# Parallel Sorting

Chris Kauffman

*Last Updated:*
*Wed Oct 20 11:57:47 AM CDT 2021*

# Logistics

## Today

- Parallel Sorting: Quicksort
- Wed: Mini-exam 2

## Reading: Grama Ch 9

- Sorting
- Focus on 9.4: Quicksort

# Quick Review

- What is Amdahl's law? What does it say about the speedup achievable by parallel programs?
- How does one calculate the following for a parallel algorithm
  - $S$: Speedup
  - $E$: Efficiency
  - $C$: Cost
- How does the Efficiency of a parallel usually change if the number of processors P increases but the problem size stays the same? If number of procs stays the same but problem size increases?
- What was the major benefit that Cannon's Algorithm provided over a naive implementation of parallel matrix multiply?
- In broad strokes, how was the LU factorization parallelized?

# Sorting

- Much loved computation problem
- What is the best complexity of general purpose (comparison-based) sorting algorithms?
- What are some algorithms which have this complexity?
- What are some other sorting algorithms which aren't so hot?
- What issues need to be addressed to parallelize any sorting algorithm?

# Partition and Quicksort

- Quicksort has $O(N \log N)$ average complexity
- In-place, low overhead sorting, recursive

## Partition

- Partition: select `pivot` value
- Rearrange elements so
  - Left array is $\leq$ `pivot`
  - Right array is $>$ `pivot`
  - `pivot` is in "middle"

```
// A is an array, lo/hi are
// inclusive boundaries
algorithm partition(A, lo, hi) is
  pivot := A[hi]
  boundary := lo
  for j := lo to hi do
    if A[j] <= pivot then
      swap A[boundary] with A[j]
      boundary++
  swap A[boundary] with A[hi]
  return boundary
```

## Quicksort

- Partition into two parts
- Recurse on both halves
- Bail out when boundaries `lo`/`hi` cross

```
algorithm quicksort(A, lo, hi) is
 if lo < hi then
   p := partition(A, lo, hi)
   quicksort(A, lo, p - 1)
   quicksort(A, p + 1, hi)
```

# Practical Parallel Sorting Setup

- Input array A of size N is already spread across P processors (no need to scatter)

```
P0: A[] = { 84 31 21 28 }
P1: A[] = { 17 20 24 84 }
P2: A[] = { 24 11 31 99 }
P3: A[] = { 13 32 26 75 }
```

- Goal: Numbers sorted across processors. Smallest on P0, next smallest on P1, etc.

```
P0: A[] = { 11 13 17 20 }
P1: A[] = { 21 24 24 26 }
P2: A[] = { 28 31 32 33 }
P3: A[] = { 75 84 84 99 }
```

- Want to use $P$ processors as effectively as possible
- Favor bulk communication over many small messages

# Exercise: Parallel Quicksort

- Find a way to parallelize quicksort
- **Hint:** The last step is each processor sorting its own data using a serial algorithm. Try to arrange data so this is possible.

```
START:
P0: A[] = { 84 32 21 28 }
P1: A[] = { 17 20 25 85 }
P2: A[] = { 24 11 31 99 }
P3: A[] = { 13 32 26 75 }

GOAL
P0: A[] = { 11 13 17 20 }
P1: A[] = { 21 24 25 26 }
P2: A[] = { 28 31 32 33 }
P3: A[] = { 75 84 85 99 }
```

```
SERIAL ALGORITHM
algorithm quicksort(A, lo, hi) is
 if lo < hi then
   p := partition(A, lo, hi)
   quicksort(A, lo, p - 1)
   quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
  pivot := A[hi]
  boundary := lo
  for j := lo to hi - 1 do
    if A[j] <= pivot then
      swap A[boundary] with A[j]
      boundary++
  swap A[i] with A[hi]
  return boundary
```

# **Answers**: Parallel Quicksort Ideas 1 / 2

```
A[] = { 84 32 21 11 | 17 20 25 85 | 24 28 31 99 | 13 33 26 75 }
        P0            P1            P2            P3

Partition(pivot=26) on each processor
A[] = { 21 11 84 32 | 17 20 25 85 | 24 28 31 99 | 13 26 33 75 }
Boundary:   ^                ^       ^                ^
Counts: P0: 2         P1: 3         P2: 1         P3: 2

Calculate which data goes where
A[] = { 21 11 84 32 | 17 20 25 85 | 24 28 31 99 | 13 26 33 75 }
        P0 P0 P2 P2   P0 P0 P1 P2   P1 P2 P3 P3   P1 P1 P3 P3

Re-arrange so values <= 26 on P0 and P1, > 26 on P2 and P3
A[] = { 21 11 17 20 | 25 24 13 26 | 84 32 85 28 | 31 99 33 75 }
        P0            P1            P2            P3

Split the world: 2 groups
A[] = { 21 11 17 20 | 25 24 13 26}|{84 32 85 28 | 31 99 33 75 }
        P0            P1            P2            P3
```

# **Answers**: Parallel Quicksort Ideas 2 / 2

```
Each half partitions on different pivot value
        P0-P1: Partition(pivot=20)  P2-P3: Partition(pivot=33)
A[] = { 11 17 20 21 | 13 25 24 26}|{28 32 84 85 | 31 33 99 75 }
Boundary:      ^       ^                ^              ^
Counts: P0: 3         P1: 1          P2: 2          P3: 2

Calculate which data goes where
A[] = { 11 17 20 21 | 13 25 24 26}|{28 32 84 85 | 31 33 99 75 }
        P0 P0 P0 P1   P0 P1 P1 P1   P2 P2 P3 P3   P2 P2 P3 P3

Re-arrange values to proper processors
A[] = { 11 17 20 13 | 21 25 24 25}|{28 32 31 33 | 84 85 99 75 }
        P0            P1             P2             P3

Split the world: 4 groups
A[] = { 11 17 20 13}|{21 25 24 25}|{28 32 31 33}|{84 85 99 75 }
        P0            P1             P2             P3

4 groups == 4 processors, all processors sort locally
A[] = { 11 13 17 20}|{21 24 25 25}|{28 31 32 33}|{75 84 85 99 }
        P0            P1             P2             P3
```

# Quicksort Difficulties

## Communication

- Determine which data go to which processors, how many send/receives are required
- Opportunity for **all-to-all communications** in MPI

## Recursing

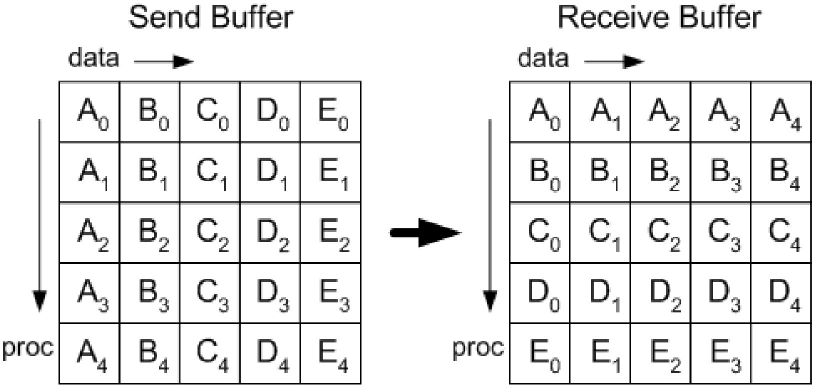
- Recursive step of algorithm requires smaller “worlds”
- Use MPI’s **communicator splitting** capability

## Pivot Value Selection

- In example, pivot values were cherry-picked to get even distribution of data among processors
- A bad pivot splits data unevenly, is annoying for serial Quicksort, shaves off processors in parallel quicksort destroying efficiency

# All-to-All Personalized Communication

All-to-all personalized communication: like every processor scattering to every other processor.




Source: Cornell University Center for Advanced Computing

# MPI_Alltoall

- Standard version: every processor gets a slice of `sendbuf`, same sized data
- Vector version allows different sized slices (appropriate for quicksort)

```
int MPI_Alltoall(
 void *sendbuf, int sendcount, MPI_Datatype sendtype,
 void *recvbuf, int recvcount, MPI_Datatype recvtype,
 MPI_Comm comm);

int MPI_Alltoallv(
 void *sendbuf, int sendcounts[], int sdispls[], MPI_Datatype sendtype,
 void *recvbuf, int recvcounts[], int rdispls[], MPI_Datatype recvtype,
 MPI_Comm comm);
```

# Exercise: Redistribution during Quicksort

- After `partition`, procs must redistribute data
- Use All-Gather so that all procs have the following table

Element Count vs Pivot

| Proc | <= | > |
|---|---|---|
| P0 | 2 | 2 |
| P1 | 3 | 1 |
| P2 | 1 | 3 |
| P3 | 2 | 2 |

Want each Proc to calculate its own Count/Displ in this table:

| P# | | P0 | P1 | P2 | P3 | | P0 | P1 | P2 | P3 | P# |
|---|---|---|---|---|---|---|---|---|---|---|---|
| P0 | RecvCount | 2 | 2 | 0 | 0 | SendCount | 2 | 0 | 2 | 0 | P0 |
| P1 | | 0 | 1 | 1 | 2 | | 2 | 1 | 1 | 0 | P1 |
| P2 | | 2 | 1 | 1 | 0 | | 0 | 1 | 1 | 2 | P2 |
| P3 | | 0 | 0 | 2 | 2 | | 0 | 2 | 0 | 2 | P3 |
| | | | | | | | | | | | |
| P0 | RecvDispl | 0 | 2 | 4 | 4 | SendDispl | 0 | 0 | 2 | 0 | P0 |
| P1 | | 0 | 0 | 1 | 2 | | 0 | 2 | 3 | 4 | P1 |
| P2 | | 0 | 2 | 3 | 4 | | 0 | 0 | 1 | 2 | P2 |
| P3 | | 0 | 0 | 0 | 2 | | 0 | 0 | 2 | 2 | P3 |

- Describe the process of calculating `RecvCount`
- Given `RecvCount`, how can one calculate `RecvDispl`

# **Answers**: Redistribution during Quicksort

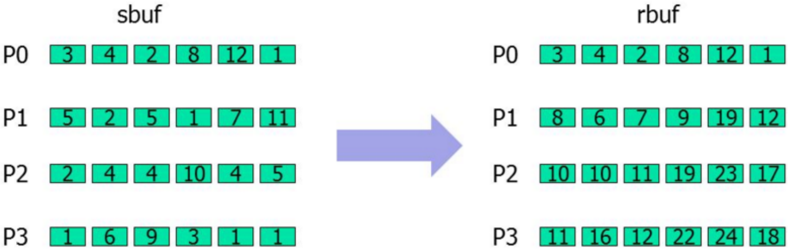
- `RecvCount` can be calculate through an iterative process
- Compute the **prefix sum** of for the table

```
Element Count vs Pivot  PS[]: PREFIX SUMS
| Proc | <= | > |        | Proc | <= | > |
|------+----+---|        |------+----+---|
| P0   |  2 | 2 |        | P0   |  2 | 2 |
| P1   |  3 | 1 |        | P1   |  5 | 3 |
| P2   |  1 | 3 |        | P2   |  6 | 6 |
| P3   |  2 | 2 |        | P3   |  8 | 8 |
```

- Know each proc must send/receive `N / P = 4` elements
- Procs receiving `<= pivot`, proc # `i`, scan column 0 for
  - First partner is proc `F` where `PS[F,0] <= 4*i`
  - Last partner is proc `L` where `PS[L,0] >= 4*(i+1)`
- Procs receiving `> pivot`, proc # `i`, scan column 1 for
  - First partner is proc `F` where `PS[F,1] <= 4*(i-2)`
  - Last partner is proc `L` where `PS[L,1] >= 4*(i-2+1)`
- Actual code will need to do some arithmetic (e.g. P1 receives 1 element from itself)
- `RecvDispl` is the **prefix sum** of `RecvCount`

# Prefix Sums / Scan

Prefix Sums or Prefix Scans are supported in parallel via MPI




```
int MPI_Scan(const void *sendbuf, void *recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

- Similar to reduction but only add on values from procs <= `proc_id`
- op can be sum/max/min/etc.
- In simple Quicksort implementations, **don't use parallel prefix scan** as this does not yield enough info to calculate send/receive partners

# Overall Flow

1. Pivot selection (open question how to do this right)
2. Broadcast of pivot value
3. Each processor `partition`'s its data
4. All-gather to get element/pivot counts
5. Calculate send/receives
6. Redistribute data via `MPI_Alltoallv()`
7. And then...

# Splitting the World

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
                   MPI_Comm *newcomm);
```

- `comm` is the old communicator (start with `MPI_COMM_WORLD`)
- `color` is which sub-comm to go into
  - Colors `0,1` splits into 2 communicators
  - Colors `0,1,2,3` splits into 4 communicators
  - Etc.
- `key` establishes rank in new sub-comm, usually `proc_id`
- `newcomm` is filled in with a new communicator
- Examine `04-mpi-code/comm_split.c`
- In Quicksort, new comm is different for lower/upper procs

# Exercise: Pivot Selection

- So far have assumed a "good" pivot can be found
- Pivot evenly splits N/2 data, half to lower # processors, half to upper

Discuss the following questions with a neighbor

1. What if the pivot is poorly selected? E.g. 1/4 below pivot, 3/4 above? Could the algorithm adapt?
2. How could one avoid a bad pivot? Discuss a some strategies
3. Is there a way to avoid recusing entirely?

# **Answers**: Pivot Selection 1/2

Discuss the following questions with a neighbor

1. What if the pivot is poorly selected? E.g. 1/4 below pivot, 3/4 above? Could the algorithm adapt?

   *With some additional computation, can split the world unevenly: 1/4 procs assigned to "low" numbers, 3/4 to "high" numbers. Still broken if a tiny fraction of the array is lower/higher than the pivot: should just try another pivot at that point or use a scheme that prevents poor pivot selection.*

2. How could one avoid a bad pivot? Discuss a some strategies

   *Lots of these exist, some mentioned in the textbook such as having a randomly selected processor compute its median and broadcast it as the pivot (main text of Grama) or have processors sample random elements, perform All-Gather, then compute a median as the common pivot (Grama Exercise 9.21).*

# **Answers**: Pivot Selection 1/2

3. Is there a way to avoid recusing entirely, e.g. single multiway pivot?

   *Grama Exercise 9.20 explores this:*

   - *Each proc samples elements, often around* $\log(N)$ *elements, and procs perform an All-Gather*
   - *All procs use common sample to select* $P-1$ *common pivots.*
   - *Elements between pivots are sent directly to final destination procs in an All-to-All communication.*
   - *Local sorting commences.*

# Exercise: Odd-Even Sort

- Variant of bubble sort which splits bubbling into odd/even phases
- $O(N^2)$ complexity of serial algorithm
- There is potential for parallelism here: what is it?
  - Consider simple case where each $P = N$: each proc hold a single number
  - What can be parallelized and how?

```
ODD_EVEN_SORT(A[]) {
  N = length(A[])
  for(r=0 to N-1){
    if(r is even){
      for(i=0; i<N-1; i+=2){
        compare_exchange(A, i, i+1);
      }
    }
    if(r is odd){
      for(i=1; i<N-1; i+=2){
        compare_exchange(A, i, i+1);
      }
    }
  }
}

COMPARE_EXCHANGE(A[], i, j){
  if(A[i] > A[j]){
    temp = A[i]
    A[i] = A[j]
    A[j] = temp
  }
}
```

# **Answers**: Odd-Even Sort

- There is potential for parallelism here: what is it?
- Consider simple case where each $P = N$: each proc hold a single number
- What can be parallelized and how?
  - *The inner loops of `compare_exchange()` can be executed in parallel as it involves communication between 2 procs to potentially exchange elements but only with a single partner.*
  - *Even iterations, lower evens exchange with higher odds*
  - *Odd iterations lower odds exchange with higher evens*
  - *Exchange can be done via a Send/Receive of elements and then "keeping" the appropriate element, min on lower proc, max on higher proc*

# Odd-Even Sort in Practice

- As before, unrealistic to have $P = N$, rather each proc holds $N/P$ elements of the array `A[]`
- `COMPARE_EXCHANGE()` becomes `COMPARE_SPLIT()`

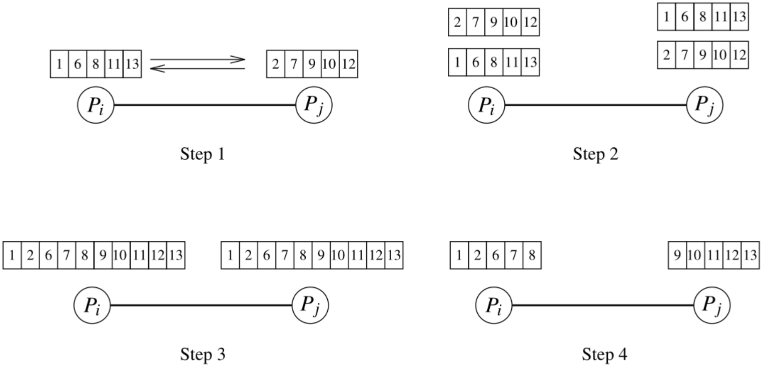



**Figure 9.2** A compare-split operation. Each process sends its block of size $n/p$ to the other process. Each process merges the received block with its own block and retains only the appropriate half of the merged block. In this example, process $P_i$ retains the smaller elements and process $P_j$ retains the larger elements.

# Analysis of Odd-Even Sort

- Initially all procs sort their local array: $O(\frac{N}{P} \log \frac{N}{P}$
- Conduct $P$ Outer iterations of `ODD_EVEN_SORT()`
- Each odd/even inner loop is done in parallel by all procs communicating with a neighbor
- Neighbor procs exchange ararys: $O(t_s + t_w \frac{N}{P})$
- Each proc then performs a compare/split: $O(\frac{2N}{P}$
- Overall complexity of parallel algorithm:

$$
\begin{aligned}
T_{par} &= O\left(\frac{N}{P} \log \frac{N}{P}\right) + P \times \left(O\left(t_s + t_w \frac{N}{P}\right) + O\left(\frac{2N}{P}\right)\right) \\
&= O\left(\frac{N}{P} \log \frac{N}{P}\right) + O(N) + O(N)
\end{aligned}
$$

Isoefficiency? : Reported in textbook as $O(P2^P)$, linear increase in $P$ requires exponential increase in problem size to maintain efficiency. Verifying this is a good exercise.

# Sorting Extras

## Odd-Even Sort to Shell Sort

- Allowing bigger "moves" in odd-even sort can improve practical efficiency of algorithm
- Shell Sort provides a mechanism for this: neighbors selected according to a "gap" scheme, less known sort with yet mysterious complexity analysis

## Sorting Hardware

- Grama Ch 9.1 discusses Sorting networks, specialized hardware which can implement sorting
- With $N$ processors, can implement Bitonic Sort in a sorting network and achieve $T_{par} = O(\log^2 N)$
- Hardware that implements sorting networks is not common but...
- Can do this in GPUs and may revisit this algorithm when we do CUDA programming