ECE 6130 Big Data and Cloud Computing Spring 2019

Homework #2 BFS Final Report

Name: Tianyu Yang

GW ID: G38878678

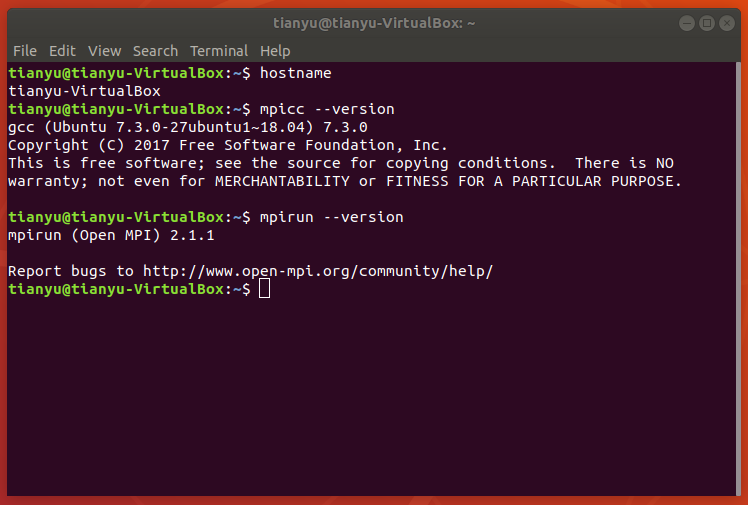
Date: 1/29/2019

Objective:

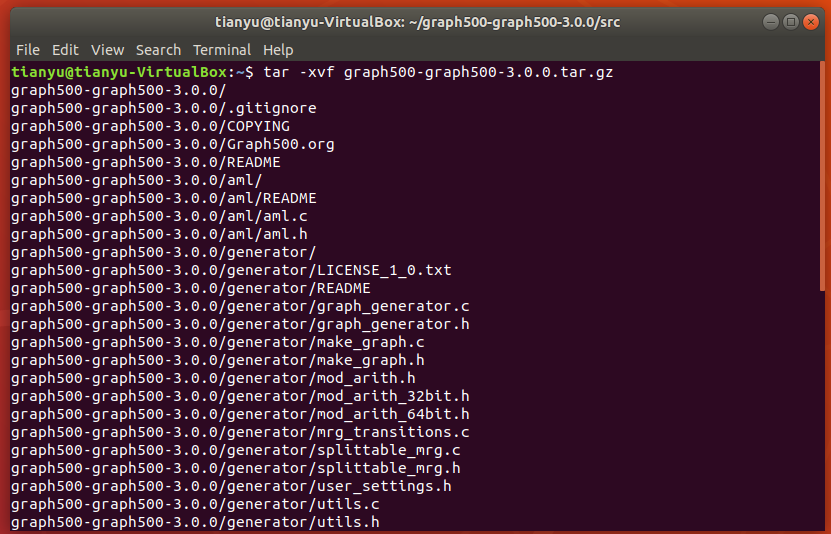
1. include results for graphs with 100, 1K, and 1M edges. Ideally your implementation should at least outperform the reference code by 2x at this point.
2. Must include results for graphs with 1B edges. The target improvement is at least 10x over the reference code.

Part 1: Preparation:

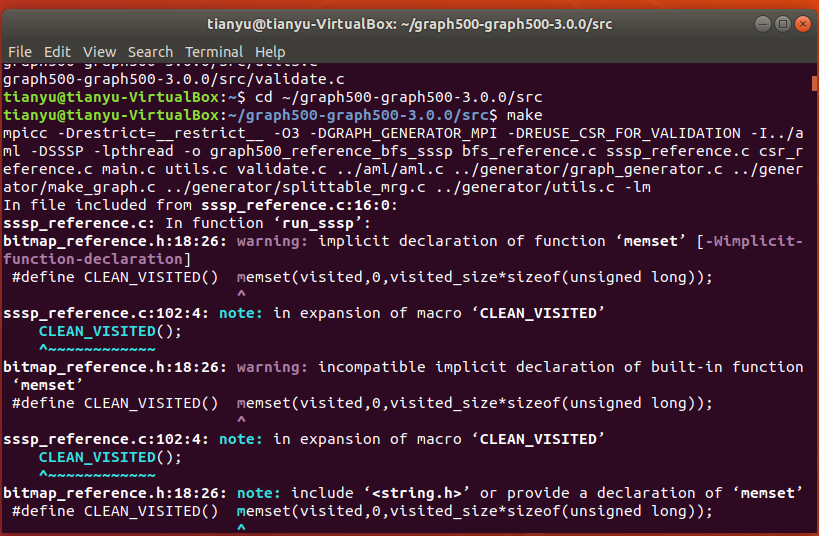
1. Install Linux on my personal workstation, using Oracle VM VirtualBox
2. Install MPI implementation on my Ubuntu installation



1. Download the reference code on Graph 500 website and decompress the tar.gz file to Home folder



1. Use cd command in the terminal to get into src folder and use make command to build the code

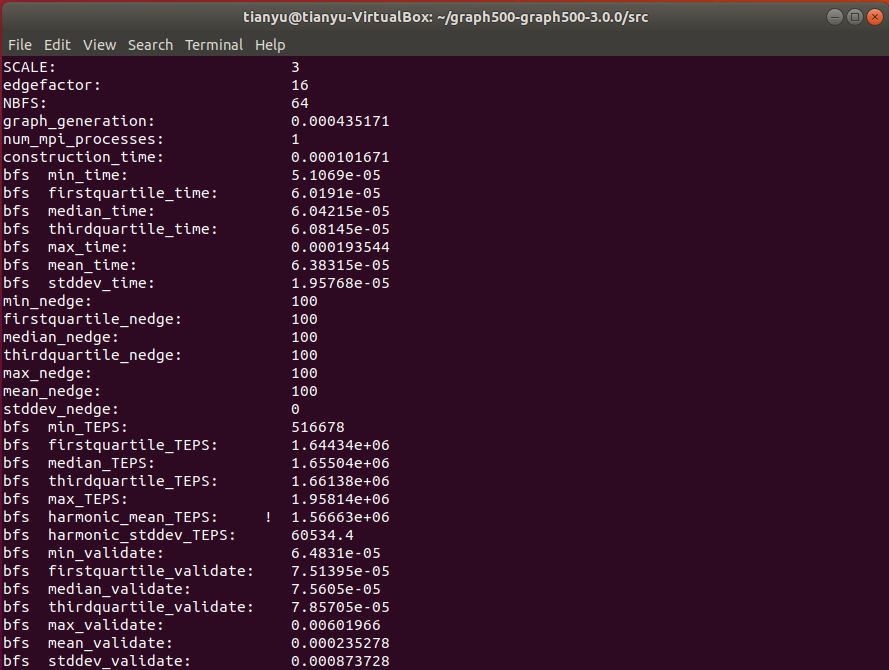


1. Obtain the shared library file for reference\_bfs file and run it in the terminal
2. Adjust the edge value such as 100, 1K, 1M and 1B, and get the results for output performance such as processing time.

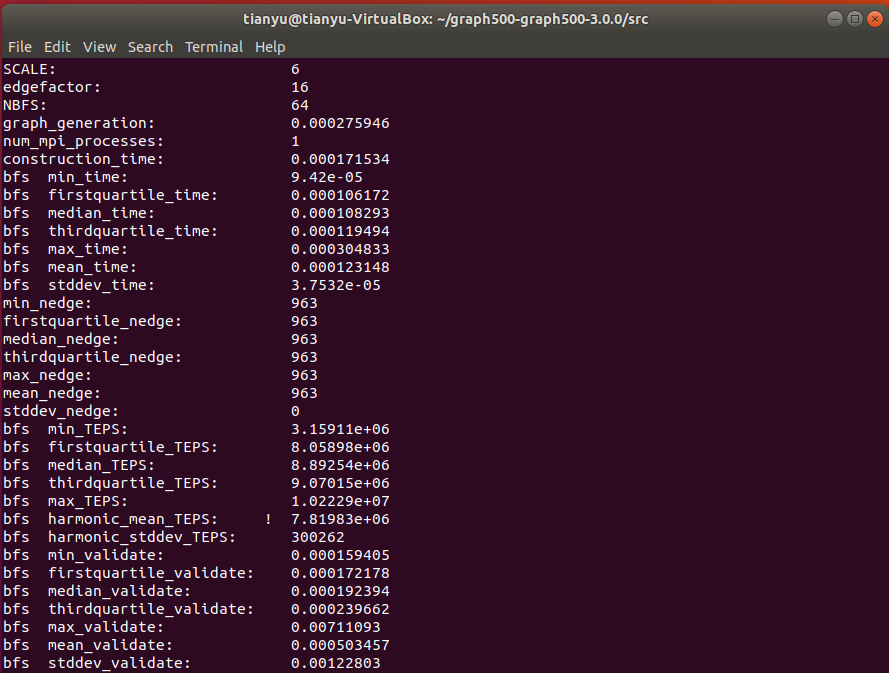
As for the number of edges. M = edgefactor \* N = edgefactor \* 2^SCALE. Therefore, when the edge value is 100, edgefactor = 16, SCALE = 3; when the edge value is 1K, edgefactor = 16, SCALE = 6; when the edge is 1M, edgefactor = 16, SCALE = 16; when the edge is 1B, edgefactor = 16, SCALE = 26;

1. Compare the results and read bfs\_reference.c code

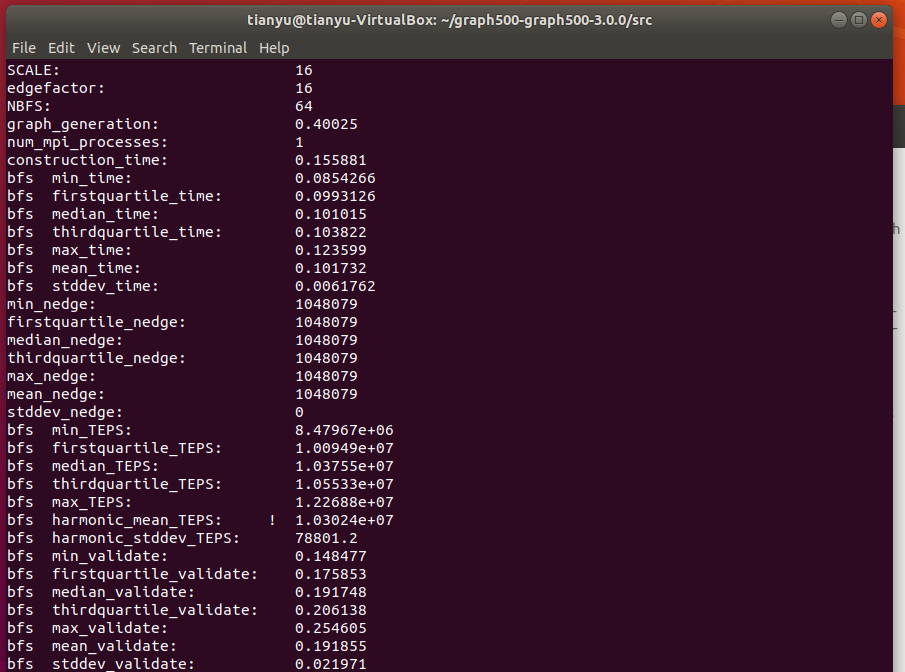
When the edge value is 100:



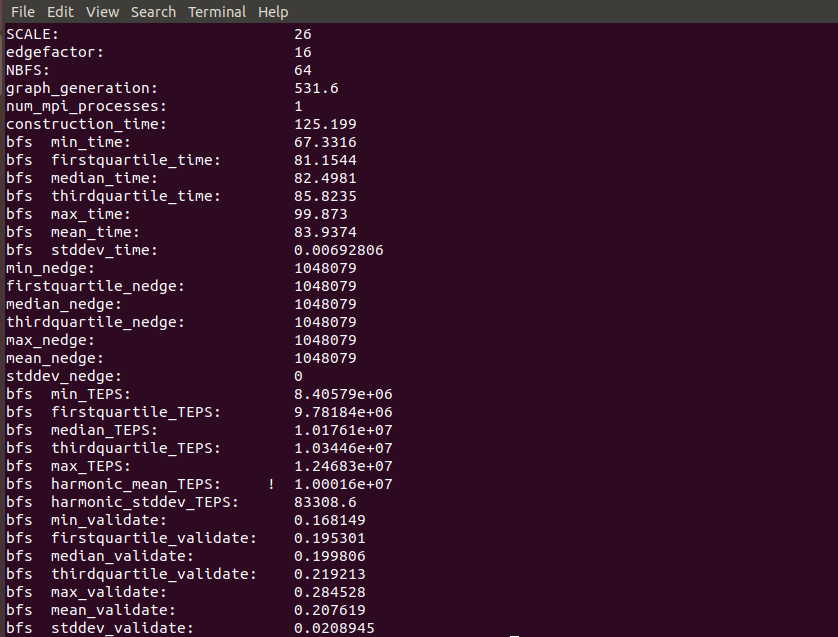
When the edge value is 1K:



When the edge value is 1M:



When the edge value is 1B: (As for 1B edges are too large for a personal computer. Therefore, I use AWS (Amazon Web Services) as platform to run cloud computing code for BFS. I use EC2 (Amazon Elastic Compute cloud) virtual machine and also develop on Linux platform.)



Most related code in bfs\_reference.c

void make\_graph\_data\_structure(const tuple\_graph\* const tg) {

int i,j,k;

convert\_graph\_to\_oned\_csr(tg, &g);

column=g.column;

rowstarts=g.rowstarts;

visited\_size = (g.nlocalverts + ulong\_bits - 1) / ulong\_bits;

aml\_register\_handler(visithndl,1);

q1 = xmalloc(g.nlocalverts\*sizeof(int)); //100% of vertexes

q2 = xmalloc(g.nlocalverts\*sizeof(int));

for(i=0;i<g.nlocalverts;i++) q1[i]=0,q2[i]=0; //touch memory

visited = xmalloc(visited\_size\*sizeof(unsigned long));

}

void run\_bfs(int64\_t root, int64\_t\* pred) {

int64\_t nvisited;

long sum;

unsigned int i,j,k,lvl=1;

pred\_glob=pred;

aml\_register\_handler(visithndl,1);

CLEAN\_VISITED();

qc=0; sum=1; q2c=0;

nvisited=1;

if(VERTEX\_OWNER(root) == rank) {

pred[VERTEX\_LOCAL(root)]=root;

SET\_VISITED(root);

q1[0]=VERTEX\_LOCAL(root);

qc=1;

}

// While there are vertices in current level

while(sum) {

#ifdef DEBUGSTATS

double t0=aml\_time();

nbytes\_sent=0; nbytes\_rcvd=0;

#endif

//for all vertices in current level send visit AMs to all neighbours

for(i=0;i<qc;i++)

for(j=rowstarts[q1[i]];j<rowstarts[q1[i]+1];j++)

send\_visit(COLUMN(j),q1[i]);

aml\_barrier();

qc=q2c;int \*tmp=q1;q1=q2;q2=tmp;

sum=qc;

aml\_long\_allsum(&sum);

nvisited+=sum;

q2c=0;

#ifdef DEBUGSTATS

aml\_long\_allsum(&nbytes\_sent);

t0-=aml\_time();

if(!my\_pe()) printf (" --lvl%d : %lld(%lld,%3.2f) visited in %5.2fs, network aggr %5.2fGb/s\n",lvl++,sum,nvisited,((double)nvisited/(double)g.notisolated)\*100.0,-t0,-(double)nbytes\_sent\*8.0/(1.e9\*t0));

#endif

}

aml\_barrier();

}

Part 2: Improved Code for 100, 1K and 1M edges

1. Modify bfs\_custom.c and main.c file code and rebuild the project by using make command in the terminal

OpenMP method some essential function code:(bfs\_custom.c)

unsigned int \*q1,\*q2;

unsigned int qc,q2c; //pointer to first free element

void make\_graph\_data\_structure(const tuple\_graph\* const tg) {

unsigned int i,j,k;

convert\_graph\_to\_oned\_csr(tg, &g);

column=g.column;

rowstarts=g.rowstarts;

visited\_size = (g.nlocalverts + ulong\_bits - 1) / ulong\_bits;

aml\_register\_handler(visithndl,1);

q1 = xmalloc(g.nlocalverts\*sizeof(int)); //100% of vertexes

q2 = xmalloc(g.nlocalverts\*sizeof(int));

for(i=0;i<g.nlocalverts;++i) q1[i]=0,q2[i]=0; //touch memory

visited = xmalloc(visited\_size\*sizeof(unsigned long));

}

void run\_bfs(int64\_t root, int64\_t\* pred) {

int64\_t nvisited;

long sum;

unsigned int i,j,k,lvl=1;

pred\_glob=pred;

aml\_register\_handler(visithndl,1);

CLEAN\_VISITED();

qc=0; sum=1; q2c=0;

nvisited=1;

if(VERTEX\_OWNER(root) == rank) {

pred[VERTEX\_LOCAL(root)]=root;

SET\_VISITED(root);

q1[0]=VERTEX\_LOCAL(root);

qc=1;

}

// While there are vertices in current level

while(sum) {

#ifdef DEBUGSTATS

double t0=aml\_time();

nbytes\_sent=0; nbytes\_rcvd=0;

#endif

//for all vertices in current level send visit AMs to all neighbours

#pragma omp parallel for{

for(i=0;i<qc;++i)

for(j=rowstarts[q1[i]];j<rowstarts[q1[i]+1];++j)

send\_visit(COLUMN(j),q1[i]);

}

aml\_barrier();

qc=q2c;int \*tmp=q1;q1=q2;q2=tmp;

sum=qc;

aml\_long\_allsum(&sum);

nvisited+=sum;

q2c=0;

#ifdef DEBUGSTATS

aml\_long\_allsum(&nbytes\_sent);

t0-=aml\_time();

if(!my\_pe()) printf (" --lvl%d : %lld(%lld,%3.2f) visited in %5.2fs, network aggr %5.2fGb/s\n",lvl++,sum,nvisited,((double)nvisited/(double)g.notisolated)\*100.0,-t0,-(double)nbytes\_sent\*8.0/(1.e9\*t0));

#endif

}

aml\_barrier();

}

Add mixing MPI and OpenMP code in main.c

#include "mpi.h"

#include <omp.h>

int iam = 0, np = 1;

MPI\_Init(&argc, &argv);

#pragma omp parallel default(shared) private(iam, np)

{

np = omp\_get\_num\_threads();

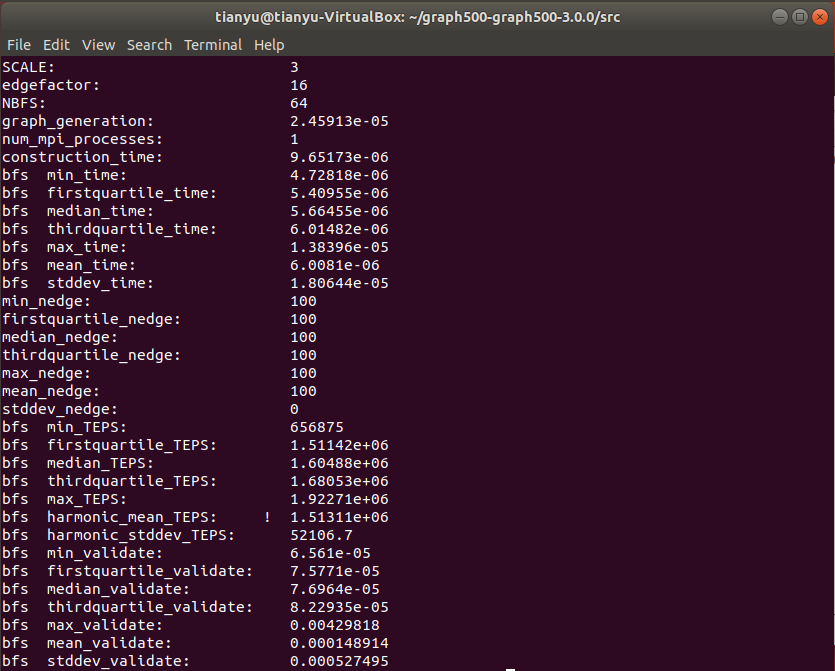
iam = omp\_get\_thread\_num();

run\_bfs(root, &pred[0]);

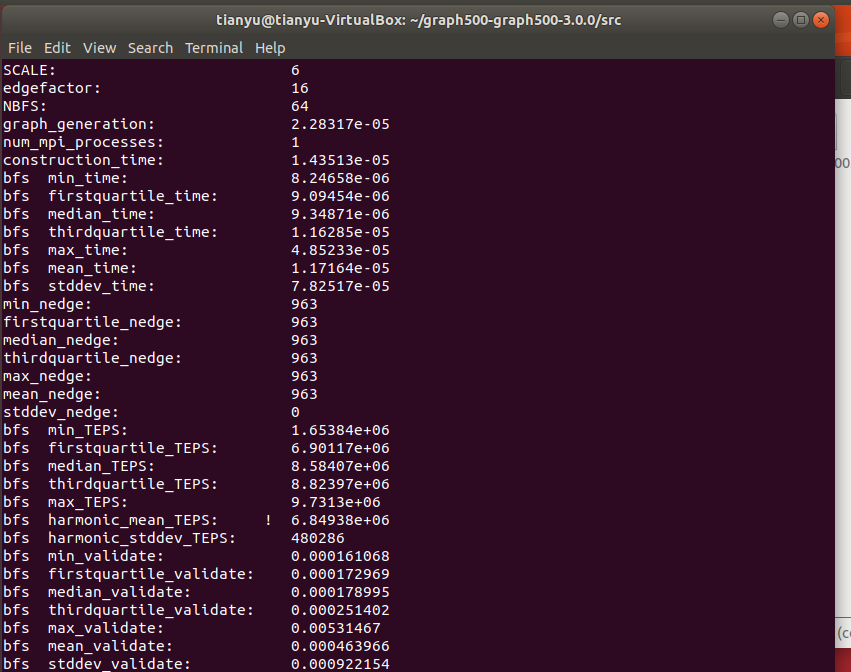
}

1. Adjust the edge value as 100, 1K and 1M and compare the output performance with referenced one

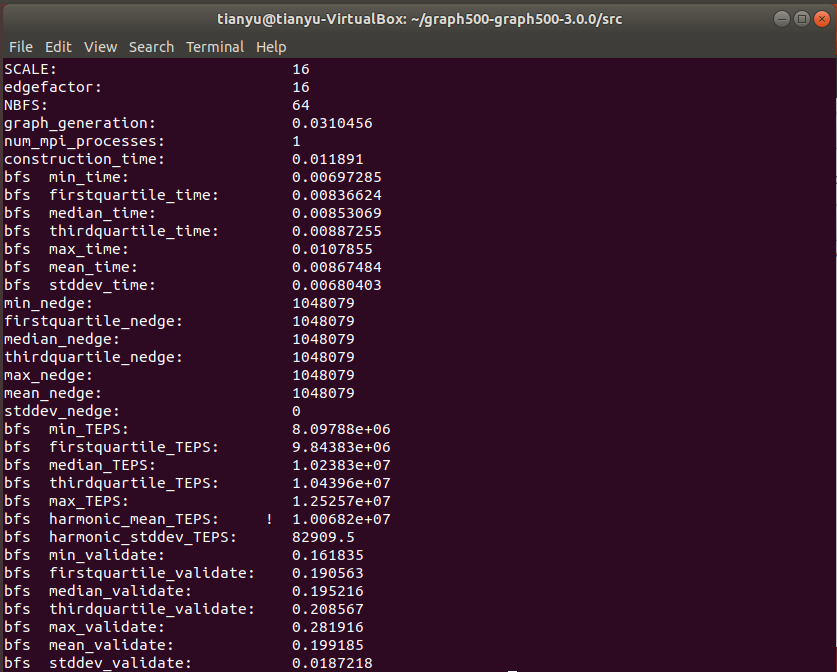
For the edge value of 100:



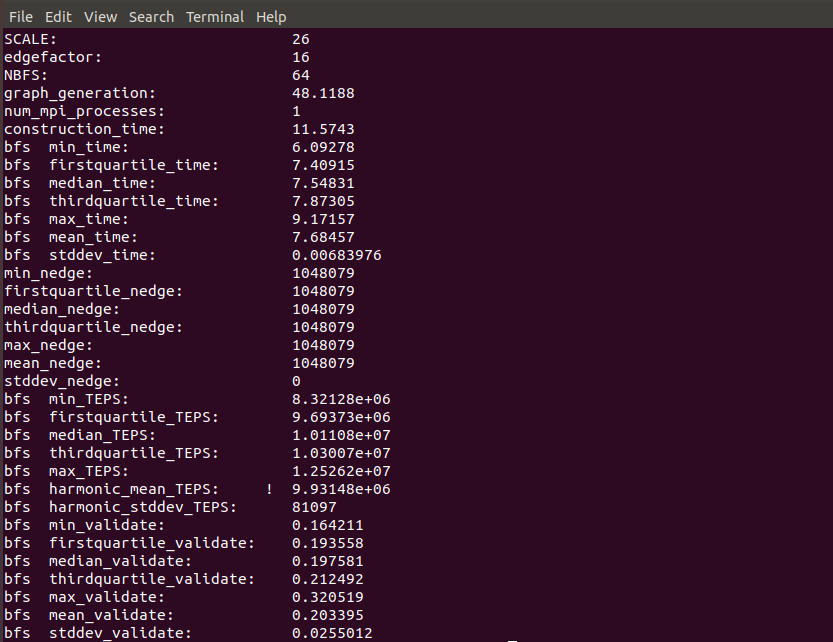
For the edge value of 1K:



For the edge value of 1M:



For the edge value of 1B:



Part 3: Output performance for the high-performance parallel BFS code

|  |  |  |  |
| --- | --- | --- | --- |
| **Edge=100** | **graph\_\_generation** | **construction\_time** | **bfs mean\_time** |
| **Reference code** | 0.000435171 | 0.000101671 | 0.000063815 |
| **Custom code** | 2.45913E-05 | 9.65173E-06 | 6.0081E-06 |
| **Multiple** | 17.69613644 | 10.53396645 | 10.62149432 |
|  |  |  |  |
| **Edge=1K** | **graph\_\_generation** | **construction\_time** | **bfs mean\_time** |
| **Reference code** | 0.000275946 | 0.000171534 | 0.000123148 |
| **Custom code** | 2.28317E-05 | 1.43513E-05 | 1.17164E-05 |
| **Multiple** | 12.08609083 | 11.95250604 | 10.51073709 |
|  |  |  |  |
| **Edge=1M** | **graph\_\_generation** | **construction\_time** | **bfs mean\_time** |
| **Reference code** | 0.40025 | 0.155881 | 0.101732 |
| **Custom code** | 0.0310456 | 0.011891 | 0.00867484 |
| **Multiple** | 12.89232613 | 13.10915819 | 11.727248 |
|  |  |  |  |
| **Edge=1B** | **graph\_\_generation** | **construction\_time** | **bfs mean\_time** |
| **Reference code** | 531.6 | 125.199 | 83.9374 |
| **Custom code** | 48.1188 | 11.5743 | 7.68457 |
| **Multiple** | 11.04765705 | 10.81698245 | 10.92284929 |

Part 4: Result analysis

OpenMP method:

OpenMP (Open Multi-Processing) is an application programming interface that supports multi-platform shared memory multiprocessing programming in C language. It consists of a set of compiler directives, library routines and environment variable that influence run-time behavior. OPENMP is a directory of C examples which illustrate the use of the OpenMP application program interface for carrying out parallel computations in a shared memory environment. The directives allow the user to mark areas of the code, such as do, while or for loops, which are suitable for parallel processing. The directives appear as a special kind of comment, so the program can be compiled and run in serial mode. However, the user can tell the compiler to "notice" the special directives, in which case a version of the program will be created that runs in parallel.

OpenMP includes a number of functions whose type must be declared in any program that uses them. A user program calling OpenMP must have the statement:

***#include <omp.h>***

In this project, I use OPENMP method to decrease run-time of BFS algorithm. This method is to use multi-threads to process the nodes by using CSR format (Compressed sparse row). I use three threads to operate the data and increase the efficiency by 10x at least.

Conclusion

By using the method above like openMP (Multi-threads), I successfully decrease the run-time of BFS by at least 10x for the edges of 100, 1K, 1M and 1B.