3D Union Find Parallel Algorithm

Student Programming Challenge [Intel Track]

23rd IEEE International Conference on High Performance Computing, Data and Analytics

Hyderabad, India

Aayush Kapadia
B.Tech (ICT with CS)
DAIICT
Gandhinagar, India
kapadiaaayush@gmail.com

Mit Naria
B.Tech (ICT with CS)
DAIICT
Gandhinagar, India
mitnaria@gmail.com

Uttamkumar Chaudhari
B.Tech (ICT with CS)
DAIICT
Gandhinagar, India
chaudharyuttam36@gmail.com

***Abstract*— This paper describes the algorithm for solving 3D Union Find Problem given in Intel Student Parallel programming challenge, HiPC 2016.**

I. Introduction

In this paper, we describe a 3D Union Find problem and a parallel algorithm for solving the problem.

II. 3D Union Find Problem

A. Defining the problem

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. For example, point (1, 1, 1) is connected to point (1, 2, 1). On the other hand, points (1, 1, 1) and (2, 2, 1) are not connected to each other. Similarly, 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.

Example: Suppose 3 points are given: (1,1,1) , (1,2,1) , (2,2,2) , then (1,1,1) and (1,2,1) are in the same partition because they are connected while (2,2,2) is in the another partition because it is not connected to any other points.

Thus, partition 1: (1,1,1) , (1,2,1).

Partition 2: (2,2,2).

B. Problem statement and input/output format

Given a set of points S on a grid, find all the connected partitions in it. Label each partition with a unique integer between 1 and total number of partitions.

C. Input format

All lines in the input file contains x,y,z where x,y,z are respective co-ordinate of the points.

Example:

X1,Y1,Z1

X2,Y2,Z2

….

Xn,Yn,Zn

n is not known.

D. Output format

For each point in the same order as input, output the partition number which is between 1 and total number of partitions.
Example:
<partition-label1>
<partition-label2>
….
<partition-labeln>

III. Serial algortihm

Preprocessing: First we count n by using system utility “wc -l” because we need an exact count of number of points. Our algorithm takes in number of points “n” as an input argument.

Algorithm: We first read all the input points and store it in an array. After that we insert into a hashMap..We have implemented our own customized hashMap specifically for this. Now we are creating a new array of edges which will be a pair of two integers, index ‘i’ and index j such that there is an edge between point ‘i’ and point ‘j’. Now for every points we would calculate it's 6 possible neighbors and find in hashMap if that neighbors exist. If neighbor exist we would add (i,j) pair into new edge array if and only if i<j .This is to avoid duplicate edges.

Now we would run our standard Union find algorithm which uses Path Compression optimization over this edge array. At the end we will get root index for every partition point. Now we want every point to label between 1 and total number of partitions. So what we have done is maintain a partition count initialized to 0. Now whenever we encounter root index (root point), we would increment partition count and then label this root vertex with value of current partition count. Now every root vertex are labelled. So now every other non root vertex will be labelled same as root vertex of their respective partitions.

Pseudocode after creating edge array:

```
for all edges in edge array :
    union(edge.first,edge.second)
```

IV. Parallel Algorithm

We use OpenMP multi threading library for parallelizing the algorithm.

The serial part till inserting the points into hashMap is same as serial algorithm. Now the part creating the edge array and then running union-find algorithm on it is parallelized. The remaining part of labelling each partition is also kept serial.

We have only parallelized the middle part of the algorithm that is creating edge array and running union find on it. This is because on GNU gprof profiling results, we have found that this part takes around 55% of the time on large dataset. So we have tried to parallelized this part.

Assuming that we are launching p threads, we have statically divided the points into p threads. That is each thread will get approximately n/p points. Now each thread will create edge array for only edges which originates from this points.

After thread has added edges into the edge array, it will perform union find algorithm over this edges. We have to add critical section on this calling of union function to avoid race conditions between threads.

V. Pseducode for parallel algorithm

Each thread divides the vertices among themselves. So suppose our thread has got vertex from index start to index end, then it will run following pseudocode for it.

```
for i in [start,end)
    add all edges originating from index I to edge array.
for edge in edgeArray :
    # pragma omp critical
    Union(edge.first,edge.second)
```

VI. Older results

We ran our code on cluster Intel Xeon E5-2620 v3 @ 2.40 GHz. [Provided by our institute]. We will refer this as Institute cluster.

For Large Dataset,

No of threads	End to end time (seconds)	Algorithm time (seconds)
2	8.858	6.756
4	6.267	4.06
6	5.278	3.134
8	4.778	2.682
10	4.555	2.434
12	4.371	2.312

We ran our code on cluster provided by Intel. We will refer this as Intel cluster.

No of threads	End to end time (seconds)	Algorithm time (seconds)
2	34.8589	23.943
4	24.312	13.400
6	20.8068	9.897
8	18.9729	8.066
10	17.9481	7.0387
12	17.2094	6.309
68	14.4062	3.4913
272	14.1874	3.2100

We have observed that on Intel cluster I/O time is about 10 seconds while on the Institute cluster it took about 2 seconds. In fact, it took about 2.3 seconds on our 4th Gen i5-4210U laptop with a simple HDD. We suspect that Intel cluster is may not be configured well. Another reason is that there is no L3 cache in Intel cluster. This depicts that I/O time is can be bottleneck. So we further optimized our code for fast I/O.

VII. Final results

After optimizing I/O, we again ran our code on institute cluster. For Large Dataset,

No of threads	End to end time (seconds)	Algorithm time (seconds)
2	7.202	6.3282
4	4.956	4.0799
6	4.0015	3.1562
8	3.6362	2.7457
10	3.3439	2.4651
12	3.3196	2.4268

We again ran our code on cluster provided by Intel.

No of threads	End to end time (seconds)	Algorithm time (seconds)
2	30.767706	24.2387
4	20.117939	13.5785
6	16.31769	9.79404
8	14.79003	8.26607
10	13.546653	7.02966
12	12.884477	6.3692
68	10.072077	3.54631
272	9.722857	3.20489

VIII. Getting best result

As we can that we get best results on intel cluster when we spawn maximum number of threads that is 272. We also get best results on institute cluster when we spawn maximum number of threads that is 12. We have attached RESULT.txt that contains hardware details.

IX. References

[1]. Algorithms by Robert Sedgewick.4th Edition.
http://algs4.cs.princeton.edu/15uf/
[2] Hardware details of institute cluster.
http://ark.intel.com/products/83352/Intel-Xeon-Processor-E5-2620-v3-15M-Cache-2_40-GHz