LonestarGPU Benchmarks

Lonestar is a collection of real world applications which shows irregular behavior. Irregular in the sense that it contains irregular data structures like graph, tree, priority queues. Many applications like N-body simulation, social networks, data mining, system modelling, compilers and many contains irregular data structures. These applications are more difficult to parallelise.

From the compiler point of view, control flow and memory access are very much data dependent. We can not statically predict the behavior of program because input data values determine the runtime behavior of the program.

LonestarGPU benchmarks are the implementation of these irregular applications in CUDA.

It contains following list of applications -

- Breadth First Search
- Single Source Shortest Path
- Minimum Spanning Tree
- Barnes Hut N-Body Simulation
- Survey Propagation
- Delaunay Mesh Refinement

1 Breadth First Search (BFS)

1.1 Introduction

Consider a graph $G = (V, E)$ with a set V which contains n vertices and a set E which contains m directed edges. Given a source vertex s, goal is to traverse the nodes of graph G in breadth first order starting at s. Each newly discovered vertex v, it will update its distance d from s and the predecessor vertex p on the shortest path to s. The pair (u, v) indicates a directed edge in the graph from u to v, and the adjacency list $A = \{v|(u, v)\epsilon E\}$ is the set of neighboring vertices incident on vertex u. We will replace each undirected edge by two directed edge containing both (u, v) and (v, u).

This algorithm measures Graph size and traversal rates in terms of number of directed edges presented in the graph. We have represented graph using adjacency matrix. We have used a compressed sparse row (CSR) sparse matrix format to store the graph in memory which contains two arrays R and C. R is an offset array which contains n+1 entries. Each entry i in R represent address of first neighbor of vertex i in C. C is an array which contains collection of adjacency list of neighbors. We have not done any preprocessing on graphs before storing in CSR format.

Algorithm 1 is the standard sequential algorithm for BFS, which uses a queue for circulating the vertices. This algorithm visits each vertices and edges exactly once, so it takes O(m + n) time.

Algorithm 1: A simple serial BFS algorithm for marking vertex distances from source

```
input  : Vertex set $V$, row offset $R$, column indices $C$, source vertex $s$
output: Array $dist[]$ with $dist[i]$ containing distance from $s$ to $i$
1  $Q = \{\}$ ;
2  for $i$ in $V$ do
3      $dist[i] \leftarrow \infty$ ;
4  end
5  $dist[s] \leftarrow 0$ ;
6  $Q$·Push($s$) ;
7  while $Q \neq \{\}$ do
8      $i = Q$·Pop() ;
9      for offset in $R[i] \ldots R[i+1]-1$ do
10         $j = C[$ offset $]$;
11         if $dist[j] == \infty$ then
12             $dist[j] = dist[i] + 1$;
13             $Q$·Push();
14         end
15     end
16 end
```

1.2 Data Structure used

LonestarGPU have used **CSR (Compressed Sparse Row)** format to store graph information and **Worklist** to store the vertices which will be accessed in perticular iteration.

1. **CSR**

   - Compressed Sparse Row (CSR) sparse matrix format is useful to store graph.

   - It contains two arrays R and C. R is an offset array which contains n+1 entries. Each entry i in R represent address of first neighbor of vertex i in C. C is an array which contains collection of adjacency list of neighbors.

2. **Worklist**

   - It handles atomic read-modify-write operations. It helps in coordinating dynamic placement of data into shared data structures and for maintaing status updates.

   - Prefix sum is a suitable approach for data placement. It is a bulk synchronous primitive which can be used to compute scatter address offset for concurrent threads given their dynamic placement requirements.

   - If we have a allocation requirement for each thread then prefix sum can be used to calculate the offset address for that thread to start writing in output array.

   - In Figure 2, prefix sum computes the scatter offset required by each thread to write its vertex to output array. Thread 0 requires 1 space and it's offset is 0 in output array. So

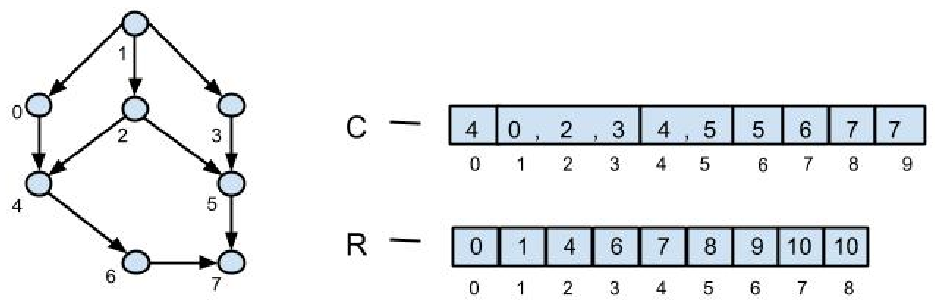

Figure 1: Sparse Graph Example and It's CSR representation

thread 0 will stores it's vertices to 0 location in output array. Similarly for other 3 threads. In the output array input order will be preserved.

- Worklist Contains following functions :
  - **Push** : Atomically push a vertex to worklist to its current index.
  - **Pop** : Atomically returns the current index vertex and decrements index by 1.
  - **display_items** : Will display all the vertices present in worklist.
  - **push_1item** : After performing ProcessEdge, all mark bit information will be returned. To put all valid mark bit vertices in output worklist in parallel, this function performs this operation. This function does prefix sum on mark bit info and computes scatter offset address in output worklist.
  - **Pop_id** : It takes array location as an parameter and returns vertex id. If array location is out of bound or invalid vertex id, it return -1. All of this is done atomically.

(Cub library is used to perform atomic store and load operations on shared data)

1.3 Parallel BFS

In above algorithm, it uses queue so it has to label vertices in increasing order of depth. Before going to the next depth, it explores the current depth vertices. So in parallel implementation of BFS it takes advantage of this, where it uses a level synchronous algorithm. This algorithm preserves the sequential ordering while processing each level in parallel. In this case a race condition may occur when multiple threads concurrently discover the vertex v. But there is no harm because all the threads will update the distance of vertex v from s to the same value, which is valid.

1.3.1 Algorithm Flow

1. Initialize all vertices distance from source vertex to infinity.
2. Push source vertex to Input queue.
3. Check all the neigbhors of vertices from Input Queue. If distance of neighbor is infinity then push that neighbor to Output queue.
4. Copy all the vertices from output queue to input queue.

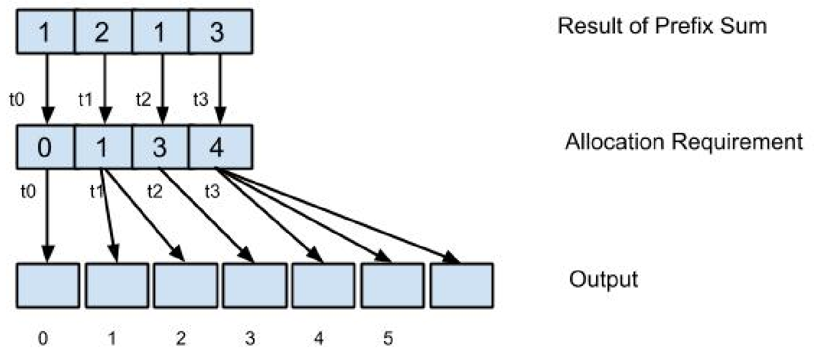

Figure 2: Example of Prefix sum for calculating scatter offsets, Input order is preserved

5. Empty the output queue.
6. Repeat steps 3,4,5 until input queue is empty.

1.4 CUDA Implementation

- LonestarGPU BFS is a implementation of parallel bfs algorithm.
- It is a work efficient parallel algorithm which performs O(m + n) work.
- This implementation uses **Vertex and Edge Frontier** which is collection of vertices and edges in perticular iteration.
- To achieve this, in each iteration it examines only the edges and vertices from that iteration's edge and vertex frontier.
- This implementation prefers to manage vertex frontier out of core, because size of average vertex frontier is small because average out-degree of vertices is small.
- While checking for the distance of neighbor, it uses lock to avoid data races.
- While pushing neighbors to output vertex frontier in parallel, it uses lock to find the correct offset for each block.

1.5 Cilk Plus Implementation

Algorithm is same as used in CUDA implementation. Instead of **Worklist**, we have used **Cilk Plus Reducers** in this implementation. For performing each iteration **cilk_for** is used, which will help in dividing task across the workers. At some places where algorithm is performing similar operations across the array, we have used combination of array notations and vectorization.

Algorithm 2: Cilk Plus parallel BFS constructed using Cilk Plus Reducers

```
input  : Vertex set $V$, row offset $R$, column indices $C$, source vertex $s$, queues
output: Array $dist[]$ with $dist[i]$ containing distance from $s$ to $i$

cilk_for $i$ in $V$ do
    $dist[i] = \infty$ ;
end
reducer_list input, output;
$dist[s] = 0$ ;
$iteration = 0$ ;
$input \cdot$Push($s$);
while $input \neq \{\}$ do
    $output = \{\}$ ;
    cilk_for $i$ in $input$ do
        for offset in $R[i] \ldots R[i+1]-1$ do
            $j = C[$ offset $]$;
            Lock $j$;
            if $dist[j] == \infty$ then
                $dist[j] = iteration + 1$;
                $output \cdot$Push();
            end
            Release $j$;
        end
    end
    $iteration++$ ;
    $input = output$ ;
end
```

1.5.1 Reducers in place of worklist

- Basic motivation behind using **Worklist** is, it handles atomic read-modify-write operations. It helps in dynamic placement of data into shared data structures and maintaining status updates. It also preserves the input order. So **LonestarGPU** implementation uses locks, prefix sum to take care of all the things mentioned above.

- **Cilk plus Reducers** addresses the problem of computing a value by updating a shared variable in parallel program without data races on that variable. Multiple strands can use Reducers safely in parallel. The runtime makes sure that each strand is using a local copy of reducers which removes the possibility of data races. At the time of synchronization reducer instances are merged into single variable.

- In this implementation, we have used **List Reducer** to replace the **Worklist**. Whenever it discovers a new vertex to add in **Ouput Vertex Frontier**, that strand will add that vertex to **List Reducer**. There is no need of explicit prefix sum and lock management. Also in discovery of vertices in **Output Vertex Frontier**, the order of vertices added is not important because in any case it has to examine all vertices in next iteration.

1.5.2 Steps and details of implementation in Cilk Plus (Line numbers are mentioned for each step from Algorithm 2)

1. **Initialization (Line 1-7)**
    - Initialize distances of all the vertices from source vertex to infinity.
    - We have used array notations and vectorization to initialize the distances.
    - Add **source** vertex to **Input Vertex Frontier**.

2. **Neighbor_Gathering (Line 9-20)**
    - Increment **Iteration**.
    - It processes all the vertices from **Input Vertex Frontier** and gathers their neighbor in parallel.
    - It uses **Cilk_for** to divide the tasks in multiple workers.
    - In **Cilk_for** loop, each iteration will have the local variables to store the information of neighbors gathered.
    - Create a **Cilk Plus Reducer List** to store the valid neighbors for the next iteration.
    - Pass all the neighbors to (**Process_Edge**).

3. **Process_Edge (Line 12-18)**
    - It validates the neighbor passed. It will check whether the neighbor should be included in next iteration or not.

- We need to use lock here because multiple strands may discover the same vertex as valid vertex and update it's distance. As explained earlier, data race does not affect the correctness of distance but it may push same vertex to **Cilk Plus List Reducer** multiple times. To avoid duplication, we need to use lock while performing the validation.

- Multiple strands will process neighbors collected in **Neighbor_Gathering** process in parallel.

- Each strand will **lock** the neighbor :

  - **If** distance from source is infinity then push it to the **Cilk Plus Reducer List**. Set it's distance from the source equal to iteration number.

  - **Else** Do not process that neighbor.

- Release the **lock**.

4. **Swap_Frontiers (Line 22)**

   - Copy all the vertices from **Cilk Plus Reducer List** to **Input Vertex Frontier**.

   - Make **Cilk Plus Reducer List** empty.

5. **Repeat 2,3,4 until Input Vertex Frontier is empty. (Line 8)**

2 Single Source Shortest Path (SSSP)

2.1 Introduction

This benchmark computes the shortest path from a source node to all nodes in a directed graph with non-negative weights. It is based on the principle of relaxation. It replaces optimization to the correct distance gradually by more accurate distance until it reaches to the optimum point.

Consider a graph $G = (V, E)$ with a set V containing n vertices and a set E containing m directed edges. The pair (u, v) indicates a directed edge in the graph from u to v, and the adjacency list $A = \{v|(u, v)\epsilon E\}$ is the set of neighboring vertices incident on vertex u. We have replaced each undirected edge by two directed edge containing both (u, v) and (v, u). For all edged (u, v), compare it's old distance from source with the new distance computed via u and update the distance of v with the minimum value. We need to do this relaxation $|V| - 1$ times. So this algorithm takes $O(|V| * |E|)$. Algorithm 3 describes the overall workflow.

2.2 Parallel SSSP

This is modified version of **Bellman Ford Algorithm**, which is similar to the BFS algorithm. The only difference is in validation checking of edge (u, v), where we do not need to check it for all the edges in each iteration.

Modification in Bellman Ford algorithm makes it similar to BFS algorithm. In Bellman Ford algorithm, it relaxes all the edges in graph in each iteration for $|v| - 1$ iterations. Each time it checks the

Algorithm 3: A serial version of Bellman Ford Algorithm for graph having non-negative Edges

```
input  : Vertex set $V$, Edge set $E$
output : Array $dist[]$ with $dist[i]$ containing shortest distance from $s$ to $i$, Array
         $predecessor[]$ with $predecessor[i]$ containing parent vertex of $i$ in shortes path form
         $s$ to $i$
1  for i in v do
2      $dist[v] = \infty$;
3      $predecessor[v] = null$ ;
4  end
5  $dist[s] = 0$;
6  for i from 1 to $|v| - 1$ do
7      for each edge (u, v) do
8          if $dist[u] + wt(u, v) < dist[v]$ then
9              $dist[v] = dist[u] + wt(u, v)$;
10             $predecessor[v] = u$;
11         end
12     end
13 end
```

current distance with alternative distance and then update the distance array, but there is no need to relax all the edges in each iteration. If vertex v has a distance value that has not ch anged since last time the edges out of v were relaxed, then there is no need to relax the edges out of v second time. So it pushes only those vertices in output vertex frontier whose distance has been updated, which makes it similar to **Parallel BFS**. This modification helps in parallelization.

2.2.1 Algorithm Flow

1. Initialize all vertices distance from source vertex to **infinity**.
2. Push source vertex to **input queue**.
3. Check all the neigbhors of vertices from **input Queue**. Let for edge **(u,v)**, **u** is vertex from input queue and **v** is it's neighbor.
   - **If** old distance of v is greater than it's distance via u then push that neighbor to **output queue**.
   - **Else** Do not push that neighbor.
4. Copy all the vertices from **output queue** to **input queue**.
5. Empty the **output queue**.
6. Repeat steps 3,4,5 until input queue is empty.

2.3 CUDA Implementation

- LonestarGPU has implemented this algorithm in CUDA.
- This implementation uses **Vertex and Edge Frontier** which is collection of vertices and edges in perticular iteration.
- It examines only the edges and vertices from that iteration's Edge and Vertex Frontier in each iteration.

- This implementation prefers to manage vertex frontier out of core, because size of average vertex frontier is small by factor d (average out-degree).
- While checking for the distance of neighbor, it uses lock to avoid data races.
- While pushing neighbors to output vertex frontier in parallel, it uses lock to find the correct offset for each block.

Algorithm 4: A Cilk Plus implementation of modified Bellman Ford Algorithm

```
input  : Vertex set $V$, row offset $R$, column indices $C$, source vertex $s$, queues
output: Array $dist[]$ with $dist[i]$ containing shortest distance from $s$ to $i$
1  cilk_for $i$ in $V$ do
2      $dist[i] = \infty$ ;
3  end
4  reducer_list $input, output$ ;
5  $dist[s] = 0$ ;
6  $iteration = 0$ ;
7  $input = \{\}$;
8  $input \cdot$Push($s$);
9  while $input \neq \{\}$ do
10     $output = \{\}$ ;
11     cilk_for $i$ in $input$ do
12         for offset in $R[i] \ldots R[i+1]-1$ do
13             $j = C[$ offset $]$;
14             Lock $j$;
15             if $dist[j] + wt(i,j) < dist[i]$ then
16                 $dist[j] = dist[i] + wt(i,j)$;
17                 $outputQ \cdot$Push();
18             end
19             Release $j$;
20         end
21     end
22     $iteration++$ ;
23     $input = output$ ;
24 end
```

(CSR and Worklist Data Structures are used for storing graph and Vertex Frontier respectively)

2.4 Cilk Plus Implementation

This algorithm is similar to CUDA implementation. Instead of **Worklist**, we have used Cilk Plus **Reducers** in this implementation. For performing each iteration in parallel we have used **cilk_for**, which helps in dividing task across the workers. At some places where it is performing similar operations across the array, we have used combination of array notations and vectorization.

2.4.1 Reducers in place of Worklist

- Basic motivation behind using **Worklist** is, it handles atomic read-modify-write operations. It helps in dynamic placement of data into shared data structures and maintaining status updates. It also preserves the input order. So **LonestarGPU** implementation uses locks, prefix sum to take care of all the things mentioned above.

- **Cilk plus Reducers** addresses the problem of computing a value by updating a shared variable in parallel program without data races on that variable. Multiple strands can use Reducers safely in parallel. The runtime makes sure that each strand is using a local copy of reducers which removes the possibility of data races. At the time of synchronization reducer instances are merged into single variable.

- In this implementation, we have used **List Reducer** to replace the **Worklist**. Whenever it discovers a new vertex to add in **Ouput Vertex Frontier**, that strand will add that vertex to **List Reducer**. There is no need of explicit prefix sum and lock management. Also in discovery of vertices in **Output Vertex Frontier**, the order of vertices added is not important because in any case it has to examine all vertices in next iteration.

2.4.2 Steps and details of implementation in Cilk Plus (Line numbers are mentioned for each step from Algorithm 4)

1. **Initialization (Line 1-8)**
    - Initialize distances of all the vertices from source vertex to infinity in **Dist** array.
    - We have used array notations and vectorization to initialize the distances in parallel.
    - Add source vertex to **Input Vertex Frontier**.

2. **Neighbor_Gathering (Line 10-22)**
    - Increment **Iteration**.
    - Process all the vertices from **Input Vertex Frontier** and gather their neighbors in parallel.
    - We have used **Cilk_for** to divide the tasks across multiple workers.
    - In **Cilk_for** loop, each iteration will have the local variables to store the information of neighbors gathered.
    - Create a **Reducer List** to store the valid neighbors for the next iteration.
    - Pass all the neighbors to (**Process_Edge**).

3. **Process_Edge (Line 13-19)**
    - It checks whether neighbor vertex should be included in next iteration or not.
    - We have used **Lock** here because multiple strands may discover the same vertex as valid vertex and update its distance. As explained earlier, data race does not affect the correctness of distance but it may push same vertex to **Cilk Plus List Reducer** multiple times. To avoid duplication, we need to use lock while performing the validation.

- Multiple strands process edge (u, v) in parallel. Where u is vertex from **Input Vertex Frontier** and v is neighbor.
- Each strand **Lock** the neighbor vertex v
  - Compare its old distance from **Dist** array with alternative distance via it's u.
  - **If** Alternative distance is minimum, it will push the neighbor v to **Reducer List** and set its distance in **Dist** array to alternative distance.
  - **Else** Do not process the neighbor v.
- Release the **Lock**.

4. **Swap_Frontiers (Line 23)**
   - Copy all the vertices from **Cilk Plus Reducer List** to **Input Vertex Frontier**.
   - Make **Cilk Plus Reducer List** empty.
5. **Repeat 2,3,4 until Input Vertex Frontier is empty. (Line 9)**

3 Minimum Spanning Tree (MST)

3.1 Introduction

This benchmark uses **Boruvka's algorithm** to compute the minimum spanning tree in a graph with edges having distinct weights. This algorithm begins with each vertex as a separate component. It finds a edge with minimum weight which connects two different components. After discovery of minimum weight edge, this algorithm merges those two components into one and reduces number of components bye one. It repeats the process until number of components remain same. In the end, the set of edges discovered is a minimum spanning tree.

3.2 Data Structures Used

Both the implementation uses **CSR (Compressed Sparse Row)** format to store graph information and **ComponentSpace** to store the component information.

- **ComponentSpace**
  - ComponentSpace uses disjoin set data structure in it.
  - Array is used to represent a set, where array value represents representative of that set.
  - ComponentSpace uses **Union-Find** algorithm with **Path Compression**. It has two main operations **Union** and **Find**.
    - **Find** - It takes element as an input and determines which set contains that element. It returns the representative of that set.
    - **Union** - It joins two subsets into one. It replaces the representative of smaller set by representative of bigger set.

- **Path Compression** is an improvement in this algorithm which reduces time required for **Find** operation from O(n) to O(log n).

- **ComponentSpace** stores following information -
  - Number of elements
  - Number of components
  - Array to store representative of each element (Which set it belongs to)
  - Array to store whether a element is representative or not

- As it unifies multiple pair of components in parallel, there is chance that there is common component in more than one pair. So we need **Locks** for the synchronization.

3.3 Parallel MST

This algorithm repeatedly relaxes the minimum weight edges. If we explicitly relaxe the edges, it will modify the graph. To avoid this, this algorithm keeps track of number of components that have been merged. After each iteration number of components reduces and size of component increases. If given graph is connected then this algorithm will terminate when number of components is one. Otherwise it will terminate when number of components remains unchanged after an iteration.

Algorithm flow :

- Find minimum weight edge going out of each vertex.

- Find minimum weight edge going out of each component (let's call it Representative) and it's partner to merge.

- Verify that representative and partner is valid.

- Merge verified representative with it's partner component.

- Repeat above steps until number of components remain unchanged.

3.4 CUDA Implementation

LonestarGPU has implemented this algorithm in CUDA on GPU. This implementation uses **ComponentSpace** data structure to manage the operations on components. It performs each step mentioned in **Algorithm flow** in parallel by dividing work across maximum block available on GPU. Every single thread available across the GPU SMs accesses a single element and processes it in parallel. For the synchronization, it uses atomic operations. This version has implemented each step from **Algorithm Flow** as separate global kernel.

3.5 Cilk Plus Implementation

We have used the same algorithm as CUDA implementation. We have used **Cilk_for** to divide work across multiple strands. We have used **Atomic** operations for the synchronization. To update the weight of minimum spanning tree and number of components after each iteration, we have used **Cilk Plus reducer_opadd**. To get the best result, we have tested it using different grain size for all **Cilk_for** loops used in this implementaion. In initialization of data, we have used array notations and vectorization. We have used **Reducer List** to store the representative of components so instead of verifying all nodes, it verifies only representative vertices.

Algorithm 5: A Cilk Plus implementation of Boruvka's algorithm

```
input  : Vertex set $V$, row offset $R$, column indices $C$
output: $mst$ containing minimum spaning tree cost
1  $Num\_Comp = |V|$;
2  $Old\_Comp = 0$;
3  while $Old\_Comp \neq Num\_Comp$ do
4      $Old\_Comp = Num\_Comp$;
5      cilk_for $i$ in $V$ do
6          $Elem\_wt[i] = Comp\_wt[i] = \infty$;
7          $Partner[i] = Comp\_to\_elem[i] = i$;
8          $Process\_in\_next\_itr = false$;
9      end
10     cilk_for $i$ in $V$ do
11         $src\_comp = Comp\_to\_elem[i]$;
12         for All edges $(i, v)$ going out of $i$ do
13             $Elem\_wt[i]$ = min(Old value, weight of $(i, v)$);
14             $Comp\_wt[src\_comp]$ = min(Old value, weight of $(i, v)$);
15             if Old value > weight of (i,v) then
16                 $dst\_comp = Comp\_to\_elem[v]$;
17                 $partner[i] = dst\_comp$;
18             end
19         end
20     end
21     cilk_for $i$ in $V$ do
22         $src\_comp = Comp\_to\_elem[i]$;
23         if $Elem\_w[i] == Comp\_wt[src\_comp]$ then
24             Mark this node as representetive;
25         end
26     end
27     cilk_for All Representative Nodes do
28         if Representative and Partner values are correct then
29             $Process\_in\_next\_itr[i] = True$;
30         end
31     end
32     cilk_for $i$ in $Process\_in\_next\_itr$ do
33         Lock src and dst components;
34         Merge smaller component into larger component;
35         $num\_Comp - -$;
36         Release lock;
37     end
38 end
```

3.5.1 Prerequisite

This implementaion uses following arrays :

- **Elem_wt** - Stores minimum weight edge going out of each vertex.
- **Comp_wt** - Stores minimum weight edge going out of each component.
- **Partner** - Stores a valid component for merging.
- **Comp_to_elem** - Stores a component representative of each vertex.
- **Process_in_next_itr** - Stores whether a representative is valid for merging or not.

3.5.2 Steps and details of Cilk Plus implementation (Line numbers are mentioned for each step from Algorithm 5)

1. **Initialization : Line(5-9)**
    - Initialize following arrays with the use of Array Notations and Vectorization :
        - **Elem_wt** : Initialize each value to Infinity.
        - **Comp_wt** : Initialize each valeue to Infinity.
        - **Partner** : Initialize each value to itself.
        - **Comp_to_elem** : Initialize each value to itself.
        - **Process_in_next_itr** : Initialize each value to false.
    - Initialize **mst_weight** to 0.
    - Initialize **Num_of_components** to number of vertices.
2. **Find_MinWt_Comp : Line(10-20)**
    - Divide total number of vertices across multiple workers using **Cilk_for** loop.
    - For each vertex, find a minimum weight edge (u,v) going out of that vertex which connects to vertex from other component and store it in **Elem_wt** array.
    - If (u,v) is minimum for the component it belongs to then update **Comp_wt** array value for that component by weight of (u,v).
    - Also update **Partner** array with the component of v.
3. **Find_Representative : Line(21-26)**
    - Divide total number of vertices across multiple workers using **Clik_for** loop.
    - For each vertex, check if it has a minimum weight edge going out of that component and connects to other component. If it is then mark it as a representative of that component.
    - This algorithm uses this representative at the time of merging two components.
4. **Verify_Representative : Line(27-31)**
    - Divide all the **representative** vertices across multiple workers using **Cilk_for** loop.

    - Verify that **Partner** array and **representative** marked are valid values.

    - If verified values are correct then update **Process_in_next_itr** array value for the **representaive** vertex to **True** for merging two components.

5. **Unify_Components : Line(32-37)**

    - For each **Process_in_next_itr** True values, merge it with **Partner** component.

    - At the time of merge, it merges smaller component with the larger one.

    - Reduce number of components by 1 and increase **mst** value by **Comp_wt** value.

6. **Repeat steps 1,2,3,4,5 until number of components remain unchanged. : Line(3)**

4 Survey Propagation (SP)

4.1 Introduction

Survey Propagation is a heuristic SAT solver based on Bayesian inference. This algorithm uses a bipartite graph which contains clauses on one side and variables on other side. Edges have either +1 if varialbe in the clause has positive value otherwise -1 value. If Variable is not present in clause then there is no edge between that clause and variable. This algorithm iteratively updates each variable.

4.2 Data Structures Used

1. CSRGraph
    - It uses Compressed Sparse Row (CSR) format to store information about clauses and variables separately.
    - It also stores information about bias values.
    - If we remove variables from graph then we have to modify the graph. So to make it easy, each variable has **marked bit** information in this data structure.

2. Edge
    - This data structure stores clause, variable and Pi values of each edge using arrays.

4.3 Parallel SP

This algorithm sets a **Epsilon** value. In each round of this algorithm, first it processes all the variables and clauses and updates the **Surveys** until all update values are below the **Epsilon** value. Then it processes all the calculated **Surveys**, find most biased variables and fix them. It then removes **fixed** variables from graph. After each round, if there are only trivial **Surveys** remaining then pass the graph to simple solver else proceed to next round.

There are two termination conditions for the algorithm. If one of them is true, algorithm would terminate. The two termination conditions are :

- When number of variables remaining are very small, pass the graph to simple solver.
- If algorithm shows no progress after some fix number of iterations then it will give up.

4.3.1 Algorithm Flow

1. Get the data from input file and build a bipartite graph.
2. Update the **Surveys** for all the edges
3. **If** updated values are above **Epsilon** then repeat step 2.
4. For each variable, if **bias** is more then threshold bias then fix that variable and remove it from graph.
5. Repeat algorithm until termination condition is true.

4.4 CUDA Implementation

- **LonestarGPU** has implemented this algorithm in CUDA on GPU. In the beginning of the implementation, they copy all the clause, variable and edge data to GPU.
- To implement the step 2 from **Algorithm Flow**, they have divided all the edges across all the allocated threads.
- While fixing the variables, it adds variables having bias value above some fixed value to bias list. So it uses atomic operation to calculate the index in bias array.
- After updating bias values, they sort the bias values for fast processing in next iteration. As it stores values of variables and their bias in separate arrays, they have used **CUDA Cub Dual buffer** to store and **Cub DeviceRadixSort** to sort.
- Where sum of all thread information is required, they have used **Cub Reduce**.

4.5 Cilk Plus Implementation

We have used same algorithm as CUDA implementation. To divide the work , like calculating pi values and updating bias, in parallel across the workers we have used **Cilk_for** loop. To calculate maximum value across the array, we have used **Cilk Plus reduces_max**. To make a reduce operation on array, we have used **Cilk Plus reducer_opadd**. While fixing the variable, we add variables having bias value above some fixed value in bias array. So to avoid **locks**, we have used **Cilk plus reducers list**.

4.5.1 Prerequisite

This implementation uses following data :

- **clauses** - It is a **CSRGraph** data structure. It stores information about the clauses present in graph.
- **vars** - It is a **CSRGraph** data structures. It stores information about variables present in graph.
- **edges** - It is a **Edge** data structure. It stores information about all edges connecting variables and clauses.
- **bias_list** - It is an array which contains list of variables which will get fixed in that round.
- **EPSILON** - It a fixed epsilon value.

Algorithm 6: A Cilk Plus implementation of Survey Propagation Algorithm

```
input  : CSRGraph clauses,vars and Edge edges
output: Modified Graph
1  Store graph information in clauses, vars and edges;
2  round = 0;
3  while Converge() do
4      Sort list of biased values of variables;
5      cilk_for All variables do
6          if it is a biased variables then
7              Fix it;
8          end
9      end
10 end
11 Converge() :
12 max_eps = 0;
13 while max_eps > EPSILON do
14     cilk_for All non-fixed variables v do
15         for All clauses C of V do
16             Calculate pi values;
17             Update Bias Values;
18             Update max_eps;
19         end
20     end
21 end
```

4.5.2 Steps and details of Cilk Plus Implementation (Line numbers are mentioned for each step from Algorithm 6)

1. **Initialization : Line(1-2)**
   - Read the input graph file.
   - Store the information in **clauses, vars** and **edges**.
   - It does not fix any variable initialy.
   - Initialize **round** to 0.

2. **Converge : Line(12-21)**
   - Calculate pi values of all the edges of the graph in parallel.
   - Update the bias values of all the variables in the graph in parallel.
   - It uses **cilk_for** loop to perform above to operations in parallel. It also uses **reducer_max** to calculate maximum bias value to compare with **EPSILON**.
   - It repeats this step until it gets maximum bias value below the **EPSILON** or it repeats this step some fixed number of iterations.

3. **Build_List : Line(4)**
   - Sort the bias list of variables to get the most biased variables in less time.
   - We have sorted the pair of variables array and corresponding bias values array using **cilkpub** library. This library sorts the given array in parallel.

4. **Decimate : Line(5-9)**
   - Get the most biased variables and mark them as fixed variables.
   - We have used **cilk_for** loop to mark fixed variables in parallel.

5. **Repeat steps 2,3,4 until it can not Converge modified graph. : Line(3)**

5 Barnes Hut N-Body Simulation

5.1 Introduction

This algortihm simulates the gravitational force acting on galactic cluster. It initializes velocities and positions of the galaxies. This algorithm calculates forces acting on each galaxies and calculates the motion of the galaxies for a number of time steps.

5.2 Parallel BH

Parallel BH uses J. Barnes and P. Hut.A hierarchical O(NlogN) force calculation algorithm. Parallelism comes from independent force calculation of each body. This algorithm uses an **Octree** data structure, which is a hierarchical tree data structure. This data structure is useful to calculate the force that n bodies in the system induce upon each other. To calculate force for n bodies, it takes n^2 operations to calculate. The Barnes Hut algorithm hierarchically partitions the total volume into smaller cells. In an Octree, each partitioned cell will represent a intermediate node. It will contain combined mass and centre of gravity of all the bodies from the cell it belongs to. All the leaf node are individual bodies present in the system.

While calculating force, if body is so far then it calculates force with that cell instead of calculating it with all the bodies from that cell. So this algorithm reduces number of operations from n^2 to nlogn. It centre of mass is not sufficiently far away then it looks into the all subcells and performs the calculations.

5.2.1 Algorithm Flow

1. Read input file.
2. Make an Octree.
3. For each body in the system, insert it in Octree.
4. Calculate centre of gravity and combined mass of each cell.
5. For each body in the system, calculate force.
6. According to calculated force, cahnge position and velocities of the bodies.
7. Repeat 2,3,4,5,6 for number of Time steps mentioned.

5.3 CUDA Implementation

- **LonstarGPU** has implemented this algorithm in CUDA on GPU.
- They have done warp based execution. It makes sure the all the threads belonging to that warp are executing same instruction. In force calculation kernel they have used warp based execution.
- To improve the executiion, this algorithm sort the bodies according to their distance.
- They have implemented **Wait-free pre-pass**, where a thread which has discoverd the child does not wait for the other threads and start processing the children.
- For synchronisation, they have used **__syncthread** and **__threadfence**.

5.4 Cilk Plus Implementation

We have implemented same algorithm as CUDA implementation. We have used **cilk plus max/min reducers** to calculate the square boundary of the area of all bodies present. To build a octree, all bodies are inserted into the octree in parallel. When more then one body is accessing same intermediate node then we have used a **lock** there to avoid the data race. Otherwise it will insert multiple bodies in parallel. We have used **cilk_for** to divide this work in parallel.

5.4.1 Prerequisite

This implementation uses following data :

- **n** - Number of bodies
- **pos_x, pos_y, pos_z** - These arrays store x,y,z coordinates of bodies present in the system.
- **vel_x, vel_y, vel_z** - These arrays store velocities of bodies in x,y,z direction.
- **mass** - This array stores the mass of all the bodies present in the system.
- **tree** - This is an array of size 8 * (n + 1). It stores the octree data structure. First n to 2*n locations are reserved to prepresent intemediate node. In octree, if place is empty then it stores -1. If place contains leaf node then it store body. If it is an intermediate node then it contains intermediate node number. Last 8 locations in array represent 8 children of root.

Algorithm 7: A Cilk Plus implementation of Barnes Hut N-Body Simulation

```
input  : mass, velocity and position arrays
output: Changed values after given number of iterations
Generate random input data;
Initialize mass,vel and pos arrays;
Initialize last 8 locations of tree array with -1 ;
for Number of timesteps do
    reducer_max max_x, max_y, max_z;
    reducer_min min_x, min_y, min_z;
    cilk_for All values in pos arrays do
        calculate max_x, max_y, max_z from pos_x, pos_y, pos_z;
        calculate min_x, min_y, min_z from pos_x, pos_y, pos_z;
    end
    $radius = max(max\_x - min\_x, max\_y - min\_y, max\_z - min\_z)$;
    cilk_for Each body do
        Compute child of root where a body should be inserted;
        if child is locked then
            Try again for this body later;
        else
            if child contains -1 then
                Insert body;
            else
                Lock child;
                Insert old and new body to next level;
                Release child;
            end
        end
    end
    cilk_for All intermediate nodes in tree do
        if All children of node are ready then
            Compute centre of mass and centre of gravity;
        else
            Wait for all children to get ready;
        end
    end
    cilk_for All leaf nodes in tree do
        Calculate offset address in sort array;
        Place it to its offset address;
    end
    cilk_for All the bodies do
        for All tree nodes do
            if Node is too far from body then
                Compute force using that cell's information;
            else
                Iterate over the individual child of that node to compute the force;
            end
        end
    end
    cilk_for For all Bodies do
        Update pos and vel arrays;
    end
end
```

5.4.2 Steps and details of Cilk Plus Implementation (Line numbers are mentioned for each step from Algorithm 7)

1. **Initialization : Line(1-3)**
   - Generate a random input data.
   - Initialize **mass, vel** and **pos** arrays.
   - Initialize last 8 locations of **tree** array with -1.
2. **Build_Box : Line(5-11)**
   - Calculate **min** and **max** values from all **pos** arrays.
   - Calculate the area of the box.
   - It calculate **max** and **min** values using **max/min reducers**.
3. **Build_Tree : Line(12-25)**
   - This function builds octree for give set of bodies.
   - For each body, it computes the child of a root it should be inserted in. Then it starts traversing from the root.
     - **If** child is **locked** then try again later.
     - **Else**
       - **If** it contains -1, then insert a body at that location.
       - **Else** : It means that there is another body present at that location. So **Lock** it and insert old body and new body to next level of tree. While inserting to the next level, it checks all above conditions again.
   - We have used **cilk_for** to insert bodies in parallel.
   - We have used **atomic locks** to avoid the data races.
4. **Summarize : Line(26-32)**
   - Compute centre of gravity and centre of mass of all intermediate cells.
   - It starts computing from leftmost intermediate node in the **tree** array because leftmost **intermediate** node will have no intermediate node in it. So it can calculate it's centre of gravity and centre of mass.
   - In case, one of the children is not ready then it will wait until it becomes ready.
5. **Sort : Line(33-36)**
   - Sorts body according to **inorder traversal** of an octree.
   - It calculates offset address in **sort** array for all **bodies** present in octree.
   - Place all bodies to their offset address to make them spatially close.
   - As we are calculating offset address, we have done it in parallel using **cilk_for**.
6. **Compute_Force : Line(37-45)**
   - Calculates force acting on each body.
   - If body is far away then it coputes force acting on it using cell information. Then there is no need to calculate force from other bodies of that cell.
   - If body is not far then it calculates force acting on it.
   - We have used **cilk_for** to divide these calculations in parallel.
   - **Sort** function makes this function faster because bodies are spatially close.

7. **Advance : Line(46-48)**
    - Updates the **pos** and **vel** arrays using calculated force.
    - This is purely parallel function where we have used vectorization with the **cilk_for** loop.
8. **Repeat 2,3,4,5,6 for given number of timesteps. : Line(4)**

6 Delaunay Mesh Refinement (DMR)

6.1 Introduction

This algorithm produces a qaulity Delaunay mesh. Qaulity Delaynay mesh is a Delauney trangulation with additional constraint that no angle of any triangle in a mesh should be less than 30 degrees. This algorithm takes a undefined mesh as an input and produces a mesh satisfying the above constraint. This algorithm fixes badly shaped triangles successively.

6.2 Parallel DMR

This algorithm performs refinement on 2D Delaunay mesh. 2D Delaunay mesh is a set of points having the property that circumcircle of any trianlge in a mesh should not contain any other point from that mesh. If there exist other point in a circumcircle then thet triangle will be marked as bad triangle. This algorithm takes set of all bad triangle and re-triangulate them. This process may affect other good neighborhood triangles, called cavity. So this algorithm iteratively fix this cavity until all the triangles present in the mesh satisfy the given constraint. Parallelism arrise from the set of bad trianlge, which will be re-triangulated.

6.2.1 Algorithm Flow

1. Check whether a bad triangle is present or not.
2. Find the affected triangles in the neighborhood by that bad triangle.
3. Re-triangulate all the triangles in the cavity.
4. Find new bad triangles and process them until there is no bad triangle in the mesh.

6.3 CUDA Implementation

- **LonestarGPU** has implemented this algorithm in CUDA on GPU.
- They have used **Worklist** data structure (Explained in BFS) to store the bad triangles of each iteration.
- Instead of using locks, they have used barrier based exclusive ownership detection which helps in avoiding conflicts. For this they have used **CUDA cub library**.
- They have distributed bad triangles to the threads to avoid the overlapping of cavities.

6.4 Cilk Plus Implementation

We have implemented this algorithm same as CUDA implementation. We have used **Reducers list** instead of **Worklist** to maintain the bad triangle of each iteration. **Reducers list** helps in managing data races. Also to divide all the bad triangles across workers, we have used **cilk_for**. When multiple workers try to access the same cavity, we have used **locks** to avoid the conflict.

6.4.1 Prerequisite

It uses following data structures :

- **Mesh** - This data structure contains information about number of elements, their (x,y) coordinates, whether element is bad or not. It stores all the information in separate arrays.

6.4.2 Steps and details of Cilk Plus implementation

1. **Initialization**
    - Read input file and stores the mesh information in **Mesh** data structure.
    - Set number of bad triangles to 0.
    - Create a **Input Reducer List** to store bad triangles.
2. **Check_Triangles**
    - Check all the triangles from **Mesh** for the given constraint.
    - **If** a triangle is bad then push it to the **Input Reducer list**.
    - Calculate total number of bad triangles.
3. **Refine**
    - It takes **Reducer List** as an input which contains all bad triangles.
    - For each bad triangle, Retriangulate bad triangle and build a cavity.
    - Push all newly created bad triangles to **Output Reducer List**. If multiple workers try to accesss same cavity then it **lock** that cavit until it processes all the bad triangle from that cavity. This happens when there exists overlapping cavities.
4. **Swap_Lists**
    - Swap the content of **Input Reducer List** with the contet of **Output Reducer List**.
    - Empty the **Output Reducer List**
5. **Repeat 2,3,4 until Input Reducer List is empty.**