A Parallel Approach to union-find algorithm implementation in finding connected partitions.

(November 2016)

Abhi Shah, Keval D. Shah, Parshwa Shah;
Enrolled in B.Tech. (Hons. In ICT with minors in Computational Science),

Dhirubhai Ambani Institute of Information and Communication Technology, Gandhinagar

***Abstract*—Union and find Algorithm applications involve manipulating objects of all types. For example, Computers in a network, web pages on the Internet, Transistors in a computer chip, Variable name aliases, Pixels in a digital photo etc. This paper presents a parallel approach to this algorithm in finding partitions of connected points from a given dataset. We implemented our algorithm on an Intel Cluster using the OpenMP Multi threading Library.**

Keywords—Connected partitions, Union-Find, Hash Table, OpenMp

I. INTRODUCTION

With increasing points in day-to-day data sets, it has become a necessity to produce algorithms that run fast with accuracy. Moreover, the extent upto which an algorithm can be implemented in parallel acts as another factor to efficient algorithms. In this paper, we present an extended application of Union and find algorithm along with other optimisation techniques to efficiently and accurately find all the connected partitions.

Section II gives a summary about the algorithm applied, Section III mentions the performance optimisation techniques, Section IV delves on the details of the parallelization techniques applied in the algorithm. Finally, Section V analyses the efficiency measurements of the algorithm with observations.

II. UNION-FIND ALGORITHM

Consider a 3-dimensional grid of points. Each point in the grid can be specified with 3 coordinates (x, y, z). *Where x, y and z are integers*. We call any two points on the grid connected to each other, if and only if they are separated by unit distance.

A *connected partition* is a set of points with following properties,

1. If a point is included in the partition, then all the points connected to it must also be included in the partition.
2. Every point in the partition must be connected to at least one other point.

For a given point, there are 6 possible other points that would be at units distance from it. Thus, for every point we check, if any of the six are present in the dataset. We connect such points into a set. This requirement is best resolved by union find Algorithm.

Find: Determine which subset a particular element is in. Find typically returns an item from this set that serves as its "representative".

Union: Join two subsets into a single subset.

The ***naive*** Union-Find Algorithm is as follows:-

1. All the points of the dataset have being assigned an id as per input order. Now, each point has a representative, which as of now is its own id.
2. For each point, in a Find call, we find the representative element of the set to which a particular point belongs.
3. When the current point is found connected to the another point, we call Union on them uniting the two sets to which they belong into a single representative set. For this, we are finding the two representative elements by calling the Find function and setting first representative of first point to second point's representative.

Pseudo Code:-

```
function Find(x)
    if x.parent == x
       return x
    else
       return Find(x.parent)
function Union(x, y)
    xRoot := Find(x)
    yRoot := Find(y)
    xRoot.parent := yRoot
```

The above classical approach is not fit for our purpose. This takes O(n) time in worst case as the resulting tree can be highly unbalanced.

III. PERFORMANCE OPTIMIZATIONS

A. ***Union by Rank*** (Weighted): Here, we maintain the rank (height) of each set. Whenever the Union call is made, we compare the ranks. The representative element of the point with smaller rank is changed to that of the point with larger rank. If both the ranks are same, any of the representative element can be assigned to both, by incrementing the rank by one. Just applying this technique alone yields a worst-case running-time of O(log n) for the Union or Find operation. The pseudocode for the improved Union goes like:-

```
function Union(x, y)
     xRoot := Find(x)
     yRoot := Find(y)
     if xRoot == yRoot
         return
     //x and y are not in same set. Merge them.
     if xRoot.rank < yRoot.rank
         xRoot.parent := yRoot
     else if xRoot.rank > yRoot.rank
         yRoot.parent := xRoot
     else
         yRoot.parent := xRoot
         xRoot.rank := xRoot.rank + 1
```

B. ***Path Compression:*** This is used to decrease the unnecessary traversal to the root for a given point when Find is used on it. The idea is that each point visited on the way to a root point may as well be attached directly to the parent point; after all, they all have the same representative. To effect this, as Find recursively traverses up the tree, it changes each point's parent reference to point to the root that it found. The resulting tree is much flatter, speeding up future operations not only on these elements but on those referencing them, directly or indirectly. The pseudo code for the improved Find goes like:-

```
function Find(x):
     if x.parent != x
        x.parent := Find(x.parent)
     return x.parent
```

Other than the above two modifications to our classical Union-Find algorithm, we use a **Hash-Table** to make it possible to check in constant time whether a point exits in the input dataset or not. This is used to find whether the 6 immediate neighbours of the current point exist in the input dataset. If they do, we call Union.

The specifications of this Hash-Table are:

- Table size is set to the total number of points in input.
- We compute the index for a given point to be used in the Hash-table from the hash function. In this function, we use three dimensions along with two large prime numbers and modulo table size.

```
int getHashIndex(Point *p)
{
    int index = (p->x*997*631 + p->y*631 + p->z);
    index = index & 0x7FFFFFFF;
    return index % hashSize;
}
```

We do the above to ensure that there is even distribution of points across the Hash-table.

IV. PARALLELIZATION TECHNIQUES APPLIED

Analysis of the initial implementation with profiling tools revealed that the program spent a significant time in the actual algorithm compared to side tasks like I/O operations.

Hence, integration with Hash-Table along with parallelization using OpenMP multi threading library for C would do the task.

1. The **#pragma omp for** construct is used to efficiently parallelize for loops, providing initial values to count based arrays and fixed increments to the output array. The construct dynamically divides the iterations among the threads to ensure maximum efficiency.

2. The **#pragma omp parallel** construct is used in the implemented algorithm, with the threads spawned diving the load among themselves. Also, the concept of private and shared variables is used, making loop variables private and Hash-table, data points shared, in order to give proper access to the required variables to all the threads without concurrency problems.

3. The **#pragma omp critical** construct is used to ensure that blocking access is given to a particular thread when it is calling the Union-Find operations. This is to make sure that multiple occurrences do not corrupt the relationships between a given point and it connectedness to other points.

There are situations in the code that have been handled using temporary variables in order to improve cache utilization, and reduce the data access time.

V. OBSERVATIONS AND INFERENCES

We observed the time by varying the number of threads that were being launched. However, the problem not being embarrassingly parallel wont scale for large number of cores as the communication between threads is more significant than the computation itself. By hit and trial, we observed that the code was giving the fastest execution for 4 threads.

So, following were our results with 3 iterations in the program.

Large Dataset

Number of threads launched = 4
Best end-to-end time (in secs) = 40.530892
Average end-to-end time (in secs) = 40.851084
Best Algorithm Time (in usecs) = 25832368
Average Algorithm Time (in usecs) = 26154830

Medium Dataset

Number of threads launched = 4
Best end-to-end time (in secs) = 9.419849
Average end-to-end time (in secs) = 9.635608
Best Algorithm Time (in usecs) = 5766563
Average Algorithm Time (in usecs) = 5977752

REFERENCES

[1] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001), Introduction to Algorithms

[2] Galler, Bernard A.; Fischer, Michael J. (May 1964), "An improved equivalence algorithm", Communications of the ACM

[3] Michael J. Quinn; “Parallel Programming in C with MPI and OpenMP”