CS406 Parallel Programming - Homework Assignment 1

Baris Sevilmis

Lab Date: 10/03/2019
Due Date: 29/03/2019

1 Problem Definition

Purpose of this homework is to implement Matrix Permanent Calculation Code as efficient as possible. Nijenhuis and Wilf Algorithm is used for efficiency, and it is be parallelized using various methods that will be explained in further sections of this report. OpenMP library is utilized for parallelization.

2 Nijenhuis and Wilf Algorithm for Matrix Permanent

Nijenhuis and Wilf Algorithm is alternative method to naive matrix permanent calculation. Algorithm 1 provides pseudocode of the sequential algorithm. Parallel version of algorithm will be provided in next section. For sake of convenience, it would be reasonable to explain algorithm shortly. Initially, x and p values are initialized by the following operations in Algorithm 1. Consecutively within a loop of $1 \rightarrow 2^{N-1} - 1$, gray code of current and previous iterations are determined to find changing bits and changing bit is used for updating x. Product of x values are used for updating p value, which at the end determines the final result of matrix permanent. From sequential perspective, implementation of the following algorithm can be done very efficiently using C/C++ by taking advantage of bitwise operations and compiler directives. From parallel perspective, there are some issues that requires handling such as breaking loop dependencies. Next section of the report will be including all the necessary details for parallel implementation issues and their solutions.

Algorithm 1 Nijenhuis and Wilf Algorithm

```
 1: procedure MATRIX PERMANENT(matrix M)
 2:     $N \leftarrow size(M)$
 3:     for $i \leftarrow [0 \rightarrow N-1]$ do
 4:         for $j \leftarrow [0 \rightarrow N-1]$ do
 5:             $sumCol \leftarrow M[i][j]$
 6:         $lastCol \leftarrow M[i][N-1]$
 7:         $x[i] \leftarrow lastCol-sumCol/2$
 8:         $p \leftarrow p*x[i]$
 9:     for $m \leftarrow [1 \rightarrow 2^{N-1} - 1]$ do
10:         $y \leftarrow GrayCode(m)$
11:         $yp \leftarrow GrayCode(m-1)$
12:         $z \leftarrow ChangingBit(y,yp)$
13:         $s \leftarrow FindSign(y,z)$
14:         $s \leftarrow ProdSign(m)$
15:         for $n \leftarrow [0 \rightarrow N-1]$ do
16:             $x[n] \leftarrow lastCol-sumCol/2$
17:             $p \leftarrow p*x[n]$
        return $result \leftarrow 4*(n\%2)-2*p$
```

3 Parallel Nijenhuis and Wilf Algorithm

Parallelizing Nijenhuis and Wilf Algorithm can not be performed directly as there are some important points to consider. It is obvious that parallelization will be mainly performed over the loop $1 \rightarrow 2^{N-1} - 1$, since N is not large on its own. Main problem for the parallelization is the dependency of x and p values over the main iteration space. In order to parallelize these values, understanding graycode plays a crucial role. Gray code values are same size as the main loop which is $1 \rightarrow 2^{N-1} - 1$, however their ordering are based on the bit differences between consecutive values. Each consecutive integer has specifically 1 bit difference. Following formulation can be used to understand this concept:

Iteration	Gray Code Value	Bitwise Representation
1	1	00000001
2	3	00000011
3	2	00000010
4	6	00000110
5	7	00000111
$\vdots$	$\vdots$	$\vdots$

Index of changing bits are used to pick specific columns of input matrix M. If sequence of changing bits are examined, it will become clear that changes over original x values are temporary. In other words, once a specific column z is added, it is subtracted after 2^z iterations. In addition, without finding changing bits we can directly compute set bit indexes of specific gray code order and add corresponding indexes of matrix M to the original x values every iteration. This way iteration space can be traversed independently, however finding set bit indexes of gray codes each iteration would be computational wise very inefficient. Therefore, this pattern should be used more carefully in order to obtain better timing results.

To increase efficiency, going over these set bits of some gray code values would be enough. Every thread needs to have an initial x value, which will be computed by going over their initial gray code values set bits only once. As corresponding set bit indexes of gray code within matrix M will be added over original x values, every thread will have their original x values. One important thing is to make x private for each thread so that race condition over x values will be resolved. Another important detail is to schedule threads so that their updates over x values will be consecutive. This way, every thread will update their x values independently from each other with correct results and update p value by with correct x products independently. By using static scheduling and using reduction over p value(to avoid race condition) this implementation achieves success with efficient results. Algorithm 2 provides pseudocode for such a parallel implementation. In the following subsections, further implementation detail will be explained, by which efficiency is further increased.

Algorithm 2 Parallel Nijenhuis and Wilf Algorithm

```
procedure PARALLEL MATRIX PERMANENT(matrix M)
    $N \leftarrow size(M)$
    $CHUNK \leftarrow 2^{N-1}/total\_threads$
    for $i \leftarrow [0 \rightarrow N-1]$ do
        for $j \leftarrow [0 \rightarrow N-1]$ do
            $sumCol \leftarrow M[i][j]$
        $lastCol \leftarrow M[i][N-1]$
        $x[i] \leftarrow lastCol-sumCol/2$
        $p \leftarrow p*x[i]$
    #pragma omp parallel firstprivate(x){
    $m \leftarrow CHUNK*tid$
    $y \leftarrow GrayCode(m)$
    $sb[:] \leftarrow FindSetBitIndexes(y)$
    for $n \leftarrow [0 \rightarrow N-1]$ do
        $x[tid][n] \leftarrow M[n][sb[:]]$
    #pragma omp for schedule(static)
    for $m \leftarrow [1 \rightarrow 2^{N-1}-1]$ do
        $y \leftarrow GrayCode(m)$
        $yp \leftarrow GrayCode(m-1)$
        $z \leftarrow ChangingBit(y,yp)$
        $s \leftarrow FindSign(y,z)$
        $prodSign \leftarrow ProdSign(m)$
        for $n \leftarrow [0 \rightarrow N-1]$ do
            $x[tid][n] \leftarrow s*M[n][sb[:]]$
            $p \leftarrow p*x[n]*prodSign$
    } return $result \leftarrow 4*(n\%2)-2*p$
```

3.1 Transpose of Matrix M

As we are using columns of matrix M for updating x, our access to matrix M can be viewed as column major access. Therefore, cache usage is not optimized. By taking transpose of matrix M, access to matrix can be converted into row major access form, which will result in a more efficient cache usage rather than going to memory every time M columns are used.

Although, taking transpose of matrix M seems useful, increase in efficiency is rather low. However, it still can be considered as an improvement. In Results section, transpose results are provided, namely TR.

3.2 NUMA

Using Non Uniform Memory Adress Space in our favor, results in more consistent and efficient results. Depending on the parallel architecture that is used, NUMA is an effective way to distribute threads across cores or sockets.

In our case our parallel platform consists of 2 sockets with 32 cores on total. Each socket has 16 core on its own. As we distribute our threads at the beginning of the parallel region by using "**proc_bind(spread)**", we achieve higher success. Important thing to remember is to use command "**export OMP_PLACES=cores;**" at the terminal environment, otherwise NUMA is not utilized. Results of NUMA on top of transpose method can be seen in Results section.

3.3 SIMD

Lastly, SIMD was used for utilization.If "**#pragma omp simd**" is used in the correct place, it affects the performance in positive manner.

SIMD was used within the main loop in order to improve performance of updating x. As we are calculating product of x in a temporary variable "**reduction(temp)**", was used to ensure calculating correct results. Important thing to remember in this part is to use **-mavx** and **-mavx2** when compiling our code to ensure performance improvement. Results of SIMD on top of NUMA and Transpose method is depicted in Results section.

4 Results

C/C++ is used as the programming language for parallel implementation as well as the sequential implementation. Testing of algorithms were done in Nebula with the following specs:

- Model name: Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 32
- Thread(s) per core: 2
- Core(s) per socket: 8
- Socket(s): 2
- NUMA node(s): 2
- Vendor ID: GenuineIntel
- CPU MHz: 1200.117, CPU max MHz: 3000.0000, CPU min MHz: 1200.0000
- L1d cache: 32K, L1i cache: 32K, L2 cache: 256K, L3 cache: 20480K

Compilation was done consecutively for -O3 and -O0 with following line of commands:
•For -O3:
export OMP_PLACES=cores;
g++ -o out3 parsrcopt.cpp -O3 -fopenmp -mavx mavx2
•For -O0:
export OMP_PLACES=cores;
g++ -o out0 parsrcopt.cpp -O0 -fopenmp -mavx mavx2

Every case was run 100 times with an input matrix of size $25x25$ on same machine consecutively and average of their results were taken into consideration. Following Table 1 and Figure 1 demonstrate their average run times for -O3. Figure 2 depict speed up results for -O3 cases and Figure 3 depict efficiency results for -O3.

Table 1: Timing results for -O3

Thread Num.	SIMD+NUMA+TR -O3	NUMA+TR -O3	TR -O3	CLASSIC -O3
1	0.381520s	0.551276s	0.580000s	0.669678s
2	0.268671s	0.346221s	0.361502s	0.413929s
4	0.148800s	0.190951s	0.192082s	0.221021s
8	0.085767s	0.119825s	0.122365s	0.138353s
16	0.049111s	0.077839s	0.099066s	0.114613s

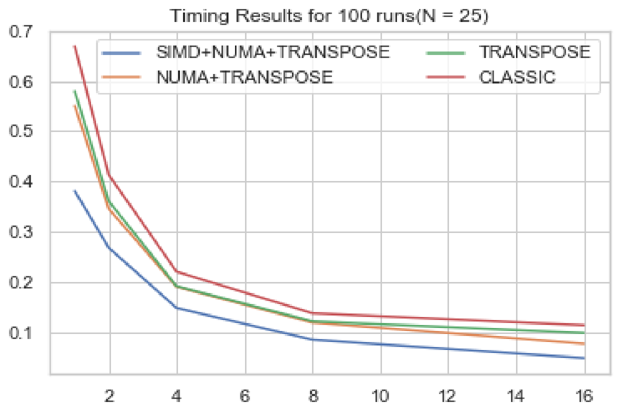

Figure 1: Average Timing Results for -O3 (100 runs)

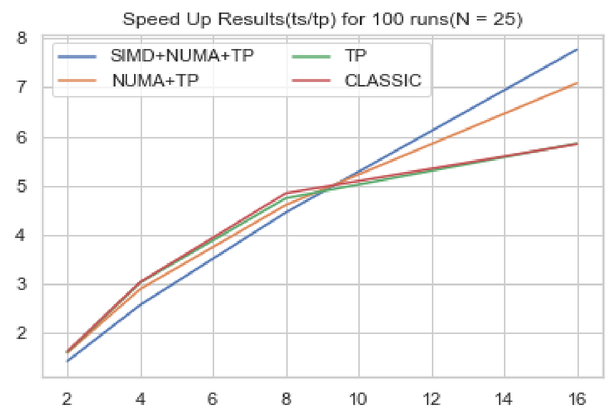

Figure 2: Speed Up Results($\frac{t_s}{t_p}$) for -O3 (100 runs)

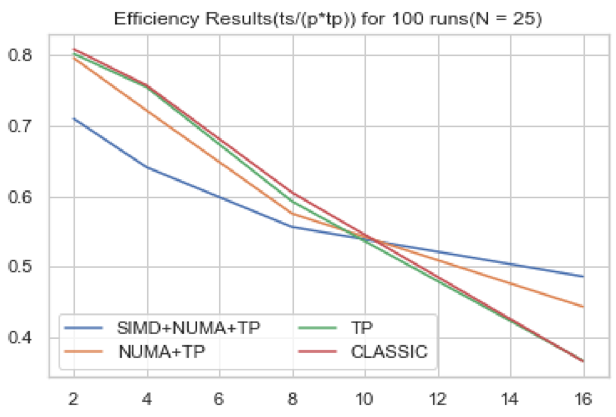

Figure 3: Efficiency Results($\frac{t_s}{p*t_p}$) for -O3 (100 runs)

As well as -O3, every case for -O0 was also run 100 times with an input matrix of size $25x25$ on same machine consecutively and average of their results were taken into consideration. Following Table 2 and Figure 4 demonstrate their average run times for -O0. Figure 5 depict speed up results for -O0 cases and Figure 6 depict efficiency results for -O0.

Table 2: Timing results for -O0

Thread Num.	SIMD+NUMA+TR -O0	NUMA+TR -O0	TR -O0	CLASSIC -O0
1	2.042397s	2.133986s	2.163870s	2.227594s
2	1.214911s	1.217536s	1.255389s	1.324422s
4	0.718879s	0.719756s	0.751527s	0.789886s
8	0.390786s	0.392677s	0.416737s	0.429124s
16	0.214891s	0.248477s	0.344692s	0.353286s

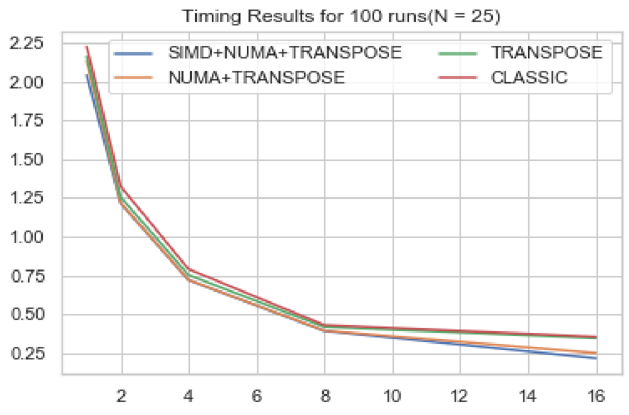

Figure 4: Average Timing Results for -O0 (100 runs)

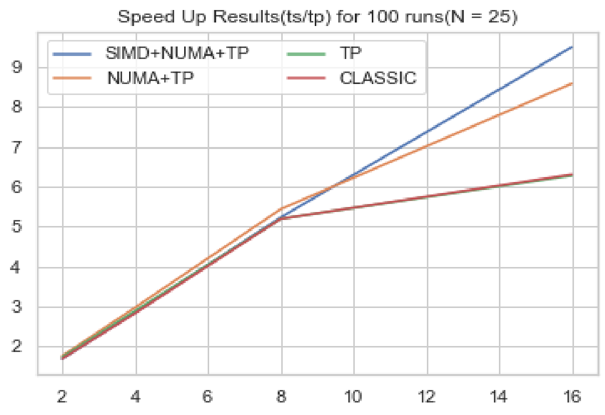

Figure 5: Speed Up Results($\frac{t_s}{t_p}$) for -O0 (100 runs)

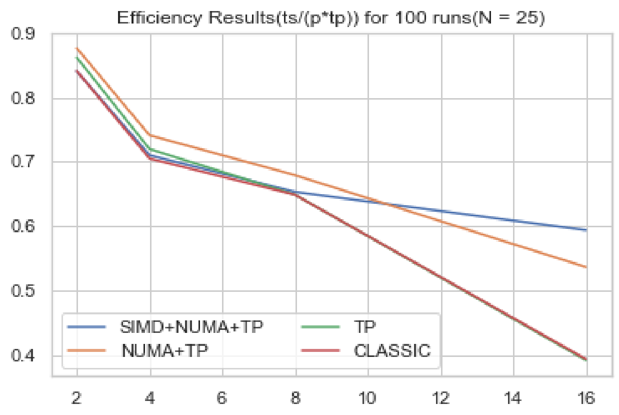

Figure 6: Efficiency Results($\frac{t_s}{p*t_p}$) for -O0 (100 runs)