15-418 Final Project Report

Jason Li and David Wise

May 12, 2015

1 Summary

We implemented a parallelized algorithm in MPI to count triangles in graphs. After optimization we were able to achieve a 3-15 times speedup over the commercial GraphLab framework on graphs with varying densities.

2 Background

A triangle in an undirected graph is what you would expect: a set of 3 vertices such that every pair is connected by an edge. The input to our algorithm is a file formatted as follows:

```
<number of vertices> <number of edges>
v1 v2
v3 v4
...
```

where $(v1, v2), (v3, v4), \ldots$ represent the edges of the graph. Our algorithm outputs a single integer, the number of triangles in the graph. The triangle count of a graph gives some measure of how clustered the graph is, and finds applications in a variety of areas such as social networks, spam detection, and link classification [1].

3 Approach

There are many known approaches to counting triangles in graph serially, but doing the computation efficiently in parallel is a more recent area of research [2]. We focused on parallelizing an algorithm in [2] called "compact forward," which we describe later in this section. Initially, we implemented the algorithm in C++ using OpenMP, but unfortunately, we observed no speedup. Since OpenMP uses the shared memory model, all of the processors are using the same memory bus to access the graph, which is stored in shared memory. We believe that the algorithm is bandwidth bound by the frequent accesses of memory in line 3(b) of the algorithm (shown later in the section).

At this point, we decided that a processor model without shared memory would be preferable. We decided to go with OpenMPI, which uses the message-passing model, in which each processor has its own memory space. This approach is much more scalable and could even be run on a distributed system. This would also make it conceivable to operate on graphs that could not be completely stored on one machine if we could find an algorithm that allowed each processor to only require a portion of the graph.

3.1 The *compact forward* algorithm

This section gives the algorithm from [2] that we parallelized.

1. Initialize the counter to 0
2. Sort the vertices in order of increasing degree, breaking ties arbitrarily. Similarly, sort the adjacency lists according to the same ordering
3. For each edge (v, w) with $v < w$:
   (a) Let u_v and u_w be the first vertices in the adjacency lists of v and w, respectively
   (b) While u_v exists and $u_v < v$ and u_w exists and $u_w < v$:
      i. if $u_v < u_w$ then set u_v to the next neighbor of v
      ii. else if $u_v > u_w$ then set u_w to the next neighbor of w
      iii. else increment the counter, and set u_v and u_w to their next neighbors

This algorithm counts all triangles (u, v, w) with $u < v < w$. It loops over all possible edges (v, w) with $v < w$, and finds all vertices $u < v$ that are neighbors of both v and w. To find all such u given an edge (u, v), it scans the two adjacency lists with a pointer in each list, and detects all common occurrences. Every such occurrence must be counted, since the algorithm always increments the lower of the two pointers, and therefore never skips over a common value.

This algorithm clearly takes $O(m)$ space, since only the graph needs to be stored. It can be proven that since the vertices are sorted by degree, the algorithm runs in $O(m^{3/2})$ time. For a proof, refer to the original compact forward paper.

Our project focused on how to parallelize this algorithm. At a first glance, it appears that each processor in a parallel machine can process the edges (v, w) independently. However, in order to count the number of vertices $u < v$ such that (u, v, w) form a triangle, the processor needs access to the adjacency lists of v and w. Therefore, if the edges (v, w) are assigned arbitrarily to the processors, then each processor needs the adjacency lists of all vertices that appear in its list of edges, which may well include the entire graph. We decided to find a way to reduce this communication cost.

Our main idea is to bucket the edges into groups so that each group contains few distinct vertices. With P processors (and P a perfect square), we can map the vertices into $\sqrt{P}$ distinct values according to a random function f, and assign edge (v, w) to the group indexed by the pair $(f(v), f(w))$. If the function f is uniform, then we expect $\frac{1}{P}$ fraction of the edges to belong to each group. Furthermore, each group only contains $\sqrt{P}$ distinct vertices. Therefore, if we assign one group to each processor, then each processor only needs the adjacency lists of $2\sqrt{P}$ vertices to perform its even share of the work.

Remember that we need to sort the vertices by degrees first. Since ties are broken arbitrarily, we impose the lexicographical ordering $u < v$ iff $(deg(u), u)$ is lexicographically smaller than $(deg(v), v)$. We can sort the vertices in $O(n)$ time with a *radix sort* on the degree. Furthermore, we need to sort the adjacency lists according to this ordering, which can be done in $O(|V| + |E|)$ time, which is $O(n + m/\sqrt{P})$ time per processor.

As for triangle counting, each processor performs independent work and there is $O(m^{3/2})$ work in total, so each processor expects $O(m^{3/2}/P)$ work for triangle counting. Therefore, the total expected work of the algorithm is $O(nP + m\sqrt{P} + m^{3/2})$, and the span is $O(n + m/\sqrt{P} + m^{3/2}/P)$. Since $O(m^{3/2})$ is usually the dominating term, we expect a near-perfect speedup in theory.

3.2 Implementation

We implemented this algorithm using the MPI C++ framework. Our code is given in the appendix.

In our implementation of this algorithm, the main difficulty we encountered was reading the input. The first problem is that `scanf` is very slow: it takes 8 seconds just to read input on the LiveJournal test case on a single processor. In addition, reading input is hard to parallelize. The function `seekg` can move a file pointer to a specified number of bytes from the beginning of a file, but since the length of each edge in the input file varies in length, we cannot `seekg` to a specific edge in the input file. Hence, the input always takes

$O(m)$ sequential time. In our implementation, we only let one processor (call it the *root* processor) read the input, and then broadcast the vertices and edges to all other processors. However, we were able to greatly reduce the time required to read the input by using a custom integer reading routine that is more fragile, but much faster:

```
#define READ_INT(n) {char c; n = getchar_unlocked() - 0; while((c = getchar_unlocked()) >= 0)
    n = (n << 3) + (n << 1) + c - 0;}
```

This routine was able to read the LiveJournal graph data in only 1.3 seconds, instead of 8.

Another implementation issue we encountered was broadcasting the input to the other processors. Theoretically, we can achieve $O(m\sqrt{P})$ bandwidth as we showed in our previous analysis, since the root node only needs to send $O(m/\sqrt{P})$ edges to each of P processors. However, we found that just broadcasting the entire list of edges using $O(mP)$ bandwidth works faster in practice. However, all our tests were only run on a single machine. In a distributed setting or on very large graphs it seems likely that broadcasting would be much less efficient.

4 Results

We ran our algorithm on four graphs from the Stanford Large Network Dataset Collection [3]. The The number of vertices, edges, and triangles in each graph is listed below:

	gplus	k5000	live	skitter
vertices	107614	5000	3997962	1696415
edges	12238285	12497500	34681189	11905298
triangles	1073677742	20820835000	177820130	28769868

The k5000 graph is the complete graph on 5000 vertices, which is obviously as dense as possible, while the live and skitter graphs are quite sparse. We found that these graphs are representative for a large class of graphs in the real world, since they varied greatly in sparsity and the number of triangles.

The machine we used, `muir.graphics.cs.cmu.edu`, had 32 Intel Xeon i7 cores.

4.1 Comparison with GraphLab

We installed and ran GraphLab on the same machine to get a baseline for comparison. GraphLab is a scalable engine for graph computation [6]. We recorded two times: the time to just count the triangles, and the total time, including reading in and processing the graph. For the times for our implementation, we ran on all perfect square numbers of cores up to 64 and took the best time. We ran our implementation and GraphLab on the same 32-core hyperthreaded machine for a fair comparison.

Graph	gplus	k5000	live	skitter
Our implementation	1.59741	1.18134	0.713308	0.272485
GraphLab	6.71067	59.108	50.5255	31.1852
Speedup relative to GraphLab	4.20097	50.03471	70.83266	114.44740

Table 1: Comparison of time to count triangles

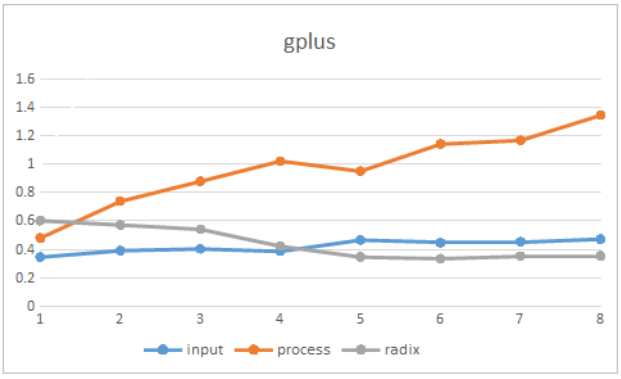
We see that our speedup for the total time taken ranged from 3.57 to 26.01.

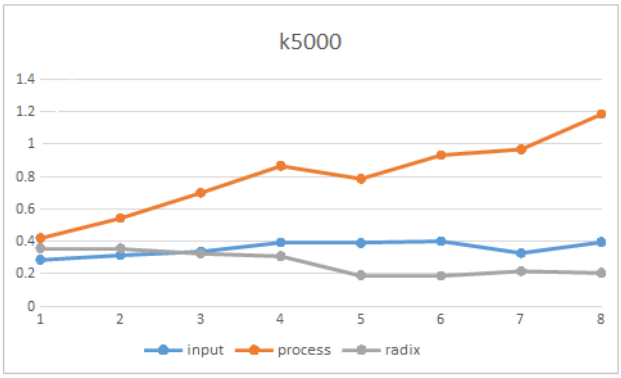
Here are our results. For the number of processes, we tried every square number from 1 to 64. For each test case and number of processes, we ran the algorithm 5 times and took the median time.

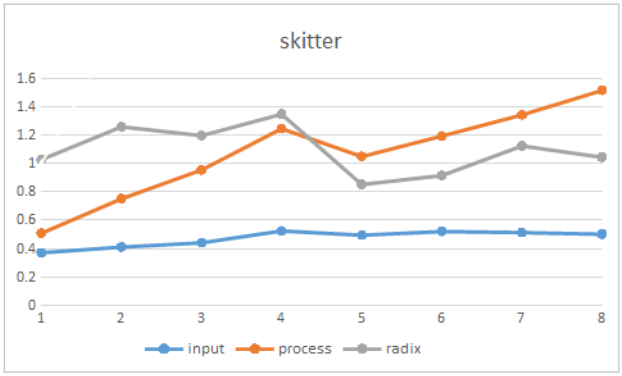
Here are the graphs for input time, preprocess time (including input), and radix sort for each test case. (A complete table of the results are provided in the Complete Results section.)

Graph	gplus	k5000	live	skitter
Our implementation	3.68976	2.77177	11.9798	2.99535
GraphLab	13.195	72.098	83.476	45.605
Speedup relative to GraphLab	3.57611	26.01154	6.96806	15.22527

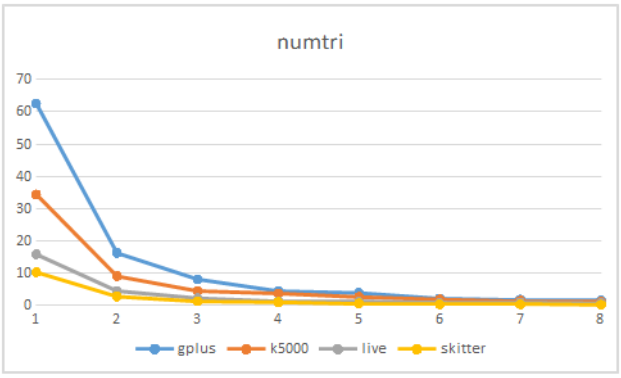
Table 2: Comparison of total time taken







Finally, here is the graph of the total running times for each test case, followed by the final speedup:



	gplus	k5000	live	skitter
speedup	17.22803	12.71895	1.94565	3.980162

The speedup factors (relative to one processor) varied from graph to graph, from only 2× on LiveJournal to 17× on Gplus. We expected worse speedup on less clustered graphs, in which the ratio of the number of triangles to the number of edges is small. This is because the radix sort, which is linear in the number

of edges, is only $\sqrt{P}$ parallel, while counting the number of triangles is P parallel. In addition, radix sort incurs more work with many processors, since its total work is $O(m\sqrt{P})$. Our assumption turned out to be true, considering that LiveJournal has $3\times$ as many edges than Gplus but only $1/6\times$ as many triangles.

5 Comparison with other sequential algorithms

Recall our triangle counting time results when we ran our parallel algorithm on one processor:

	gplus	k5000	live	skitter
Counting	62.486	34.4093	15.8704	10.3336

We tried another algorithm which starts off by sorting the vertices and the adjacency lists by degree, but then does the following:

3 For each edge u:

(a) Consider the set of all neighbors v such that $u < v$. For each pair in this set, check if that pair is an edge in the graph.

To ensure efficient constant-time checking, the edges are checked using an offline sweep (refer to the code). However, this makes the memory usage $O(m^{3/2})$, so we were unable to perform this algorithm on k5000.

	gplus	k5000	live	skitter
Counting	81.876557	?	17.893766	2.291446

Although the theoretical complexity is still $O(m^{3/2})$ once the vertices are sorted by degree, this algorithm performs well in practice on sparse graphs, but poorly on dense graphs.

In comparison to this sequential algorithm, it appears that our parallel algorithm serves as an excellent sequential version when only using one processor. The above algorithm, while performing very well on skitter, did not excel in the other denser graphs.

6 Further improvements

From a theoretical perspective, input is a bottleneck, since it must be read sequentially from a file in $O(m)$ time. However, consider a model in which the input graph is not stored in a file, but in a database that allows efficient retrieval of the adjacency list of any vertex. In that case, input can indeed be parallelized to $O(m/\sqrt{P})$ time. The total bandwidth from the database server is still $O(m\sqrt{P})$, so bandwidth might still be the limiting factor.

7 Generalizing to k-cliques

The compact forward algorithm can be generalized to count k-cliques instead of triangles (or 3-cliques). Instead of looping over all edges (v, w) and finding the number of vertices u common to the adjacency lists of v and w, we now loop over all $(k-1)$-cliques and find the number of vertices common to all $k-1$ adjacency lists. Therefore, we run compact forward on $(k-1)$-clique finding, and every time a $(k-1)$-clique is found, we do this extra step of counting the number of common neighbors. The space requirement is still $O(|V| + |E|)$ for each processor.

In general, if the vertex degrees are sorted, then this algorithm will run in time $O(m^{k/2})$. Therefore, for large values of k, we do not expect to be able to count k-cliques efficiently. Nevertheless, if we bucket the groups into $\sqrt[k]{P}$ buckets where P is a perfect k-th power, then we can assign each of P processors to a

k-tuple of buckets. Each processor will get $O(m/\sqrt[k]{P})$ edges, and counting the k-cliques will take $O(m^{k/2}/P)$ time. Overall, the work becomes $O(nP + mP^{1-1/k} + m^{k/2})$, the span is $O(n + m/\sqrt[k]{P} + m^{k/2}/P)$, and the space requirement per processor is $O(n + m/\sqrt[k]{P})$. Since the $m^{k/2}$ term dominates the other terms by a wide margin, we expect near-perfect speedup in theory.

8 Appendix

8.1 Complete Results

The input column is the time to read the input. Processing is the time to read the input and broadcast the graph data. Radix is the time to perform the radix sorts, Counting is the time to count the triangles, and Total is the total time.

# processors	Input	Processing	Radix	Counting	Total
1	0.344631	0.479327	0.601724	62.486	63.5673
4	0.390783	0.738109	0.571868	16.2976	17.7936
9	0.402798	0.878927	0.540968	8.1544	9.93864
16	0.385469	1.0218	0.421714	4.49102	7.94248
25	0.465214	0.950506	0.345385	3.91986	5.60942
36	0.445782	1.14153	0.334108	2.18326	4.62567
49	0.451992	1.16783	0.351549	1.84438	3.74731
64	0.47159	1.34522	0.353155	1.59741	3.68976

Table 3: Time in seconds on gplus graph

# processors	Input	Processing	Radix	Counting	Total
1	0.286515	0.420204	0.357917	34.4083	35.254
4	0.31504	0.543664	0.357283	9.1053	10.1702
9	0.337594	0.700071	0.324966	4.52421	8.84579
16	0.392608	0.86596	0.309142	3.80202	5.3782
25	0.38989	0.786669	0.1913	2.68293	4.01197
36	0.401435	0.932992	0.18826	1.90129	3.26371
49	0.328532	0.968561	0.216887	1.51875	2.77177
64	0.395828	1.18499	0.205708	1.18134	2.85244

Table 4: Time in seconds on k5000 graph

# processors	Input	Processing	Radix	Counting	Total
1	1.30881	1.9437	5.50524	15.8704	23.3085
4	1.41322	2.82458	5.45237	4.53343	14.5233
9	1.45713	3.65775	4.88237	2.28016	12.1036
16	1.38312	3.7568	4.10192	1.3738	11.9798
25	1.63259	4.79692	4.0261	1.14757	17.3923
36	1.56411	7.75282	4.86698	0.785297	19.1065
49	1.71383	13.9677	5.50551	0.980394	25.3576
64	1.71619	11.9125	5.88192	0.713308	24.4834

Table 5: Time in seconds on live graph

# processors	Input	Processing	Radix	Counting	Total
1	0.370143	0.507592	1.02752	10.3336	11.8813
4	0.410203	0.751175	1.2593	2.81684	4.96346
9	0.440556	0.954779	1.19705	1.31154	4.61572
16	0.52436	1.24625	1.34836	1.07794	3.94638
25	0.494116	1.04915	0.85817	0.67643	2.98513
36	0.520116	1.19375	0.915212	0.380698	3.18838
49	0.512794	1.34311	1.12384	0.397659	2.99535
64	0.501132	1.51599	1.04451	0.272483	3.20555

Table 6: Time in seconds on skitter graph

8.2 Code

Below is the code for our MPI implementation.

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cassert>
#include <vector>

#include <mpi.h>

using namespace std;

int proc_id, num_procs;
int num_blocks;

void mpi_init()
{
    // Initialize the MPI environment. The two arguments to MPI Init are not
    // currently used by MPI implementations, but are there in case future
    // implementations might need the arguments.
    MPI_Init(NULL, NULL);

    // Get the number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    // Get the remap of the process
    MPI_Comm_rank(MPI_COMM_WORLD, &proc_id);
}
void mpi_end()
{
    // Finalize the MPI environment. No more MPI calls can be made after this
    MPI_Finalize();
}

int num_V, num_E;
int* edges;
int* degrees;
vector<vector<int> > low_nbr;

int* block;
```

```
#define READ_INT(n) {char c; n = getchar_unlocked() - '0'; while((c = getchar_unlocked()) >= '0')
    n = (n << 3) + (n << 1) + c - '0';}

void input(char* filename)
{
    int block1 = proc_id / num_blocks, block2 = proc_id % num_blocks;

    // ***** INPUT *****

    FILE* file;
    int* input_edges;
    int num_input_edges;
    if(proc_id == 0) // only root reads from file
    {
        freopen(filename, "r", stdin);
        READ_INT(num_V);
        READ_INT(num_input_edges);
    }

    MPI_Bcast(&num_V, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&num_input_edges, 1, MPI_INT, 0, MPI_COMM_WORLD);

    input_edges = new int[2 * num_input_edges];

    if(proc_id == 0)
    {
        for(int i = 0; i < num_input_edges * 2; i++)
            READ_INT(input_edges[i]);
    }

    MPI_Bcast(input_edges, 2 * num_input_edges, MPI_INT, 0, MPI_COMM_WORLD);

    block = new int[num_V];
    for(int i = 0; i < num_V; i++)
        block[i] = i % num_blocks;

    degrees = new int[num_V];
    memset(degrees, 0, num_V * sizeof(int));
    num_E = 0;
    for(int i = 0; i < num_input_edges; i++)
    {
        int u = input_edges[2 * i];
        int v = input_edges[2 * i + 1];
        degrees[u]++;
        degrees[v]++;
        if(block[u] == block1 || block[u] == block2)
            num_E++;
        if(block[v] == block1 || block[v] == block2)
            num_E++;
    }
    edges = new int[num_E * 2];
    int next_E = 0;
    for(int i = 0; i < num_input_edges; i++)
    {
        int u = input_edges[2 * i];
```

```
        int v = input_edges[2 * i + 1];
        if(block[u] == block1 || block[u] == block2)
        {
            edges[next_E++] = u;
            edges[next_E++] = v;

        }
    }
    for(int i = 0; i < num_input_edges; i++)
    {
        int u = input_edges[2 * i];
        int v = input_edges[2 * i + 1];
        if(block[v] == block1 || block[v] == block2)
        {
            edges[next_E++] = v;
            edges[next_E++] = u;
        }
    }

    delete[] input_edges;
}

void radix_sort()
{
    int block1 = proc_id / num_blocks, block2 = proc_id % num_blocks;

    // ***** RADIX SORT *****

    int* remap = new int[num_V];
    int next_id = 0;

    int* radix_sizes = new int[num_V];
    int* radix_next = new int[num_V];
    int** radix = new int*[num_V];
    memset(radix_sizes, 0, num_V * sizeof(int));
    for(int v = 0; v < num_V; v++)
        radix_sizes[degrees[v]]++;
    for(int d = 0; d < num_V; d++)
        if(radix_sizes[d] > 0)
            radix[d] = new int[radix_sizes[d]];
    memset(radix_next, 0, num_V * sizeof(int));
    for(int v = 0; v < num_V; v++)
        radix[degrees[v]][radix_next[degrees[v]]++] = v;

    for(int d = 0; d < num_V; d++)
        for(int i = 0; i < radix_sizes[d]; i++)
        {
            int v = radix[d][i];
            remap[v] = next_id++;
        }
    assert(next_id == num_V);

    int* new_block = new int[num_V];
    for(int v = 0; v < num_V; v++)
        new_block[remap[v]] = block[v];
    delete[] block;
```

```
    block = new_block;

    // reuse radix for another radix sort
    for(int d = 0; d < num_V; d++)
        if(radix_sizes[d] > 0)
            delete[] radix[d];
    memset(radix_sizes, 0, num_V * sizeof(int));

    for(int i = 0; i < num_E; i++)
    {
        int& v = edges[2 * i];
        int& w = edges[2 * i + 1];
        v = remap[v];
        w = remap[w];
    }

    for(int i = 0; i < num_E; i++)
    {
        int w = edges[2 * i], v = edges[2 * i + 1];
        if(v < w)
            radix_sizes[v]++;
    }

    for(int v = 0; v < num_V; v++)
        if(radix_sizes[v] > 0)
            radix[v] = new int[radix_sizes[v]];

    memset(radix_next, 0, num_V * sizeof(int));

    for(int i = 0; i < num_E; i++)
    {
        int w = edges[2 * i], v = edges[2 * i + 1];
        if(v < w)
            radix[v][radix_next[v]++] = w;
    }

    // using vectors for low_nbr seems to work better for num_triangles()
    low_nbr = vector<vector<int> >(num_V);
    for(int v = 0; v < num_V; v++)
        for(int i = 0; i < radix_sizes[v]; i++)
        {
            int w = radix[v][i];
            low_nbr[w].push_back(v);
        }

    delete[] remap;
    for(int v = 0; v < num_V; v++)
        if(radix_sizes[v])
            delete[] radix[v];
    delete[] radix_next;
    delete[] radix_sizes;
}

long long num_triangles()
{
    int block1 = proc_id / num_blocks, block2 = proc_id % num_blocks;
```

```
    // ***** COUNTS NUMBER OF TRIANGLES *****

    double start_time = MPI_Wtime();

    long long counter = 0;
    for(int i = 0; i < num_E; i++)
    {
        int w = edges[2 * i], v = edges[2 * i + 1];
        if(!(v < w && block[v] == block1 && block[w] == block2)) continue;
        int v_ptr = 0;
        int w_ptr = 0;
        while(v_ptr < low_nbr[v].size() && low_nbr[w][w_ptr] < v)
        {
            if(low_nbr[v][v_ptr] < low_nbr[w][w_ptr])
                v_ptr++;
            else if(low_nbr[v][v_ptr] > low_nbr[w][w_ptr])
                w_ptr++;
            else
            {
                counter++;
                v_ptr++;
                w_ptr++;
            }
        }
    }

    return counter;
}

int main(int argc, char** argv)
{
    assert(argc == 2);
    char* filename = argv[1];

    mpi_init();

    double start_time = MPI_Wtime();

    num_blocks = -1;
    for(int i = 1; i <= num_procs; i++)
        if(i * i == num_procs)
            num_blocks = i;
    assert(num_blocks != -1);

    input(filename);

    radix_sort();

    long long num_tri = num_triangles();

    if(proc_id == 0)
    {
        long long total = num_tri;
        for(int proc = 1; proc < num_procs; proc++)
        {
```

```
            long long add = 0;
            MPI_Recv(&add, 1, MPI_LONG_LONG, proc, 0, MPI_COMM_WORLD, NULL);
            total += add;
        }
        printf("***** found %lld triangles in %lf seconds (ignoring input time) *****\n", total,
             MPI_Wtime() - start_time);
    }
    else
        MPI_Send(&num_tri, 1, MPI_LONG_LONG, 0, 0, MPI_COMM_WORLD);

    mpi_end();
}
```

Here is the code for the other sequential algorithm we tried:

```
#include <iostream>
#include <cstdio>
#include <vector>
#include <ctime>
#include <cassert>

using namespace std;

int num_V, num_E;
vector<vector<int> > graph;

double cur_time = 0;
double time_from_last()
{
    double new_time = (double)clock() / CLOCKS_PER_SEC;
    double diff = new_time - cur_time;
    cur_time = new_time;
    return diff;
}

#define READ_INT(n) {char c; n = getchar_unlocked() - '0'; while((c = getchar_unlocked()) >= '0')
     n = (n << 3) + (n << 1) + c - '0';}

void input_and_sort()
{
    // *** INPUT ***
    READ_INT(num_V);
    READ_INT(num_E);
    vector<pair<int, int> > edges(num_E);
    vector<int> degrees(num_V);

    for(int e = 0; e < num_E; e++)
    {
        int u, v;
        READ_INT(u);
        READ_INT(v);
        edges[e] = make_pair(u, v);
        degrees[u]++;
        degrees[v]++;
    }
```

```
    printf("Input took %lf seconds\n", time_from_last());



    // *** RADIX SORT ***
    vector<vector<int> > radix(num_V);
    vector<int> remap(num_V);
    int next_id = 0;

    // radix-sort vertices by degree
    for(int v = 0; v < num_V; v++)
        radix[degrees[v]].push_back(v);

    for(int d = 0; d < num_V; d++)
        for(int i = 0; i < radix[d].size(); i++)
        {
            int v = radix[d][i];
            remap[v] = next_id++;
        }
    assert(next_id == num_V);

    // radix-sort graph adjacency lists
    radix = vector<vector<int> >(num_V);
    for(int e = 0; e < num_E; e++)
    {
        int u = remap[edges[e].first];
        int v = remap[edges[e].second];
        if(u > v) swap(u, v);
        radix[v].push_back(u);
    }

    graph = vector<vector<int> >(num_V);
    for(int v = 0; v < num_V; v++)
        for(int i = 0; i < radix[v].size(); i++)
        {
            int u = radix[v][i];
            graph[u].push_back(v);
        }

    printf("Radix sort took %lf seconds\n", time_from_last());
}

long long num_triangles()
{
    // *** COUNTS NUMBER OF TRIANGLES ***
    vector<vector<int> > edges_to_check(num_V);

    // count triples (u, v, w) with u < v < w
    for(int u = 0; u < num_V; u++)
    {
        for(int i = 0; i < graph[u].size(); i++)
        {
            int v = graph[u][i];
            for(int j = i + 1; j < graph[u].size(); j++)
            {
                int w = graph[u][j];
```

```
                edges_to_check[w].push_back(v);
            }
        }
    }

    long long counter = 0;
    vector<int> next_index(num_V, 0);
    for(int v = 0; v < num_V; v++)
        for(int i = 0; i < edges_to_check[v].size(); i++)
        {
            int u = edges_to_check[v][i];
            while(next_index[u] < graph[u].size() && graph[u][next_index[u]] < v)
                next_index[u]++;
            if(next_index[u] < graph[u].size() && graph[u][next_index[u]] == v)
                counter++;
        }

    return counter;
}

void verify(int result)
{
    int answer;
    if(scanf("%d", &answer) == 1)
    {
        if(result == answer) printf("Correct result!\n");
        else printf("INCORRECT RESULT.\n");
    }
}

int main()
{
    input_and_sort();

    int result = num_triangles();
    printf("Number of triangles is %lld\n", result);
    printf("Triangle counting took %lf seconds\n", time_from_last());
    verify(result);
}
```

References

[1] A. Pavan, Kanat Tangwongsan, Srikanta Tirthapura, Kun-Lung Wu. Counting and Sampling Triangles from a Graph Stream. Proceedings of the VLDB Endowment VLDB Endowment Hompage archive, Volume 6 Issue 14, September 201, Pages 1870-1881.

[2] Matthiou Latapy. Theory and Practice of Triangle Problems in Very Large (Sparse (Power-Law)) Graphs. arXiv:cs/0609116v1. September, 2006.

[3] Irene Finocchi, Marco Finocchi, Emanuele G. Fusco. Clique counting in MapReduce: theory and experiments. arXiv:1403.0734v2. July, 2014.

[4] 15-418 Final Report, Shu-Hao Yu, YiCheng Qin.

`http://www.cs.cmu.edu/afs/cs/user/shuhaoy/www/Final_Project.pdf`.

[5] Stanford Large Network Dataset Collection. https://snap.stanford.edu/data/.

[6] GraphLab. https://dato.com/products/create/open_source.html.