Maxim Dossioukov
CSC 656-01, F23
Coding Project #3

Part 5 – Analyzing Results

Table 1:
Rows are problem sizes, Columns are results for each: Blas, Basic, Vectorized, OMP1, OMP4, OMP16, OMP64

Results are times in seconds

	Problem Size(N)	Blas	Basic	Vectorized	omp-1	omp-4	omp-16	omp-64
Time Results	1024	0.00016	0.00354	0.00024	0.00355	0.0015	0.00049	0.00069
	2048	0.0005	0.01426	0.0011	0.01427	0.00427	0.002	0.00179
	4096	0.00406	0.05722	0.00472	0.05726	0.01518	0.00462	0.00364
	8192	0.01831	0.22923	0.01954	0.2294	0.05808	0.0156	0.01366
	16384	0.07539	0.91834	0.07957	0.92001	0.23041	0.05853	0.05818

Table 2:
Rows are problem sizes, Columns are results for each: Blas, Basic, Vectorized, OMP1, OMP4, OMP16, OMP64

Results are MFLOPS calculated by:
(N / 1m) / time.

Based on the code I wrote:

```
for(int row = 0; row < n; row++){
        row_offset = row * n;
        for (int col = 0; col < n; col++){
        y[row] += A[row_offset + col] * x[col];
        }
}
```

The outer loop runs N times
Row offset is 1 multiplication operation.
The inner loop runs N times
Inside the loop:
y+= A[row_offset + col] * x[col] involves 2 array accesses, 2 additions and multiplication.

Resulting in a 5n^2 run time. This is plugged into N. So the calculation is:
(5n^2 / 1m)/time

MFLOPS	Problem Size(N)	Blas	Basic	Vectorized	omp-1	omp-4	omp-16	omp-64
	1024	32768	1481.039548	21845.33333	1476.867606	3495.253333	10699.7551	7598.376812
	2048	41943.04	1470.653576	19065.01818	1469.622985	4911.362998	10485.76	11715.93296
	4096	20661.59606	1466.027263	17772.47458	1469.622985	4911.362998	18157.16017	23045.62637
	8192	18325.74113	1463.788858	17172.17605	1462.704098	5777.278237	21509.25128	24564.00586
	16384	17803.12084	1461.525448	16867.88086	1458.872491	5825.169394	22931.44165	23069.39292

Table 3:
Rows are problem sizes, Columns are results for each: Blas, Basic, Vectorized, OMP1, OMP4, OMP16, OMP64

Result is the % bandwidth calculated by:

$$\frac{\left(\frac{8(N)}{\frac{1,000,000,000}{Time}}\right)}{204.8}$$

same as last CP #2

% of mem bandwidth	Problem Size(N)	Blas	Basic	Vectorized	omp-1	omp-4	omp-16	omp-64
	1024	0.25	0.01129943503	0.1428571429	0.00995024875	0.02797202797	0.08	0.06896551724
	2048	0.16	0.005610098177	0.05755395683	0.00494437577	0.07142	0.01616161616	0.04395604395
	4096	0.039408867	0.002796225096	0.02768166089	0.00244760593	0.00911680911	0.03595505617	0.03747072599
	8192	0.01747678864	0.001395977839	0.01344537815	0.00122845406	0.00454351838	0.01262825572	0.01866977829
	16384	0.008489189548	0.0006969096413	0.008043232374	0.0006956446126	0.0023046453	0.00922190201	0.00826873385

1. Comparing the results of your basic and vectorized implementations at N=16384, which code has better performance in terms of MFLOP/s, and by how much? Which code has better memory system utilization, and by how much?

   In terms of MFLOP/s The vectorized code performs better. Practically a 11x performance difference. However the memory utilization was higher on the vectorized code, although not by a significant amount ~1%

Basic	Vectorized
1481.039548	21845.33333
1470.653576	19065.01818
1466.027263	17772.47458
1463.788858	17172.17605
1461.525448	16867.88086

2. Comparing the results of your basic and OpenMP 4-way parallel implementation at N=16384, which code has better performance in terms of MFLOP/s and by how much? Which code has better memory system utilization, and by how much?

Basic	Vectorized	omp-1	omp-4	omp-16	omp-64
1481.039548	21845.33333	1476.867606	3495.253333	10699.7551	7598.376812
1470.653576	19065.01818	1469.622985	4911.362998	10485.76	11715.93296
1466.027263	17772.47458	1469.622985	4911.362998	18157.16017	23045.62637
1463.788858	17172.17605	1462.704098	5777.278237	21509.25128	24564.00586
1461.525448	16867.88086	1458.872491	5825.169394	22931.44165	23069.39292

The basic and omp-1 code are very similar but once we start looking at the omp-16 and omp-64 code we can see 16x MFLOP/s improvement. However our memory usage does not spike once again similar to the vectorized code comparison.

I want to note that I believe that my calculations are incorrect and this is likely the reason why I am not noting a difference of significant change.

3. Looking at the results of your OpenMP implementation at N=16384, what is the ***speedup*** of this code going from 1 to 4 threads, from 1 to 16 threads, and from 1 to 64 threads? Use your runtime data to compute these speedup metrics.

From 1 → 4 threads our speed up is ~ 4x
From 1→ 16 threads our speed up is ~ 15x
From 1→ 64 threads our speed up is ~ 16x and not much different than the 16 thread ex.