Cache Efficient Parallel Partition

Alek Westover
June 26, 2019

Abstract

This is a modified version of the "Algorithm Overview" paragraph in the "Analysis of Grouped Partition" section of the other writeup that I am doing.

Algorithm Overview. We now describe the ***Grouped Partition*** algorithm. We logically divide the array $A = A[0], A[1], \ldots, A[n-1]$ into blocks P_j each of b adjacent elements, where b is the ***block size***. Note that the P_js are defined in a different way then in the Strided algorithm when P_j contained elements spaced throughout the array. This is equivalent to setting $|P_j| = 1$ for the P_js in the Strided Algorithm and then making P_j a collection of cache blocks. Define $\mathbf{pred}(\boldsymbol{j})$ to be the number of predecessors in P_j. There are n/b of these blocks P_j.

The algorithm generates a random array $X = X[0], X[1], \ldots, X[s-1]$ where each element in X contains an integer chosen randomly at uniform from $[0, g-1]$, where g is the number of groups. The values in X determine g groups $G_0, G_1, \ldots, G_{g-1}$ of P_js. The first group is

$$G_0 = \{X[0], X[1] + g, X[2] + 2 \cdot g, \\ \ldots, X[s-1] + (s-1) \cdot g\}.$$

This means that group G_0 is a collection of indices for specific parts P_j, indicating that these parts belong to group G_0. Similarly, group G_y is defined as

$$G_y = \{(X[0]+i) \bmod g, (X[1]+i) \bmod g+g, \\ \ldots, (X[s-1]+i) \bmod g + (s-1) \cdot g\}.$$

Intuitively this means that, for group G_y, on each chunk of size g of the array, take $X[j]$ and add the group's index i (wrapping around if there is overflow beyond the number of groups) to get to the index of the P_j that belongs to group G_y from this chunk of the array. Note that we do not need to store the indices of each P_j that belongs to each group because between X and the group index the P_js that belong to the group G_y are uniquely determined. We call our algorithm in-place because the size s of X is made $\Theta(\frac{\log n}{\delta^2})$, for $\delta \in (0, 1)$ of our choice, which is minuscule compared to the array of size n that is an input to the partition problem.

Define U_y to be the union of all parts that belong to group G_y, that is

$$U_y = \bigcup_{j \in G_y} P_j.$$

Define μ_y to be the number of predecessors in U_y divided by the number of elements in U_y. This is analogous to the definition of μ: the number of predecessors in A divided by n.

Once the algorithm has generated X, it performs a serial partition on each collection U_y in parallel. After partitioning the collection U_y the Grouped Partition algorithm stores the index v_y, which is the index in A of the first successor in G_y, in array of size

s. Recall that s is small, and that the algorithm already created an array X of size s, so this does not change the asymptotics for memory consumption of the algorithm. After the Grouped Partition algorithm finishes performing in parallel a serial partition of each group, the algorithm computes in serial $v_{min} = \min v_y$, and $v_{max} = \max v_y$ from the stored values v_y generated by each serial partition. Note that we cannot just have a variable that all threads can write to that stores the current lowest v_y and current highest v_y and then overwrite this in each thread when it finishes its group as we would try in a serial algorithm because this would cause data races in the parallel algorithm. Thus, because for all y

$$v_{min} \leq v_y \leq v_{max},$$

we can determine that all elements of A with index less than v_{min} are predecessors and elements of the array with index greater or equal to than v_{max} are successors. Thus, by recursing on the subarray $A[v_{min}], \ldots, A[v_{max} - 1]$, we complete the partitioning of the array. We recursively apply the Grouped Partition algorithm to the subproblem. The base case for the recursion is that when the algorithm can no longer make a substantial number of groups, in which case it partitions its input array in serial.