Deterministic Parallel Programming Exercises

MPRI 2.37.1

February 27, 2020

1 MergeShuffle

Bacher, Bodini, Hollender, and Lumbroso have proposed the following subroutine for producing a randomly shuffled union of two randomly shuffled arrays.[1]

```c
// Shuffle A[lo, mid) and A[mid, hi) into A[lo, hi).
void shuffle_halves(int *A, size_t lo, size_t mid, size_t hi) {
  size_t i = lo, j = mid;
  while (1) {
    if (flip_coin()) {
      if (j == hi) break;
      swap(A[i], A[j++]);
    } else {
      if (i == j) break;
    }
    i++;
  }
  for (; i < hi; i++)
    swap(A[i], A[rand(i - lo + 1)]);
}
```

Questions.

1. Use this routine to implement a parallel shuffling algorithm in Cilk.
2. State the work and span of your proposal. Is it work-efficient?
3. Propose a coarsened version of your algorithm.

2 Matrix Multiplication

Let A and B be $n \times n$ matrices, with n a power of 2. The equation below expresses their product AB in terms of four submatrices A_{ij} and B_{ij}.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{bmatrix} + \begin{bmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{bmatrix}$$

Questions.

1. Implement a parallel matrix multiplication algorithm using this decomposition.
2. State the work and span of your algorithm. Is it work-efficient?
3. Discuss the space complexity of your implementation. Do you see an alternative implementation with lower space usage? How would its span compare to the previous one?

[1] `https://arxiv.org/abs/1508.03167`

3 Spanning Forests

A *spanning forest* for an undirected graph $G = (V, E)$ is a set $F \subseteq E$ such that (V, F) is a maximal acyclic subgraph of G. If the graph comes equipped with edge weights $w : E \to \mathbb{N}$, one may want F to be a *minimum* spanning forest, i.e., one minimizing $\sum_{f \in F} w(f)$.

Kruskal's algorithm computes a minimum spanning forest in a greedy manner, using a disjoint-set data structure to represent its result. The algorithm runs in $O(|E| \log |E|)$ time.[2]

```
disjoint_set_t *kruskal(graph_t *G) {
  disjoint_set_t *F = ds_make(number_of_nodes(G));
  sort_edges_by_weight(G);      /* Useless when G is not weighted. */
  for (edge_t *e = first_edge(G); e != last_edge(G); e = next_edge(G, e)) {
    int u = ds_find(F, e->u), v = ds_find(F, e->v);
    if (u != v) {
      /* Links the set represented by u to the set represented by v, keeping the
         representative of v as the representative of the resulting union. */
      ds_link(F, u, v);
    }
  }
  return F;
}
```

The goal of this exercise is to sketch internally-deterministic versions of spanning forest computations, using the deterministic-reservations framework to parallelize Kruskal's algorithm.

For each algorithm, you should provide at least the two functions `bool reserve(int i)` and `bool commit(int i)` expected by the deterministic-reservations framework, as well as any auxiliary state and initialization code required for their operation. The `reserve(i)` function should return `false` to discard iteration `i`, never calling `commit(i)`; the `commit(i)` function should return `true` to mark iteration `i` as processed, and `false` to retry it next round.

These parallel implementations have to make assumptions on the commutativity of the operations acting upon the disjoint-set data structure. You may assume the following:

- calls to `ds_find` commute with each other,
- calls to `ds_link(F, y1, x1)` and `ds_link(F, y2, x2)` commute when `y1 != y2`,
- calls to `ds_link(F, y1, x1)` and `ds_find(F, x2)` commute when `x1 == x2`.

Questions.

1. Propose a parallel implementation for the unweighted case. It does not have to return the same spanning forest as the sequential algorithm.

   Hint: you may want to use the `write_max()` primitive.

2. Propose a parallel implementation for the weighted case. It does not have to return the same spanning forest as the sequential algorithm, but should return a minimum spanning forest. You may assume as given an efficient parallel comparison sort.

3. Propose an implementation of the disjoint-set data structure that respects the commutativity properties stated above. Its operations do not have to run in better than linear time.

[2]We assume a general comparison sort is required. Special-purpose sorts, when relevant, can decrease running time to $O(|E|\,\alpha(|V|))$, where α is the inverse of the Ackermann function.