Parallel Programming Assignment 2
Parallel Delta Stepping Algorithm Implementation and Experimentation using MPI

Ayush Raina, 22148

March 24, 2025

Delta Stepping Algorithm

Delta-Stepping is a parallel algorithm for solving the **Single-Source Shortest Path (SSSP)** problem, particularly effective on graphs with both **light** and **heavy** edges. It was introduced by Ulrich Meyer and Peter Sanders to achieve high performance on shared and distributed memory systems by exploiting parallelism in edge relaxation.

Problem Definition

Given a graph $G = (V, E)$, where V represents the set of vertices and E represents the set of weighted edges with non-negative weights, and a source vertex $s \in V$, the objective of the algorithm is to compute the shortest path distance $d(v)$ from the source s to every vertex $v \in V$.

Key Concept: Light and Heavy Edges

Edges are categorized based on a threshold parameter $\Delta > 0$:

- **Light Edges**: Edges with weight $w(e) \leq \Delta$
- **Heavy Edges**: Edges with weight $w(e) > \Delta$

This distinction allows the algorithm to process light edges concurrently while deferring the processing of heavy edges, thereby enhancing parallel efficiency.

Algorithm Workflow

1. **Initialization:**
   - Set the distance to the source vertex s as $d(s) = 0$, and to all other vertices as infinity $d(v) = \infty$ for $v \neq s$.
   - Initialize a collection of buckets $B[i]$, where each bucket stores vertices with distances in the range:
   $$B[i] = \{v \mid i \times \Delta \leq d(v) < (i+1) \times \Delta\}$$
2. **Bucket Processing Loop:**
   - While there are non-empty buckets:
     - Extract and process the **smallest non-empty bucket** $B[i]$.
     - Relax **light edges** of vertices in the bucket and update distances.
     - Move vertices to appropriate future buckets based on updated distances.
3. **Handling Heavy Edges:**
   - After all light edges are processed, relax heavy edges.
   - If the distance of a vertex changes, place it in the appropriate bucket.
4. **Termination:**
   - The algorithm terminates when all buckets are empty, ensuring all shortest paths are computed.

Mathematical Formulation

For a vertex v with current distance $d(v)$, when relaxing an edge (v, u) with weight w:

$$d(u) = \min(d(u), d(v) + w)$$

If $w \leq \Delta$, the edge is considered **light** and processed immediately. Otherwise, it is **heavy** and deferred.

Parallelization Strategy

In a distributed-memory setting using MPI, the graph is partitioned across multiple processes. Each process handles a subset of vertices and their outgoing edges. Communication occurs to share updated distances:

- **Local Processing:** Each process updates its local vertices.
- **Exchange Updates:** Use collective operations (e.g., `MPI_Alltoallv`) to share changes across processes.
- **Synchronization:** Ensure global consistency via reduction operations.

Experimental Setup

All the experiments are done on teaching cluster provided to us. The graph used in experiementation contains 1,971,281 vertices and 5,533,214 edges with randomly assigned weights $w(e) \in [1, 100]$. We are choosing source vertex to be 0. Following is the execution times for 1,2,4,8,16,32,64 processors along with Speedup and Efficiency for different Delta Values

Delta	1 Proc	2 Proc	4 Proc	8 Proc	16 Proc	32 Proc	64 Proc
1	4.503	3.515	3.100	3.026	2.939	3.467	8.079
5	4.536	3.523	3.091	2.825	2.947	3.399	8.108
10	4.530	3.631	3.090	2.908	3.053	3.412	7.903
20	4.507	3.536	3.042	2.892	2.990	3.421	8.117
30	4.493	3.619	3.121	2.942	2.983	3.434	7.930
40	4.500	3.525	3.120	2.938	2.973	3.490	8.353
50	4.504	3.538	3.186	2.965	3.000	3.420	7.992
60	4.507	3.516	3.185	2.928	2.916	3.583	8.651
70	4.498	3.455	3.096	2.877	2.971	3.563	8.428
80	4.536	3.459	3.064	3.042	2.922	4.855	8.156
90	4.505	3.440	3.259	3.032	2.945	3.570	8.059
100	4.511	3.543	3.148	2.955	2.964	3.376	8.425

Table 1: Execution times (in seconds) for different processor counts and Delta values

In the above table, the execution time for involving more than 1 processor is reported for the processor which took the maximum time. Here are some plots to visualize the above data.

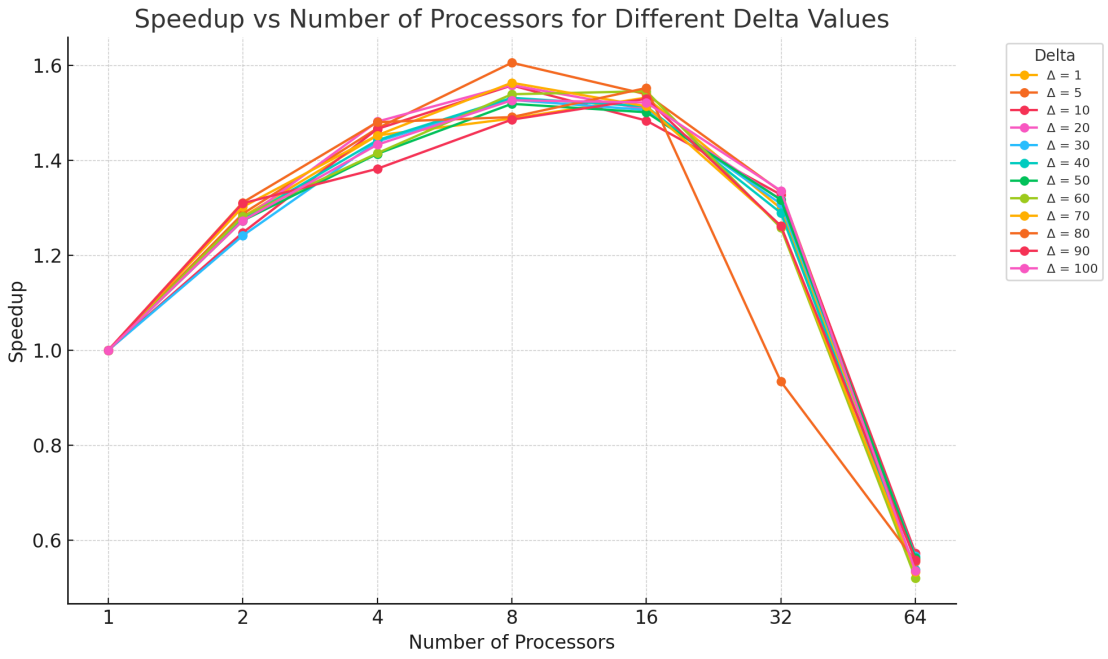

Figure 1: Speedup vs Number of Processors for Different Delta Values

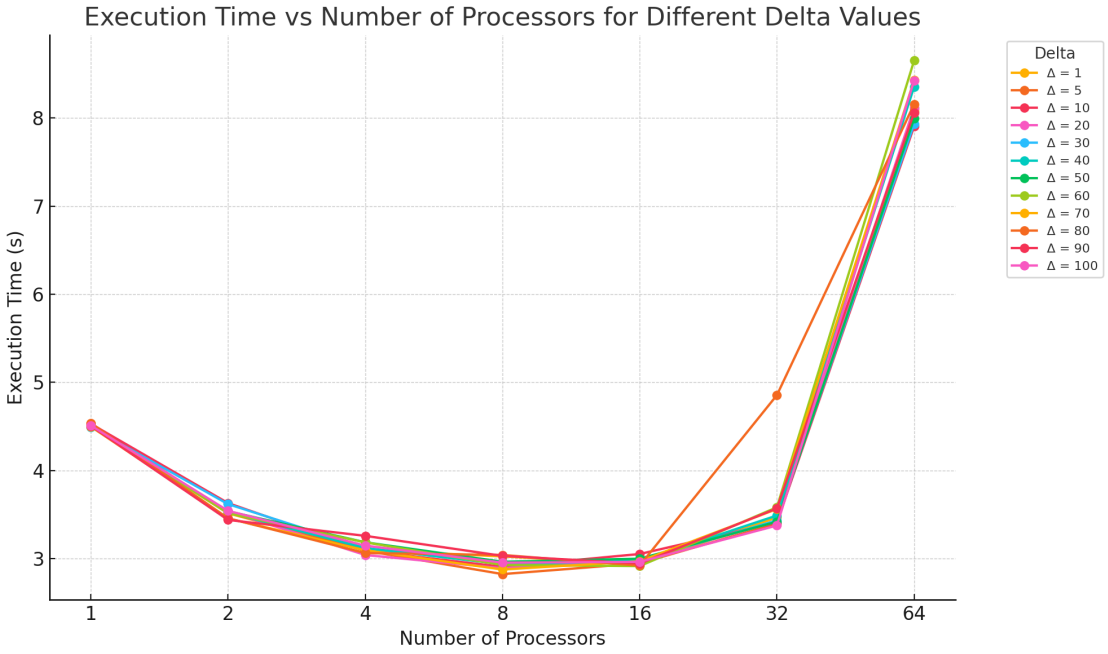

Figure 2: Execution Time vs Number of Processors for Different Delta Values

Parallel Overhead Calculation

We can also calculate Parallel Overhead which quantifies extra work incurred due to parallelization - such as communication, synchronization and load imbalance. It is defined as $PO = T_N \times P - T_1$ where T_N is the execution time for N processors and T_1 is the execution time for 1 processor. Here is the plot for Parallel Overhead for different Delta Values.

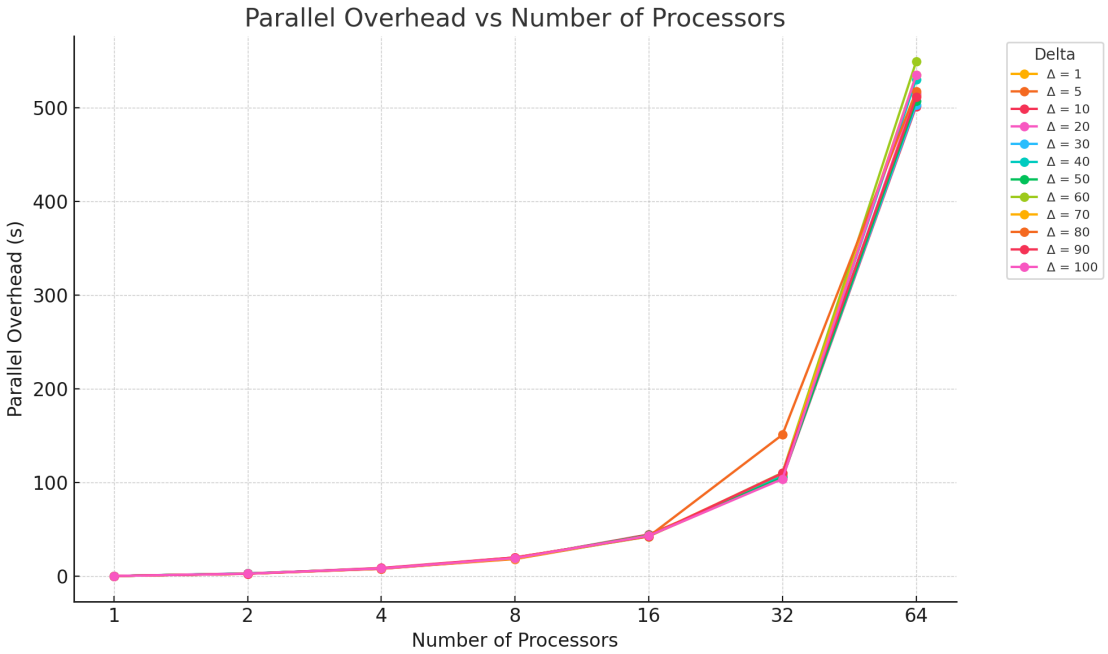

Figure 3: Parallel Overhead vs Number of Processors for Different Delta Values

One more reason, I think for huge increase in overhead when we move to 64 processor from 32 processors is because the machine in which we are running the code has 32 cores. I tried all these experiments with my 24 core machine also in which same pattern followed. Initially execution time kept decreasing until 16 cores, then if I use **–oversubscribe** flag to run with 32 and 64 processors, the execution time increased.

Here is efficiency plot for different Delta Values.

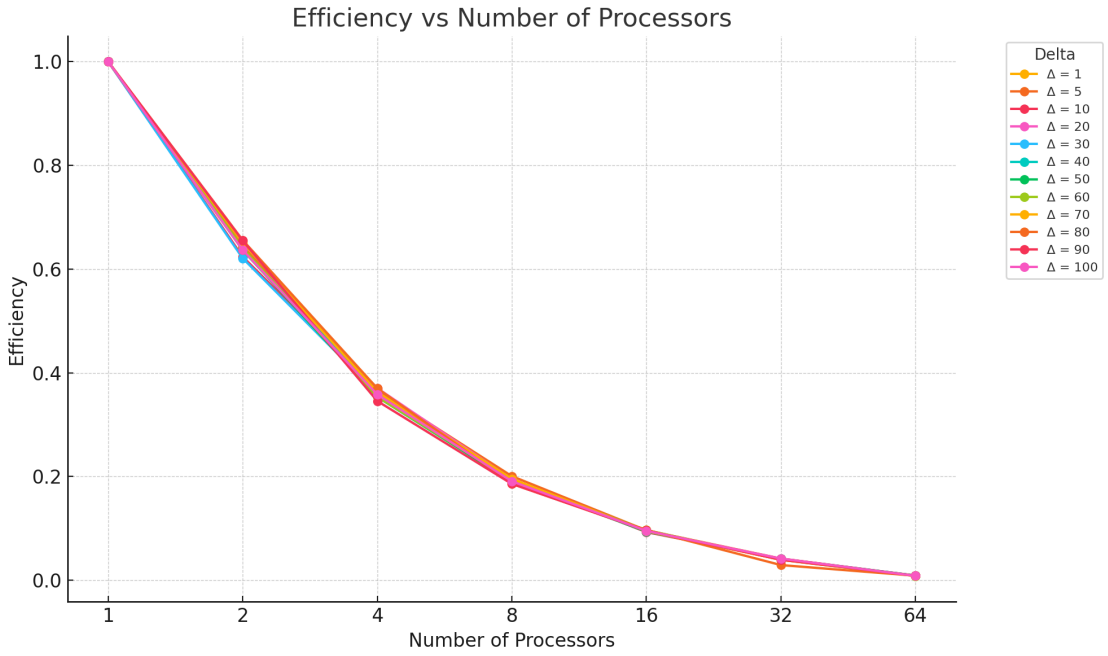

Figure 4: Efficiency vs Number of Processors for Different Delta Values

From above plot we can see that even we are lowering the execution time till 8 processors, but still the resource utilization is not up to the mark. This is because of communication overheads, synchronization overheads as Delta Stepping Algorithm is a bulk synchronization algorithm, and non trivial programming using MPI. My implementation is around 1000 lines of code and I used **modulus** operator to distribute the vertices among processors, due to which all processors have got almost equal number of vertices to process. So **load imbalance** is not an issue, **communication and synchronization overheads** are the main reasons for low efficiency.

Hybrid Implementation Using Open MP and CUDA

We implemented the same Delta Stepping Algorithm using Open MP and CUDA. We used Open MP for parallelizing the bucket processing loop and CUDA for parallelizing the edge relaxation step. We used 8 threads for Open MP and 256 threads per block for CUDA.Here is the execution times for different Delta Values.

Table 2: Execution Times for Different Delta Values (OpenMP + CUDA)

Delta Value	Execution Time (s)
1	6.1021
5	3.4281
10	3.3989
20	1.5955
30	2.4969
40	1.5924
50	1.3124
60	1.3066
70	1.61707
80	1.5882
90	2.2041
100	1.3141

Assuming Sequential Execution Time (T_seq): 4.503 seconds

$$\text{Speedup} = \frac{T_{seq}}{T_{par}} \tag{1}$$

and

$$\text{Efficiency} = \frac{\text{Speedup}}{P} \tag{2}$$

Where T_{par} is the execution time for parallel implementation and P is the number of processors which is 8 in this case.

Here are the plots for speedup, execution times and efficiency for OpenMP + CUDA implementation for different Delta Values.

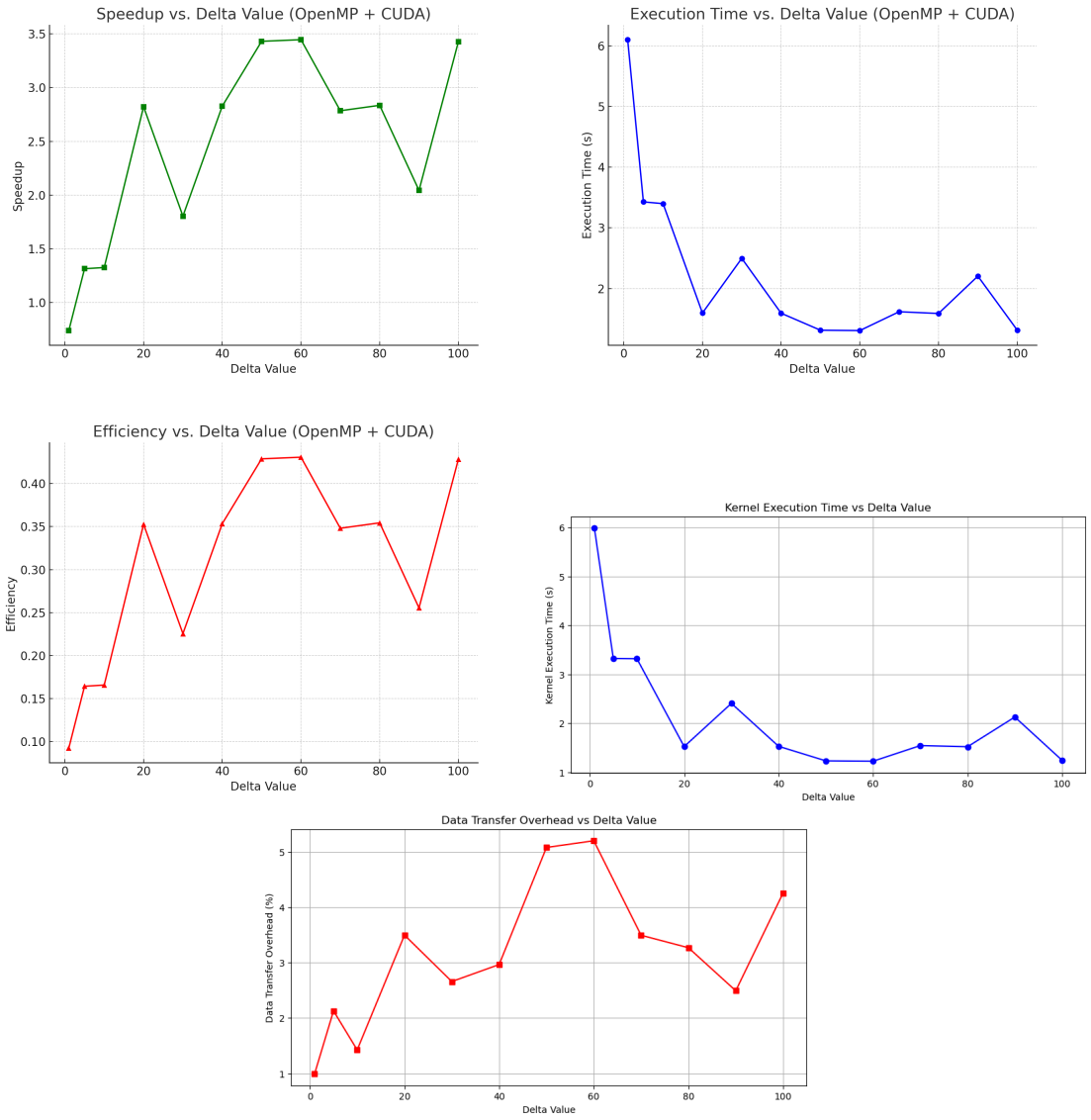

Figure 5: Speedup vs Delta Values for OpenMP + CUDA Implementation