CS 267
Lecture 25: Searching and Sorting

Aydin Buluc

[with some slides from Charles Leiserson and some slides from "Introduction to Parallel Computing" book]

https://sites.google.com/lbl.gov/cs267-spr2019/

Outline

- (Optional) recap Master Method for solving recurrences. Because many of these algorithms are recursive.

- Searching for the kth smallest element in parallel
  - A serial algorithm to warm up
- Sorting in shared memory
- Sorting in distributed memory

- One representative algorithm per problem

The Master Method (Optional)

The ***Master Method*** for solving recurrences applies to recurrences of the form

$$T(n) = a\,T(n/b) + f(n)\ ,^{*}$$

where $a \geq 1$, $b > 1$, and f is asymptotically positive.

IDEA: Compare $n^{\log_b a}$ with $f(n)$.

* The unstated base case is $T(n) = \Theta(1)$ for sufficiently small n.

Master Method — CASE 1

$$T(n) = a\,T(n/b) + f(n)$$

$$n^{\log_b a} \gg f(n)$$

Specifically, $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$.

Solution: $T(n) = \Theta(n^{\log_b a})$.

Master Method — CASE 2

$$T(n) = a\,T(n/b) + f(n)$$

$$n^{\log_b a} \approx f(n)$$

Specifically, $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$.

Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n))$.

Ex(qsort): a =2, b=2, k=0 ➔ $T_1(n)=\Theta(n \lg n)$

Master Method — CASE 3

$$T(n) = a\,T(n/b) + f(n)$$

$$n^{\log_b a} \ll f(n)$$

Specifically, $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and $f(n)$ satisfies the ***regularity condition*** that $a\,f(n/b) \leq c\,f(n)$ for some constant $c < 1$.

Solution: $T(n) = \Theta(f(n))$.

Master Method Summary

$$T(n) = a\,T(n/b) + f(n)$$

CASE 1: $f(n) = O(n^{\log_b a - \epsilon})$, constant $\epsilon > 0$

$\Rightarrow T(n) = \Theta(n^{\log_b a})$.

CASE 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, constant $k \geq 0$

$\Rightarrow T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

CASE 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$, constant $\epsilon > 0$, and regularity condition

$\Rightarrow T(n) = \Theta(f(n))$.

Outline

- (Optional) recap Master Method for solving recurrences. Because many of these algorithms are recursive.

- Searching for the kth smallest element in parallel
  - **A serial algorithm to warm up**
- Sorting in shared memory
- Sorting in distributed memory

- One representative algorithm per problem

Selection problem

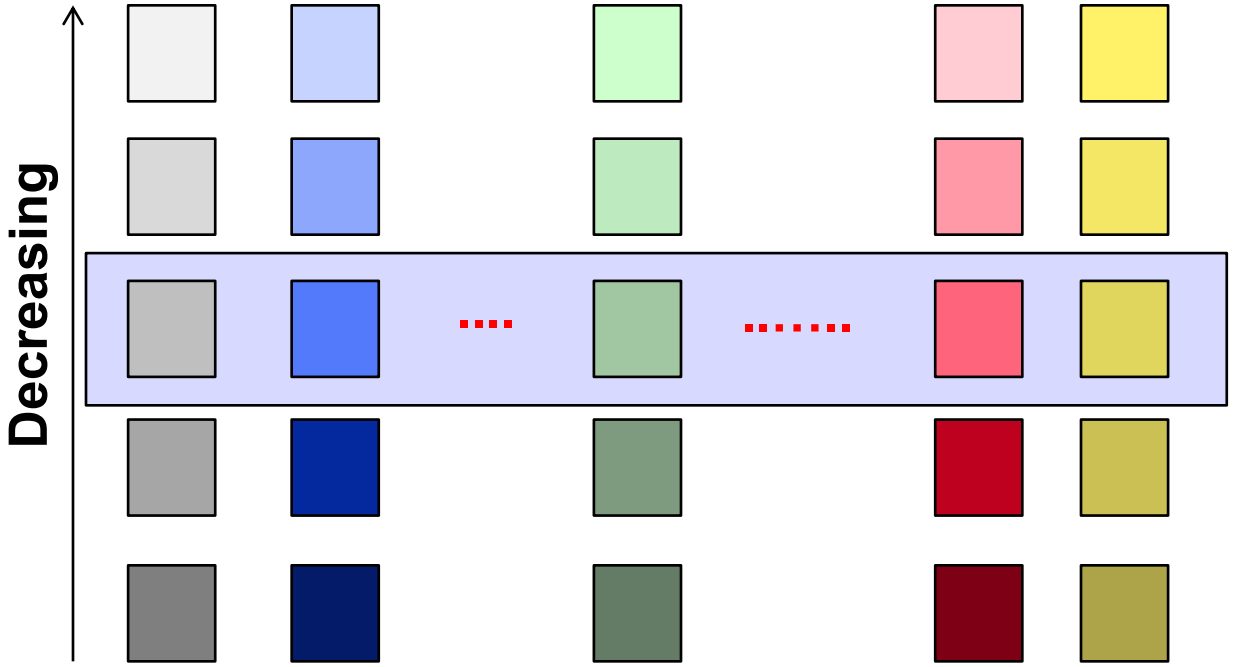
- **Task:** Find the **kth** smallest element in a **list of length N**
- **Common case:** Finding the median and the quartiles
- Fundamental building block for various algorithms (example: MCL clustering)

- **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
  - Trivia: How many of the authors (out of 6 total) are Turing Laureates?

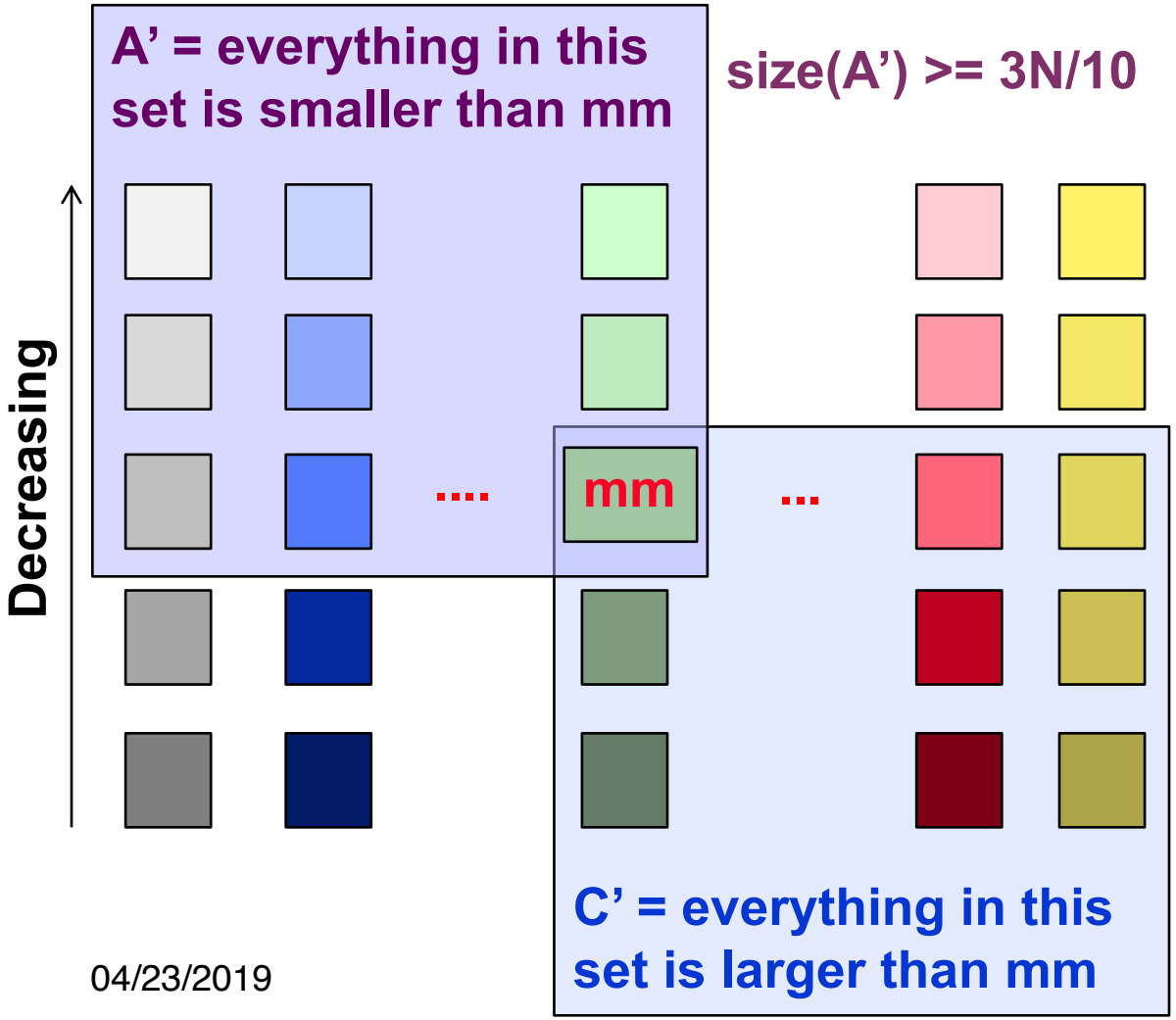
Median of the Medians Algorithm for Selection

- 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 (M)**



Median of the Medians Algorithm for Selection

- Base case: the recursive call will find the "median", assume the green list holds the ***median of the medians (mm)***



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

Median of the Medians Algorithm for Selection

Partition the whole list into three district sets: A, B, C

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)**

```
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 on O(7N/10)
        return SELECT(A, k)
    else if (if k > |A| + |B|)     // recurse on O(7N/10)
        return SELECT(C, k -|A|-|B|)
}
```

Median of the Medians Algorithm for Selection

T(n) <= T(n/5) + T(7n/10) + O(n)

Why O(7N/10)?

A excludes members of C' (and C excludes members of A')

```
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 on O(7N/10)
        return SELECT(A, k)
    else if (if k > |A| + |B|)     // recurse on O(7N/10)
        return SELECT(C, k -|A|-|B|)
}
```

Median of the Medians Algorithm for Selection

$T(n) <= T(n/5) + T(7n/10) + O(n)$

$T(n) <= T(9n/10)+O(n)$

$n^{\log_b a} <= f(n)$

because

$n^{\log_{10} 9} <= O(n)$

(actually $n^{\log_{10} 9} = 1$ because n^t asymptotically goes to 1 for $t<1$).

Master method:

$$T(n) = a\,T(n/b) + f(n)$$

Hence, $T(n) = O(n)$

Now you know why we have lists of 5 (and not fewer) elements

Outline

- (Optional) recap Master Method for solving recurrences. Because many of these algorithms are recursive.

- **Searching for the kth smallest element in parallel**
  - A serial algorithm to warm up
- Sorting in shared memory
- Sorting in distributed memory

- One representative algorithm per problem

Parallel Selection via Median of the Medians?

- 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

$T_{comm} = O(\alpha \log(n/p^2) + \beta (\log(n/p^2) + k))$
$T_{comp} = O(n/p + k)$

Bader, David A., and Joseph JáJá. "Practical parallel algorithms for dynamic data redistribution, median finding, and selection.", 1996

Parallel Selection via Median of the Medians

- Is there a simpler algorithm that doesn't load balance?
- **IDEA:** Ignore load imbalance instead of dealing with it!
- **Weighted median of the medians**

Given p distinct elements $m_1, m_2, \ldots, m_p$ with corresponding positive weights $w_1, w_2, \ldots, w_p$ such that $\Sigma_{1<=i<=p} w_i = 1$, the **weighted median** is the element **M** that satisfies $\Sigma_{i,mi<M} w_i <= 1/2$ and $\Sigma_{i,mi>M} w_i <= 1/2$

Sanguthevar Rajasekaran, Wang Chen, and Shibu Yooseph. "Unifying themes for network selection.", 1994

Einar L.G. Saukas, and Siang W. Song. "Efficient selection algorithms on distributed memory computers.", 1998

Weighted Median of the Medians Algorithm

```
PARALLELSELECT(S, k) // find kth smallest in S
{
    lm = SELECT(S,|S|/2);          // find local median
    LMS = MPI_Allgather(lm,0);     // exchange medians
    wmm = WeightedMedian(LMS);      // redundant
computation
    [A,B,C] = PARTITION(S,wmm); // same as in serial
    MPI_Allreduce(size(A), &ls, MPI_SUM);  // less than
    MPI_Allreduce(size(B), &eq, MPI_SUM);  // equal to

    if (ls < k <= ls + eq)    // solution found
        return wmm;
    else if (k <= ls)                 // recurse on O(3N/4)
        return PARALLELSELECT(A,k)
    else if (if k > ls + eq)          // recurse on O(3N/4)
        return PARALLELSELECT( C, k-|A|-|B|)
}
```

Weighted Median of the Medians Algorithm

Where does 3N/4 come from?

By definition of weighted median:

$\mathbf{\Sigma_{i,mi<M}\ w_i <= 1/2}$ and $\mathbf{\Sigma_{i,mi>M}\ w_i <= 1/2}$

Replace weights with #elements in each processor n_i:

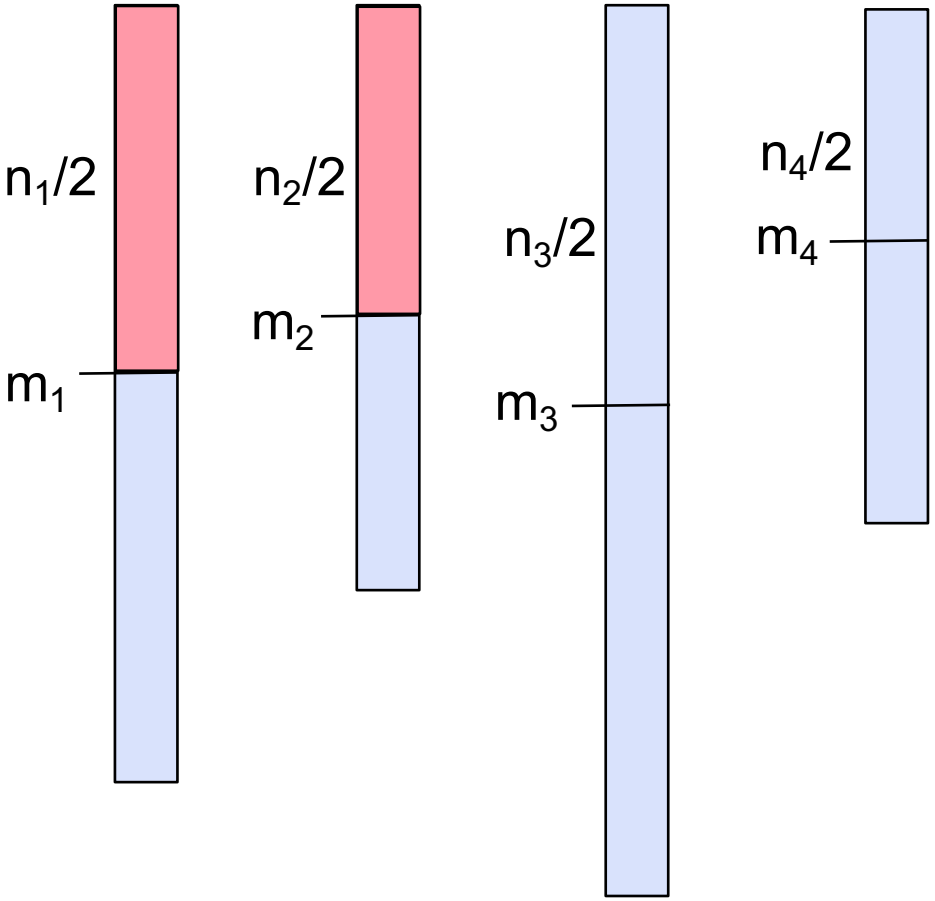
$\mathbf{\Sigma_{i,mi<M}\ n_i <= N/2}$ and $\mathbf{\Sigma_{i,mi>M}\ n_i <= N/2}$

At processor i, **#elements <= m_i** is at least **$n_i/2$** (by median definition). Half of those elements are also smaller than M

1. Hence, total #elements (across all processor) that are less than or equal to M is N/4
2. "Greater than or equal to" case is symmetric

We will discard either (1) or (2), hence left with most 3N/4

Weighted Median of the Medians Algorithm



$\Sigma_{1<=i<=p}\ n_i = N$

$\Sigma_{i,mi<M}\ n_i <= N/2$

$\Sigma_{i,mi<M}\ (n_i)/2 <= N/4$

Weighted Median of the Medians Algorithm

$T_{comm}(n) <= T_{comm}(3n/4) + O(\alpha \log(p) + \beta p)$

- Number of recursive calls is O(log n)
- log(p) latency factor comes from gather/reduce

$T_{comm}(n) = O(\alpha \log(p) \log(n) + \beta\ p \log(n))$

- We can coarsen the base case: "gather" and locally solve when the remaining number of items are n/p
- Number of recursive calls become O(log(p))
- Extra bandwidth cost to "gather": O(n/p)

$T_{comm}(n) = O(\alpha \log^2(p) + \beta\ (p \log(p) + n/p))$

$T_{comp}(n) = O(n/p * \log(n))$

Because (a) we can't guarantee that local lists are going to shrink and (b) this is the cost of locally sort at the end

Parallel Selection Recap

- There are faster, ***randomized*** algorithms
- There is no experimental study in the last decade
- **Potential class project** to evaluate variants.

Al-Furiah, I., Aluru, S., Goil, S. and Ranka, S." Practical algorithms for selection on coarse-grained parallel computers". IEEE Transactions on Parallel and Distributed Systems (1997).

Bader, David A. "An improved, randomized algorithm for parallel selection with an experimental study." Journal of Parallel and Distributed Computing 64.9 (2004): 1051-1059.

Outline

- (Optional) recap Master Method for solving recurrences. Because many of these algorithms are recursive.

- Searching for the kth smallest element in parallel
  - A serial algorithm to warm up
- **Sorting in shared memory**
- Sorting in distributed memory

- One representative algorithm per problem

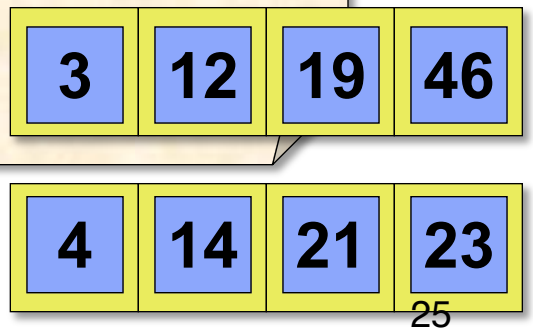
MERGESORT

- Mergesort is an example of a recursive sorting algorithm.
- It is based on the divide-and-conquer paradigm
- It uses the merge operation as 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)

Merging Two Sorted Arrays

```
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--;
  } }
```

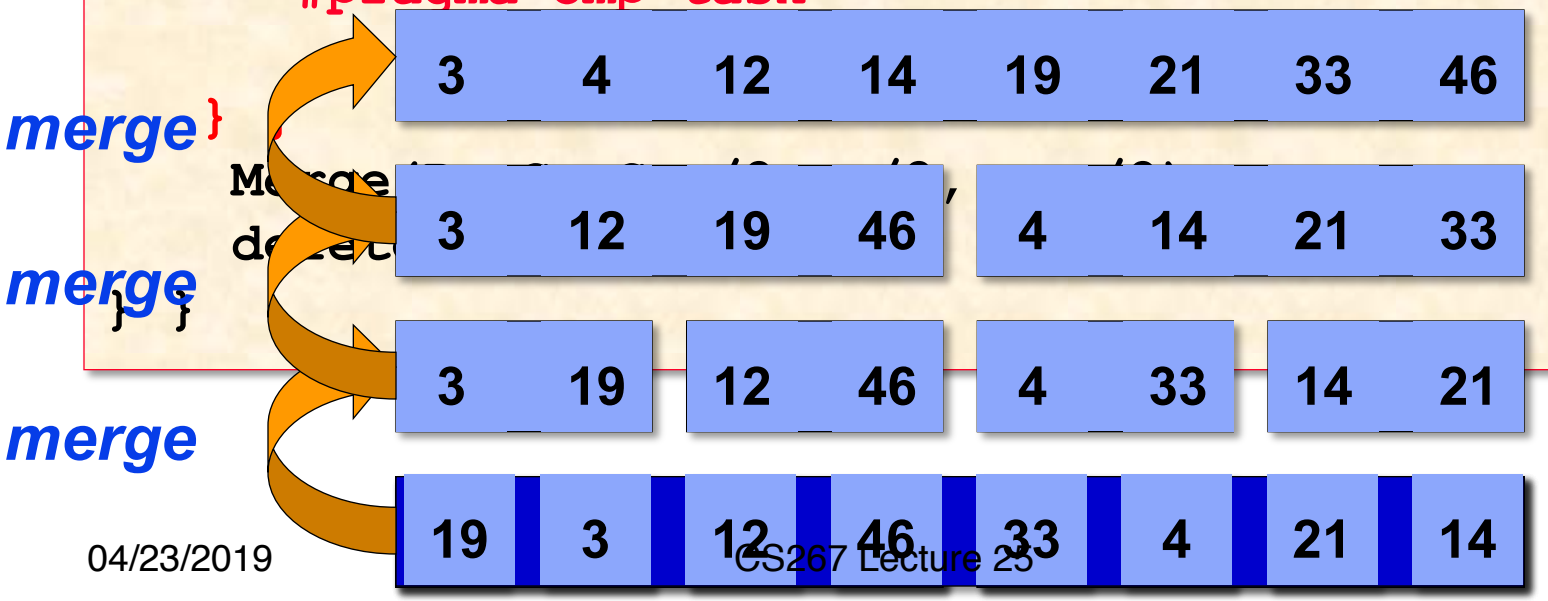
Time to merge n elements = Θ(n).



Parallel Merge Sort

```
template <typename T>
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
      }
      Merge
      delet
  } }
```

A: input (unsorted)
B: output (sorted)
C: temporary



Work of Merge Sort

```
template <typename T>
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/2);
   } }
    Merge(B, C, C+n/2, n/2, n/2);
    delete[] C;
} }
```

CASE 2 (of Master Theorem):
$n^{\log_b a} = n^{\log_2 2} = n$
$f(n) = \Theta(n^{\log_b a} \lg^0 n)$

Work: $T_1(n) = 2T_1(n/2) + \Theta(n) = \Theta(n \lg n)$

Span of Merge Sort

```
template <typename T>
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
      #pragma omp task
            MergeSort(C+n/
   } }
    Merge(B, C, C+n/2, n/2, n-
    delete[] C;
} }
```

CASE 3 (of Master Theorem):
$n^{\log_b a} = n^{\log_2 1} = 1$
$f(n) = \Theta(n)$

Span: $T_\infty(n) = T_\infty(n/2) + \Theta(n) = \Theta(n)$

Parallelism of Merge Sort

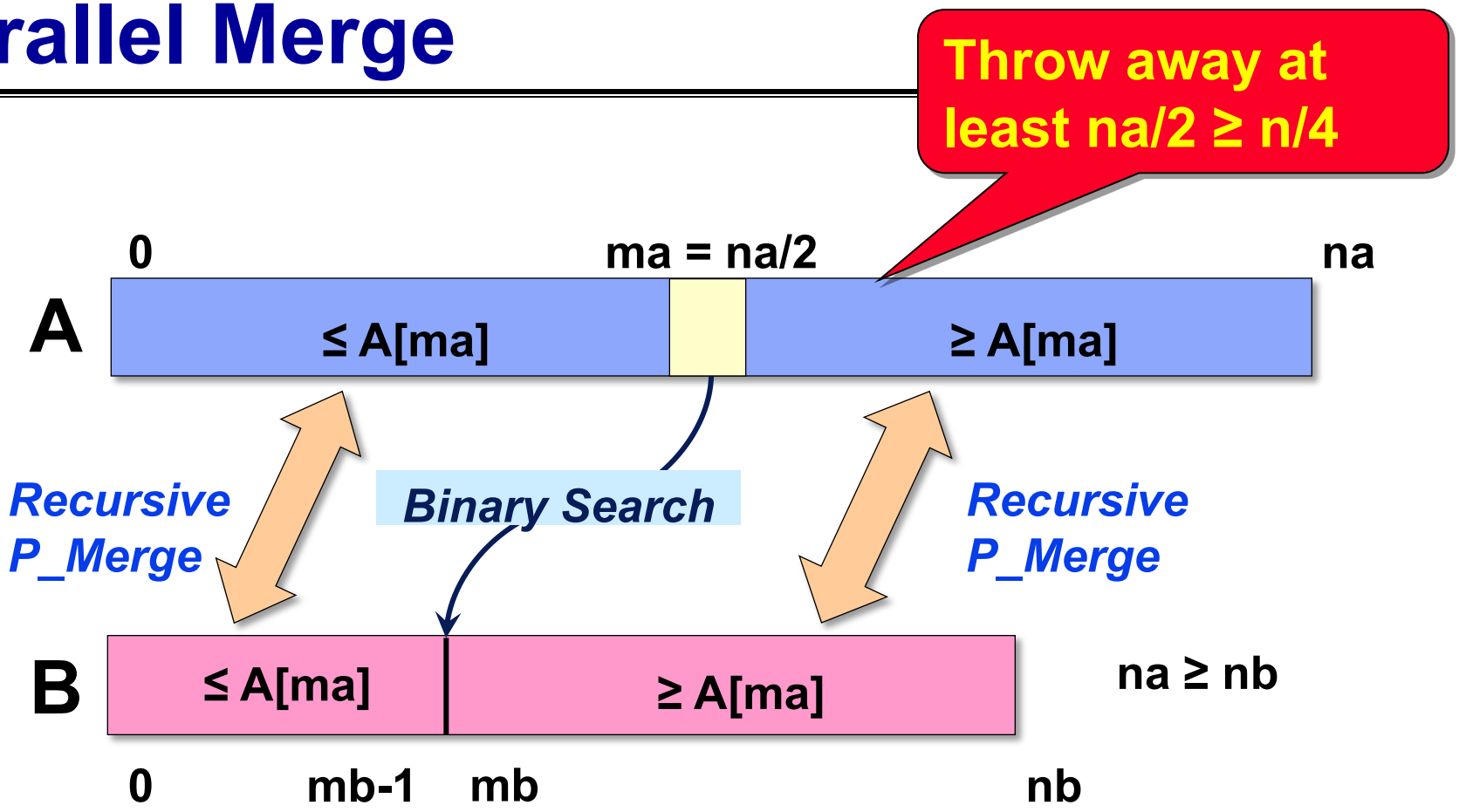
Work: $T_1(n) = \Theta(n \lg n)$

Span: $T_\infty(n) = \Theta(n)$

Parallelism: $$\frac{T_1(n)}{T_\infty(n)} = \Theta(\lg n)$$

We need to parallelize the merge!

Parallel Merge



KEY IDEA: 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 .

Parallel 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
} }
```

Coarsen base cases for efficiency.

Span of Parallel 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,
    #pragma omp task
        P_Merge(C+ma+mb+1,A+
    } } // implicit taskwait
} }
```

CASE 2 (of Master Theorem):

$n^{\log_b a} = n^{\log 4/3\, 1} = 1$

$f(n) = \Theta(n^{\log_b a} \lg^1 n)$

Span: $T_\infty(n) = T_\infty(3n/4) + \Theta(\lg n)$

$= \Theta(\lg^2 n)$

Work of Parallel 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
} }
```

HAIRY!

Work: $T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n)$, where $1/4 \le \alpha \le 3/4$.

Claim: $T_1(n) = \Theta(n)$.

Analysis of Work Recurrence

Work: $T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n)$,
where $1/4 \le \alpha \le 3/4$.

Substitution method: Inductive hypothesis is $T_1(k) \le c_1 k - c_2 \lg k$, where $c_1, c_2 > 0$. Prove that the relation holds, and solve for c_1 and c_2.

$$\begin{aligned} T_1(n) &= T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n) \\ &\le c_1(\alpha n) - c_2 \lg(\alpha n) \\ &\quad + c_1(1-\alpha)n - c_2 \lg((1-\alpha)n) + \Theta(\lg n) \end{aligned}$$

Analysis of Work Recurrence

Work: $T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n)$,
where $1/4 \le \alpha \le 3/4$.

$$
\begin{aligned}
T_1(n) &= T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n) \\
&\le c_1(\alpha n) - c_2 \lg(\alpha n) \\
&\quad + c_1(1-\alpha)n - c_2 \lg((1-\alpha)n) + \Theta(\lg n)
\end{aligned}
$$

Analysis of Work Recurrence

Work: $T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n)$,
where $1/4 \le \alpha \le 3/4$.

$$T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n)$$
$$\le c_1(\alpha n) - c_2\lg(\alpha n) + c_1(1-\alpha)n - c_2\lg((1-\alpha)n) + \Theta(\lg n)$$
$$\le c_1 n - c_2\lg(\alpha n) - c_2\lg((1-\alpha)n) + \Theta(\lg n)$$
$$\le c_1 n - c_2(\lg(\alpha(1-\alpha)) + 2\lg n) + \Theta(\lg n)$$
$$\le c_1 n - c_2\lg n - (c_2(\lg n + \lg(\alpha(1-\alpha))) - \Theta(\lg n))$$
$$\le c_1 n - c_2\lg n$$

by choosing c_2 large enough. Choose c_1 large enough to handle the base case.

Parallelism of P_Merge

Work: $T_1(n) = \Theta(n)$

Span: $T_\infty(n) = \Theta(\lg^2 n)$

Parallelism: $$\frac{T_1(n)}{T_\infty(n)} = \Theta(n/\lg^2 n)$$

Parallel Merge Sort

```
template <typename T>
void P_MergeSort(T *B, T *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    T C[n];
#pragma omp parallel {
  #pragma omp single {
    #pragma omp task
        P_MergeSort(C, A,
    #pragma omp task
        P_MergeSort(C+n/2
    } }
    P_Merge(B, C, C+n/2, n/2
} }
```

CASE 2:

$n^{\log_b a} = n^{\log 2^2} = n$

$f(n) = \Theta(n^{\log_b a} \lg^0 n)$

Work: $T_1(n) = 2T_1(n/2) + \Theta(n) = \Theta(n \lg n)$

Parallel Merge Sort

```
template <typename T>
void P_MergeSort(T *B, T *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    T C[n];
#pragma omp parallel {
  #pragma omp single {
    #pragma omp task
        P_MergeSort(C, A,
    #pragma omp task
        P_MergeSort(C+n/2
    } }
    P_Merge(B, C, C+n/2, n/      -n/2);
} }
```

CASE 2:

$n^{\log_b a} = n^{\log 2^2} = n$

$f(n) = \Theta(n^{\log_b a} \lg^0 n)$

Span: $T_\infty(n) = T_\infty(n/2) + \Theta(\lg^2 n) = \Theta(\lg^3 n)$

Parallelism of P_MergeSort

Work: $T_1(n) = \Theta(n \lg n)$

Span: $T_\infty(n) = \Theta(\lg^3 n)$

Parallelism: $$\frac{T_1(n)}{T_\infty(n)} = \Theta(n/\lg^2 n)$$

Outline

- (Optional) recap Master Method for solving recurrences. Because many of these algorithms are recursive.

- Searching for the kth smallest element in parallel
  - A serial algorithm to warm up
- Sorting in shared memory
- **Sorting in distributed memory**

- One representative algorithm per problem

Bucket and Sample 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 $\Theta(n\log(n/m))$.

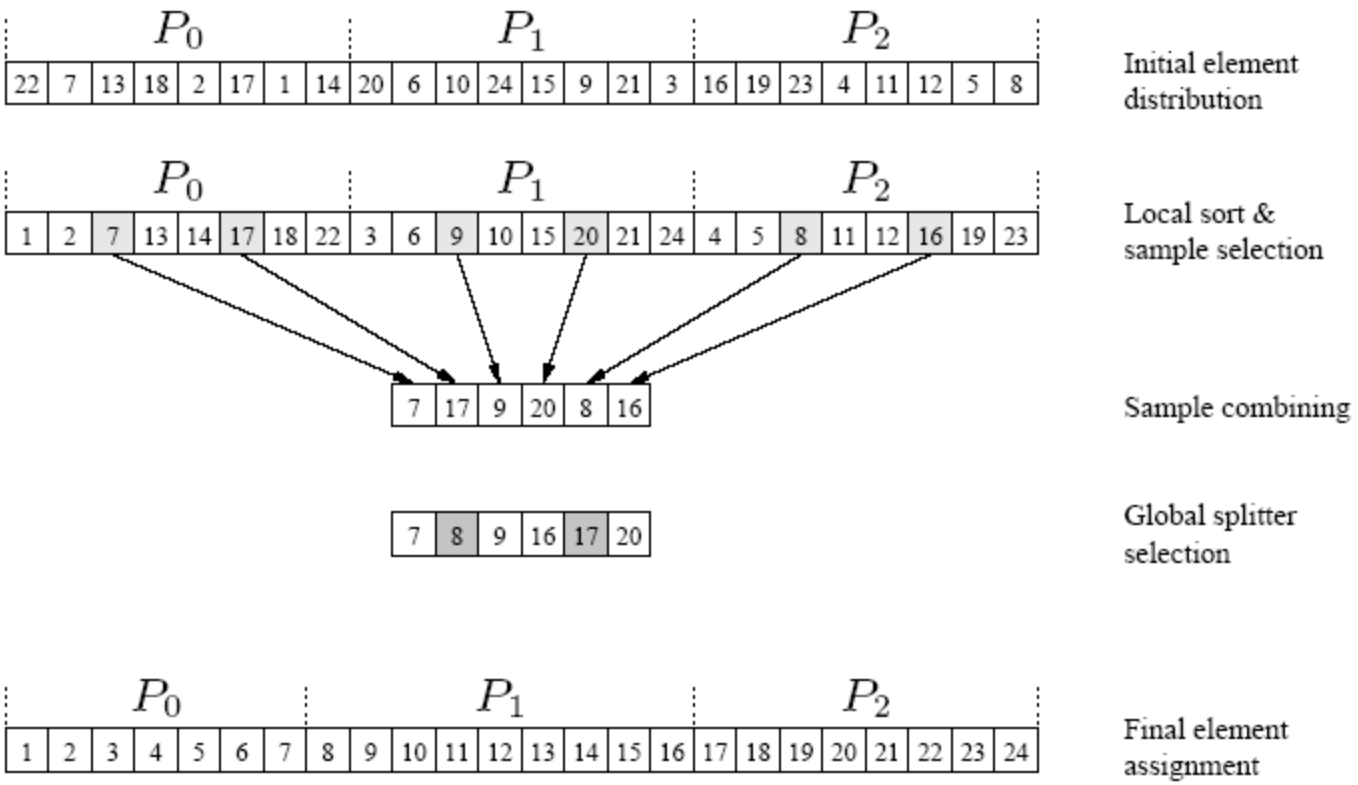
Parallel Bucket Sort

- Parallelizing bucket sort is relatively simple. We can select $m = p$.
- In this case, 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

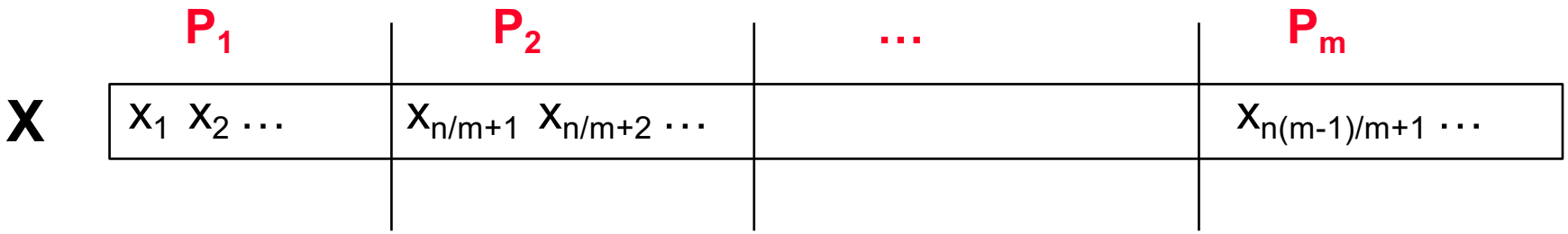
- **Generalization of Quicksort**: from 1 pivot to p pivots. This is done by suitable splitter selection.
- The splitter selection method divides the n elements into m blocks of size n/m each, and sorts each block by using quicksort.
- Choose m-1 **evenly spaced** samples from each sorted block.
- The 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 Sorting by **Regular Sampling**" by Shi and Schaeffer

Parallel Sample Sort



An example of the execution of sample sort on an array with 24 elements on three processes.

"Balance" proof of regular sampling



- 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**
- Pick $y_{m/2}$, $y_{m+m/2}$, … as the final splitters and broadcast them

- **P_i** gets entries x_i such $y_{(i-2)m+m/2} < x_i <= y_{(i-1)m + m/2} = y_{(m-i)m - m/2}$
- Each sample y_i is larger than at least n/m^2 elements of **X**
- So, $y_{(i-2)m+m/2}$ is larger than $lb=((i-2)m+m/2)(n/m^2)$ elements of **X** and
- $y_{(m-i)m - m/2}$ is smaller than $ub=((m – i)m - m/2 + 1)(n/m^2) - 1$ elements of **X**
- $n-(lb+ub) = n - (n-2n/m+n/m^2) = 2n/m - n/m^2 <= 2n/m$
- At worst, load imbalance is 2X.
- In practice, it is much better load balanced

Parallel Sample Sort

- 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!
- **Possible quick hack:** Find the median (recall: parallel selection), exchange data, and recurse on p/2 processors each
- **Alternative** ("A novel parallel sorting algorithm for contemporary architectures" by Cheng, Shah, Gilbert, Edelman): Use exact splitters. The parallel selection algorithm is generalized
- **Alternative (CLASS PROJECT):** Parallelize merging of Y using less space per processor

Parallel Sample Sort: Complexity Analysis

- 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 $\Theta(p^2)$, 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 $O(n/p)$.

Parallel Sample Sort: Analysis

- The total time is given by:

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta\left(p^2\log p\right)}^{\text{sort sample}} + \overbrace{\Theta\left(p\log\frac{n}{p}\right)}^{\text{block partition}} + \overbrace{\Theta(n/p).}^{\text{communication}}$$

- The formula above (wrongly) assumes that communication and computation costs have the same constants (no α/β)

Further reading (beyond Sample Sort):

E. Solomonik and L.V. Kale. Highly scalable parallel sorting, 2010

Hari Sundar, Dhairya Malhotra, and George Biros. "Hyksort: a new variant of hypercube quicksort on distributed memory architectures.", 2013