S
{
    M = DIVIDEANDSORT(S,5); // O(N), M: list of medians
    mm = SELECT(M,|M|/2);  // recurse on O(N/5)
    [A,B,C] = PARTITION(S,mm);  // O(N)
    if (|A| < k <= |A| + |B|)
        return x;
    else if (k <= |A|), // recurse
        return SELECT(A, k)
    else if (if k > |A| + |B|) // recurse
        return SELECT(C, k -|A|-|B|)
}
```

Parallel Version

- Let each processor compute its local median
- Find global median of medians
- Discard elements and recurse
- Problem:
  - Load imbalance surfaces as algorithm proceeds
  - Requires data redistribution
    - → extra communication
    - → extra programming hassle

Divide and Conquer: Mergesort

- Mergesort: a recursive sorting algorithm.
- Based on the divide-and-conquer paradigm
- The merge operation is its fundamental component (which takes in two sorted sequences and produces a single sorted sequence)
- Drawback of mergesort: Not in-place (uses an extra temporary array)

Divide and Conquer: Mergesort

```
function merge_sort(list m) is
    // Base case. A list of zero or one elements is sorted, by definition.
    if length of m ≤ 1 then
        return m

    // Recursive case. First, divide the list into equal-sized sublists
    // consisting of the first half and second half of the list.
    // This assumes lists start at index 0.
    var left := empty list
    var right := empty list
    for each x with index i in m do
        if i < (length of m)/2 then
            add x to left
        else
            add x to right

    // Recursively sort both sublists.
    left := merge_sort(left)
    right := merge_sort(right)

    // Then merge the now-sorted sublists.
    return merge(left, right)
```

Divide and Conquer: Mergesort
Merging two arrays

```
function merge(left, right) is
    var result := empty list

    while left is not empty and right is not empty do
        if first(left) ≤ first(right) then
            append first(left) to result
            left := rest(left)
        else
            append first(right) to result
            right := rest(right)

    // Either left or right may have elements left; consume them.
    // (Only one of the following loops will actually be entered.)
    while left is not empty do
        append first(left) to result
        left := rest(left)
    while right is not empty do
        append first(right) to result
        right := rest(right)
    return result
```

C++ Merge

```
template <typename T>
void Merge(T *C, T *A, T *B, int na, int nb) {
  while (na>0 && nb>0) {
    if (*A <= *B) {
      *C++ = *A++; na--;
    } else {
      *C++ = *B++; nb--;
    } }
  while (na>0) {
    *C++ = *A++; na--;
  }
  while (nb>0) {
    *C++ = *B++; nb--;
  }
}
```

Parallel Mergesort

```
template <typename T>
// B sorted output, A unsorted input, C temporary

void MergeSort(T *B, T *A, int n) {

    if (n==1) { B[0] = A[0]; }

    else {

    T* C = new T[n];

    #pragma omp parallel {

        #pragma omp single {

            #pragma omp task

                MergeSort(C, A, n/2);

            #pragma omp task

                MergeSort(C+n/2, A+n/2, n-n/2);

        }
    }

    Merge(B, C, C+n/2, n/2, n-n/2);

    delete[] C;

    }
}
```

Parallelizing Merge
(pragma notes)

- Omp parallel identifies parallel section
- Omp single picks one thread to be the “producer” and installs a barrier at the end where the threads wait for a new task
- New tasks are spawned and then complete at the end of the parallel construct

Mergesort: Parallelizing Merge

- If the total number of elements to be merged in the two arrays is $n = na + nb$, the total number of elements in the larger of the two recursive merges is at most $(3/4)n$
- Use binary search to find the median in the sorted array

Mergesort: Parallelizing Merge

```
template <typename T>
void P_Merge(T *C, T *A, T *B, int na, int nb) {
    if (na < nb) { P_Merge(C, B, A, nb, na); }
    else if (na==0) { return; }
    else {
        int ma = na/2;
        int mb = BinarySearch(A[ma], B, nb);
        C[ma+mb] = A[ma];
        #pragma omp parallel {
            #pragma omp single {
            #pragma omp task
                P_Merge(C, A, B, ma, mb);
            #pragma omp task
                P_Merge(C+ma+mb+1,A+ma+1,B+mb,na-ma-1,nb-mb);
            }
         } // implicit taskwait
    }
}
```

Bucket sort

- In Bucket sort, the range [a,b] of input numbers is divided into *m* equal sized intervals, called *buckets*.
- Each element is placed in its appropriate bucket.
- If the numbers are uniformly divided in the range, the buckets can be expected to have roughly identical number of elements.
- Elements in the buckets are locally sorted.
- The runtime of this algorithm is $\Theta(n \log(n/m))$.

Parallel Bucket sort

- Parallelizing bucket sort is relatively simple: select m = p.
- Each processor has a range of values it is responsible for.
- Each processor runs through its local list and assigns each of its elements to the appropriate processor.
- The elements are sent to the destination processors using a single all-to-all personalized communication.
- Each processor sorts all the elements it receives.
- Load Imbalance: the assumption that the input elements are uniformly distributed over an interval [a, b] is not realistic.

Serial Quicksort

```
// Sorts a (portion of an) array, divides it into subpartitions, then recurses
quicksort(A, lo, hi) {
  if lo >= 0 && hi >= 0 && lo < hi then
    p := partition(A, lo, hi) // p is the pivot
    quicksort(A, lo, p)
    quicksort(A, p + 1, hi)
```

Serial Quicksort, cont'd

```
algorithm partition(A, lo, hi) {
 // Pivot value
  pivot := A[ floor((hi + lo) / 2) ] // The value in the middle of the array
  i := lo - 1 // Left index
  j := hi + 1 // Right index

  loop forever
    // Move the left index to the right at least once and while the element at
    // the left index is less than the pivot
    do i := i + 1 while A[i] < pivot

    // Move the right index to the left at least once and while the element at
    // the right index is greater than the pivot
    do j := j - 1 while A[j] > pivot

    // If the indices crossed, return
    if i >= j then return j

    // Swap the elements at the left and right indices
    swap A[i] with A[j]
```

Parallelizing Quicksort

- The depth of Quicksort's divide-and-conquer tree directly impacts the algorithm's scalability,
- Depth is highly dependent on the choice of pivot
- Hard to parallelize the partitioning efficiently in-place. The use of scratch space simplifies the partitioning step, but increases the algorithm's memory footprint and constant overheads.

Parallel Sample sort

- Parallelizing bucket sort is relatively simple: select m = p.
- Each processor has a range of values it is responsible for.
- Each processor runs through its local list and assigns each of its elements to the appropriate processor.
- The elements are sent to the destination processors using a single all-to-all personalized communication.
- Each processor sorts all the elements it receives.
- Load Imbalance: the assumption that the input elements are uniformly distributed over an interval [a, b] is not realistic.

Parallel Sample Sort

- Generalization of Quicksort: from 1 pivot to p pivots. This is done by suitable splitter selection.
- The splitter selection method divides the n elements into m blocks of size n/m each, and sorts each block by using quicksort.
- Choose m-1 evenly spaced samples from each sorted block.
- The m(m-1) elements selected from all the blocks represent the sample used to determine the buckets.
- This scheme guarantees that the number of elements ending up in each bucket is less than 2n/m

Parallel Sample Sort, cont'd

- Sort n/m sized blocks locally
- Pick (m-1) regularly spaced samples from each block
- Merge/sort these m(m-1) samples in a single processor to form Y
- Size(Y) = p(p-1)
- One processor merges pieces of Y or all merge redundantly

- How can this go wrong at large scale?
- Sorting on 20,000 cores: Each sample (splitter candidate) is at least 12 bytes (value: 4, global index: 8). Only the p(p-1) splitters take 4.8GB, per core!

Parallel Sample Sort, cont'd

- The internal sort of n/p elements requires time $\Theta((n/p)\log(n/p))$, and the selection of p – 1 sample elements requires time $\Theta(p)$.
- The time for an all-to-all broadcast is $\Theta(p2)$, the time to internally sort the p(p – 1) sample elements is $\Theta(p^2 \log p)$, and selecting p – 1 evenly spaced splitters takes time $\Theta(p)$.
- Each process can insert these p – 1 splitters in its local sorted block of size n/p by performing p – 1 binary searches in time $\Theta(p\log(n/p))$.
- The time for reorganization of the elements is $O(n/p)$.
- Total time is sum *but* assumes that communication and computation costs have the same constants (they don't)

Radix Sort

- Treat keys as multidigit numbers, where each digit is an integer from <0…(m-1)> where m is the radix
- The radix m is variable, and chosen to minimize running time
- Radix sort divides the keys into buckets based on one or more digits of the key
- Most significant digit (MSD) vs. least significant (LSD)

Radix Sort (MSD)

- Divide the array into buckets numbered 0..255
- Based on the value of the first byte of the key
- Recursively sort each bucket into sub-buckets, based on the value of subsequent bytes in the key
- MSD radix sort is a recursive divide-and conquer algorithm
- Divides the array into smaller and smaller partitions, so locality tends to be good
- O(N) steps are needed for each level of partitioning

Radix Sort (LSD)

- Divide the array into buckets numbered 0..255
- Based on the value of the last byte of the key
- But do not recursively sort each bucket: Instead divide the full array into separate buckets based on the second last key

Serial (and Paralle) Radix Sort

COUNTING-SORT
  HISTOGRAM-KEYS
    do $i \leftarrow 0$ **to** $2^r - 1$
      $Bucket[i] \leftarrow 0$
    do $j \leftarrow 0$ **to** $N - 1$
      $Bucket[D[j]] \leftarrow Bucket[D[j]] + 1$
  SCAN-BUCKETS
    $Sum \leftarrow 0$
    do $i \leftarrow 0$ **to** $2^r - 1$
      $Val \leftarrow Bucket[i]$
      $Bucket[i] \leftarrow Sum$
      $Sum \leftarrow Sum + Val$
  RANK-AND-PERMUTE
    do $j \leftarrow 0$ **to** $N - 1$
      $A \leftarrow Bucket[D[j]]$
      $R[A] \leftarrow K[j]$
      $Bucket[D[j]] \leftarrow A + 1$

- Loop dependencies in all three phases
- Solution: Use a separate set of buckets for each processor
- Each processor takes care of N/P keys where P is number of processors.

Sorting Networks

- Network of comparators designed for sorting
- Comparator : two inputs x and y; two outputs x' and y'

1) Increasing (denoted ⊕): $x' = \min(x,y)$ and $y' = \max(x,y)$
2) Decreasing (denoted Ɵ) : $x' = \max(x,y)$ and $y' = \min(x,y)$

- Sorting network speed is proportional to its depth

Bitonic sorting networks

- Network structure: a series of columns
- Each column consists of a vector of comparators (in parallel)
- Bitonic sequence: sequence of elements $<a_0, a_1, \ldots , a_{n-1}>$
  - There exists i such that $<a_0, \ldots ,a_i>$ is monotonically increasing and $<a_{i+1},\ldots , a_{n-1}>$ is monotonically decreasing or
  - There exists a cyclic shift of indicies such that the above is satisfied.
  - Example:
    - 〈1,2,4,7,6,0〉: first increases and then decreases (or vice versa)
    - 〈8,9,2,1,0,4〉: cyclic rotation of 〈0,4,8,9,2,1〉
- Bitonic sorting network
  - sorts n elements in Θ(log2 n) time
  - network kernel: rearranges a bitonic sequence into a sorted one

Bitonic rearrangement

- Let $s = <a_0, a_1, \dots , a_{n-1}>$
- $a_{n/2}$ is the beginning of the decreasing seq.
- Let $s_1 = <\min\{a_0, a_{n/2}\}, \min\{a_1, a_{n/2+1}\} \dots \min\{a_{n/2-1}, a_{n-1}\}>$
- Let $s_2 = <\max\{a_0, a_{n/2}\}, \max\{a_1, a_{n/2+1}\} \dots \max\{a_{n/2-1}, a_{n-1}\}>$
- In sequence s_1 there is an element $b_i = \min\{a_i, a_{n/2+i}\}$
  - all elements before b_i are from increasing
  - all elements after b_i are from decreasing
- Sequence s_2 has a similar point
- Sequences s_1 and s_2 are bitonic

Sorting via Bitonic Merging Network

- Sorting network can implement bitonic merge algorithm:
- Network structure
  - $\log_2 n$ columns
  - each column has n/2 comparators
- Bitonic merging network with n inputs: $\oplus BM[n]$: produces an increasing sequence
- Replacing $\oplus$ comparators by $\ominus$ comparators: $\ominus BM[n]$: produces a decreasing sequence

What about unordered lists?

- To use the bitonic merge for n items, we must first have a bitonic sequence of n items.
- Two elements form a bitonic sequence
- Any unsorted sequence is a concatenation of bitonic sequences of size 2
- Merge those into larger bitonic sequences until we end up with a bitonic sequence of size n

Sorting a bitonic sequence

- Use bitonic split recursively,

  INPUT: a bitonic sequence of size n

  Phase 1: 2 bitonic sequences of size n/2

  Phase 2: 4 bitonic sequences of size n/4

  …

  …

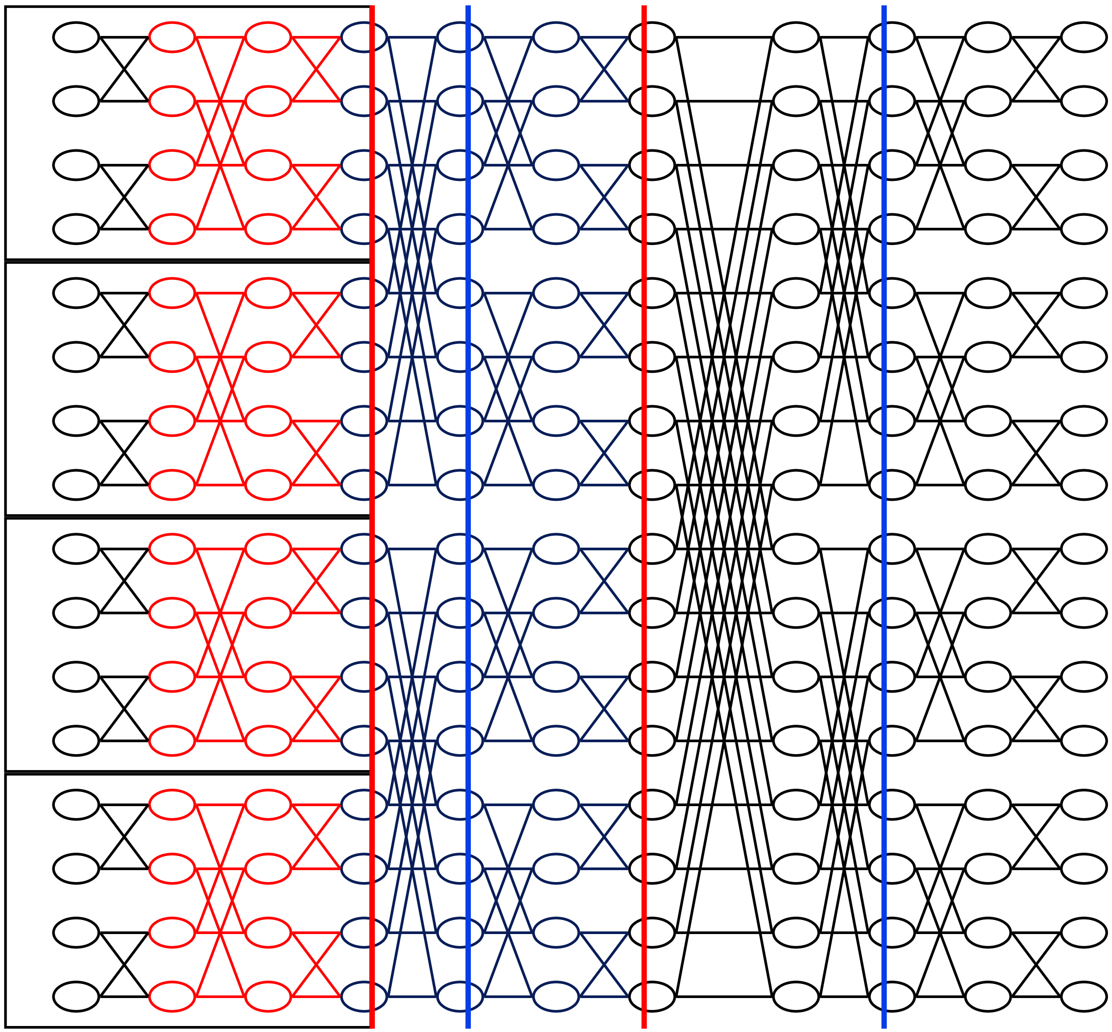
  Phase (log n): n bitonic sequences of size 1
- A sorted sequence can be generated by concatenating the n bitonic sequences of size 1

Complexity of Parallel Bitonic Sort for n processors

- The last stage of an n-element bitonic sort needs to merge n-elements, and has a depth of log(n)
- Other stages perform a complete sort of n/2 elements
- Depth: $d(n) = d(n/2) + \log(n)$
- $d(n) = 1 + 2 + 4 + \ldots + \log(n) = \theta(\log^2 n)$
- Complexity: $T(n) = \theta(\log^2 n)$

Bitonic Sort for n=16: all dependencies shown

Block Layout



lg N/p stages are local sort – Use best local sort

remaining stages involve

Block-to-cyclic, local merges (i - lg N/P cols)

cyclic-to-block, local merges (lg N/p cols within stage)