Raghav Pandya(rp835) - Midterm Report

We first start by analyzing the problem which is also popularly known as the Gauss-Siedel or stencil operation.

Identifying Dependencies

- Each row elements depends on the element to the left
- Each column depends on the previous column

Approaches

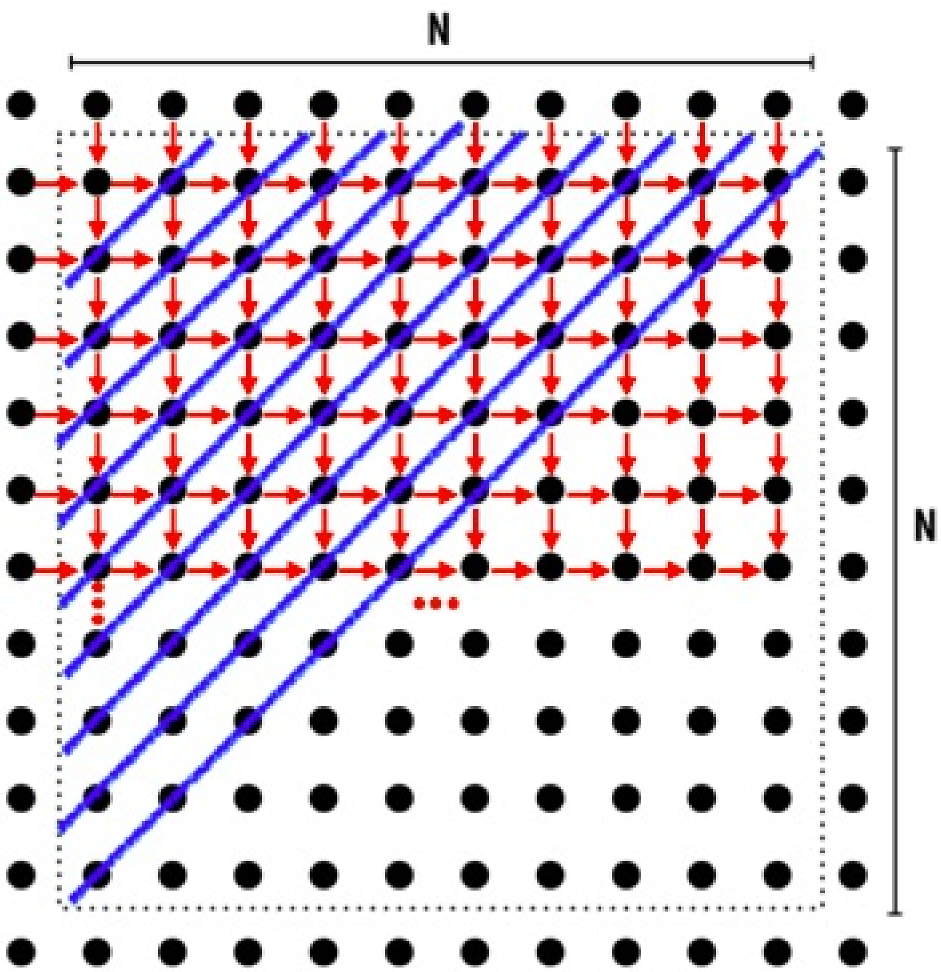
We will discuss and implement two ways of solving the problem and then discuss the tradeoffs of each approach:

- Wave Solver
- Red and Black Solver

Wave Solver

We can observe from the order of computaions for different elements of the matrix that a wave pattern exists. All elements in a single diagonal can be computed parallely but the computation of diagonals is sequential as next diagonal depends on the values from previous values.

See the below figure to get the idea.



> *Blue lines represent the diagonals*
>
> *Elements in the diagonal can be computed parallely*
>
> *Each diagonal should wait on the completion for diagonals on it's left*

```
for (int k = 1; k <= diagonals; ++k)
   {
     // printf("Diagonal: %d\n", k);
     #pragma omp parallel for num_threads(THREAD_COUNT) p
rivate(tmp, i, j) reduction(+:diff)
```

```
    for (i = (k <= n ? 1 : (k - n + 1)); i <= k; i++)
      {
        if(i <= n)
        {
          j = k + 1 - i;
          // printf("Thread: %d on (%d, %d)\n", omp_get_
thread_num(), i, j);
          tmp = A[i][j];
          A[i][j] = 0.2*(A[i][j] + A[i][j-1] + A[i-1][j]
 + A[i][j+1] + A[i+1][j]);
          diff += fabs(A[i][j] - tmp);
        }
      }
  }
```

The outer loop iterates over the diagonals sequentially and the inner loop calculates the elements on a particular diagonal.

We achieve the parallelism with :

```
#pragma omp parallel for num_threads(THREAD_COUNT) privat
e(tmp, i, j) reduction(+:diff)
```

We can specify the number of threads to be used for computation and workload is equally distributed among threads. We also use the `reduction(+:diff)` directive which instructs threads to accumalate the difference locally and then combine at the end when thread operations are done. This improves the parallelism.

Advantages:

- Easy and Effective parallelism
- Scaling: As the workload is distributed equally, it scales well
- Correctness: Exactly same result as the serial approach

Disadvantages:

- Frequent synchronizations between diagonals equal to " 2N - 1 "
- Not much parallelism in initial and final diagonals

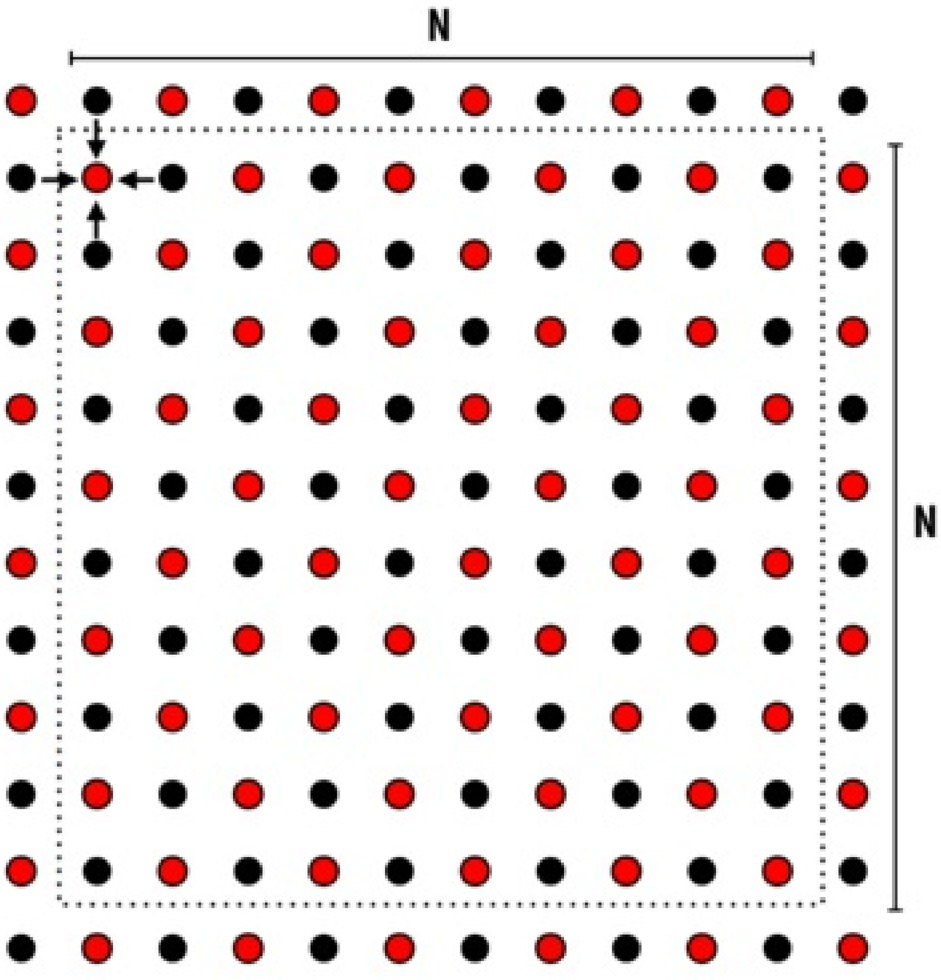
Red and black coloring approach

We slightly change the algorithm with the domain knowledge that an approximate solution is acceptable for applications using Gauss Siedel.

Key Ideas:

- Change the order in which cells are updated
- New algorithm converges in same(approximate) number of iterations
- Change is acceptable for applications utilizing the Gauss Siedel method

We split the elements into red and black color scheme such that no red element depends on black and vice versa



Firstly, red cells are computed in parallely and then the black cells are computed. We repeat the process until we acheive the desired convergence.

```
#pragma omp parallel num_threads(THREAD_COUNT) private(tm
p, i, j) reduction(+:diff)
    {
      #pragma omp for
      for (i = 1; i <= n; ++i)
      {
        for (j = 1; j <= n; ++j)
        {
```

```
            if ((i + j) % 2 == 1)
            {
              // Computation for Red Cells
            }
          }
        }
        #pragma omp barrier
      }
      #pragma omp parallel num_threads(THREAD_COUNT) privat
e(tmp, i, j) reduction(+:diff)
      {
        #pragma omp for
        for (i = 1; i <= n; ++i)
        {
          for (j = 1; j <= n; ++j)
          {
            if ((i + j) % 2 == 0)
            {
              // Computation for Black Cells
            }
          }
        }
        #pragma omp barrier
```

Advantages:

- Faster compared to wave approach and for smaller dataset as well

- Uniform parallelism

Disadvantages:

- Approximate solution

Analysis

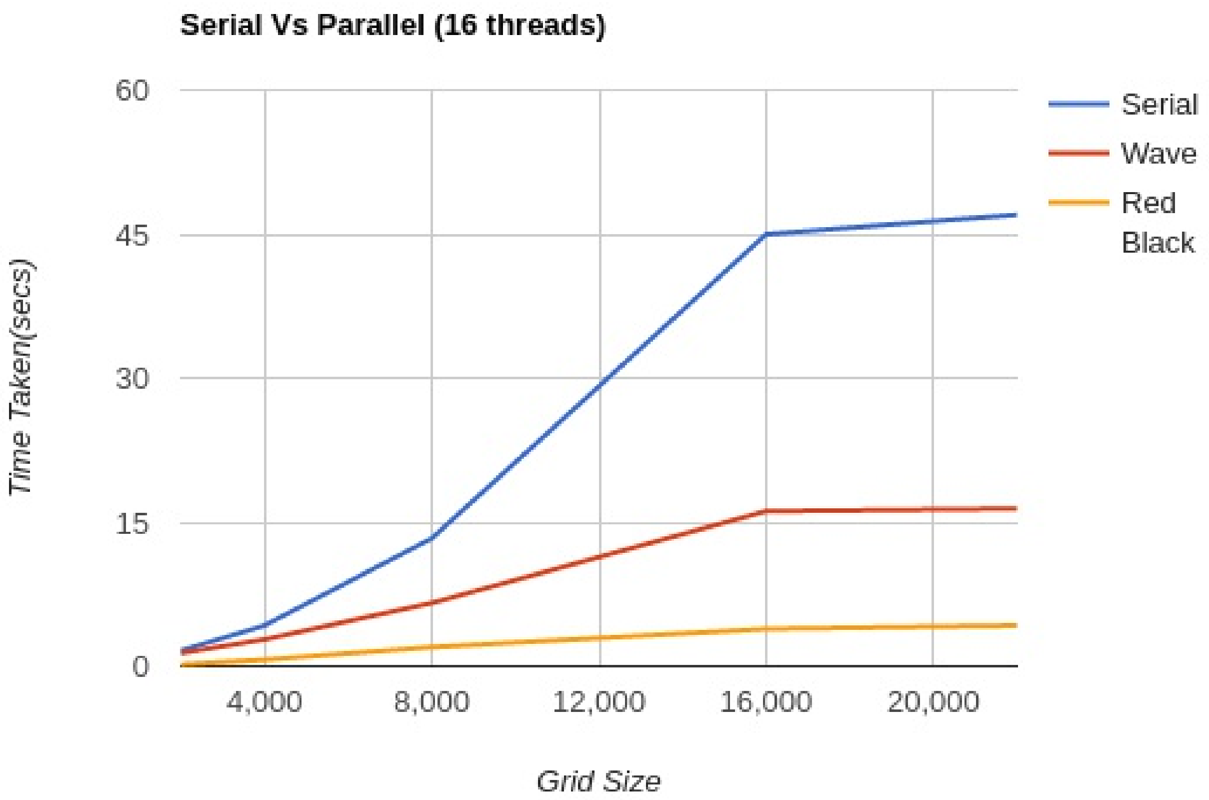
COMPARISON- SERIAL VS PARALLEL WAVE VS PARALLEL RED BLACK

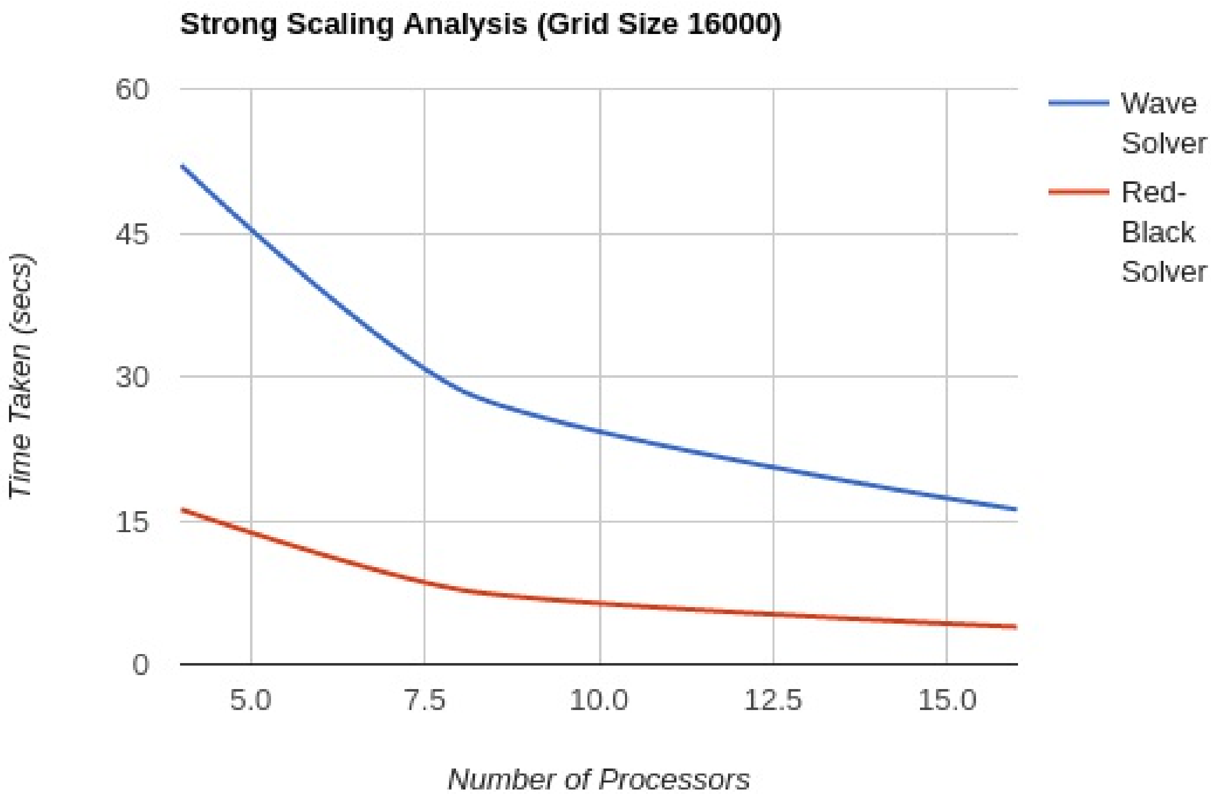
Grid Size	Serial		Parallel - Wave						Parallel - Red and Black					
			Thread : 4		Thread : 8		Thread : 16		Thread : 4		Thread : 8		Thread : 16	
	Iterations (LIMIT 20)	Time (secs)	Iterations (LIMIT 20)	Time (secs)	Iterations (LIMIT 20)	Time (secs)	Iterations (LIMIT 20)	Time (secs)	Iterations (LIMIT 20)	Time (secs)	Iterations (LIMIT 20)	Time (secs)	Iterations (LIMIT 20)	Time (secs)
500	20	0.0050	20	0.0340	20	0.0523	20	0.0871	20	0.0492	20	0.0257	20	0.0145
1000	20	0.5356	20	0.5087	20	0.5989	20	0.7889	20	0.1908	20	0.0973	20	0.0504
2000	20	1.6447	20	1.4077	20	1.1352	20	1.3814	20	0.7598	20	0.3818	20	0.1931
4000	20	4.2563	20	3.8001	20	2.8531	20	2.8003	20	2.0147	20	1.3107	20	0.6968
8000	20	13.3027	20	15.6991	20	7.6473	20	6.5929	20	6.2065	20	3.5011	20	1.9485
16000	20	44.9898	20	52.0419	20	28.6567	20	16.1537	12	16.1056	12	7.8037	12	3.9213
22000	10	47.1963	10	55.1821	10	31.3894	10	16.4176	7	18.2006	7	8.6333	7	4.2673
27000	7	49.6843	7	62.0270	7	32.0548	7	17.7709	5	19.5079	5	9.8606	5	4.7562
32000	5	49.7049	5	62.0577	5	32.1571	5	19.2416	4	22.8012	4	12.2923	4	6.1369

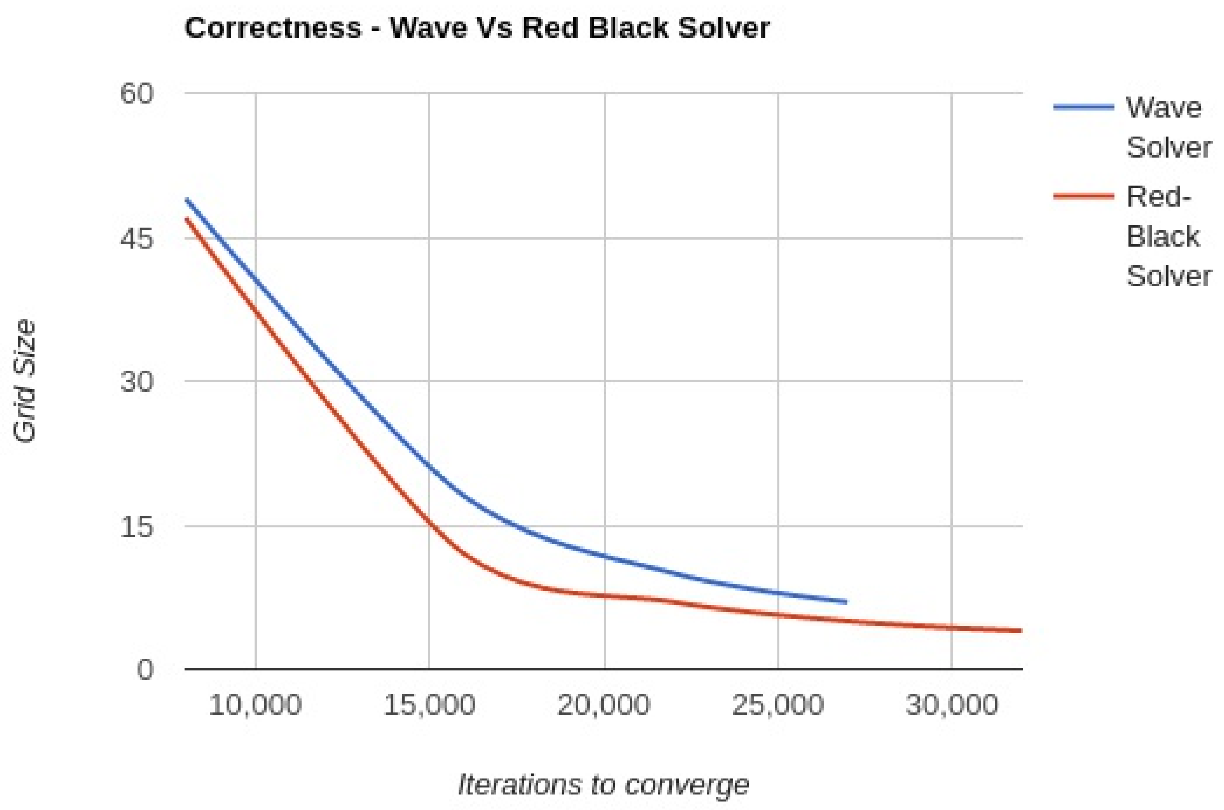
CORRECTNESS(ITERATIONS TO CONVERGE): WAVE VS RED & BLACK

GRID SIZE	Wave	Red andBlack	Difference
8000	49	47	2
16000	18	12	4
22000	10	7	3
27000	7	5	2
32000	5	4	1

Charts


Serial Vs Parallel (16 threads)
Time Taken(secs)
60
45
30
15
0
4,000
8,000
12,000
16,000
20,000
Grid Size
Serial
Wave
Red Black


Strong Scaling Analysis (Grid Size 16000)
Time Taken (secs)
60
45
30
15
0
5.0
7.5
10.0
12.5
15.0
Number of Processors
Wave Solver
Red-Black Solver



Observation

- Red black performs very well for all range of Grid size
- Red black is the fastest
- Wave Solver is completely accurate and the result exactly matches the Serial Implementation
- Approximate solution given by red black solver is acceptable

Running the code

To run the Parallel Wave Solver in ELF:

```
$ g++ -o wave_solver parallel_omp_wave.cpp -fopenmp
$ ./wave_solver
```

To run the Parallel Red and Black Solver in ELF:

```
$ g++ -o rb_solver parallel_rb.cpp -fopenmp
$ ./wave_solver
```