CSC-726 — Parallel Algorithms
Assignment 1

Esteban Murillo, Lawton Manning

Thursday 3rd October, 2019

1 Introduction to sequential problem

A naïve approach to solving the Knapsack problem is to create the entire dynamic programming table in memory and solve it sequentially, either row by row left-to-right or column by column left-to-right. We can explore the combination of memory optimizations, cache optimizations, and parallelization of the dynamic programming solution.

2 Optimizations

2.1 Cache optimization

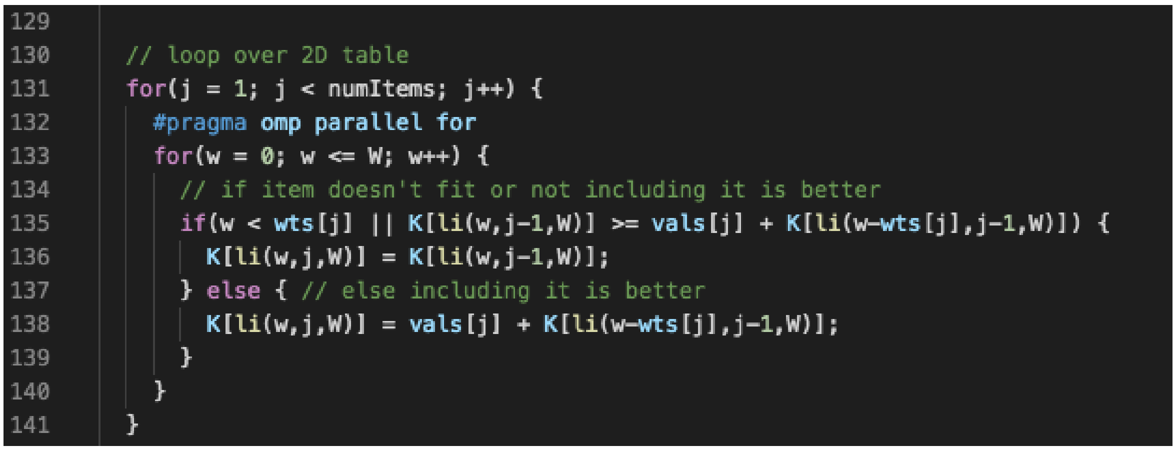
There are two ways that a two-dimensional matrix can be stored in memory: row major and column major. Our matrix was stored in column major. To be sure that we maintain spatial locality, the matrix should be iterated sequentially as it exists in memory. This means that all nested loops should have an outer column loop and an inner row loop, and so in column-by-column order.

2.2 Memory optimization

For dynamic programming, all data dependencies exist between adjacent columns. Specifically, any possible entry $K_{i,j}$ can only depend on $K_{k,j-1}$ where $k \leq i$. So, it is possible to compute the entire table by only storing two adjacent columns at any one time.

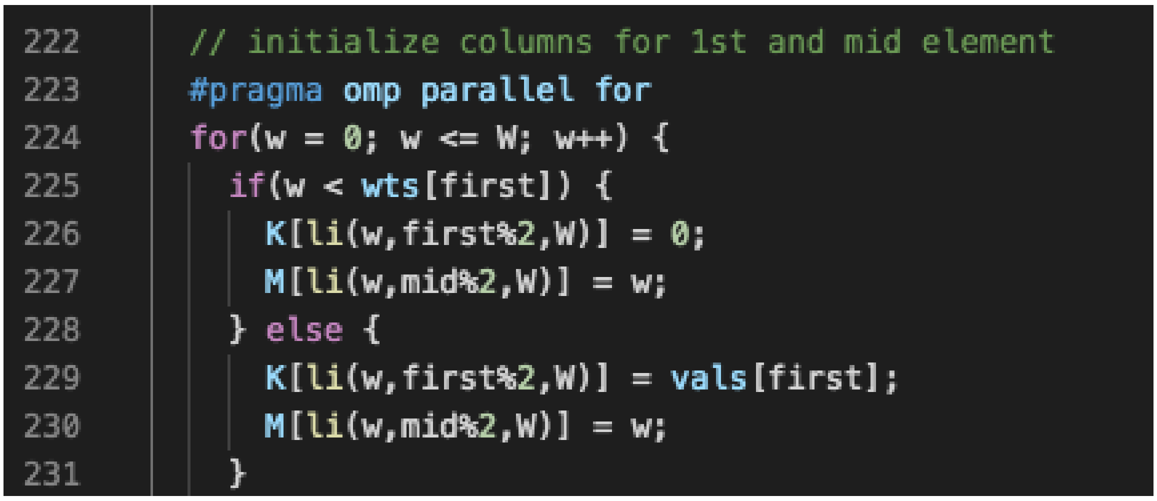
2.3 Parallel optimization

From the previous section on memory, any parallelization has to follow the same dependencies laid out by the problem. With two adjacent columns, we can parallelize the computation of up to all the rows of the current column. From Figure 1, the inner loop representing the rows has been parallelized for each column j. Similar changes have been



```
129
130     // loop over 2D table
131     for(j = 1; j < numItems; j++) {
132       #pragma omp parallel for
133       for(w = 0; w <= W; w++) {
134         // if item doesn't fit or not including it is better
135         if(w < wts[j] || K[li(w,j-1,W)] >= vals[j] + K[li(w-wts[j],j-1,W)]) {
136           K[li(w,j,W)] = K[li(w,j-1,W)];
137         } else { // else including it is better
138           K[li(w,j,W)] = vals[j] + K[li(w-wts[j],j-1,W)];
139         }
140       }
141     }
```

Figure 1: Code after switching line 131 for line 133



```
// initialize columns for 1st and mid element
#pragma omp parallel for
for(w = 0; w <= W; w++) {
  if(w < wts[first]) {
    K[li(w,first%2,W)] = 0;
    M[li(w,mid%2,W)] = w;
  } else {
    K[li(w,first%2,W)] = vals[first];
    M[li(w,mid%2,W)] = w;
  }
```

Figure 2: Sample code of how *for* loops were optimized

made for other nested loops on n and W in the program. For each version discussed, the bottleneck has been this loop which is $O(nW)$. By parallelizing the rows, we can achieve a span of $nlog(W)$, where $log(W)$ is the work necessary to spawn a thread for each w in W.

There have been other parallel additions such as in *convert_path_to_used* where the work is normally $O(n)$ but is parallelized to produce a span of $O(log(n))$. However, this function exists as a supplementary to another more complex function and so parallelizing it does not affect the bottleneck.

3 Analysis

As seen on Figure 3, in an ideal world we would be getting 2x results every single time we double the number of threads. Even though the speed up is not perfect we can still analyze if the “investment” is worth it. As seen on Figure 3, there is a point in which the return might not be worth it anymore. Of course, that we are allowed to experiment with as many threads as we want, up to 44. But in a real life scenario, we would want to see how much is enough.

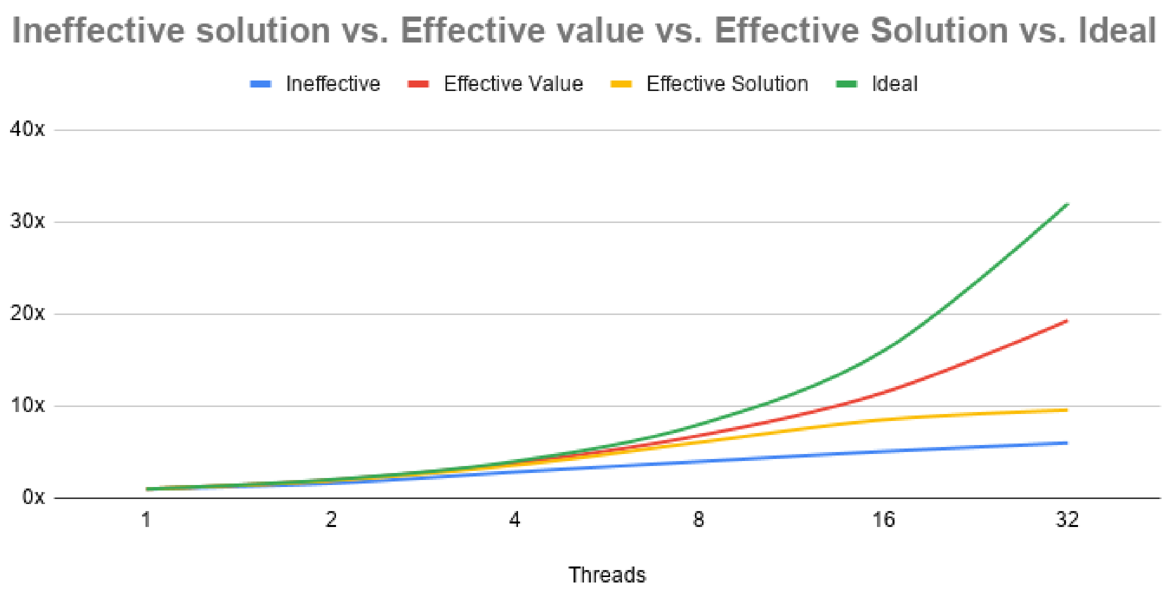


Figure 3: Ideal scenario where speedup is linear

Threads	Ineffective solution	Effective value	Effective solution
1	1.0000	1.0000	1.0000
2	1.6089	1.9511	1.8897
4	2.8213	3.8120	3.5695
8	3.9747	6.7969	6.0602
16	5.0680	11.4658	8.5097
32	5.9779	19.2836	9.5468

Table 1: Yielded speedup for every single Knapsack version

Figure 3 shows how much faster the algorithm got. In Table 1, we can see that we stop having significant speedup values after solving the Knapsack problem with 8 threads. Of course that does not mean that if we go further we are not going to get faster results, but just as we mentioned, the improvements are going to diminish after this point.

Comparing the speedups in Table 1, we can see that the speedup goes like $log(P)$ where P is the number of processors for the Ineffective solution. This makes sense as the work is nW and span is $nlog(P)$, giving a $T(p)$ of:

$$T(p) = \frac{nW}{nlog(P)} = \frac{W}{log(P)} \tag{1}$$

Similarly, the work for the Effective solution is also nW with a span of $nlog(n) \cdot log(P)$ so that its $T(p)$ is:

$$T(p) = \frac{nW}{nlog(n)log(P)} = \frac{W}{log(n)log(P)} \tag{2}$$

When we compare Equation 1 to Equation 2, we see that when n and W are kept constant, the they are within a constant factor of $\frac{1}{log(P)}$. So, they should have similar speedups of $+1$ for every doubling of P. However, the Effective solution has a boost over the ineffective. We would attribute this a combination of the memory efficiency of not loading an entire table into memory and only solving a subsection of the problem which lessens the load on each processor.

For the Effective value, the speedups are more pronounced as it has all of the advantages of the cache and memory improvements and does not compute the path through the dynamic solution.

4 Conclusion

From all data shown we could say that the way in which data is accessed is more important than the actual parallelization of our algorithm. Going back to Figure 3, we see that the Effective solution without that many changes was marginally better than the ineffective solution, despite the fact that it did more work to find its answer within a constant factor. Clearly, a combination of memory efficiency and parallelization are both important in producing optimal algorithms.