qsort\_omp.h:

#ifndef QSORT\_OMP

#define QSORT\_OMP

#include <stdlib.h>

#include <omp.h>

#include "edge.h"

// Number of threads participating in the sorting.

#define NUM\_THREADS 2

void qsort\_omp(edge\* edges, edge\* buffer, size\_t size) {

int N = NUM\_THREADS;

omp\_set\_num\_threads(N);

// "Split" input array into N parts.

int i;

int \*index = malloc((N+1)\*sizeof(int));

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

index[i] = (i\*size)/N;

index[N] = size;

// Each thread will qsort a part.

#pragma omp parallel for private(i)

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

qsort(edges+index[i], index[i+1]-index[i], sizeof(edge), compareEdges);

// Finally, a parallel mergesort merges the sorted parts.

do {

#pragma omp parallel for private(i)

for(i = 0; i < N; i += 2) {

edge\* A = edges+index[i];

edge\* B = edges+index[i+1];

edge\* bfr = buffer+index[i];

int aSize = index[i+1]-index[i];

int bSize = index[i+2]-index[i+1];

int ai=0, bi=0, j=0;

while (ai < aSize && bi < bSize) {

if (A[ai].weight <= B[bi].weight)

bfr[j++] = A[ai++];

else

bfr[j++] = B[bi++];

}

int k;

if (ai < aSize)

for (k = ai; k < aSize; k++) bfr[j++] = A[k];

else

for (k = bi; k < bSize; k++) bfr[j++] = B[k];

}

N /= 2;

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

index[i] = (i\*size)/N;

index[N] = size;

} while(N > 1);

edge \*tmp = edges;

edges = buffer;

buffer = tmp;

}

#endif

disjointset.h:

#ifndef DISJOINT

#define DISJOINT

#include <stdlib.h>

// Disjoint Set (ds) type.

typedef struct dsNodeType {

struct dsNodeType\* parent;

uint32\_t rank;

} dsNode;

// Set that contains all vertices as dsNodes.

dsNode \*dsSet;

// MakeSet operation.

inline void dsMakeSet(uint32\_t nVerts) {

// Callocates (init to 0) room for all vertices.

// parent = NULL and rank = 0.

dsSet = calloc(nVerts, sizeof(dsNode));

}

// Find operation.

inline dsNode\* dsFind(dsNode \*n) {

if (n->parent == NULL) return n;

n->parent = dsFind(n->parent);

return n->parent;

}

// Union operation.

// Assuming that n and m belong to different sets.

inline void dsUnion(dsNode\* n, dsNode\* m) {

if (n->rank < m->rank) {

n->parent = m;

} else if (n->rank > m->rank) {

m->parent = n;

} else {

n->parent = m;

n->rank += 1;

}

}

#endif

edge.h:

#ifndef EDGE

#define EDGE

// Edge type.

typedef struct edgeType {

uint32\_t u;

uint32\_t v;

uint32\_t weight;

} edge;

// Edge comparison.

int compareEdges(const void \*a, const void \*b) {

const edge \*pa = (const edge \*)a;

const edge \*pb = (const edge \*)b;

if (pa->weight > pb->weight)return 1;

else if (pa->weight < pb->weight) return -1;

return 0;

}

#endif

main.c:

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#include <inttypes.h>

#include "edge.h"

#include "disjointset.h"

#include "qsort\_omp.h"

#define VERBOSE 0

FILE \*input = NULL;

// MPI related variables.

MPI\_Datatype mpiEdge; // MPI datatype of edge.

int mpiNp; // Total # of processors.

int mpiRank; // Rank of a processor.

uint32\_t nVerts; // Total # of vertices.

uint32\_t nEdges; // Total # of edges.

edge \*edges; // Contains the edges of a processor.

edge \*edges\_sort\_buffer; // Used for parallel quicksorting of 'edges' for performance.

uint32\_t edgeCount;

edge \*mst; // Contains the MST edges of a processor. Also used as send/recv MPI buffer.

uint32\_t mstEdgeCount;

uint32\_t mstLength;

// Timing variables.

double commTime;

double procTime;

double parseTime;

// Checks if (l <= x < r)

inline int in(uint32\_t x, uint32\_t l, uint32\_t r) {

return ((x >= l) && (x < r));

}

// Finalizes MPI and frees buffers.

inline void finalize() {

MPI\_Type\_free(&mpiEdge);

MPI\_Finalize();

free(edges);

free(mst);

free(dsSet);

}

// Force kill due to initialization error.

inline void die(char \*msg) {

printf("%d: %s\n",mpiRank, msg);

if (input != NULL) fclose(input);

MPI\_Type\_free(&mpiEdge);

MPI\_Finalize();

exit(0);

}

// Initializes MPI and mpiEdge datatype.

inline void initMPI(int argc, char \*\*argv) {

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &mpiRank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &mpiNp);

MPI\_Type\_contiguous(3, MPI\_UINT32\_T, &mpiEdge);

MPI\_Type\_commit(&mpiEdge);

}

// Initializes and validates input.

inline void initInput(char \*filename) {

if ((mpiNp & (mpiNp - 1)) != 0)

die("\* Please make sure that the number of processors is a power of 2\n");

if(filename == NULL || NULL == (input = fopen(filename, "rb")))

die("\* Provide a valid input file argument.\n");

fread(&nVerts, sizeof(uint32\_t), 1, input);

fread(&nEdges, sizeof(uint32\_t), 1, input);

if (nVerts/mpiNp < 2)

die("\* The number of vertices per process must be at least 2.\n");

}

// Parses the edges that correspond to the current processor.

inline void parseEdges() {

uint32\_t vPerProc = nVerts/mpiNp; // Vertices per processor.

uint32\_t firstVert = mpiRank \* vPerProc; // First vertex index.

uint32\_t lastVert = (mpiRank + 1) \* vPerProc; // Last vertex index.

// Last processor case.

if (mpiRank == mpiNp - 1) {

lastVert += nVerts%mpiNp;

vPerProc += nVerts%mpiNp;

}

// Mem allocation.

edges = malloc(vPerProc \* nVerts \* sizeof(edge));

edges\_sort\_buffer = malloc(vPerProc \* nVerts \* sizeof(edge));

// Allocating x2 because of merging. The first part of mst buffer contains the

// MST edges and the second part is used as a receive MPI buffer.

mst = malloc(2 \* (nVerts - 1) \* sizeof(edge));

// Edge parsing.

edgeCount = 0;

uint32\_t i;

edge e;

for (i = 0; i < nEdges; i++) {

fread(&e, sizeof(edge), 1, input);

if (in(e.u, firstVert, lastVert) || in(e.v, firstVert, lastVert)) {

edges[edgeCount++] = e;

}

}

fclose(input);

parseTime = MPI\_Wtime() - parseTime;

// Waiting for all processors to complete parsing.

MPI\_Barrier(MPI\_COMM\_WORLD);

}

// Calculates the MST.

inline void calculateMst() {

// Parallel quicksort.

qsort\_omp(edges, edges\_sort\_buffer, edgeCount);

free(dsSet);

dsMakeSet(nVerts);

// Looping trhough all edges in increasing weight order.

mstEdgeCount = 0;

mstLength = 0;

uint32\_t i = 0;

for(; i < edgeCount; i++) {

// Find parent sets of the nodes.

dsNode \*vParent = dsFind(&dsSet[edges[i].v]);

dsNode \*uParent = dsFind(&dsSet[edges[i].u]);

// If they are from two different sets, merge them.

if(vParent != uParent) {

mst[mstEdgeCount++] = edges[i];

mstLength += edges[i].weight;

dsUnion(vParent, uParent);

}

}

}

int main(int argc, char \*\*argv) {

// Initialization and input file parsing.

initMPI(argc, argv);

parseTime = MPI\_Wtime();

initInput("test\_data.bin");

parseEdges();

// Processing.

double tmpTime;

procTime = MPI\_Wtime();

// In case of 1 processor, act as serial application.

if (mpiNp == 1) {

calculateMst();

} else {

int processors = mpiNp;

int pow2 = 1; // Used to find out where to send/recv.

MPI\_Status mpiStatus;

int recvEdgeCount;

while(processors > 1) {

// Calculate the local MST using 'edges' buffer.

calculateMst();

// Communication part.

if((mpiRank/pow2)%2 != 0) {

// Send from the first half of 'mst'.

MPI\_Send(mst, mstEdgeCount, mpiEdge, (mpiRank-pow2), 0, MPI\_COMM\_WORLD);

break; // Processor did his job and can now exit.

} else {

// Receive into the second half of 'mst'.

tmpTime = MPI\_Wtime();

MPI\_Recv(mst+mstEdgeCount, nVerts-1, mpiEdge, (mpiRank+pow2), 0, MPI\_COMM\_WORLD, &mpiStatus);

MPI\_Get\_count(&mpiStatus, mpiEdge, &recvEdgeCount);

commTime += MPI\_Wtime() - tmpTime; // Communication time is equal to the waiting-to-recv time.

}

// Pointer swap between 'edges' and 'mst'.

// 'edges' will be reused to calculate a new local MST.

// 'mst' will contain that new local MST.

edge \*tmp = edges;

edges = mst;

mst = tmp;

edgeCount = mstEdgeCount + recvEdgeCount; // New edge count after merging.

processors /= 2;

pow2 \*= 2;

}

// Only the root will execute this last calculation.

if(mpiRank == 0) calculateMst();

}

procTime = (MPI\_Wtime() - procTime) - commTime;

if(VERBOSE)

printf("%d: Parse time: %.3fs\n%d: Comm time: %.3fs\n%d: Proc time: %.3fs\n"

, mpiRank, parseTime, mpiRank, commTime, mpiRank, procTime);

if(mpiRank == 0) {

printf("Verex: %d\n", nVerts);

printf("MST length: %d\n", mstLength);

printf("Total Time %.3fs\n", parseTime + commTime + procTime);

printf("Without I/O %.3fs\n", commTime + procTime);

}

finalize();

return 0;

}