Implementation Description for the Practical Part of the Parallel Algorithms Lecture

Noah Wahl

1 Approach

This implementation uses `TBB` as parallelization framework for the Dynamic Program to solve the 0/1 Knapsack Problem. The main optimizations are the usage of a suitable partitioner for the `tbb::parallel_for` loop, using 32 bit where possible and filling known values up front.

2 Solving Data Dependencies

A row $r, 1 \leq r \leq n+1$ of the 0/1 Knapsack Problem formulated as a Dynamic Problem depends only on values of its direct predecessor $r-1$. Therefore a parallel implementation may work simultaneously on the columns of a single row without problems. Another observation is that for an item with weight w the values of the first $w-1$ columns are the same as in the previous row.

The problem also can be solved by using only 2 `std::vector<>` representing adjacent rows. One of those vectors will be read-only while the other one is filled with values. Using 32 instead of 64 bit to store the values in these tables leads to significant runtime improvements. 32 bit may be used if the sum of all values fits into a 32 bit unsigned integer.

After solving the current row with this method the vectors switch their purpose; the newly filled row becomes the read-only vector and the other vector becomes the next row to be filled with new values. To prevent copying or reallocation, `std::swap` is used.

3 Parallel Implementation of the Inner Loop

Naturally, the approach to solve this Dynamic Program consists of two `for` loops. The outer loop iterates the rows while the inner loop iterates the columns of a row. Because of the data dependencies to the previous row, the outer loop cannot be worked on concurrently without communicating which columns already have been solved in the previous row. This implementation does not make the effort to parallelize the outer loop but focuses solemnly on the inner loop. The simplest way to achieve parallel execution of the inner loop is via `tbb::parallel_for` which splits the row into work packages that are then worked on by threads in parallel. Because the size of the rows does not change the partition of the row can be optimized by using the `tbb::affinity_partitioner` which outperforms the standard partitioner roughly by a factor of 4. Letting this partitioner decide on chunk sizes and perform load balancing across iterations produces better results than manually setting chunk sizes with `tbb::simple_partitioner` and the provided cache affinity mitigates (but not prevents) false sharing. Because TBB is rather high-level, reducing false sharing while not clashing with TBB's internal cache optimizations is not trivial. This implementation uses alignment of row entries by 4 or 8 bytes (for 32 or 64 bit integers). This does not change the size of cells but simply aligning the values helps the partitioner.

4 Comment on the Speedup

The speedup of this implementation is solid for up to 16 threads where it achieves a peak relative speedup of roughly 6. For higher thread counts the chunk size becomes too small and therefore the scheduling overhead starts to dominate. TBB excels at parallelizing long running tasks but solving Knapsack cells requires a comparatively small amount of clock cycles. Therefore the range of row entries to compute for every thread must be large to hit the target of roughly 100'000 cycles which is recommended in the documentation. However this not consistently achievable with the provided instances.