CS406 HW1

Adnan Kaan Ekiz

March 2019

1 Introduction

For the Homework 1 of the Parallel Computing Course, main idea was to efficiently create a matrix permanent calculation code. During the process of implementing the code, it is important to recognize and briefly mention some of the key parts which seems crucial. For that reason, homework 1 report will present some idea about the algorithm and how it is implemented for sequential and parallel run, some of the tricks and steps used in the implementation phase, and time result of impovements with an overall analysis.

2 Algorithm

To achieve the ultimate goal of low complexity, for this homework, I have decided to use the matrix permanent algorithm introduced by **Nijenhuis and Wilf**. For the sequential implementation, pseudocode of the algorithm as follows:

Algorithm 1 Matrix Permanent Algorith proposed by Nijenhuis and Wilf

```
1:  procedure PERMANENT[MATRIX M,ROWSIZE N]
2:      lastColumn ← M[:N-1]
3:      sum ← M[:]
4:      x ← lastColumn - (sum/2)
5:      for i in x do
6:          p ← p * x[i]
7:      for i in [0,exp(2,n-1)] do
8:          z ← findChangingBit()
9:          s ← findSign()
10:         prodSign ← exp(-1,i)
11:         x ← x ∗ M[:z]
12:         p ← p + [x1*x2....]
        return 4*(remainder(N,2)-2)*p
```

Hence, algorithm has a complexity of $O(2^{n-1}n)$, which is one of the best algorithms in terms of complexity. Main idea in the algorithm is to create a gray code order for the changing bits, and use the location of the bit to update the x array for the next computation to come.

2.1 Sequential Implementation

Given the pseudocode of the sequential algorithm above, algorithm has 2 different loop part:

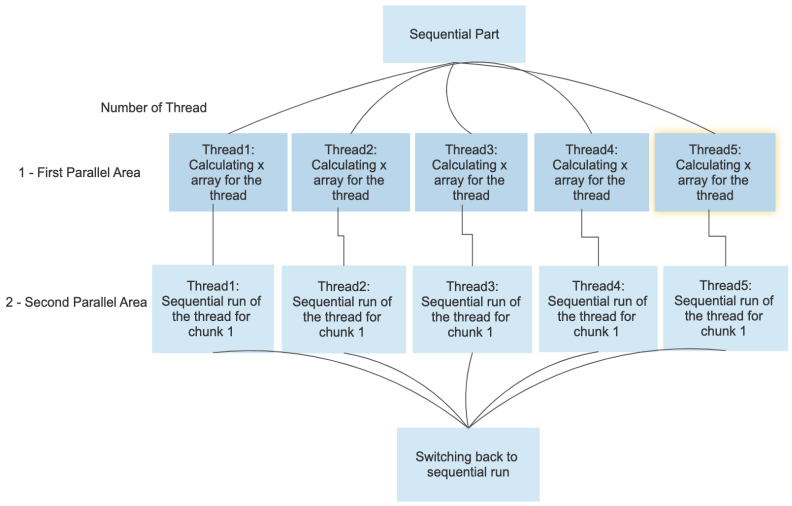
1. In the first loop, computation of the x array is being created. This array will further used for the calculation of the permanent value at the end of the algorithm.

2. During the each iteration of the second loop, algorithm compares the current version of the bits in gray code with previous iteration's bits, hence finding the changing bit as well as the sign which determines whether the column of the matrix will be incremented or decremented from the elements of the x array.

3. Finally, updating the p variable and determining the final result.

2.2 Parallel Implementation

In terms of parallel implementation, computation of the initial x array by substracting the sums of the row from the last column takes very little time. Therefore, determining the value in sequential run can be more efficient due to possible overhead time caused by threads in parallel area. For the other computations, structure of the parallel implementation is given in below:



Then, algorithm opens a parallel region before the main for loop in order to determine the distinct starting locations of different threads. Determining the starting location for each thread helps us to derive the gray coded bits and allow algorithm to easily update the x array. At the beginning, **Nijenhuis and Wilf** used the last known value of x array to find the current x array by using other values such as z(changing bit location) and s(sign). This proposition created a bottleneck to find the value of x array without finishing the iterations up until the value of certain desired x array and bits needed for that. In general, algorithm has the following distinct steps to find x without the need of earlier values of x:

1. Copying the initial x values calculated in sequential region for each thread under the name of x_{spec}
2. Depending on the thread id, divides the maximum work to number of threads and finds the starting location for the iteration of each thread
3. For each 1 in the gray code number(calculated by doing the following: (starting location) XOR (starting location >> 1)), algorithm adds the n^{th} column values to copied x_{spec} array.
4. For each thread, algorithm behaves as a sequential run with calculated x_{spec} arrays

3 Tricks

3.1 SIMD

When compiling the code, I apply pragma omp simd directive i to indicate that multiple iterations of the loop can be executed concurrently by using SIMD instructions. [1]SIMD (Single Instruction Multiple Data) is a computer science method of combining multiple operations into a single computer instruction. The employment of SIMD results in significant savings in speed for algorithms designed to work in parallel.

3.2 Transpose Matrix Idea

It is beneficial in the homework to access the data stored in matrix in row-wise fashion rather than accessing in column-wise fashion. It helps because when accessing the data in column-wise fashion, program uses the data stored in cache much worse. This operation helps us to increase spatial locality and hence, data access rate.

3.3 Using Bitwise Operations

It also helps to use bitwise operations when calculating exponential values. In homework, it showed that using bitwise operations decreases the total time of

the run. It was also beneficial to use $--builtin-ctz$ function (builtin method is provided by GCC to count the number of leading zero's in variable) rather than using functions such as log2. These small tricks helped whole algorithm gain speed up slightly especially for big values of N.

4 Results

To compare the final results of the parallel algorithm, average of 10 tests for each setting will be shown in addition to optimization flags used while compiling the code. Experiment has a 3 different setting for each compiler option which is -O3 and -O0. These settings are:

1. With transpose matrix, without using simd
   g++ -o parallel.out example_par_v5.cpp -fopenmp -O3
   g++ -o parallel.out example_par_v5.cpp -fopenmp -O0

2. Without simd or transpose matrix
   g++ -o parallel.out example_par_v5.cpp -fopenmp -O3
   g++ -o parallel.out example_par_v5.cpp -fopenmp -O0

3. With simd and transpose matrix
   g++ -o parallel.out example_par_v5.cpp -fopenmp -mavx -mavx2 -O0
   g++ -o parallel.out example_par_v5.cpp -fopenmp -mavx -mavx2 -O3

Setting	1	2	3
Using -O0	Average Time(s)		
Thread Num(1)	2.651178	2.67756	2.984285
Thread Num(2)	1.491256	1.436022	1.584764
Thread Num(4)	0.8683932	0.8226553	0.9152636
Thread Num(8)	0.5231727	0.461918	0.5356832
Thread Num(16)	0.3498677	0.2459405	0.3082424
Using -O3	Average Time(s)		
Thread Num(1)	0.8180378	0.9062742	0.8214403
Thread Num(2)	0.4920578	0.5152223	0.5279467
Thread Num(4)	0.2741389	0.2721268	0.2625418
Thread Num(8)	0.1593877	0.1502378	0.1480599
Thread Num(16)	0.09887453	0.08749265	0.08516193

5 Conclusion

In conclusion, when I have used transpose matrix to retrieve data from the matrix in addition to using simd actions, run time of the code has decreased and it improved the run time of the code. When I used transpose matrix but didn't

include the simd actions when compiling the code. It did some improvement when thread number was higher but didn't really increased the time when thread number is smaller or equal than 4.

As expected, run time has increased when no transpose matrix and simd function is used. Therefore, it proved that using the transpose matrix and simd functions decreases the total time of parallelyzed code. In general, parallelyzed and improved version of the algorithm has about 9.64 speed-up compared to the sequential version of the algorithm.

6 References

1 http://www.tech-faq.com/simd.html

2 https://www.go4expert.com/articles/builtin-gcc-functions-builtinclz-t29238/

3 http://www.tech-faq.com/simd.html

4 https://www.mathworks.com/matlabcentral/fileexchange/53784-matrix-permanent-using-nijenhuis-wilf-in-cmex