CS420 – Lecture 20

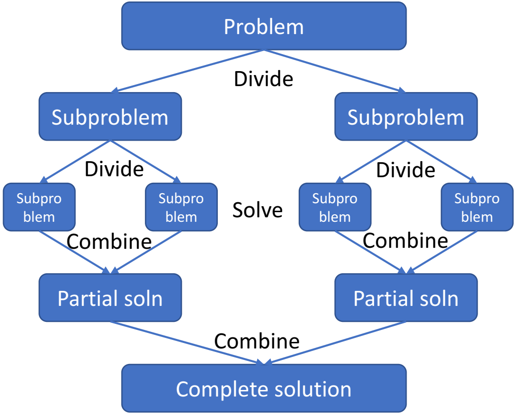
Omri Mor

Fall 2018

Divide-and-Conquer

Divide-and-Conquer

- Suppose you have a large problem to solve, but directly solving it is slow
  - Repeatedly *divide* it into smaller problems until you can solve
  - Then *combine* solutions to solve larger problems
- Applications:
  - Sorting
  - Fast multiplication and matrix multiplication
  - Fast Fourier Transform
  - ...



Sorting

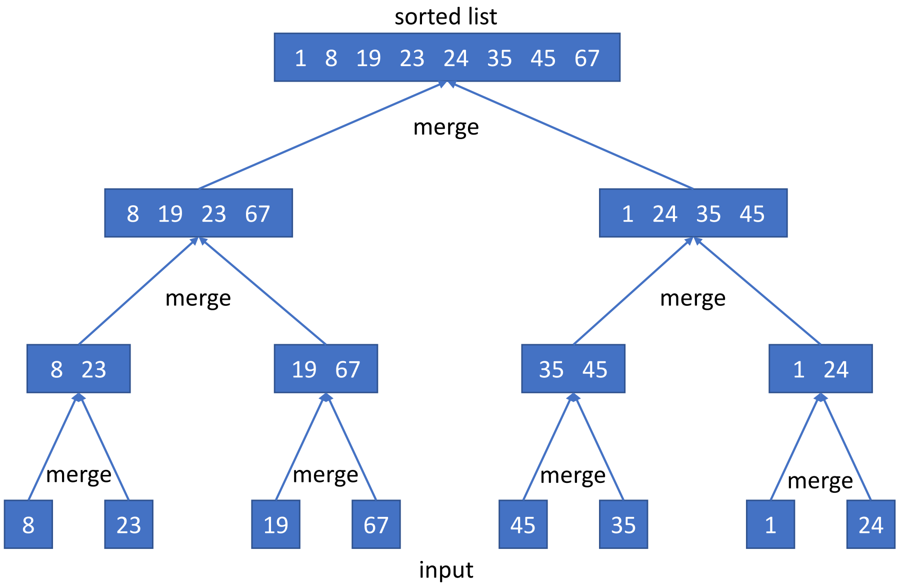
- Given a list and a comparator, order the list
- Many algorithms with different theoretical and empirical performance
- Serial comparison-based sort can be done in $\Omega(n \log n)$ time
- Many algorithms can be faster on real-world data
- Built-in serial implementations: `qsort(array, count, size, compare)`, `std::sort(first, last)`
- `http://sortbenchmark.org/`

8 23 19 67 45 35 1 24

$\downarrow$

1 8 19 23 24 35 45 67

Merge sort


sorted list
1 8 19 23 24 35 45 67
merge
8 19 23 67
1 24 35 45
merge
merge
8 23
19 67
35 45
1 24
merge
merge
merge
merge
8
23
19
67
45
35
1
24
input

Serial Merge Sort

```
void mergesort(int[] list, int start, int end) {
  if (start + 1 < end) {
    int mid = (start + end) / 2;
    mergesort(list, start, mid);
    mergesort(list, mid, end);
    merge(list, start, mid, end);
  }
}
```

```
void merge(int[] list, int start, int mid, int end) {
  // copy_add_sentinal: copy list[a:b], append MAX_INT
  int *left = copy_add_sentinal(list, start, mid);
  int *right = copy_add_sentinal(list, mid, end);
  int i = 0;
  int j = 0;
  for (int k = start; k < end; k++) {
    if (left[i] <= right[j]) {
      list[k] = left[i];
      i++;
    } else {
      list[k] = right[j];
      j++;
    }
  }
}
```

Note: Beware branch prediction!

OpenMP Merge Sort

```
void omp_mergesort(int[] list, int start, int end) {
  if (start + 1 < end) {
    int mid = (start + end) / 2;

    #pragma omp parallel sections
    {
      #pragma omp parallel section
      omp_mergesort(list, start, mid);

      #pragma omp parallel section
      omp_mergesort(list, mid, end);
    }

    merge(list, start, mid, end);
  }
}
```

Tasking

- `omp parallel sections` defines blocks that are executed in parallel
- `omp_mergesort` is called recursively: our threads spawn threads!
- `omp_set_nested(1)`: enable nested parallelism
- Must beware of oversubscription and spawning too many tasks

Our code from before, but limit the number of tasks...

```
void omp_mergesort(int[] list, int start, int end, int threads) {
  if (start + 1 < end) {
    if (threads <= 1) {
      mergesort(list, start, end);
      return;
    }

    int mid = (start + end) / 2;

    #pragma omp parallel sections
    {
      #pragma omp parallel section
      omp_mergesort(list, start, mid, threads / 2);

      #pragma omp parallel section
      omp_mergesort(list, mid, end, threads - threads / 2);
    }

    merge(list, start, mid, end);
  }
}
```

Performance analysis

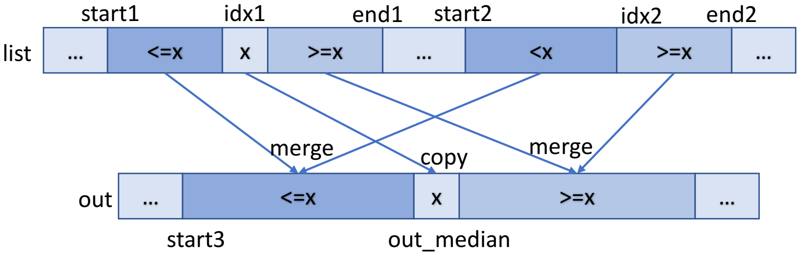
- At each level the list is split in half and processed in parallel
- ... But we have to merge the results of the lower level

Performance analysis

- At each level the list is split in half and processed in parallel
- ... But we have to merge the results of the lower level
- Takes $\mathcal{O}(n)$ time to merge at the top level!
- Need to parallelize `merge` as well

OpenMP merge

- Can use divide-and-conquer for the merge step as well, but the algorithm is more complicated
  - Split the larger list in half
  - Split the other list into two parts based on the midpoint of the first list
  - Merge the pairs of lists recursively
- Will require a `binary_search` subroutine
- Will need to use some secondary lists: no longer always merging contiguous lists
- Parallel merge will run in $\mathcal{O}(\log^2 n)$ time, and parallel mergesort in $\mathcal{O}(\log^3 n)$ time



```
void omp_merge(int[] list, int start1, int end1, int start2, int end2,
               int* out, int start3) {
  if (threads <= 1) {
    merge(list, start1, end1, start2, end2, out, start3); return;
  }

  int len1 = end1 - start1;
  int len2 = end2 - start2;
  // Assume the first list is the longest; can swap if needed.
  if (len1 == 0)
    return;
  int median_idx1 = (start1 + end1) / 2;
  int median_idx2 = binary_search(list, start2, end2, list[median_idx1]);
  int out_median = out_start + (median_idx1 - start1) + (median_idx2 - start2);
  out[out_median] = list[median_idx1];

  #pragma omp parallel sections
  {
    #pragma omp section
    omp_merge(list, start1, median_idx1, start2, median_idx2, out, out_start,
              threads / 2);

    #pragma omp section
    omp_merge(list, median_idx1+1, end1, median_idx2, end2, out, out_start,
              threads - threads / 2);
  }
}
```

MPI merge sort

- A simple MPI implementation is similar to OpenMP
- Send the chunk to be sorted to an MPI process
- But this results in severe load imbalance
- Instead pick multiple pivot points at the beginning to ensure every process has work