

# Line Segment Intersection

## Problem Description

Write a threaded code to find pairs of input line segments that intersect within three-dimensional space. The input file of line segments and the output results file will be specified on the command line.

Input format: The first line of the text file will contain a single integer, N, which will be the number of line segments contained in the file. The remaining N lines will have the four (printable) character name of the line segment and 6 integers representing the two (x,y,z) endpoints of the line segment. The first three numbers will be one endpoint, the last three will be the coordinates of the other endpoint. The precision required for this problem will be that of a 32-bit float type on the judging platforms.

Output format: Use some human readable format. List each pair of line segments from the input data set intersect. If there are no intersections, then print a message stating that fact.

## Input File Example

```
8
AAAA -6 1 3 -8 4 -4
BBBB 3 -9 0 -3 -1 6
CCCC -1 -7 2 -4 -8 -4
DDDD 0 0 0 -1 6 2
EEEE 0 3 8 5 3 -7
FFFF 3 6 4 -4 0 -2
GGGG 8 -1 0 7 9 0
HHHH 11 8 0 10 -4 -3
```

## Output File Example

Segments that intersect:

```
DDDD FFFF
```

## Vector Arithmetic

A line segment  $L$  between points  $P_1(x_1, y_1, z_1)$  and  $P_2(x_2, y_2, z_2)$  is given by the equation  $P = P_1 + t(P_2 - P_1)$  for all  $0 \leq t \leq 1$

Two lines  $L_i = P_{i1}P_{i2}$  and  $L_j = P_{j1}P_{j2}$  intersect iff

1.  $L_i$  and  $L_j$  are parallel and  $P_{j1}$  lies on  $L_i$  between  $P_{i1}$  and  $P_{i2}$  or  $P_{j2}$  lies on  $L_i$  between  $P_{i1}$  and  $P_{i2}$  or  $P_{i1}$  lies on  $L_j$  between  $P_{j1}$  and  $P_{j2}$ . This can be checked using the equation defined above.

OR

1. They are not parallel
  - (a) The cross product of the directions of parallel lines is the 0 vector. i.e. the lines are parallel if  $N_{ij} = (P_{i2} - P_{i1}) \times (P_{j2} - P_{j1}) = (0, 0, 0)$
2. They lie in the same plane
  - (a) Lines lie in the same plane if the dot product  $(P_{i1} - P_{j1}) \cdot N_{ij} = 0$
3. They have a common point
  - (a)  $P_{i1} + t_i(P_{i2} - P_{i1}) = P_{j1} + t_j(P_{j2} - P_{j1})$  for some  $t_i, t_j$  such that  $0 \leq t_i, t_j \leq 1$

## Serial Algorithm

The original implementation was a brute-force algorithm as described below:

1. Let  $N$  be the number of lines
2. Let  $R$  be a vector for holding pairs of intersecting lines
3. Check if segment  $L_i$  intersects segment  $L_j$  for all  $0 \leq i < N-1, i < j < N$ 
  - (a) If  $L_i$  and  $L_j$  intersect, add the pair  $\langle L_i, L_j \rangle$  to  $R$
4. Write each pair in  $R$  to the output file

## Aha!

Then came the realization that the lines could be partitioned based on their coordinates into 3 categories per dimension. This would significantly reduce the number of comparisons.

For some value  $v_x$ , the categories based on the x-coordinate would be:

1. Segments for which x-coordinates of both end points are greater than or equal to  $v_x$
2. Segments for which x-coordinates of both end points are less than  $v_x$
3. Segments which do not belong to either of the above categories

Each of the above partitions can be further categorized on the basis of the y-coordinate based on some value  $v_y$  and again on the basis of the z-coordinate for some value  $v_z$ . Such a classification distributes the lines into 27 ( $3 \times 3 \times 3$ ) categories.

Such a classification can be stored in a 3D array (call it *division*) with each dimension of the array representing the categories for one spatial dimension. The indices chosen for the categories were:

- 0: neither category
- 1: both coordinates < v
- 2: both coordinates >= v

Therefore,

division[0][0][0]	contains segments that cannot be categorized by any coordinate
division[0][0][1]	contains segments that have both z coordinates < v <sub>z</sub>
...	
division[1][0][2]	contains segments that have both x coordinates < v <sub>x</sub> and both z coordinates >= v <sub>z</sub>
...	
division[2][2][2]	contains segments that have both x coordinates >= v <sub>x</sub> , both y coordinates >= v <sub>y</sub> and both z coordinates >= v <sub>z</sub>

The best distribution of random lines was observed when the mean of all the points was used as the basis for categorization.

i.e.  $(v_x, v_y, v_z) = \Sigma(P_{n1} + P_{n2}) / (2 * N)$  for all  $0 \leq n < N$

### **Modified Parallelizable Algorithm**

1. Let N be the number of lines
2. Compute the mean of all the end points  $(v_x, v_y, v_z)$
3. Let R be a vector for holding pairs of intersecting lines
4. Let division[3][3][3] be a 3-dimensional array of vectors of line segments
5. Add segment  $L_n = P_{n1}P_{n2}$  to division[a<sub>n</sub>][b<sub>n</sub>][c<sub>n</sub>] for all  $0 \leq n < N$  where,
  - (a)  $P_{n1} = (x_{n1}, y_{n1}, z_{n1})$  and  $P_{n2} = (x_{n2}, y_{n2}, z_{n2})$
  - (b) a<sub>n</sub> =
    - i. 1 if  $x_{n1} < v_x$  and  $x_{n2} < v_x$
    - ii. 2 if  $x_{n1} \geq v_x$  and  $x_{n2} \geq v_x$
    - iii. 0 otherwise
  - (c) b<sub>n</sub> =
    - i. 1 if  $y_{n1} < v_y$  and  $y_{n2} < v_y$
    - ii. 2 if  $y_{n1} \geq v_y$  and  $y_{n2} \geq v_y$
    - iii. 0 otherwise
  - (d) c<sub>n</sub> =
    - i. 1 if  $z_{n1} < v_z$  and  $z_{n2} < v_z$
    - ii. 2 if  $z_{n1} \geq v_z$  and  $z_{n2} \geq v_z$
    - iii. 0 otherwise
6. For each category, division[a][b][c] for all  $0 \leq a, b, c < 3$ 
  - (a) Check if segment L<sub>i</sub> intersects segment L<sub>j</sub> for all  $0 \leq i < N_{abc}-1, i \leq j < N_{abc}$  where N<sub>abc</sub> is the number of elements in division[a][b][c].
    - i. If L<sub>i</sub> and L<sub>j</sub> intersect, add the pair <L<sub>i</sub>, L<sub>j</sub>> to R
7. For each pair of categories, division[a][b][c] and division[d][e][f] such that
  - (a=0 or d=0 or a=d) and
  - (b=0 or e=0 or b=e) and

- (c=0 or f=0 or c=f)
- (a) Check if segment  $L_i$  of division[a][b][c] intersects segment  $L_j$  of division[d][e][f] for all  $0 \leq i < N_{abc}$ ,  $0 \leq j < N_{def}$  where  $N_{abc}$  and  $N_{def}$  are the number of segments if division[a][b][c] and division[d][e][f] respectively.
- i. If  $L_i$  and  $L_j$  intersect, add the pair  $\langle L_i, L_j \rangle$  to R
8. Write each pair in R to the output file

## Parallel Algorithm

The serial algorithm described above can be easily parallelized using the task pool model. Steps 6 and 7 of the serial algorithm above define 184 computationally independent tasks.

Efficiency of the algorithm is affected when computationally intensive tasks are present at the end of the queue. Hence, in order to balance the load better, a priority queue was used with the task pool. The priority of a task is determined by the number of pairs of lines to be checked for intersections. Greater the number of comparisons, higher the priority. Priorities can be computed as follows:

- For tasks generated in step 6 of the algorithm, the following pairs of lines must be checked:

	$L_1$	$L_2$	$L_3$	...	$L_{N-1}$	$L_N$
$L_1$		Y	Y	...	Y	Y
$L_2$			Y	...	Y	Y
$L_3$				...	Y	Y
...	...	...	...	...	...	...
$L_{N-1}$				...		Y
$L_N$				...		

A 'Y' indicates a pair of lines to be compared.  $L_1$  must be compared with N-1 other lines,  $L_2$  with N-2 other lines and so on. Obviously, the number of comparisons is the summation of integers from 1 to N-1.

$$\text{Number of comparisons} = \sum_{i=1}^{N-1} i \text{ (for } i = 1 \text{ to } N-1) = (N-1) * N / 2$$

- For tasks generated in step 7 of the algorithm, the following pairs of lines must be checked:

	$L_{j1}$	$L_{j2}$	$L_{j3}$	...	$L_{jM-1}$	$L_{jM}$
$L_{i1}$	Y	Y	Y	...	Y	Y
$L_{i2}$	Y	Y	Y	...	Y	Y
$L_{i3}$	Y	Y	Y	...	Y	Y
...	...	...	...	...	...	...

$L_{iN-1}$	Y	Y	Y	...	Y	Y
$L_{iN}$	Y	Y	Y	...	Y	Y

In this case,  $\{ L_{i1}, L_{i2}, \dots, L_{iN} \}$  and  $\{ L_{j1}, L_{j2}, \dots, L_{jM} \}$  are disjoint sets. Each of the  $N$  lines in the first set must be compared with each of the  $M$  lines in the second set. Therefore, the number of comparisons here is  $M * N$ .

Additionally, floating point arithmetic was avoided, for the sake of accuracy, and to overcome the overhead of tolerances, by storing fractions in the numerator / denominator format. For example, the code used to check whether a point lies on a segment is as follows:

```
class Line {
public:

    char name[8];
    Point p1;
    Point p2;
    Point dir; //    dir = p2 - p1

    //    ... Other variables and methods ...

    inline int contains(Point &p) {
        int rN = 0; //    numerator
        int rD = 0; //    denominator
        if (dir.x != 0) {
            rD = dir.x;
            rN = p.x - p1.x;
        } else if (dir.y != 0) {
            rD = dir.y;
            rN = p.y - p1.y;
        } else if (dir.z != 0) {
            rD = dir.z;
            rN = p.z - p1.z;
        }

        //    rN / rD must be between 0 and 1
        if ((rD * rN >= 0 && ABS(rD) >= ABS(rN)) &&
            (p.x - p1.x) * rD == dir.x * rN &&
            (p.y - p1.y) * rD == dir.y * rN &&
            (p.z - p1.z) * rD == dir.z * rN )
            return 1;

        return 0;
    }
};
```

Another approach attempted was parallelizing the brute force algorithm using the data-parallel model and mapping data to threads statically. But this approach was found to be significantly slower than the categorization-based algorithm. This is due to the fact that categorization renders a substantial percentage of comparisons unnecessary. For example,  $division[0][0][1]$  need not be compared with  $division[0][0][2]$  since the lines in these categories lie in spatially different regions.

A major problem encountered while implementing the parallel solution was cache thrashing. Since tasks are prioritized based on number of comparisons, divisions with larger sizes tend to dominate the initial few tasks. If pointers to the original segment objects are used in comparisons, the same segments getting used in different threads leads to substantial performance degradation. This issue was resolved by creating per-task copies of the required segments. This problem is evident in the following instance:

For an input data set of 160000 lines, the parallel brute force algorithm generated results in 109.37 sec whereas the parallel category-based algorithm took 178.65 sec to generate the same result. After the vector copy logic was implemented, the time for the parallel category-based algorithm was reduced to 25.81 seconds.

## Performance

The hardware configuration used for testing purposes was as follows:

Processor: Core 2 Quad Q8200 2.33GHz

RAM: 4GB DDR3 1333MHz + 8GB swap

The following program was used to generate test data:

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.PrintWriter;

public class TestGenerator {
    static final byte alphabet[] =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789".getBytes();
    int nLines;
    int min;
    int max;

    public TestGenerator(int nLines, int min, int max) {
        this.nLines = nLines;
        this.min = min;
        this.max = max;
    }

    public void generate(String filename) throws FileNotFoundException {
        int range = max - min;
        int x1, y1, z1;
        int x2, y2, z2;
        char name[] = new char[4];
        PrintWriter fout = new PrintWriter(new FileOutputStream(filename));
        fout.println(nLines);
        for (int i = 0; i < nLines; i++) {
            x1 = (int) (Math.random() * range + min);
            y1 = (int) (Math.random() * range + min);
            z1 = (int) (Math.random() * range + min);
            x2 = (int) (Math.random() * range + min);
            y2 = (int) (Math.random() * range + min);
            z2 = (int) (Math.random() * range + min);
```

```

        for (int j = 0; j < 4; j++)
            name[j] = (char) alphabet[
                (int) (Math.random() * alphabet.length)];

        fout.println(new String(name) + " " + x1 + " " + y1 + " " + z1
            + " " + x2 + " " + y2 + " " + z2);
    }

    fout.close();
}

```

This code generates an input file called *filename* containing *nLines* lines with end-points in the range *min* to *max*.

The following results were observed:

# Lines	Range	# Intersections	Brute Force		Categorization	
			Serial	Parallel	Serial	Parallel
10000	0 – 100	685	1.6	0.44	0.46	0.14
20000	0 – 100	2730	6.36	1.66	1.7	0.48
40000	0 – 1000	20	24.29	6.55	6.46	1.7
80000	0 – 1000	59	98.92	26.88	25.46	6.57
160000	0 – 1000	258		109.37	101.06	25.81

## Conclusion

From the above table it is evident that the speed of the categorization-based serial algorithm is comparable to the speed of the parallel brute-force algorithm on 4 cores. Scaling is almost linear for both algorithms.