**Parallelizing Matrix Multiplication**

Matrix multiplication is an important problem that is used in many areas including linear algebra, scientific computing, and machine learning. The matrix multiplication problem takes two square N X N square matrices A and B as its inputs and produces an N X N square matrix C = A X B as its output.

**Serial Algorithms**

**Serial Naïve (SN):** The naïve algorithm for performing matrix multiplication takes O(N3) operations, since we need to compute N2 entries of the matrix C and each entry cij of C requires us to perform N multiplications of the N elements in the ith row of A with the N elements in the jth column of B and then add these products using N-1 additions. For N = 1000, this problem requires *1e9* operations and it is challenging to exceed much beyond this number of operations on a single processor [1]. Several theoretical improvements to this naïve algorithm continue to be made with the current best algorithm by Virginia Williams taking O(N2.373) operations [1,2]. The benefits of these improved algorithms can be realized only when N is more than 1000 [1], and even then, the improvements offered may not be practical due to the inherent high communication costs. Therefore, it is worth investigating whether parallelization produces any improvement in performance over serial versions of this algorithm.

**Serial Optimized(SO):** For large values of N, it is hard to store all the three matrices A, B and C in the cache. As a result, the performance of the naïve algorithm degrades due to the high number of cache misses. This can be avoided by blocking optimization described in [3] in which blocks of submatrices that can be stored in the cache are multiplied and the multiplication is performed such that all the operations involving a submatrix block is completed when it is already available in the cache. Therefore, this optimization improves performance by increasing temporal locality of the inner loop of the matrix multiplication. The matrix multiplication algorithm with blocking optimization is derived from [3]. I have implemented the blocking version of this algorithm and determined different blocking factors for different values of N.

**Parallel Algorithms**

Matrix multiplication is an inherently parallelizable problem since we can compute submatrices of the result C matrix in parallel this can result in significant speedup over the serial methods.

**Parallel Naïve (PN):** A naïve approach to parallelize the multiplication algorithm using 4 Posix threads is to partition the result matrix entries into slices and allocate each thread to perform the computation. The slices could be made of a contiguous group of rows or a contiguous group of columns of the result matrix. The current algorithm creates slices of rows of size N/4 and allocates the computation of the entries in each slice to one thread. If N is a multiple of 4 then the computation of the remainder entries is allocated to the one of the four threads. For example, if N = 250, threads 1 to and 3 compute entries 1-60, 61-120, 121-180, and 181-250 respectively. The input matrixes are not partitioned and instead each thread works using all of the input matrices. However, since the input matrices are not modified there is no need for locks to synchronize the access from these threads to the input matrices.

**Parallel Optimized (PO):** The algorithm PN exploits the parallelism inherent in the multiplication algorithm, but does not exploit the cache usage in each thread. This can lead to suboptimal performance within each thread due to cache misses. This is especially problematic, when N becomes larger than 4000 entries. In fact, in these cases, the degradation due to cache misses can dominate the gains obtained through parallelization and can make the algorithm PN perform no better (or in fact even worse for N = 8192) than the serial optimized algorithm. So, the final parallel optimized algorithm, **PO** performs optimization along both the dimensions -- 1) parallel threads and 2) blocking within each thread. While the number of threads is fixed to 4 in the problem, the blocking factor was determined by experiments as done for the algorithm SO.

**Experiments**

The four algorithms coded in C with Posix threads were run on the linux.uchicago.edu machines for different values of N ranging from 32 to 12,000 entries using four automated scripts – serial\_scr (runs SO), *sernopt\_scr* (runs SN), parallel\_scr(runs PO) and *parnopt\_scr*(runs PN). The entries of the input matrices A and B were all set to the same value, ENTRY. In our experiments we set ENTRY = 1 since this made it easy to test that the algorithm was performing correctly. For ENTRY =1, and matrix dimension N, all the entries of the C matrix must have the value N. This was checked by setting a configuration parameter in these algorithm to write the matrix C to a file for each N. This parameter was set to value 0 while measuring performance so that the overhead of writing these matrices may be avoided.

These scripts were run individually executing the corresponding algorithm for different values of N one after another starting with value 32 and progressively increasing to end at value 12,000. The performance of each run was measured using the Linux time command and the resulting *real*, *user* and *system* time values were automatically output to a corresponding output file. We used the *real* time value to plot the speedup of algorithm B over algorithm A: Speedup = (real time value of Serial A/real time value of B).

The blocking factors used in SO and NO were determined experimentally for each value of N and varying the blocking factor from 16 to 256 and choosing the one that gave the best performance in terms of *real* time.

The results of these runs are given the graphs below. The first graph plots depicts the performance of the algorithms SO and PO in terms of the real time values output by the linux time command. The X-axis is the matrix dimensions and the Y-axis is the time taken in seconds. The second graph plots the speedup obtained by parallelization. It shows that we get the best, 4.5 speed up for N= 2048, and around 4X speedup up to N = 8192. And, this shows the benefits of parallelization using just 4 threads. The last graph plots the performance of SO and PO for N = 1K, 2K, 4K and 8K sizes. Based on the graph it can be concluded that the best performance for these values of N are observed with the blocking factor of 128 except for N = 8K where blocking factor of 32 is better.

References

[1] <https://stanford.edu/~rezab/classes/cme323/S16/notes/Lecture03/cme323_lec3.pdf>

[2] V. V. Williams, Multiplying matrices in O(n 2.373) time, Stanford University, (2014).

[3] R. Bryant and D. R. O’Hollaran, CS:APP2e Web Aside MEM:BLOCKING: Using Blocking to Increase Temporal Locality∗ 2012.