CS406 HW1 Report

Kerem Yildirir

March 2019

Contents

1 Problem Definition

Permanent of a matrix is an imminent feature of a matrix, like determinant. Computing the permanent of a matrix is considered to be a more difficult problem than computing the determinant.
Arithmetically, computing the permanent of a matrix is as follows:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = aei + bfg + cdh + ceg + bdi + afh$$

The most efficient algorithm for computing the permanent of a matrix is Ryser algorithm, which has the complexity of $O(2^{n-1}n^2)$. In this homework, an efficient variant of the Ryser algorithm, Nijenhuis and Wilf algorithm, which has complexity $O(2^{n-1}n)$ is implemented.

2 Sequential Implementation

Input: Square Matrix M of Size NxN
Output: Permanent of M

```
lastCol = M[:,N-1]                                   ▷ Last column of M
sumCol = Columnwise summation of M
 x = lastCol - sumCol/2
 p = ∏_i^N x[i]
for i in 1 → 2^(n-1) do
    y = ith grey code number
    z = Changed bit of y
    s = Sign of z (1 if changed bit was 0, -1 otherwise)
    prodSign = (−1)^i
    x += s*M[:,z]
     p += prodSign * ∏_i^N x[i]
return (4 * (n % 2) – 2 ) * p
```

In sequential algorithm, it looks like the values in the vector x are dependant on each other at every iteration. However, since we either add or subtract the column z to x at each iteration, it can be observed that the values of x vector can be computed independently at any given iteration by the following equation:

```
x = Initial computation of x
for t in set bits of y do          ▷ y is the grey code value computed at ith iteration
    x += M[t,:]                    ▷ Add column t of M to the x
```

Sequential algorithms runs around 0.8 seconds using O3 optimization for input size 25.

3 Parallel Implementation

Since each iteration of the loop is independent, it can be divided into multiple threads such that the work is shared. However, computing the x values from scratch for each iteration is an expensive strategy. Hence, in the parallel implementation, for each thread, we calculate the initial starting x value, then continue with the sequential algorithm for each thread. Since each thread only works in their own predetermined chunk, these computations can be done in parallel. Naive parallel algorithm is as follows:

chunkSize = $2^{n-1}/numThreads$
for each thread **do**
startPoint = threadId*ChunkSize
Calculate x vector for starting point
for i=threadId*chunkSize → threadId*chunkSize + chunkSize **do**
▷ Each thread works in their sliced region, on their private x value, p is shared
y = ith grey code number
z = Changed bit of y
s = Sign of z (1 if changed bit was 0, -1 otherwise)
prodSign = $(-1)^i$
x += s*M[:,z]
p += prodSign * $\prod_i^N x[i]$

Starting from the I consecutively made 4 optimizations on the implementation. Details of the optimizations are explained in Improvements section. Results of the parallel algorithm can be seen on the following tables and charts:

4 Results

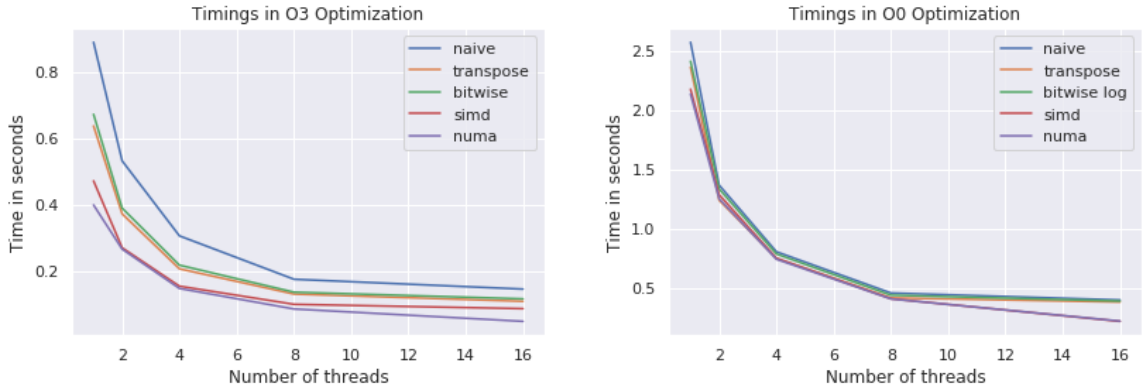

Figure 1: Timings on input size N=25

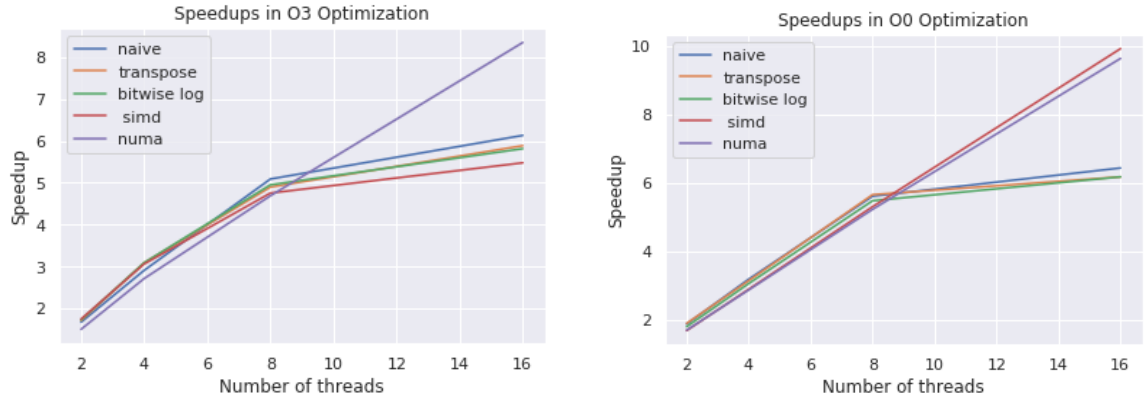

Figure 2: Speedups on input size N=25

Method	1	2	4	8	16
naive	0.8913	0.5328	0.3062	0.1751	0.1453
bitwise	0.6736	0.3899	0.2177	0.1360	0.1158
trans	0.6383	0.3725	0.2066	0.1303	0.1084
simd	0.4723	0.2702	0.1542	0.0993	0.0862
numa	0.3996	0.2659	0.1472	0.0852	0.0479

Method	1	2	4	8	16
naive	2.5723	1.3675	0.8061	0.4581	0.3995
bitwise	2.4118	1.3352	0.7875	0.4399	0.3902
trans	2.3616	1.2408	0.7505	0.4172	0.3822
simd	2.1759	1.2864	0.7510	0.4098	0.2193
numa	2.1355	1.2525	0.7426	0.4083	0.2216

Figure 3: Average timings for O0 optimization on input size N = 25

Method	2	4	8	16
naive	1.6731	2.9109	5.0917	6.1330
bitwise	1.7274	3.0936	4.9520	5.8163
trans	1.7133	3.0898	4.8986	5.8876
simd	1.7481	3.0633	4.7555	5.4783
numa	1.5029	2.7152	4.6903	8.3520

Method	2	4	8	16
naive	1.8810	3.1910	5.6154	6.4396
bitwise	1.8064	3.0627	5.4829	6.1806
trans	1.9034	3.1468	5.6607	6.1798
simd	1.6915	2.8975	5.3091	9.9228
numa	1.7050	2.8757	5.2308	9.6371

Figure 4: Speedups for O3 and O0 optimization on input size N = 25

Method	2	4	8	16
naive	0.8365	0.7277	0.6365	0.3833
bitwise	0.5000	0.4477	0.3583	0.2104
trans	0.2772	0.2500	0.1982	0.1191
simd	0.1838	0.1610	0.1250	0.0720
numa	0.0900	0.0813	0.0702	0.0625

Method	2	4	8	16
naive	0.9405	0.7978	0.7019	0.4025
bitwise	0.5000	0.4239	0.3794	0.2138
trans	0.3024	0.2500	0.2249	0.1227
simd	0.1593	0.1364	0.1250	0.1168
numa	0.0885	0.0746	0.0678	0.0625

Figure 5: Parallel efficiency for O3 and O0 optimization on input size N = 25

5 Improvements

On top of the Naive implementation that described earlier, I first used bitwise operations to compute the logarithm function. Then, I took the transpose of the matrix in order to have better memory access. Then I applied SIMD(Single Instruction Multiple Data) functions. Finally I exploited the NUMA architectire in our system. At the end with all these optimizations, I was able to gain 8.35 speedup in O3 and 9.63 speedup in O0 with 16 threads. It can be seen that the biggest impact on the running time was using SIMD instruction and NUMA architecture. However in O0, using NUMA did not make a positive impact in 16 threads, however it was slighly better on 1,2,4,8 threads. It can be said that that it doesn't make a huge difference in O0. In O3 however, it halved the performance with 16 threads. All the experiments are done on with input size N = 25 as it was the only test input among our test cases. Just for clarification, following chart shows the running time of the final system on all inputs.

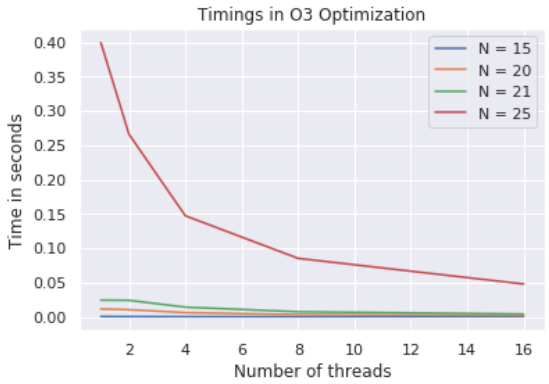

Figure 6: Running time of all inputs with final system

5.1 Bitwise Operations and Compiler Intrinsics

In the naive implementing, when computing the change bit in the grey code sequence, log2 function from cmath library is used. Cmath implementation does not offer very good performance, hence it is more efficient to compute the logarithm using bitwise operations. Another improvement for performance is using builtin compiler functions while calculating the indices of set bits in y values.(GCC has **__builtin_popcount()** for counting the number of set bits and **__builtin_ffs()** for finding the index of the most significant bit) Instead of looping through all the bits, these instructions allows us to compute the indices of set bits faster.

5.2 SIMD

The most expensive part of of the algorithm is the product operation that being done on the vector x at each iteration of the loop. On the initial algorithm there are N-1 multiplications at each iteration, summing up to $2^{n-1} * (N-1)$ multiplications total. Using OpenMp's **#pragma omp simd** clause before the loop and **-mavx** and **-mavx2** during compilation vectorizes this multiplication and gives us a signigicant improvement on the performance when compiled with O3. This implementation also minimizes false sharing as it prevents the threads from accessing the same cache line by distributing them to chunks.

5.3 Memory Access Optimizations

5.3.1 Memory access

Another bottleneck on the performance is memory access. When looked at the algorithm, it can be seen that at each iteration, each thread accesses a column of the matrix M and the vector x. Taking the transpose of the matrix allows us to fit the matrix in the cache lines and have fast row major access to the matrix from the cache line, increasing spatial locality. However, since the matrix size is relatively small, it does not have a huge impact on the performance for this implementation.

5.3.2 Spreading threads in NUMA architecture

Our server has NUMA(Non-uniform memory access) memory architecture with 32 cores and 2 sockets, meaning that there are 16 cores per socket in the machine. In the naive implementation, M is shared and kept in the core its initialized due to first touch initialization policy.This results in communication overhead between threads. X is private for each thread, but it is also kept where first initialized. Using **#pragma omp spread** and **OMP_PLACES=cores**, we can distribute threads to cores and minimize the need for intercommunication required during the execution. Since we are using at most 16 threads, we are using one socket for all the cores. If we distribute threads to sockets, it would require a communication between sockets, hence it results in slightly worse performance.

6 Instructions for compiling

For compilation, declare the environment variable **OMP_PLACES** to **cores**. Then proceed with compilation. O3 executable is out3, O0 executable is out0.
export OMP_PLACES=cores
g++ permanent_hw1.cpp -o out3 -mavx -mavx2 -fopenmp -O3 -std=c++11
g++ permanent_hw1 -o out0 -mavx -mavx2 -fopenmp -O0 -std=c++11