CS420 – Lecture 21

Omri Mor

Fall 2018

Radix Sort

Comparison Sort

- List of items and comparator
- Return list of items, ordered by comparator

Comparison Sort

- List of items and comparator
- Return list of items, ordered by comparator
- Time Complexity?
  - Best: depends on algorithm
  - Average: $O(n \log n)$
  - Worst?

Comparison Sort

- List of items and comparator
- Return list of items, ordered by comparator
- Time Complexity?
  - Best: depends on algorithm
  - Average: $O(n \log n)$
  - Worst? $O(n \log n)$, $O(n^2)$, ...

Comparison Sort

- List of items and comparator
- Return list of items, ordered by comparator
- Time Complexity?
  - Best: depends on algorithm
  - Average: $O(n \log n)$
  - Worst? $O(n \log n)$, $O(n^2)$, ...
- Space Complexity?
  - Depends on algorithm
  - $O(1)$, $O(\log n)$, $O(n)$, ...

Comparison Sort: Examples

- Insertion sort:
  - $O(n)$ best
  - $O(n^2)$ average and worst
  - $O(1)$ space

Comparison Sort: Examples

- Insertion sort:
  - $O(n)$ best
  - $O(n^2)$ average and worst
  - $O(1)$ space
- Merge sort:
  - $O(n \log n)$ best, average, and worst
  - $O(n)$ space

Comparison Sort: Examples

- Insertion sort:
  - $O(n)$ best
  - $O(n^2)$ average and worst
  - $O(1)$ space
- Merge sort:
  - $O(n \log n)$ best, average, and worst
  - $O(n)$ space
- Quick sort:
  - $O(n \log n)$ best and average
  - $O(n^2)$ worst
  - $O(\log n)$ space

Non-Comparative Sort

- Will no one rid me of this troublesome comparator?
- Must restrict scope: can't sort arbitrary items
  - Focus on integers
  - But not strictly limited—requires "positional notation"
- Algorithm outline:

Non-Comparative Sort

- Will no one rid me of this troublesome comparator?
- Must restrict scope: can't sort arbitrary items
  - Focus on integers
  - But not strictly limited—requires "positional notation"
- Algorithm outline:
  1. Take least significant group of digits (or bits)

Non-Comparative Sort

- Will no one rid me of this troublesome comparator?
- Must restrict scope: can't sort arbitrary items
  - Focus on integers
  - But not strictly limited—requires "positional notation"
- Algorithm outline:
  1. Take least significant group of digits (or bits)
  2. Group keys based on digits/bits, keeping order within group

Non-Comparative Sort

- Will no one rid me of this troublesome comparator?
- Must restrict scope: can't sort arbitrary items
  - Focus on integers
  - But not strictly limited—requires "positional notation"
- Algorithm outline:
  1. Take least significant group of digits (or bits)
  2. Group keys based on digits/bits, keeping order within group
  3. Repeat with next significant digits/bits

Decimal Example

Histogram I

0	1	2	3	4	5	6
23	45	43	54	76	14	13

Digit	Count
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

Histogram I

0	1	2	3	4	5	6
23	45	43	54	76	14	13

Digit	Count
0	0
1	0
2	0
3	1
4	0
5	0
6	0
7	0
8	0
9	0

Histogram I

0	1	2	3	4	5	6
23	45	43	54	76	14	13

Digit	Count
0	0
1	0
2	0
3	1
4	0
5	1
6	0
7	0
8	0
9	0

Histogram I

0	1	2	3	4	5	6
23	45	43	54	76	14	13

Digit	Count
0	0
1	0
2	0
3	2
4	0
5	1
6	0
7	0
8	0
9	0

Histogram I

0	1	2	3	4	5	6
23	45	43	54	76	14	13

Digit	Count
0	0
1	0
2	0
3	2
4	1
5	1
6	0
7	0
8	0
9	0

Histogram I

0	1	2	3	4	5	6
23	45	43	54	76	14	13

Digit	Count
0	0
1	0
2	0
3	2
4	1
5	1
6	1
7	0
8	0
9	0

Histogram I

0	1	2	3	4	5	6
23	45	43	54	76	14	13

Digit	Count
0	0
1	0
2	0
3	2
4	2
5	1
6	1
7	0
8	0
9	0

Histogram I

0	1	2	3	4	5	6
23	45	43	54	76	14	13

Digit	Count
0	0
1	0
2	0
3	3
4	2
5	1
6	1
7	0
8	0
9	0

Exclusive Scan I

0	1	2	3	4	5	6
23	45	43	54	76	14	13

Digit	Count
0	0
1	0
2	0
3	3
4	2
5	1
6	1
7	0
8	0
9	0

$sum = 0$

Exclusive Scan I

0	1	2	3	4	5	6
23	45	43	54	76	14	13

Digit	Count
0	0
1	0
2	0
3	3
4	2
5	1
6	1
7	0
8	0
9	0

$sum = 0$

Exclusive Scan I

0	1	2	3	4	5	6
23	45	43	54	76	14	13

Digit	Count
0	0
1	0
2	0
3	3
4	2
5	1
6	1
7	0
8	0
9	0

$sum = 0$

Exclusive Scan I

0	1	2	3	4	5	6
23	45	43	54	76	14	13

Digit	Count
0	0
1	0
2	0
3	3
4	2
5	1
6	1
7	0
8	0
9	0

$sum = 0$

Exclusive Scan I

0	1	2	3	4	5	6
23	45	43	54	76	14	13

Digit	Count
0	0
1	0
2	0
3	0
4	2
5	1
6	1
7	0
8	0
9	0

$sum = 3$

Exclusive Scan I

0	1	2	3	4	5	6
23	45	43	54	76	14	13

Digit	Count
0	0
1	0
2	0
3	0
4	3
5	1
6	1
7	0
8	0
9	0

$sum = 5$

Exclusive Scan I

0	1	2	3	4	5	6
23	45	43	54	76	14	13

Digit	Count
0	0
1	0
2	0
3	0
4	3
5	5
6	1
7	0
8	0
9	0

$sum = 6$

Exclusive Scan I

0	1	2	3	4	5	6
23	45	43	54	76	14	13

Digit	Count
0	0
1	0
2	0
3	0
4	3
5	5
6	6
7	0
8	0
9	0

$sum = 7$

Exclusive Scan I

0	1	2	3	4	5	6
23	45	43	54	76	14	13

Digit	Count
0	0
1	0
2	0
3	0
4	3
5	5
6	6
7	7
8	0
9	0

$sum = 7$

Exclusive Scan I

0	1	2	3	4	5	6
23	45	43	54	76	14	13

Digit	Count
0	0
1	0
2	0
3	0
4	3
5	5
6	6
7	7
8	7
9	0

$sum = 7$

Exclusive Scan I

0	1	2	3	4	5	6
23	45	43	54	76	14	13

Digit	Count
0	0
1	0
2	0
3	0
4	3
5	5
6	6
7	7
8	7
9	7

$sum = 7$

Emplace I

0	1	2	3	4	5	6
23	45	43	54	76	14	13

Digit	Count
0	0
1	0
2	0
3	0
4	3
5	5
6	6
7	7
8	7
9	7

Emplace I

0	1	2	3	4	5	6
23	45	43	54	76	14	13

Digit	Count
0	0
1	0
2	0
3	0
4	3
5	5
6	6
7	7
8	7
9	7

Emplace I

0	1	2	3	4	5	6
23	45	43	54	76	14	13
23						

Digit	Count
0	0
1	0
2	0
3	1
4	3
5	5
6	6
7	7
8	7
9	7

Emplace I

0	1	2	3	4	5	6
23	45	43	54	76	14	13
23						

Digit	Count
0	0
1	0
2	0
3	1
4	3
5	5
6	6
7	7
8	7
9	7

Emplace I

0	1	2	3	4	5	6
23	45	43	54	76	14	13
23					45	

Digit	Count
0	0
1	0
2	0
3	1
4	3
5	6
6	6
7	7
8	7
9	7

Emplace I

0	1	2	3	4	5	6
23	45	43	54	76	14	13
23					45	

Digit	Count
0	0
1	0
2	0
3	1
4	3
5	6
6	6
7	7
8	7
9	7

Emplace I

0	1	2	3	4	5	6
23	45	43	54	76	14	13
23	43				45	

Digit	Count
0	0
1	0
2	0
3	2
4	3
5	6
6	6
7	7
8	7
9	7

Emplace I

0	1	2	3	4	5	6
23	45	43	54	76	14	13
23	43				45	

Digit	Count
0	0
1	0
2	0
3	2
4	3
5	6
6	6
7	7
8	7
9	7

Emplace I

0	1	2	3	4	5	6
23	45	43	54	76	14	13
23	43		54		45	

Digit	Count
0	0
1	0
2	0
3	2
4	4
5	6
6	6
7	7
8	7
9	7

Emplace I

0	1	2	3	4	5	6
23	45	43	54	76	14	13
23	43		54		45	

Digit	Count
0	0
1	0
2	0
3	2
4	4
5	6
6	6
7	7
8	7
9	7

Emplace I

0	1	2	3	4	5	6
23	45	43	54	76	14	13
23	43		54		45	76

Digit	Count
0	0
1	0
2	0
3	2
4	4
5	6
6	7
7	7
8	7
9	7

Emplace I

0	1	2	3	4	5	6
23	45	43	54	76	14	13
23	43		54		45	76

Digit	Count
0	0
1	0
2	0
3	2
4	4
5	6
6	7
7	7
8	7
9	7

Emplace I

0	1	2	3	4	5	6
23	45	43	54	76	14	13
23	43		54	14	45	76

Digit	Count
0	0
1	0
2	0
3	2
4	5
5	6
6	7
7	7
8	7
9	7

Emplace I

0	1	2	3	4	5	6
23	45	43	54	76	14	13
23	43		54	14	45	76

Digit	Count
0	0
1	0
2	0
3	2
4	5
5	6
6	7
7	7
8	7
9	7

Emplace I

0	1	2	3	4	5	6
23	45	43	54	76	14	13
23	43	13	54	14	45	76

Digit	Count
0	0
1	0
2	0
3	3
4	5
5	6
6	7
7	7
8	7
9	7

Histogram II

0	1	2	3	4	5	6
23	43	13	54	14	45	76

Digit	Count
0	0
1	2
2	1
3	0
4	2
5	1
6	0
7	1
8	0
9	0

Exclusive Sum II

0	1	2	3	4	5	6
23	43	13	54	14	45	76

Digit	Count
0	0
1	0
2	2
3	3
4	3
5	5
6	6
7	6
8	7
9	7

Emplace II

0	1	2	3	4	5	6
23	43	13	54	14	45	76
13	14	23	43	45	54	76

Digit	Count
0	0
1	2
2	3
3	3
4	5
5	6
6	6
7	7
8	7
9	7

Serial Radix Sort I

```
void sort(int32_t **keys_p, size_t N)
{
        int32_t *keys = *keys_p;
        int32_t *new_keys = malloc(sizeof(int32_t) * N);
        size_t hist[2];

        /* int32_t is 4 bytes = 32 bits */
        for (int r = 0; r < 32; r++) {
                /* reset histogram */
                memset(hist, 0, sizeof(hist));

                /* compute histogram of key bits */
                for (size_t i = 0; i < N; i++) {
                        size_t bit = (keys[i] >> r) & 0x1;
                        hist[bit]++;
                }

                /* exclusive scan on histogram */
                size_t sum = 0;
                for (int i = 0; i < 2; i++) {
```

Serial Radix Sort II

```
                size_t tmp = hist[i];
                hist[i] = sum;
                sum += tmp;
        }

        /* put into place */
        for (size_t i = 0; i < N; i++) {
                size_t bit = (keys[i] >> r) & 0x1;
                size_t index = hist[bit]++;
                new_keys[index] = keys[i];
        }

        /* swap keys and new_keys */
        int32_t *tmp = keys;
        keys = new_keys;
        new_keys = tmp;
    }
    *keys_p = keys;
    free(new_keys);
}
```

Serial Radix Sort: Analysis

- Time
  - Histogram: $O(n)$
  - Scan: $O(1)$
  - Emplace: $O(n)$
  - Each repeated k times
    - In general, k must be at least $\log n$ if all elements are distinct
    - Here, k is a constant—32
    - So depending on how you consider it, time is $O(n \log n)$ or $O(n)$

Serial Radix Sort: Analysis

- Time
  - Histogram: $O(n)$
  - Scan: $O(1)$
  - Emplace: $O(n)$
  - Each repeated k times
    - In general, k must be at least $\log n$ if all elements are distinct
    - Here, k is a constant—32
    - So depending on how you consider it, time is $O(n \log n)$ or $O(n)$
- Space
  - Histogram: $O(1)$
  - Extra array: $O(n)$

Parallel Radix Sort

Parallel Radix Sort

- Split array into P equal chunks
- Compute local histograms in parallel
- Compute global histogram
- Emplace in parallel

Global Histogram

- Each processor p knows how many elements it has in each bin i
- To emplace in the right array offset, must know:
  - Sum of counts for all processors of all bins prior to i
    - $\sum_{n=0}^{i-1} \sum_{k=0}^{P} H_{k,n}$
    - Exclusive scan over a reduction on the local histograms
  - Sum of counts for all processors prior to p of bin i
    - $\sum_{k=0}^{p-1} H_{k,i}$
    - Exclusive scan over the local histograms
- Adding these together allows each processor to know its initial position in the global array for each bin

Analysis

- Histogram and emplace are $O(\frac{n}{P})$
- Reduction and exclusive scan are $O(\log P)$ in parallel
- So total is $O(\frac{n}{P} + \log P)$ or $O((\frac{n}{P} + \log P) \log n)$
  - $P = 1$: $O(n)$ or $O(n \log n)$
  - $P = n$: $O(\log n)$ or $O(\log^2 n)$
- So with an infinite number of processors, can do in $O(\log n)$ time

OpenMP Radix Sort

OpenMP Radix Sort I

```
void sort_omp(int32_t **keys_p, size_t N)
{
        int32_t *keys = *keys_p;
        int32_t *new_keys = malloc(sizeof(int32_t) * N);
        size_t global_hist[2];

        #pragma omp parallel
        {
                const int num_threads = omp_get_num_threads();
                size_t local_hist[2];

                /* int32_t is 4 bytes = 32 bits */
                for (int r = 0; r < 32; r++) {
                        /* reset global histogram, single thread */
                        #pragma omp single nowait
                        memset(global_hist, 0, sizeof(global_hist));

                        /* reset local histogram */
                        memset(lcoal_hist, 0, sizeof(local_hist));
```

OpenMP Radix Sort II

```
/* compute local histogram of key bits */
#pragma omp for schedule(static) nowait
for (size_t i = 0; i < N; i++) {
        size_t bit = (keys[i] >> r) & 0x1;
        local_hist[bit]++;
}

/* loop over each thread in order */
#pragma omp for schedule(monotonic: static, 1) ordered
for (int t = 0; t < num_threads; t++) {
        /* compute reduction and exclusive scan */
        #pragma omp ordered
        for (size_t i = 0; i < 2; i++) {
                size_t tmp = global_hist[i];
                global_hist[i] += local_hist[i];
                local_hist[i] = tmp;
}
```

OpenMP Radix Sort III

```
/* exclusive scan over global histogram */
#pragma omp single
{
        size_t sum = 0;
        for (int i = 0; i < 2; i++) {
                size_t tmp = global_hist[i];
                global_hist[i] = sum;
                sum += tmp;
        }
}

/* compute indices */
for (int i = 0; i < 2; i++) {
        local_hist[i] += global_hist[i];
}
```

OpenMP Radix Sort IV

```
                        /* put into place */
                        #pragma omp for schedule(static)
                        for (size_t i = 0; i < N; i++) {
                                size_t bit = (keys[i] >> r) & 0x1;
                                size_t index = local_hist[bit]++;
                                new_keys[index] = keys[i];
                        }

                        /* swap keys and new_keys */
                        #pragma omp single
                        {
                                int32_t *tmp = keys;
                                keys = new_keys;
                                new_keys = tmp;
                        }
                }
        }
        *keys_p = keys;
        free(new_keys);
}
```

OpenMP Comments

- Reduction and exclusive scans done serially
- Can be done in parallel using OpenMP tasks
- OpenMP 5.0 has `scan` directive
  - Implementation simpler than using tasks
  - OpenMP 5.0 is new—standard released November 2018
  - Compiler support is sparse or nonexistent

MPI Radix Sort

MPI Radix Sort I

```
/* distributed array, each process has N elements */
void sort_mpi(int32_t **keys_p, size_t N)
{
        int32_t *keys = *keys_p;
        int32_t *new_keys = malloc(sizeof(int32_t) * N);
        MPI_Request *reqs = malloc(sizeof(MPI_Request) * N * 2);
        MPI_Count local_hist[2];
        MPI_Count global_hist[2];
        MPI_Count prior_hist[2];

        /* int32_t is 4 bytes = 32 bits */
        for (int r = 0; r < 32; r++) {
                /* reset local histogram */
                memset(local_hist, 0, sizeof(local_hist));

                /* compute local histogram of key bits */
                for (size_t i = 0; i < N; i++) {
                        size_t bit = (keys[i] >> r) & 0x1;
                        local_hist[bit]++;
                }
```

MPI Radix Sort II

```
/* compute reduction on histogram */
MPI_Allreduce(local_hist, global_hist, 2,
              MPI_COUNT, MPI_SUM, MPI_COMM_WOLRD);

/* compute histogram for prior processes */
MPI_Exscan(local_hist, prior_hist, 2,
           MPI_COUNT, MPI_SUM, MPI_COMM_WORLD);

/* exclusive scan over global histogram */
MPI_Count sum = 0;
for (int i = 0; i < 2; i++) {
        MPI_Count tmp = global_hist[i];
        global_hist[i] = sum;
        sum += tmp;
}

/* compute starting indices */
for (int i = 0; i < 2; i++) {
        local_hist[i] = global_hist[i] + prior_hist[i];
}
```

MPI Radix Sort III

```
/* post receives */
for (size_t i = 0; i < N; i++) {
        MPI_Irecv(&new_keys[i], 1, MPI_INT32_T, MPI_ANY_SOURCE,
                  i, MPI_COMM_WORLD, &reqs[i]);
}

/* send keys to right place */
for (size_t i = 0; i < N; i++) {
        size_t bit = (keys[i] >> r) & 0x1;
        MPI_Count index = local_hist[bit]++;
        int dest = index / N;
        int local_index = index % N;
        MPI_Isend(&keys[i], 1, MPI_INT32_T, dest,
                  local_index, MPI_COMM_WORLD, &reqs[i+N]);
}

/* complete communication */
MPI_Waitall(2*N, reqs, MPI_STATUSES_IGNORE);
```

MPI Radix Sort IV

```
            /* swap keys and new_keys */
            int32_t *tmp = keys;
            keys = new_keys;
            new_keys = tmp;
        }
        *keys_p = keys;
        free(new_keys);
        free(reqs);
}
```

MPI Tag Limit

- Need one send and receive for each element in the array
- Use wildcard source (`MPI_ANY_SOURCE`) since it's impossible to know who has the element
- Each receive needs a different tag—match based on local index
- MPI-3.1, §8.1.2, pp. 335:

  > *Tag values range from 0 to the value returned for `MPI_TAG_UB`, inclusive. These values are guaranteed to be unchanging during the execution of an MPI program. In addition, the tag upper bound value must be at least* 32767*. An MPI implementation is free to make the value of `MPI_TAG_UB` larger than this; for example, the value* $2^{30} - 1$ *is also a valid value for `MPI_TAG_UB`.*

- So if we have more than $2^{15} - 1$ elements, this might not work

MPI One-Sided Radix Sort I

```
/* distributed array, each process has N elements */
void sort_mpi(int32_t **keys_p, size_t N)
{
        int32_t *keys = *keys_p;
        int32_t *new_keys = NULL;
        MPI_Win win;
        MPI_Win new_win;
        MPI_Request *reqs = malloc(sizeof(MPI_Request) * N);
        MPI_Count local_hist[2];
        MPI_Count global_hist[2];
        MPI_Count prior_hist[2];

        MPI_Win_create(keys, sizeof(int32_t) * N, sizeof(int32_t),
                       MPI_INFO_NULL, MPI_COMM_WORLD, &win);
        MPI_Win_allocate(sizeof(int32_t) * N, sizeof(int32_t),
                         MPI_INFO_NULL, MPI_COMM_WORLD, &new_keys, &new_win);
```

MPI One-Sided Radix Sort II

```
/* int32_t is 4 bytes = 32 bits */
for (int r = 0; r < 32; r++) {
        /* reset local histogram */
        memset(local_hist, 0, sizeof(local_hist));

        /* compute local histogram of key bits */
        for (size_t i = 0; i < N; i++) {
                size_t bit = (keys[i] >> r) & 0x1;
                local_hist[bit]++;
        }

        /* compute reduction on histogram */
        MPI_Allreduce(local_hist, global_hist, 2,
                      MPI_COUNT, MPI_SUM, MPI_COMM_WOLRD);

        /* compute histogram for prior processes */
        MPI_Exscan(local_hist, prior_hist, 2,
                   MPI_COUNT, MPI_SUM, MPI_COMM_WORLD);
```

MPI One-Sided Radix Sort III

```
/* exclusive scan over global histogram */
MPI_Count sum = 0;
for (int i = 0; i < 2; i++) {
        MPI_Count tmp = global_hist[i];
        global_hist[i] = sum;
        sum += tmp;
}

/* compute starting indices */
for (int i = 0; i < 2; i++) {
        local_hist[i] = global_hist[i] + prior_hist[i];
}
```

MPI One-Sided Radix Sort IV

```
/* lock RMA window */
MPI_Win_lock_all(0, new_win);

/* put keys in right place */
for (size_t i = 0; i < N; i++) {
        size_t bit = (keys[i] >> r) & 0x1;
        MPI_Count index = local_hist[bit]++;
        int dest = index / N;
        MPI_Aint disp = index % N;
        MPI_Rput(&keys[i], 1, MPI_INT32_T, dest,
                 disp, 1, MPI_INT32_T, new_win, &reqs[i]);
}

/* complete puts */
MPI_Waitall(N, reqs, MPI_STATUSES_IGNORE);
/* unlock RMA window */
MPI_Win_unlock_all(new_win);
/* wait for everyone */
MPI_Barrier(MPI_COMM_WORLD);
```

MPI One-Sided Radix Sort V

```
            /* swap keys, new_keys and win, new_win */
            int32_t *tmp_keys = keys;
            MPI_Win tmp_win = win;
            keys = new_keys;
            win = new_win;
            new_keys = tmp_keys;
            new_win = tmp_win;
        }
        *keys_p = keys;
        MPI_Win_free(new_win);
        MPI_Win_free(win);
        free(reqs);
}
```