Parallel computing - Parallel matrix multiplication

Andi Škrgat - r0876363

December 4, 2021

Diagonal matrix product

For parallelizing diagonal matrices I used parallelization of only outer loop with static scheduling because in C++ arrays are stored as set of rows so when having large arrays there won't be constant penalization because of cache misses. Also, every iteration should take approximately same time so there is no reason to use dynamic or guided scheduling.

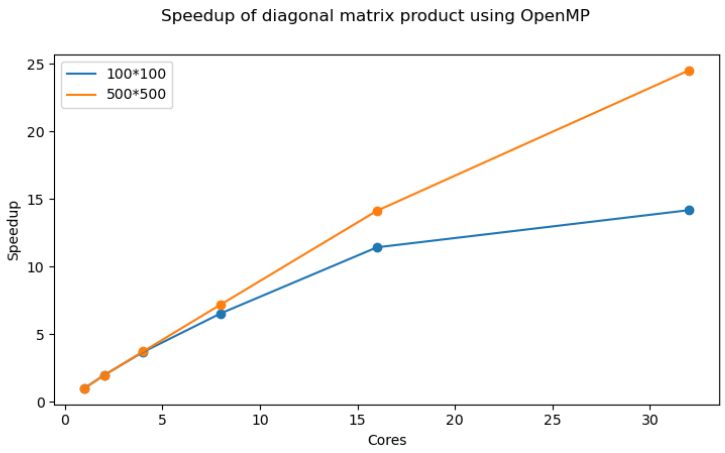

Figure 1: Benchmark for diagonal matrix product.

Full matrix product

For full matrix multiplication I tried several approaches. By parallelizing only $\boldsymbol{I}$ loop I got same results as by parallelizing inner $\boldsymbol{J}$ loop and as parallelizing using collapse command where $\boldsymbol{I}$ and $\boldsymbol{J}$ loops are both combined into one larger iteration space. Reason is that no matter what approach we use, there will always be same problem with fetching elements from right matrix because of cache misses but this applies only if the array is large enough.

This is again because in C++ arrays are stored as set of rows. Parallelizing ***K*** loop doesn't make sense because all threads are using same memory location - result(i,j) so the only way in which updating this variable could work is by using critical section but then parallelization wouldn't have sense because of huge penalty implied on each thread. I also managed to boost performance by using SIMD reduction on inner ***K*** loop and when using this feature speedup grows with factor ≈ 1.67.

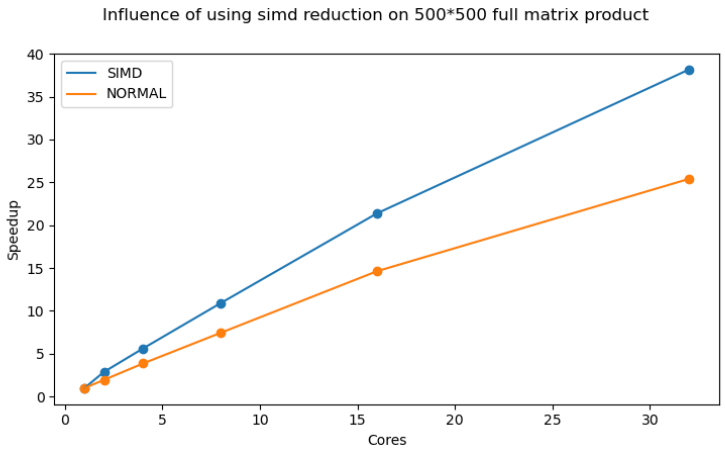

Figure 2: Influence of using SIMD reduction operator on full matrix product.

Full*blocked matrix product

Parallelizing block implementation produces best results and it was done by merging iteration space of variable i and j into larger one using collapse command. There is one very important thing one should take care about and that is block size. From theoretical perspective block size should correspond to the following term:

$$block\_size \approx \sqrt{\frac{L1\_cache\_size}{3 * sizeof(double)}}$$

so all matrices could fit into the cache. For running on VSC supercomputer where L1 cache is of size 64KB this gave value of 52. However, this didn't produce great speedups when more cores were used. Thing that worked best was setting block size to the number inversely proportional to number of cores by using expression:

$$block\_size = \frac{N}{omp\_get\_max\_threads()}$$

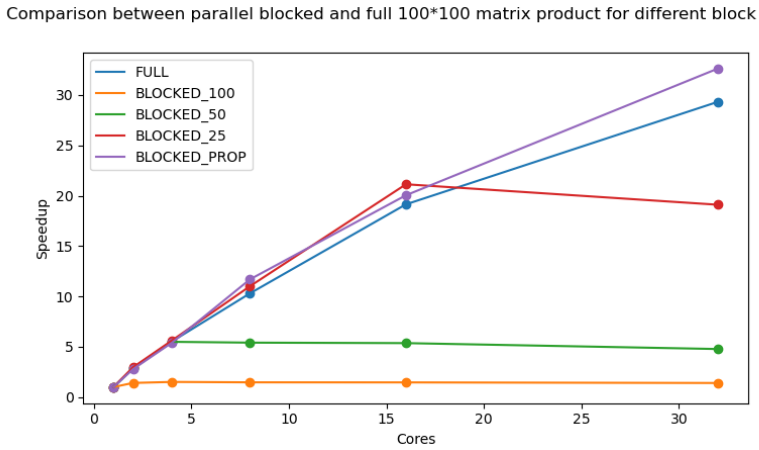

Figure 3: Only when block size is set to number inversely proporional to the number of cores, blocked implementation was able to beat full times matrix product that uses SIMD reduction.

Triangular matrix product

Triangular matrix product is specific because every core won't have same amount of work to do. I parallelized only outer loop but my main goal was proving that in this case dynamic or guided scheduling would work better than static scheduling. From the plots below this assumption can be confirmed because it can be seen that best speedups are obtained when using dynamic scheduling with chunk size equal to 1.

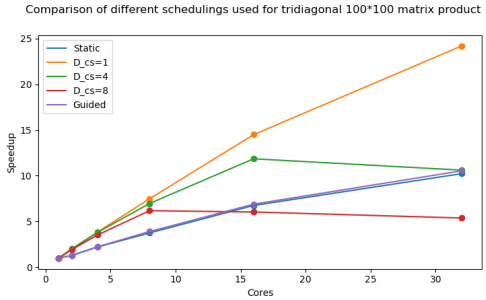

Figure 4: Comparison of different schedulings used for tridiagonal matrix product. For example D_cs=4 means: dynamic scheduling using chunk size equal to 4.