

PPA-F25 hw4

Logistics

This homework consists of 100 points total, with points for individual tasks as indicated below. There are 2 coding tasks in this assignment.

Please submit your work on Gradescope.

For the coding tasks, you will edit and submit `DedupSort.sml`

Due Date: This submission is due at **5:00pm EDT on Tuesday, Oct 7**. Course policy on late submissions is available on the course website. (<https://cs.nyu.edu/~shw8119/courses/f25/3033-121-ppa/>)

Setup

- Follow the instructions on the course website to access one of the **crunchy** compute servers.
- Install MaPLe. For the **crunchy** servers, you can:
 - Download <https://cs.nyu.edu/~shw8119/courses/f25/3033-121-ppa/resources/mpl-845f76f.tgz> and copy it to the server
 - Unpack by running `tar xzf mpl-845f76f.tgz`. This will create a directory `mpl-845f76f/`
 - The compiler, `mpl`, is located at `mpl-845f76f/bin/mpl`
 - Add `mpl-845f76f/bin` to your `PATH` so that you can access it easily: `export PATH="$(pwd -P)/mpl-845f76f/bin:$PATH"`. We recommend updating your `~/.bashrc` file or other configuration file as appropriate.

Deduplicate, Sort, and Count

Suppose you have a large unsorted collection of keys, potentially with a lot of duplicates. You want to sort, deduplicate, and count the number of duplicates for every key.

In a language like C++, you might write a sequential solution like so:

```
std::vector<std::pair<key, long>>
dedup_sort(std::vector<key> input_keys)
{
    std::map<key, long> counts;
    for (key k : input_keys) { counts[k]++; }
    std::vector<std::pair<key, long>> uniques;
    uniques.assign(counts.begin(), counts.end());
    return uniques;
}
```

In this assignment, you will develop a parallel version of `dedup_sort` in MaPLe that is faster than this C++ code.

Benchmarking scripts to test the C++ code are provided in `cpp/`. Here is an example recent run on `crunchy1.cims.nyu.edu`, with 10M input keys but only 10K unique keys (each key appears approximately 1000 times).

```
$ make -C cpp dedup_sort
$ cpp/dedup_sort 10000000 10000
n 10000000
k 10000
target-sortedness 60
```

```

generating input... (might take a moment)
input 0 0 0 4706 0 7202 8717 9733 1109 0 7228 0 9533 0 8943 0 0 8612 0 0 ...
warming up for 3 seconds...
warmup done.
time 1.26184
time 1.25829
time 1.27189
time 1.26425
time 1.27402
time 1.27286
time 1.26657
time 1.27134
time 1.25235
time 1.2658
average 1.26592
(0,1008) (1,1071) (2,953) (3,971) (4,952) (5,996) (6,1021) (7,1052) (8,1028) (9,987) ...
num unique 10000

```

Preliminaries: Divide-And-Conquer Parallel Merging

To start, take a look at the functions `merge` and `merge_tree` in `ReferenceImplementations.sml`. These functions implement the divide-and-conquer parallel merge that we discussed in lecture.

```

fun merge_tree (cmp: 'a * 'a -> order) (s1: 'a Seq.t, s2: 'a Seq.t) =
  if Seq.length s1 = 0 then
    TFlatten.leaf s2
  else if Seq.length s2 = 0 then
    TFlatten.leaf s1
  else
    let
      val n1 = Seq.length s1
      val n2 = Seq.length s2
      val mid1 = n1 div 2
      val pivot = Seq.nth s1 mid1
      val mid2 = BinarySearch.search cmp s2 pivot

      val l1 = Seq.take s1 mid1
      val r1 = Seq.drop s1 (mid1 + 1)
      val l2 = Seq.take s2 mid2
      val r2 = Seq.drop s2 mid2

      val (outl, outr) =
        ForkJoin.par (fn _ => merge_tree cmp (l1, l2),
                      fn _ => merge_tree cmp (r1, r2))
    in
      TFlatten.node
        (TFlatten.node (outl, TFlatten.leaf (Seq.singleton pivot)), outr)
    end

fun merge (cmp: 'a * 'a -> order) (s1: 'a Seq.t, s2: 'a Seq.t) =
  TFlatten.flatten (merge_tree cmp (s1, s2))

```

Comparison Functions

Note that MaPLe uses the convention that comparison functions are encoded with the type `'a * 'a -> order`, which returns one of LESS, EQUAL, or GREATER:

```
datatype order = LESS | EQUAL | GREATER
```

For example, you could define your own comparison function on integers as:

```
fun int_compare (x: int, y: int) : order =  
  if x < y then LESS  
  else if x = y then EQUAL  
  else GREATER
```

Tree Flattening

The implementations above make use of a structure `TFlatten` which makes it easy to build a tree of sequences and then flatten them at the end. The functions `TFlatten.leaf` and `TFlatten.node` are used to construct leaves and nodes of a tree, respectively:

```
structure TFlatten:  
sig  
  type 'a tree  
  type 'a t = 'a tree  
  datatype 'a view = Leaf of 'a Seq.t | Node of 'a t * 'a t  
  
  val size: 'a tree -> int (* O(1) work and span *)  
  val leaf: 'a Seq.t -> 'a tree (* O(1) work and span *)  
  val node: 'a tree * 'a tree -> 'a tree (* O(1) work and span *)  
  val view: 'a tree -> 'a view (* O(1) work and span *)  
  
  val flatten: 'a tree -> 'a Seq.t (* O(n+m) work  
    * O(polylog(n) + polylog(m)) span  
    * where n is the number of leaves  
    * and m is the output size *)  
end
```

Binary Searching

The code above also uses an implementation of `BinarySearch`, to split the right-hand-side sequence at an appropriate “middle” point.

```
structure BinarySearch:  
sig  
  (* Assuming `s` is sorted with respect to the comparison function `cmp`,  
   * The call `search cmp s x` returns an index `i` such that `x` would  
   * appear at index `i` in the sorted order.  
   *)  
  (* O(log|s|) work and span.  
  *)  
  val search: ('a * 'a -> order) -> 'a Seq.t -> 'a -> int  
  ...  
end
```

Cost Recurrences

The above implementation of `merge_tree` adheres to the following work and span recurrences, $W(n, m)$ and $S(n, m)$, for two input sequences of size n and m .

Work:

$$W(n, m) = W(n/2, \alpha \cdot m) + W(n/2, (1 - \alpha) \cdot m) + \log m \quad (\text{where } 0 \leq \alpha \leq 1/2)$$

$$W(0, m) = m \quad (\text{base case})$$

$$W(n, 0) = n \quad (\text{base case})$$

Span:

$$S(n, m) = \max(S(n/2, \alpha \cdot m), S(n/2, (1 - \alpha) \cdot m)) + \log m \quad (\text{where } 0 \leq \alpha \leq 1/2)$$

$$S(0, m) = \log m \quad (\text{base case})$$

$$S(n, 0) = \log n \quad (\text{base case})$$

You may assume that $W(n, m) \in O(n + m)$ and $S(n, m) \in O(\text{polylog}(n) + \text{polylog}(m))$.

The `merge` function pays additionally to perform the flattening. Note that flattening in this case has the same work and span bounds. So, overall, the cost of `merge` is $O(n + m)$ work and $O(\text{polylog}(n) + \text{polylog}(m))$ span.

Adapting D/C Merge for Deduplication

Task 1 (50 points). In `DedupSort.sml`, implement the following function:

```
val dedup_merge: ('a * 'a -> order)
                -> ('a * int) Seq.t * ('a * int) Seq.t
                -> ('a * int) Seq.t
```

The inputs to this function are two sequences of key-count pairs (k, c) , with keys of type `'a` and counts of type `int`.

You may assume that these sequences have been sorted and deduplicated according to their keys `k` (using the comparison function that is also given as input).

The goal of `dedup_merge` is to take two such input sequences, each of which has individually been deduplicated and sorted, and to merge them.

For example, on inputs $\langle (5, 10), (6, 1), (100, 1042) \rangle$ and $\langle (0, 3), (5, 2), (7, 1) \rangle$ using normal integer comparison for keys, the output should be $\langle (0, 3), (5, 12), (6, 1), (7, 1), (100, 1042) \rangle$. Note that the key 5 appears in both inputs and needs to be appropriately deduplicated and counted in the output.

On two inputs of size n and m , your implementation must have $O(n + m)$ work and $O(\text{polylog}(n) + \text{polylog}(m))$ span.

We recommend thinking about adapting `ReferenceImplementations.merge`.

You can test your code by adding test inputs to `test.sml` and running the following. Note that if you add additional test cases you will need to recompile.

```
$ make test
$ ./test
```

Parallel Deduplicating Sort

Task 2 (50 points). In `DedupSort.sml`, implement the following function:

```
val dedup_sort: ('a * 'a -> order)
               -> 'a Seq.t
               -> ('a * int) Seq.t
```

This function takes a sequence of keys of type `'a` and should return a sorted (by key) sequence of key-count pairs (k, c) , where c is the number of times key k appears in the input. The comparison function given as input determines the sorted order.

For example, on input $\langle 1, 0, 5, 1, 0, 0, 2 \rangle$ (and normal integer comparison), the output should be $\langle (0, 3), (1, 2), (2, 1), (5, 1) \rangle$.

In contrast to the other tasks so far in this course, in this task, the goal is to **optimize its performance in practice**. Specifically, you will compare it against the C++ implementation and the goal is to get real parallel speedups in comparison to the C++ code. The goal is also to get good scalability, as the number of cores increases.

Your implementation does not need to meet any particular cost bounds. Your score will be determined by performance in practice.

You **don't** have to use your `dedup_merge` implementation as a subroutine, but we recommend trying it, e.g., in some sort of deduplicating mergesort. (Other algorithms are possible, but this approach in particular we feel is interesting and worth trying.)

You can test your implementation for correctness as follows, similar to `dedup_merge` above. Note that this only tests correctness, not performance.

```
$ make test
$ ./test
```

You can test performance as follows. This is also how we will measure performance for grading, taking the reported average.

```
$ make main
$ ./main @mpl procs <P> -- 10000000 10000 -warmup 2 -repeat 10
```

The rubric for Task 2 is as follows. At the moment, we plan to use `crunchy1.cims.nyu.edu` for performance grading.

- 20 points for correctness.
- 30 points for performance. **Note: you must pass *all* of the correctness tests to be eligible for performance points.**
 - 10 points for scalability. Specifically:
 - * (5 points) on 30 processors, **at least** 8x *self*-speedup
 - * (5 points) on 60 processors, **at least** 15x *self*-speedup
 - 20 points for performance vs C++. Specifically:
 - * (10 points) on 1 processor, **at most** 4x slower than the C++ `dedup_sort`
 - * (10 points) on 60 processors, **at least** 4x faster than the C++ `dedup_sort`

A few recommendations:

- In any divide-and-conquer implementation, it is often faster to switch to a sequential algorithm at small input sizes. Try experimenting with different thresholds for the switch from parallel to sequential.
- When experimenting with thresholds, it is convenient to be able to pass a threshold as a command-line parameter, so that you don't have to recompile in between. You can do this in MaPLe

using the `CommandLineArgs` structure. For example, with `val x = CommandLineArgs.parseInt "param" 42`, the value `x` will by default be 42; if you pass `-param 1000` at the command-line, it will have the value 1000.

- Feel free to use the functions in `ReferenceImplementations`. The functions named `XXX_serial` are all sequential, i.e., no parallelism, and reasonably well optimized (although certainly not as fast as they could be; the serial dedup-merge in particular could probably be further optimized quite a bit).
- When measuring performance, be careful about competing jobs on the same machine. Someone else might be logged in at the same time and hogging processors, slowing down your runs. Use `top` or `htop` to see what else is running on the machine.