```cpp
#include "Chunk.hpp"
#include "Jacoby.hpp"
#include <fstream>
#include <string>
#include <sstream>
#include <utility>
#include "Heat.hpp"
#include <omp.h>

using namespace std;
using namespace heat;

static pair<double, double> result (string filename, Chunk &chunk, double t,
                    const Functor *an, vector<double> *res);

int main (int argc, char *argv[])
{
    size_t N = 512, M = 512;
    string out, decomposition = ".";
    const Functor *Z = &Functor::get("zero1");
    const Functor *E = &Functor::get("edge1");
    const Functor *H = &Functor::get("hole1");
    const Functor *F = &Functor::get("foo1");
    const Functor *analitic = NULL;
    int c;

    double T = 0.03, t_step = -1, t=0;

    MPI_Init(&argc, &argv);
    MPI_Comm comm = MPI_COMM_WORLD;
    int rank, size;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);

    while ( (c = getopt(argc, argv, "n:m:o:d:t:T:Z:E:H:F:a:")) != -1) {
        switch (c){
        case 'n':
            N = atoi(optarg);
            break;
        case 'm':
            M = atoi(optarg);
            break;
        case 'o':
            out = optarg;
            break;
        case 'd':
            decomposition = optarg;
            break;
        case 'Z':
            Z = &Functor::get(optarg);
            break;
        case 'E':
            E = &Functor::get(optarg);
            break;
        case 'H':
            H = &Functor::get(optarg);
            break;
        case 'F':
            F = &Functor::get(optarg);
            break;
        case 't':
            t_step = atof(optarg);
            break;
        case 'T':
            T = atof(optarg);
            break;
        case'a':
            analitic = &Functor::get(optarg);
            break;
        }
    }
    if (t_step < 0) {
        t_step = T*2;
```

```
        t = t_step;
    } else {
        t = 0;
    }

    stringstream lstream;
    lstream << decomposition << "/" << rank;

    fstream local (lstream.str());
    fstream global (decomposition + "/global");

    if (rank == 0) {
        cout <<"Preparing data..." <<endl;
    }
    Jacoby2 jacoby(N, M, *F);
    Chunk chunk(local, global, *Z, *E, *H, jacoby.T());
    MPI_Barrier(MPI_COMM_WORLD);
    if (rank == 0) {
        cout <<"Calculating, using " << omp_get_num_procs()
            << " OpenMP threads." <<endl;
        cout <<"Will stop on T=" << T <<endl;
        if (t_step < T) {
            cout <<"Will generate result on each t=" << t_step <<endl;
        }
    }

    fstream defs;
    if (rank == 0) {
        defs.open((out + "-eps.txt").c_str(), ios_base::out);
    }
    vector<double> results;
    if (!out.empty()) {
        result(out, chunk, jacoby.T(), analitic, NULL);
    };

    double time = MPI_Wtime();
    while (chunk.step()*jacoby.T() < T) {
        bool get_eps = false;
        if (chunk.step()*jacoby.T() > t && !out.empty()){
            get_eps = true;
            const Chunk::Values *v = chunk.result();
            if (v != NULL) {
                results = *v;
            }
            t += t_step;
        }
        chunk.step(jacoby);
        if (rank == 0 && chunk.step() % 100 == 0) {
            cout << "Step=" << chunk.step() << " T=" <<
                chunk.step() *jacoby.T() << endl;
        }
        if (get_eps){
            pair<double, double> def = result(out,
                    chunk, jacoby.T(), analitic, &results);
            if (rank == 0) {
                defs << "Step=" << chunk.step() << " T=" <<
                    chunk.step() *jacoby.T() << " Eps=" << def.first;
                if (analitic != NULL) {
                    defs << " Analytic distance="<< def.second;
                }
                defs << endl;
            }
        }
    }
    time = MPI_Wtime() - time;
    if (rank == 0) {
        cout.precision(5);
        cout <<"Finish on step " << chunk.step() << " with math-time=" <<
            chunk.step() * jacoby.T() << " and run-time=" << time <<
            "s (+/-" << MPI_Wtick() <<"s)"<< endl;
    }

    if (!out.empty()) {
```

```
        result(out, chunk, jacoby.T(), analitic, NULL);
    };

    MPI_Finalize();
    return 0;
}
static pair<double, double> result (string filename, Chunk &chunk, double t,
                    const Functor *an, vector<double> *res)
{
    const Chunk::Values *v = chunk.result();
    static size_t n = 0;
    if (v == NULL) return pair<double, double>(0,0);

    stringstream sname;
    stringstream aname;
    sname << filename << "-" << n <<".ppm";
    aname << filename << "-an-" << n++ <<".ppm";
    fstream f (sname.str().c_str(), ios_base::out);
    f << "P3" << endl;
    f << "# " << filename << endl;
    f << chunk.N() << " " << chunk.M() << endl;
    f << 255 << endl;
    fstream a;
    if (an != NULL) {
        a.open(aname.str(), ios_base::out);
        a << "P3" << endl;
        a << "# " << filename << endl;
        a << chunk.N() << " " << chunk.M() << endl;
        a << 255 << endl;
    }

    double min = NAN, max = NAN;
    min = 0;
    max = 2;

    size_t i = 0;
    double d_eps = 0, d_an = 0;
    for (Chunk::Values::const_iterator it = v->begin();
            it != v->end(); ++it, ++i) {
        double v = *it;
        v = (v - min)/(max - min);
        f << R(v) << " " << G(v) << " " << B(v) << endl;
        if (an != NULL){
            pair<double,double> pos= Vertex::dpos(i, chunk.N(), chunk.M());
            double v_an = (*an)(pos.first, pos.second, chunk.step()*t);
            if (!isnan(v)) {
                d_an = std::max(d_an, fabs(v_an - *it));
            }
            v_an = (v_an - min)/(max - min);
            a << R(v_an) << " " << G(v_an) << " " << B(v_an) << endl;
        }
        if (res != NULL && res->size() > 0) {
            if (!isnan(v) && !isnan((*res)[i]) ) {
                d_eps += square((*res)[i] - *it);
            }
        }
    }
    f.close();
    if (an != NULL) {
        a.close();
    }
    cout.precision(5);
    return pair<double, double>(sqrt(d_eps), d_an);
}
```

```cpp
#include "Chunk.hpp"

#include <algorithm>
#include <iostream>

using namespace std;

Chunk::Chunk(istream &chunkfile, istream &fullfile,
                const Functor &zero, const Functor &edge, const Functor &hole,
                double t)
    : m_step(0)
    , f_edge(edge)
    , f_hole(hole)
    , f_zero(zero)
    , m_T(t)
    , m_none_border(0)
{
    size_t index;
    chunkfile >> m_rank;
    fullfile >> m_N;
    fullfile >> m_M;
    fullfile >> m_chunks;

    ItMapping mapping(m_N*m_M,m_vertecies.end());
    while(!chunkfile.eof()) {
        chunkfile >> index;
        VertexSet::iterator it = m_vertecies.insert(
                Vertex(index, m_N, m_M, m_step)).first;
        it->set(f_zero(it->x(), it->y(), 0));
        it->set(it->get(0), 1);
        mapping[index] = it;
    }
    m_decomposition.resize(m_N*m_M);
    for (size_t i = 0; (i < m_N*m_M) && !fullfile.eof(); ++i) {
        fullfile >> m_decomposition[i];
    }
    connect(m_decomposition, mapping);
    if (m_rank != 0) {
        m_decomposition.resize(0);
    }
    for (VertexSet::iterator it = m_vertecies.begin();
            it != m_vertecies.end(); ++it)
    {
        m_iterators.push_back(it);
    }
    sort(m_iterators.begin(), m_iterators.end(), vertex_comporator);
    for (NeighbourSet::iterator it = m_neighbors.begin();
            it != m_neighbors.end(); ++it)
    {
        m_it_neighbors.push_back(it);
    }
}
void Chunk::connect(const std::vector<int> &decomposition, const ItMapping &map)
{
    for (VertexSet::iterator it = m_vertecies.begin(); it != m_vertecies.end();
            ++it)
    {
        connect (decomposition, map, *it);
        if (it->type() ==it->I) {
            m_none_border++;
        }
    }
}
void Chunk::connect(const Mapping &decomposition, const ItMapping &map,
        const Vertex &v)
{
    v.set(f_zero(v.x(), v.y(), 0));
    v.set(f_zero(v.x(), v.y(), 0), 1);
    connect (decomposition, map, v, v.i() - 1, v.j(), Vertex::Left);
    connect (decomposition, map, v, v.i() + 1, v.j(), Vertex::Right);
    connect (decomposition, map, v, v.i(), v.j() - 1, Vertex::Top);
    connect (decomposition, map, v, v.i(), v.j() + 1, Vertex::Bottom);
}
```

```
void Chunk::connect(const std::vector<int> &decomposition, const ItMapping &map,
        const Vertex &v, size_t i, size_t j, Vertex::Direction d)
{
    size_t index = Vertex::index(i,j,m_N,m_M);
    int rank = decomposition[index];

    if (rank < 0) {
        connect_edge(v, d, index, rank);
    } else {
        if (rank != m_rank) {
            connect_neighbor(v, d, index, rank);
        } else {
            connect_inner(v, d, index, map);
        }
    }
}
void Chunk::connect_edge(const Vertex &v, Vertex::Direction d, size_t index,
        int rank)
{
    CondVertexSet::iterator it = m_hole.insert(
            CondVertex<Functor>(index, m_N, m_M, m_step,m_T,
            rank < -1 ? f_hole : f_edge,
            rank < - 1 ? Vertex::Hole : Vertex::Edge)).first;
    if (it == m_hole.end()) {
        exit(1);
    }
    v.set(*it, d);
    it->set(v, Vertex::reverse(d));
}
void Chunk::connect_neighbor(const Vertex &v, Vertex::Direction d, size_t index,
        int rank)
{
    NeighbourSet::iterator it = m_neighbors.insert(Neighbour(*this, rank,
                f_zero)).first;
    if (it == m_neighbors.end()) {
        exit(1);
    }
    it->add(index, &v, d);
}
void Chunk::connect_inner(const Vertex &v, Vertex::Direction d, size_t index,
        const ItMapping &map)
{
    VertexSet::iterator it = map[index];
    if (it == m_vertecies.end()) exit(1);
    v.set(*it, d);
    it->set(v, Vertex::reverse(d));
}
size_t Chunk::N() const
{
    return m_N;
}
size_t Chunk::M() const
{
    return m_M;
}
size_t &Chunk::step() const
{
    return m_step;
}
const Chunk::Values *Chunk::result()
{
    if (m_rank != 0) {
        result_slave();
        return NULL;
    }
    result_master();
    return &m_result;
}
void Chunk::result_slave()
{
    Values vals;
    result (vals);
```

```
    MPI_Send(vals.data(), vals.size(), MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
}
void Chunk::result_master()
{
    typedef pair<size_t, Values>ResPair;
    vector<ResPair> rcv(m_chunks, ResPair(0,Values()));
    m_result.resize(m_N*m_M);
    for (size_t i = 0; i < m_decomposition.size(); ++i) {
        int rank = m_decomposition[i];
        if (rank < 0) continue;
        rcv[rank].first++;
    }
    for (int i = 0; i < (int)m_chunks; ++i) {
        rcv[i].second.resize(rcv[i].first, 0);
        rcv[i].first = 0;
        if (i == m_rank) {
            result (rcv[i].second);
            continue;
        }
        result_master(rcv[i].second, i);
    }
    for (size_t i = 0; i < m_decomposition.size(); ++i) {
        int rank = m_decomposition[i];
        if (rank< 0) {
            m_result[i] = NAN;
            continue;
        }
        m_result[i] = rcv[rank].second[rcv[rank].first];
        rcv[rank].first++;
    }
}
void Chunk::result_master(Values &vals, int rank)
{
    double buf [vals.size()];
    memset (buf, 0, sizeof (buf));
    if (MPI_Recv(buf, vals.size(), MPI_DOUBLE, rank, 1, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE) < 0)
    {
        exit(1);
    }
    for (size_t i=0; i<vals.size(); ++i) {
        vals[i] = buf[i];
    }
}
void Chunk::result (Values &vals)
{
    vals.resize(m_vertecies.size());
    size_t i=0;
    for (VertexSet::iterator it = m_vertecies.begin();
            it != m_vertecies.end(); ++i, ++it)
    {
        vals[i] = *it;
    }
}
bool Chunk::vertex_comporator(VertexSet::iterator &v1,
            VertexSet::iterator &v2)
{
    if (v1->type() == Vertex::B && v2->type() != Vertex::B) {
        return true;
    }
    if (v2->type() == Vertex::B && v1->type() != Vertex::B) {
        return false;
    }
    return v1->distance() < v2->distance();
}
```

```
#ifndef _CHUNK_HPP_
#define _CHUNK_HPP_

#include <istream>
#include <vector>
#include <set>
#include <omp.h>

#include "Vertex.hpp"
#include "Neighbour.hpp"
#include "Params.hpp"

class Chunk {
public:
    typedef std::set<Vertex> VertexSet;
    typedef std::set<Neighbour> NeighbourSet;
    typedef std::set<CondVertex<Functor> > CondVertexSet;
    typedef std::vector<double> Values;
    typedef std::vector<int> Mapping;
    typedef std::vector<VertexSet::iterator> ItMapping;
    typedef std::vector<VertexSet::iterator> Iterators;
    typedef std::vector<NeighbourSet::iterator> Neighbours;
public:
    Chunk (std::istream &chunkfile, std::istream &fullfile,
            const Functor &zero, const Functor &edge, const Functor &hole,
            double t);

    template <typename F>
    size_t step(const F &f)
    {
        int neighbors = m_neighbors.size();
        ++m_step;
#pragma omp parallel
        {
#pragma omp for
            for (int i = 0; i<(int)m_none_border; ++i){
                VertexSet:: iterator &it = m_iterators[i];
                if (neighbors  > 0 &&
                        omp_get_thread_num() < (int)m_neighbors.size() &&
                        i % 8 == 0)
                {
#pragma omp critical
                    if (m_it_neighbors[omp_get_thread_num()]->try_step(f)) {
                        --neighbors;
                    }
                }
                it->set(f(*it, *it->top(), *it->right(), *it->bottom(),
                            *it->left(), m_step, it->x(), it->y()));
            }
        }
        if (neighbors > 0) for (NeighbourSet::iterator it = m_neighbors.begin();
                it != m_neighbors.end(); ++it)
        {
            it->step(f);
        }
        return m_step;
    }
    size_t N() const;
    size_t M() const;
    size_t &step() const;

    const Values *result();
private:
    void connect(const Mapping &decomposition, const ItMapping &);
    void connect(const Mapping &decomposition, const ItMapping &,
            const Vertex &v);
    void connect(const Mapping &decomposition, const ItMapping &,
            const Vertex &v,
            size_t i, size_t j, Vertex::Direction d);
    void connect_edge(const Vertex &v, Vertex::Direction, size_t index,
                      int rank);
    void connect_neighbor(const Vertex &v, Vertex::Direction, size_t index,
                      int rank);
```

```
    void connect_inner(const Vertex &v, Vertex::Direction, size_t index,
            const ItMapping &);
    void result_slave();
    void result_master();
    void result_master(Values &vals, int rank);
    void result (Values &vals);
    static bool vertex_comporator(VertexSet::iterator &v1,
            VertexSet::iterator &v2);
private:
    size_t m_N, m_M, m_chunks;
    int m_rank;
    mutable size_t m_step;
    VertexSet m_vertecies;
    NeighbourSet m_neighbors;
    CondVertexSet m_hole;
    const Functor &f_edge, &f_hole, &f_zero;
    double m_T;
    Values m_result;
    Mapping m_decomposition;
    Iterators m_iterators;
    size_t m_none_border;
    Neighbours m_it_neighbors;
};

#endif /* Chunk.hpp */
```

```cpp
#include <unistd.h>
#include "Decompositor.hpp"
#include "Params.hpp"

int main (int argc, char *argv[])
{
    Decompositor d;
    char c;

    while ( (c = getopt(argc, argv, "n:m:p:f:o:P:")) != -1) switch (c){
    case 'n':
        d.N(atoi(optarg));
        break;
    case 'm':
        d.M(atoi(optarg));
        break;
    case 'f':
    case 'o':
        d.path(optarg);
        break;
    case 'p':
        d.proc(atoi(optarg));
        break;
    case 'P':
        d.hole(HoleParams::get(optarg)->hole());
        break;

    }

    return d.run();
}
```

```
#ifndef _DECOMPOSITOR_HPP_
#define _DECOMPOSITOR_HPP

#include "Hole.hpp"
#include "Matrix.hpp"
#include "Heat.hpp"
#include <string>
#include <cstring>
#include <metis.h>
#include <fstream>
#include <sstream>
#include <iostream>
#include <set>
#include <map>
#include <algorithm>

class Decompositor {

public:
    typedef Matrix<double>::position position;
public:
    Decompositor()
        : m_hole(new Holes())
        , m_N(256), m_M(256)
        , m_path(".")
        , m_proc(1)
    {}
    ~Decompositor()
    {
        delete m_hole;
    }
    void hole(const Hole *hole)
    {
        if (m_hole != NULL) {
            Holes *h = new Holes (m_hole, hole);
            delete m_hole;
            m_hole = h;
        } else {
            m_hole = hole == NULL ? NULL : hole->copy();
        }
    }
    void N(size_t in_N)
    {
        m_N = in_N;
    }
    void M(size_t in_M)
    {
        m_M = in_M;
    }
    void dimetion (size_t N, size_t M)
    {
        m_N = N;
        m_M = M;
    }
    template <typename T>
    void path (T path)
    {
        m_path = path;
    }
    void proc (size_t proc)
    {
        m_proc = proc;
    }
    bool hole (position p) const
    {
        bool h = m_hole->contains(((double)p.first)/m_N,
                    ((double)p.second)/m_M);
        return h || p.first == 0 || p.second == 0 ||
            p.first == (ssize_t)m_N - 1 ||
            p.second == (ssize_t)m_M -1;
    }
    int run() const
    {
```

```
        idx_t nvtxs = m_N * m_M;
        idx_t ncon = 1;
        idx_t nedjs = nvtxs * 2 - m_N - m_M;
        idx_t *xadj = new idx_t [nvtxs+1];
        idx_t *vwgt = new idx_t [nvtxs];
        idx_t *adjncy = new idx_t [2*nedjs];
        idx_t nparts = m_proc;
        idx_t o_objval;
        idx_t *o_part = new idx_t [nvtxs];

        xadj[0] = 0;
        for (size_t i=0, j = 0; i< m_N*m_M; ++i) {
            position p = pos(i), n_p;
            if (this->hole(p)) {
                xadj[i+1]  = j;
                vwgt[i] = 0;
                continue;
            }
            ssize_t nb;
            if (( nb = index(position(p.first - 1, p.second))) >= 0) {
                adjncy[j] = nb;
                ++j;
            }
            if (( nb = index(position(p.first + 1, p.second))) >= 0) {
                adjncy[j] = nb;
                ++j;
            }
            if (( nb = index(position(p.first, p.second - 1))) >= 0) {
                adjncy[j] = nb;
                ++j;
            }
            if (( nb = index(position(p.first, p.second + 1))) >= 0) {
                adjncy[j] = nb;
                ++j;
            }
            xadj[i+1]  = j;
            vwgt[i] = 1;
        }

#if 0
        std::cout << "xadj[";
        for (idx_t i = 0; i <= nvtxs; ++i) {
            std::cout << xadj[i] << ",";
        }
        std::cout << "]" <<std::endl;
        std::cout << "adjncy[";
        for (idx_t i = 0; i < 2*nedjs; ++i) {
            std::cout << adjncy[i] << ",";
        }
        std::cout << "]" << std::endl;
#endif

        std::cout << "Running decompositor with nvtxs=" << nvtxs << " nedjs=" <<
            nedjs << " nparts=" << nparts << std::endl;
        int result;
        if (m_proc > 1) {
            result = METIS_PartGraphKway(&nvtxs, &ncon, xadj, adjncy, vwgt,
                     NULL, NULL, &nparts, NULL, NULL, NULL, &o_objval, o_part);
        } else {
            memset(o_part, 0, sizeof (idx_t) * nvtxs);
            result = METIS_OK;
        }
        std::cout << "Decomposition done with result='" << str_result(result)
            <<"' (" << result << ")" << std::endl;
        if (result != METIS_OK) return result;

        std::fstream global (m_path + "/global", std::ios_base::out);
        std::fstream ppm (m_path + "/map.ppm", std::ios_base::out);
        global <<  m_N << " "<< m_M << " " << m_proc << std::endl;
        ppm << "P3" << std::endl;
        ppm << "# " << "map.ppm" << std::endl;
        ppm << m_N << " " << m_M << std::endl;
        ppm << 255 << std::endl;
```

```
        std::fstream ranks[m_proc];

        std::vector <std::pair<idx_t, size_t> > sort_chunks;
        for (size_t i = 0; i < m_proc; ++i) {
            sort_chunks.push_back(std::pair<idx_t, size_t>(i, m_M*m_N + 10));
        }
        for (size_t i = 0; i < (size_t)nvtxs; ++i) {
            position p = pos(i);
            if (!hole(p)) {
                size_t v = sort_chunks[o_part[i]].second;
                sort_chunks[o_part[i]] =
                    std::pair<idx_t, size_t>(o_part[i], std::min(i,v));
            }
        }
        std::sort(sort_chunks.begin(), sort_chunks.end(), [](
                    const std::pair<idx_t, size_t> &a,
                    const std::pair<idx_t, size_t> &b) -> int {
                return a.second < b.second;
                });
        std::map <idx_t, idx_t> sort_map;
        for (size_t i = 0; i < m_proc; ++i) {
            sort_map[sort_chunks[i].first] = i;
        }

        for (size_t i=0; i< m_proc; ++i) {
            std::stringstream name;
            name << m_path << "/" << i;
            ranks[i].open(name.str(), std::ios_base::out);
            ranks[i] << i << std::endl;
        }
        for (size_t i = 0; i < (size_t)nvtxs; ++i) {
            position p = pos(i);
            if (hole(p)) {
                if (m_hole->contains(((double)p.first)/m_N,
                    ((double)p.second)/m_M))
                {
                    global << -2 << std::endl;
                } else {
                    global << -1 << std::endl;
                }
                ppm << 255 << " " << 255 << " " << 255 << std::endl;
            } else {
                global << sort_map[o_part[i]] << std::endl;
                ranks [sort_map[o_part[i]]] << i << std::endl;
                ssize_t n1 =index(position(p.first - 1, p.second));
                ssize_t n2 =index(position(p.first + 1, p.second));
                ssize_t n3 =index(position(p.first, p.second - 1));
                ssize_t n4 =index(position(p.first, p.second + 1));
                if (n1 < 0 || n2 < 0 || n3 < 0 || n4 < 0 ||
                    sort_map[o_part[i]] != sort_map[o_part[n1]] ||
                    sort_map[o_part[i]] != sort_map[o_part[n2]] ||
                    sort_map[o_part[i]] != sort_map[o_part[n3]] ||
                    sort_map[o_part[i]] != sort_map[o_part[n4]])
                {
                    ppm << 0 << " " << 0 << " " << 0 << std::endl;
                } else {
                    double v = sort_map[o_part[i]];
                    v /= m_proc;
                    ppm << heat::R(v) << " " << heat::G(v) << " " << heat::B(v) << s
td::endl;
                }
            }
        }
        for (size_t i=0; i< m_proc; ++i) {
            ranks[i].close();
        }
        global.close();

        delete [] xadj;
        delete [] adjncy;
        delete [] o_part;
        return 0;
    }
```

```
    static const std::string &str_result(int result)
    {
        static const std::string ok ("METIS_OK");
        static const std::string in ("METIS_ERROR_INPUT");
        static const std::string mem ("METIS_ERROR_MEMORY");
        static const std::string er ("METIS_ERROR");
        static const std::string none ("NONE");

        switch (result) {
        case METIS_OK:
            return ok;
        case METIS_ERROR_INPUT:
            return in;
        case METIS_ERROR_MEMORY:
            return mem;
        case METIS_ERROR:
            return er;
        default:
            return none;
        }
    }
    position pos(size_t index) const
    {
        position res = {index / m_M, index %m_M};
        return res;
    }
    ssize_t index (const position &pos) const
    {
        if (hole(pos)) {
            return -1;
        }
        if (pos.first < 0 || pos.second < 0 || pos.first >= (ssize_t)m_N ||
                pos.second >= (ssize_t)m_M)
        {
            return -1;
        }
        size_t res = pos.first * m_M + pos.second;
        return res;
    }
    void color (int val, int c[3]) const
    {
        if (val < 0) {
            c[0] = c[1] = c[2] = 255;
            return;
        }
        c[0] = c[1] = c[2] = 16;
        for (size_t i = 0; i<m_proc/3; ++i) {
            c[val % 3] += 1;
            val /= 3;
        }
        double mul = 255. / m_proc;
        c[0]=(c[0]*mul + m_proc/4);
        c[1]=(c[1]*mul + m_proc/4);
        c[2]=(c[2]*mul + m_proc/4);
        c[0] %= 255;
        c[1] %= 255;
        c[2] %= 255;
    }
private:
    Hole *m_hole;
    size_t m_N, m_M;
    std::string m_path;
    size_t m_proc;
};


#endif /* Decompositor.hpp */
```

```cpp
#include "Exchanger.hpp"
#include <cstdlib>
#include <utility>

#include <iostream>
Exchanger::~Exchanger()
{
    for (size_t i=0; i<size(); ++i) {
        delete m_bottom[i];
        delete m_top[i];
        delete m_corner[i];
        delete m_left[i];
        delete m_right[i];
    }
}
Exchanger::net &Exchanger::at(size_t i)
{
    return Step::at(i);
}
const Exchanger::net &Exchanger::at(size_t i) const
{
    return Step::at(i);
}
double Exchanger::at(size_t i, size_t j, size_t l) const
{
    position pos = local(i, j);

    if (pos.first < 0) {
        if(pos.second < 0) {
            return corner(TL, abs(pos), l);
        } else if (pos.second < (ssize_t)Hj()) {
            return top(abs(pos), l);
        } else {
            return corner(TR, abs(pos), l);
        }
    } else if (pos.first < (ssize_t)Hi()) {
        if(pos.second < 0) {
            return left(abs(pos), l);
        } else if (pos.second < (ssize_t)Hj()) {
            return Step::at(l).at(pos);
        } else {
            return right(abs(pos), l);
        }
    } else {
        if(pos.second < 0) {
            return corner(BL, abs(pos), l);
        } else if (pos.second < (ssize_t)Hj()) {
            return bottom(abs(pos), l);
        } else {
            return corner(BR, abs(pos), l);
        }
    }
}
double Exchanger::at(size_t i, size_t j, size_t l)
{
    return ((const Exchanger *)this)->at(i,j, l);
}

double Exchanger::at(const position &pos, size_t l) const
{
    return Exchanger::at(pos.first, pos.second, l);
}
double Exchanger::at(const position &pos, size_t l)
{
    return Exchanger::at(pos.first, pos.second, l);
}

Exchanger::edge_t &Exchanger::top(size_t t)
{
    return *m_top [t];
}
Exchanger::edge_t &Exchanger::bottom(size_t t)
{
```

```
    return *m_bottom [t];
}
Exchanger::edge_t &Exchanger::left(size_t t)
{
    return *m_left [t];
}
Exchanger::edge_t &Exchanger::right(size_t t)
{
    return *m_right [t];
}
Exchanger::corner_t &Exchanger::corner(size_t t)
{
    return *m_corner [t];
}
const Exchanger::edge_t &Exchanger::top(size_t t) const
{
    return *m_top [t];
}
const Exchanger::edge_t &Exchanger::bottom(size_t t) const
{
    return *m_bottom [t];
}
const Exchanger::edge_t &Exchanger::left(size_t t) const
{
    return *m_left [t];
}
const Exchanger::edge_t &Exchanger::right(size_t t) const
{
    return *m_right [t];
}
const Exchanger::corner_t &Exchanger::corner(size_t t) const
{
    return *m_corner [t];
}
double Exchanger::top(const position &pos, size_t l) const
{
    return top(l)[pos.first][pos.second];
}
double Exchanger::bottom(const position &pos, size_t l) const
{
    return bottom(l)[pos.first][pos.second];
}
double Exchanger::left(const position &pos, size_t l) const
{
    return left(l)[pos.second][pos.first];
}
double Exchanger::right(const position &pos, size_t l) const
{
    return right(l)[pos.second][pos.first];
}
double Exchanger::corner(CType c, const position &pos, size_t l) const
{
    return corner(l)[c][pos];
}
Exchanger::position Exchanger::abs(const position &pos) const
{
    position res = pos;
    if (res.first < 0) {
        res.first = -1 - res.first;
    } else while (res.first >= (ssize_t)Hi()) {
        res.first -= Hi();
    }
    if (res.second < 0) {
        res.second = -1 - res.second;
    } else while (res.second >= (ssize_t)Hj()) {
        res.second -= Hj();
    }
    return res;
}
void Exchanger::v_next()
{
    size_t s = size() - 1;
    auto t = m_top[s];
```

```
    auto b = m_bottom[s];
    auto l = m_left[s];
    auto r = m_right[s];
    auto c = m_corner[s];

    for (size_t i = size() - 1; i >0; --i) {
        m_top[i] = m_top[i-1];
        m_bottom[i] = m_bottom[i-1];
        m_right[i] = m_right[i-1];
        m_left[i] = m_left[i-1];
        m_corner[i] = m_corner[i-1];
    }
    Step::v_next();

    m_top[0] = t;
    m_bottom[0] = b;
    m_left[0] = l;
    m_right[0] = r;
    m_corner[0] = c;
}
```

```
/*
 * Class, represents the thread which interacts with other threads via
 * exchages.
 */

#ifndef _EXCHANGER_HPP_
#define _EXCHANGER_HPP_

#include "Step.hpp"
#include <array>

class Exchanger : public Step {
public:
    typedef typename Step::matrix matrix;
    typedef typename Step::line line;
    typedef typename Step::raw raw;
    typedef typename Step::column column;
    typedef typename Step::value_type value_type;
    typedef typename Step::reference reference;
    typedef typename Step::pointer pointer;
    typedef typename Step::const_reference const_reference;
    typedef typename Step::const_pointer const_pointer;
    typedef typename Step::position position;
    typedef std::vector<line> edge_t;
    typedef std::array<matrix, 4> corner_t;

public:
    enum CType {TL=0, TR=1, BR=2, BL=3};

public:

    template <typename ... Args>
    Exchanger(size_t overlap, size_t len, size_t index, size_t N, size_t M,
              size_t Hi, size_t Hj, Args ... args)
        : Step(len, index, N, M, Hi, Hj, args...)
    {
        for (size_t i=0; i<len; ++i) {
            m_bottom.push_back(new edge_t(overlap, line(Hj)));
            m_top.push_back(new edge_t(overlap, line(Hj)));
            m_left.push_back(new edge_t(overlap, column(Hi)));
            m_right.push_back(new edge_t(overlap, column(Hi)));
            m_corner.push_back(new corner_t({
                        matrix(overlap, overlap),
                        matrix(overlap, overlap),
                        matrix(overlap, overlap),
                        matrix(overlap, overlap)}));
        }
    }
    virtual ~Exchanger();

    net &at(size_t i = 0);
    const net &at(size_t i = 0) const;

    /* This operator gets the indexes from all over the entire net */
    double at(size_t i, size_t j, size_t l = 0) const;
    double at(size_t i, size_t j, size_t l = 0);
    double at(const position &pos, size_t l = 0) const;
    double at(const position &pos, size_t l = 0);

    edge_t &top(size_t t);
    edge_t &bottom(size_t t);
    edge_t &left(size_t t);
    edge_t &right(size_t t);
    corner_t &corner(size_t t);

    const edge_t &top(size_t t) const;
    const edge_t &bottom(size_t t) const;
    const edge_t &left(size_t t) const;
    const edge_t &right(size_t t) const;
    const corner_t &corner(size_t t) const;

    double top(const position &pos, size_t l = 0) const;
    double bottom(const position &pos, size_t l = 0) const;
```

```
    double left(const position &pos, size_t l = 0) const;
    double right(const position &pos, size_t l = 0) const;
    double corner(CType c, const position &pos, size_t l = 0) const;

    position abs(const position &pos) const;

public:
    virtual void v_next();
    const size_t W = 0;

private:
    std::vector<edge_t *> m_top;
    std::vector<edge_t *> m_bottom;
    std::vector<edge_t *> m_left;
    std::vector<edge_t *> m_right;
    std::vector<corner_t *> m_corner;
};

#endif /* exchanger.hpp */
```

```cpp
/*
 * Test of Step class
 */

#include "Exchanger.hpp"
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <unistd.h>

using namespace std;

template <typename T> class SimpleNet;
class SimpleStep : public Exchanger {
public:
    template <typename ...Args>
    SimpleStep(SimpleNet<SimpleStep> &net, Args ...args)
        : Exchanger(args...)
        , m_net(net)
    {}
    double calc(const position &pos);
    virtual void on_start(unsigned step);
private:
    SimpleNet<SimpleStep> &m_net;
};
template <typename T>
class SimpleNet {
public:
    template <typename ...Args>
    SimpleNet(size_t N, size_t M, size_t Hi, size_t Hj, Args... args)
    {
        Net<> tmp(0, N, M, Hi, Hj);
        m_length = tmp.Nc() * tmp.Mc();
        for (size_t i = 0; i<m_length; ++i) {
            m_net.push_back(new T(*this, 1, 2, i, N, M, Hi, Hj, args...));
        }
        m_net[0]->Step::at(0, 0) = 1;
    }
    ~SimpleNet() {
        for (auto s : m_net) {
            delete s;
        }
    }
    void next() {
        for (int i = m_length - 1; i >= 0; --i) {
            m_net[i]->next();
        }
    }
    template<typename ...Args>
    double at(Args ... pos) const {
        size_t i = (*m_net[0])->global_index(pos...);
        return (*m_net[i]).at(pos...);
    }
    void print () const {
        for (size_t i = 0; i < m_net[0]->N(); ++i) {
            for (size_t j = 0; j < m_net[0]->M(); ++j) {
                cout.width(5);
                cout << long(at(i,j)) << " ";
            }
            cout << endl;
        }
    }
    T *operator [] (ssize_t i) {
        if (i < 0 || i > (ssize_t)m_length) {
            return NULL;
        }
        return m_net[i];
    }

    const T *operator [] (size_t i) const{
        return m_net[i];
    }
private:
```

```
    vector<T *> m_net;
    size_t m_length;
};
double SimpleStep::calc(const position &pos)
{
    double result;
    result = at(pos);
    if (pos.first > 0) {
        result += at(pos.first - 1, pos.second);
    }
    if (pos.second > 0) {
        result += at(pos.first, pos.second - 1);
    }
    if (pos.first > 0 && pos.second > 0) {
        result += at(pos.first - 1, pos.second - 1);
    }
    return (long)result % 100000;

}
void SimpleStep::on_start(unsigned step)
{
    SimpleStep *nb[4] = {m_net[at().index_top()], m_net[at().index_bottom()],
        m_net[at().index_left()], m_net[at().index_right()]};
    if (nb[0]) top(Exchanger::W)[0] = (*nb[0])->bottom();
    if (nb[1]) bottom(Exchanger::W)[0] = (*nb[1])->top();
    if (nb[2]) left(Exchanger::W)[0] = (*nb[2])->right();
    if (nb[3]) right(Exchanger::W)[0] = (*nb[3])->left();
}

    #define N 10
    #define M 20
    #define Hi 5
    #define Hj 10
    #define Steps 20
    #define Count (N/Hi * M/Hj)
static void net()
{
    SimpleNet<SimpleStep> net(N, M, Hi, Hj);
    net.print();
    for (size_t s = 0; s < Steps; ++s) {
        cout << endl << endl << "=======================================";
        cout << endl << "Step " << s << endl;
        net.next();
        net.print();
    }
}
int main()
{
    net();
    return 0;
};
```

```cpp
#include "Heat.hpp"
#include <iostream>
#include <fstream>

using namespace std;
using namespace heat;

int main ()
{
    fstream v("vert.ppm", ios_base::out);

    v << "P3" << endl;
    v << "# " << "vert.ppm" << endl;
    v << "16 256" << endl;
    v << 255 << endl;


    for (size_t i = 0; i < 256; ++i)
        for (size_t j = 0; j < 16; ++ j)
        {
            v << R(i/255.) << " " << G(i/255.) << " " << B(i/255.) <<endl;
        }
    v.close();

    fstream h("horisontal.ppm", ios_base::out);

    h << "P3" << endl;
    h << "# " << "horisontal.ppm" << endl;
    h << "256 16" << endl;
    h << 255 << endl;
    for (size_t j = 0; j < 16; ++ j)
        for (size_t i = 0; i < 256; ++i)
        {
            h << R(i/255.) << " " << G(i/255.) << " " << B(i/255.) <<endl;
        }
    h.close();
    return 0;
}
```

```cpp
#include <cmath>
#include <string>

#ifndef _HEAT_HPP_
#define _HEAT_HPP_

namespace heat {

static const double PI = 3.14159265359;

/* v^2 */
static inline size_t R(double v, size_t max = 0xff)
{
    if (std::isnan(v)) return max;
    if (v > 1) v = 1;
    if (v < 0) v = 0;
    return v * v * max;
};
/* sin(PI*v) */
static inline size_t G(double v, size_t max = 0xff)
{
    if (std::isnan(v)) return max;
    if (v > 1) v = 1;
    if (v < 0) v = 0;
    return sin(PI*v) * max;
};
/* (1-v)^2 */
static inline size_t B(double v, size_t max = 0xff)
{
    if (std::isnan(v)) return max;
    if (v > 1) v = 1;
    if (v < 0) v = 0;
    return (1 - v) * (1 - v) * max;
};

};

#endif /* Heat.hpp */
```

```
#ifndef _HOLE_HPP_
#define _HOLE_HPP_

#include <vector>
#include <utility>
#include <initializer_list>
#include <cstddef>
#include <unistd.h>

class Hole {
public:
    typedef std::pair<double, double> position;
public:
    virtual ~Hole() {}
    virtual bool contains (double i, double j) = 0;
    bool contains (const position &pos)
    {
        return contains(pos.first, pos.second);
    }
    virtual Hole *copy () const = 0;
};

class RectangleHole : public Hole {
public:
    RectangleHole (double i1 = 0, double j1 = 0, double i2 = 0, double j2 = 0)
        : m_i1(i1) , m_j1(j1), m_i2(i2), m_j2(j2)
    {
        if (m_i1 > m_i2) std::swap(m_i1, m_i2);
        if (m_j1 > m_j2) std::swap(m_j1, m_j2);
    }
    RectangleHole (const RectangleHole &other)
        : RectangleHole(other.m_i1, other.m_j1, other.m_i2, other.m_j2)
    {}
    virtual bool contains (double i, double j)
    {
        if (!((i >= m_i1) && (i < m_i2) && (j >= m_j1) && (j < m_j2))) {
            return false;
        }
        return true;
    }
    virtual RectangleHole *copy () const
    {
        return new RectangleHole(*this);
    }
private:
    double m_i1, m_j1, m_i2, m_j2;
};
template <typename F>
class FHole : public Hole {
public:
    FHole(const F &f)
        : m_f(f)
    {}
    operator const F& () const{
        return m_f;
    }
    virtual bool contains (double i, double j) {
        return m_f(i,j);
    }
    virtual Hole *copy () const {
        return new FHole(*this);
    }
private:
    const F &m_f;
};
template <typename F>
static inline FHole<F> fHole (const F &f)
{
    return FHole<F> (f);
}

class Holes : public Hole {
public:
```

```
    template <typename ... Args>
    Holes (Args... args)
    {
        append(args...);
    }
    ~Holes()
    {
        for (auto h: m_holes) delete h;
    }
    void append(){};
    template <typename ... Args>
    void append(const Hole &h, Args ... args)
    {
        m_holes.push_back(h.copy());
        append (args...);
    }
    template <typename ... Args>
    void append(const Hole *h, Args ... args)
    {
        append(*h, args...);
    }
    template <typename ... Args>
    void append(const Holes &other, Args ... args)
    {
        for (auto h: other.m_holes) append(h);
        append (args...);
    }
    template <typename ... Args>
    void append(const Holes *h, Args ... args)
    {
        append (*h, args...);
    }
    virtual bool contains (double i, double j)
    {
        for (auto h: m_holes) {
            if (h->contains(i,j)) {
                return true;
            }
        }
        return false;
    }
    virtual Holes *copy() const {
        return new Holes (this);
    }
private:
    std::vector<Hole *> m_holes;
};

#endif /* Hole.hpp */
```

```cpp
#ifndef _JACOBY_HOLE_HPP_
#define _JACOBY_HOLE_HPP_

#include "Jacoby.hpp"
#include "Hole.hpp"
#include <cmath>

#include <iostream>
class Jacoby_Hole : public Jacoby {
public:
    template <typename ... Args>
    Jacoby_Hole (const Hole &h, const EdgeCondition &holeCondition,
                 Args ... args)
        : Jacoby(args...)
        , m_hole(h.copy())
        , m_holeCondition(holeCondition)
    {}
    ~Jacoby_Hole() {
        delete m_hole;
    }
    virtual double method(const position &pos)
    {
        bool edge;
        if (m_hole->contains(pos, edge)) {
            if (edge) return m_holeCondition(step(), pos.first, pos.second);
            return NAN;
        }
        return Jacoby::method(pos);
    }
private:
    Hole *m_hole;
    const EdgeCondition &m_holeCondition;
};

class HoleCondition : public EdgeCondition {
public:
    template <typename F>
    HoleCondition (const F &f, size_t N, size_t M = 0, double Lx = 1,
            double Ly = 0)
        : m_f(f)
        , Hx(Lx/(M == 0 ? N : M))
        , Hy((Ly == 0 ? Lx : Ly)/N)
    {}
    virtual double calc (size_t step, size_t i, size_t j) const
    {
        return m_f(j*Hx, i*Hy);
    }
private:
    std::function<double(double, double)> m_f = [](double x, double y)
        -> double { return 0; };
    double Hx, Hy;
};

#endif /* Jacoby-Holes.hpp */
```

```cpp
#include "Manifest.hpp"
#include "Params.hpp"

#include <algorithm>

class Jacoby : public Manifest {
public:
    template <typename F, typename ... Args>
    Jacoby (const F &f, Args ... args)
        : Manifest(args ...)
        , m_Hx((1.)/M())
        , m_Hy((1.)/N())
        , m_T(std::min(m_Hx*m_Hx/2, m_Hy*m_Hy/2)/2)
        , m_f(f)
    {}
    virtual double method(const position &pos)
    {
        double add_i = at(pos.first + 1, pos.second) - 2 * at(pos) +
            at(pos.first - 1, pos.second);
        double add_j = at(pos.first, pos.second + 1) - 2 * at(pos) +
            at(pos.first, pos.second - 1);
        double result =  at(pos) + m_T*(add_i/(m_Hy*m_Hy) +
                    add_j/(m_Hx*m_Hx) +
                    m_f(pos.second*m_Hx, pos.first*m_Hy, step()*m_T));
        return result;
    }
    double time() const
    {
        return m_T * step();
    }
private:
    double m_Hx;
    double m_Hy;
    double m_T;
    std::function<double(double, double, double)> m_f =
        [](double x, double y, double t) -> double { return 0; };

};
class Jacoby2 {
public:
    Jacoby2(size_t N, size_t M, const Functor &f)
        : m_Hx((1.)/N)
        , m_Hy((1.)/M)
        , m_T(std::min(m_Hx*m_Hx/2, m_Hy*m_Hy/2)/20)
        , m_f(f)
    {}

    double operator () (double v, double t, double r, double b, double l,
            size_t step, double x, double y) const
    {
        double add_i = l - 2 * v + r;
        double add_j = b - 2 * v + t;
        double result =  v + m_T*(add_i/(m_Hy*m_Hy) +
                    add_j/(m_Hx*m_Hx) + m_f(x, y, step*m_T));
        return result;
    }
    double T() const {
        return m_T;
    }

private:
    double m_Hx;
    double m_Hy;
    double m_T;
    const Functor &m_f;
};
```

```cpp
/*
 * Test of Step class
 */

#include "Jacoby-Holes.hpp"
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <unistd.h>
#include <cmath>

using namespace std;

#define PI 3.14159265359

template<typename T>
void print(const Matrix<T> &m)
{
    for (size_t i = 0; i < m.N(); ++i) {
        for (size_t j = 0; j < m.M(); ++j) {
            cout.precision(4);
            cout.width(7);
            cout << std::fixed;
            cout << m[i][j] << " ";
        }
        cout << endl;
    }
}

    #define N 10
    #define M 20
    #define Hi 5
    #define Hj 10
    #define Eps 0.01
    #define Count (N/Hi * M/Hj)
void jacoby(int &argc, char **&argv)
{
    MPI_Init(&argc, &argv);
    MPI_Comm comm = MPI_COMM_WORLD;
    int rank;
    auto zero =
        [](double x, double y) -> double { return sqrt(abs(x*x - y*y)); };
    auto left = [](double y) -> double { return sin(2*PI*y); };
    auto right = [](double y) -> double { return 1-cos(2*PI*y); };
    auto top = [](double x) -> double { return 0; };
    auto bottom = [](double x) -> double { return sin(2*PI*x); };
    auto f = [](double x, double y, double t) -> double {
        return sin(PI*x) * sin(PI*y) * sin(2*PI*t);
    };

    SplitEdgeCondition edge (N, M);
    Holes holes (RectangleHole(N/2, M/4, N/2 + N/3, (3*M)/4));
    HoleCondition hole_cond(zero, N, M);

    edge.zero(zero);
    edge.left(left);
    edge.right(right);
    edge.top(top);
    edge.bottom(bottom);

    MPI_Comm_rank(comm, &rank);
    cout << "Got runk " << rank << endl;
    Jacoby_Hole jkb(holes, hole_cond, f, edge, comm, 1, 2, rank, N, M, Hi, Hj);
    jkb.next(); // Fill up borders;
    do {
        jkb.next();
    } while(jkb.eps() > Eps);
    Matrix<double> *result = jkb.sync_results();
    if (result != NULL) {
        print(*result);
    }
    MPI_Finalize();
}
```

```
int main(int argc, char *argv[])
{
    jacoby(argc, argv);
    return 0;
};
```

```cpp
#include "Jacoby-Holes.hpp"
#include "Heat.hpp"
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <unistd.h>
#include <cmath>
#include <fstream>

using namespace std;
using namespace heat;

template<typename T>
void print(const Matrix<T> &m)
{
    for (size_t i = 0; i < m.N(); ++i) {
        for (size_t j = 0; j < m.M(); ++j) {
            cout.precision(4);
            cout.width(7);
            cout << std::fixed;
            cout << m[i][j] << " ";
        }
        cout << endl;
    }
}
int draw (Matrix<double> *result, double time, double T, const char *file);
int run(int rank, int N, int M, int Hi, int Hj, double eps, double T,
        const char *file);
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    MPI_Comm comm = MPI_COMM_WORLD;
    int rank, size;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);
    int N = 0, M;

    switch (size) {
    case 8:
        N=2;
        M=4;
        break;
    case 32:
        N=4;
        M=8;
        break;
    case 128:
        N=8;
        M=16;
        break;
    case 512:
        N=16;
        M=32;
        break;
    default:
        do {
            ++N;
            M=size/N;
        } while (N < M);
        break;
    }
    if (rank >= N*M) {
        MPI_Finalize();
        return 0;
    }

    int c;
    double eps = 0.1;
    int len = 512;
    const char *file = NULL;
    double T = 0;
    while ( (c = getopt(argc, argv, "n:e:o:t:")) != -1) switch (c){
    case 'n':
```

```
            len = atoi(optarg);
            if (len <=0 ) {
                MPI_Finalize();
                return 1;
            }
            break;
        case 'e':
            eps = atof(optarg);
            break;
        case 't':
            T = atof(optarg);
            break;
        case 'o':
            file = optarg;
            break;
        }

        run(rank, len, len, len/N, len/M, eps, T, file);

        MPI_Finalize();
        return 0;
}
int run(int rank, int N, int M, int Hi, int Hj, double eps, double T,
            const char *file)
{
        auto zero =
            [](double x, double y) -> double { return x+y - 1; };
        auto hole_edge =
            [](double x, double y) -> double { return 2; };
        auto left = [](double y) -> double { return 2*sin(2*PI*y); };
        auto right = [](double y) -> double { return 1-cos(2*PI*y); };
        auto top = [](double x) -> double { return 0; };
        auto bottom = [](double x) -> double { return 2*sin(2*PI*x); };
        auto f = [](double x, double y, double t) -> double {return 0;};

        SplitEdgeCondition edge (N, M);
        Holes holes (
                RectangleHole(N/4, M/8, 3*N/8, M/4),
                RectangleHole(N/8, 5*M/8, N/4, 6*M/8),
                RectangleHole(5*N/8, 6*M/8, 6*N/8, 7*M/8),
                RectangleHole(6*N/8, 3*M/8, 7*N/8, M/2),
                RectangleHole(3*N/8, M/2, N/2, 5*M/8),
                RectangleHole(3*N/8 - N/16, M/2 - M/16, N/2 - N/16, 5*M/8 - M/16));
        HoleCondition hole_cond(hole_edge, N, M);
        MPI_Comm comm = MPI_COMM_WORLD;

        edge.zero(zero);
        edge.left(left);
        edge.right(right);
        edge.top(top);
        edge.bottom(bottom);

        Jacoby_Hole jkb(holes, hole_cond, f, edge, comm, 1, 2, rank, N, M, Hi, Hj);
        double time = MPI_Wtime();
        jkb.next(); // Fill up borders;
        do {
            jkb.next();
            if (rank == 0 && jkb.step() % 100 == 0) {
                cout << "Step: " << jkb.step() << "; Eps: " << jkb.eps() <<
                    "; T: " << jkb.time() << endl;
            }
        } while((jkb.eps() > eps) && (T <= 0 || jkb.time() < T));
        time = MPI_Wtime() - time;
        Matrix<double> *result = jkb.sync_results();
        if (result == NULL) {
            return 0;
        }

        return draw(result, time, jkb.time(), file);
}
int draw (Matrix<double> *result, double time, double T, const char *file)
{
        double summ = 0;
```

```
    double max = NAN, min = NAN;

    for (size_t i = 0; i < result->N(); ++i) {
        for (size_t j = 0; j < result->M(); ++j) {
            double v = (*result)[i][j];
            if(!isnan(v)) summ += v;
            if (isnan(max) || max < v) max = v;
            if (isnan(NAN) || min > v) min = v;
        }
    }
    cout.precision(4);
    cout << "Time: " << time << "; Summ: " << summ << "; T: " << T << endl;

    if(file == NULL) return 0;

    fstream f(file, ios_base::out);
    f << "P3" << endl;
    f << "# " << file << endl;
    f << result->N() << " " << result->M() << endl;
    f << 255 << endl;

    for (size_t i = 0; i < result->N(); ++i) {
        for (size_t j = 0; j < result->M(); ++j) {
            double v = (*result)[i][j];
            v = (v - min)/(max - min);
            f << R(v) << " " << G(v) << " " << B(v) << endl;
        }
    }
    f.close();
    return 0;
}
```

```
CXX         =   mpicxx
CC          =   mpicc
CFLAGS      =   -fopenmp -g -O2 -Wall -Werror -DJACOBY_SYNC -std=gnu++11
LD          =   mpicc
LDFLAGS     =   -fopenmp -lm
METIS       =   -lmetis

all: decompositor calculation

tests: matrix-test step-test exchanger-test mpi-test jacoby-test

matrix-test.o: Matrix-test.cpp Matrix.hpp
    $(CXX) $(CFLAGS) -c -o $@ $<

matrix-test: matrix-test.o
    $(CXX) $(LDFLAGS) -o $@ $^


Step.o: Step.cpp Step.hpp Net.hpp Matrix.hpp
    $(CXX) $(CFLAGS) -c -o $@ $<

step-test.o: Step-test.cpp Step.hpp Matrix.hpp Net.hpp
    $(CXX) $(CFLAGS) -c -o $@ $<

step-test: step-test.o Step.o
    $(CXX) $(LDFLAGS) -o $@ $^


Exchanger.o: Exchanger.cpp Exchanger.hpp Step.hpp Net.hpp Matrix.hpp
    $(CXX) $(CFLAGS) -c -o $@ $<

exchanger-test.o: Exchanger-test.cpp Exchanger.hpp Step.hpp Matrix.hpp Net.hpp
    $(CXX) $(CFLAGS) -c -o $@ $<

exchanger-test: exchanger-test.o Exchanger.o Step.o
    $(CXX) $(LDFLAGS) -o $@ $^

MPI.o: MPI.cpp MPI.hpp MPI.hpp MPI.hpp MPI.hpp
    $(CXX) $(CFLAGS) -c -o $@ $<

mpi-test.o: MPI-test.cpp MPI.hpp Exchanger.hpp Step.hpp Matrix.hpp Net.hpp
    $(CXX) $(CFLAGS) -c -o $@ $<

mpi-test: mpi-test.o Exchanger.o Step.o MPI.o
    $(CXX) $(LDFLAGS) -o $@ $^

Manifest.o: Manifest.cpp Manifest.hpp MPI.hpp Exchanger.hpp Step.hpp Matrix.hpp Net.
hpp
    $(CXX) $(CFLAGS) -c -o $@ $<

Jacoby-test.o: Jacoby-test.cpp Jacoby-Holes.hpp Hole.hpp Jacoby.hpp Manifest.hpp MPI
.hpp Exchanger.hpp Step.hpp Matrix.hpp Net.hpp
    $(CXX) $(CFLAGS) -c -o $@ $<

jacoby-test: Jacoby-test.o Manifest.o Exchanger.o Step.o MPI.o
    $(CXX) $(LDFLAGS) -o $@ $^

main.o: main.cpp Jacoby-Holes.hpp Hole.hpp Jacoby.hpp Manifest.hpp MPI.hpp Exchanger
.hpp Step.hpp Matrix.hpp Net.hpp Heat.hpp
    $(CXX) $(CFLAGS) -c -o $@ $<

main: main.o Manifest.o Exchanger.o Step.o MPI.o
    $(CXX) $(LDFLAGS) -o $@ $^

heat.o: Heat.cpp Heat.hpp
    $(CXX) $(CFLAGS) -c -o $@ $<

heat: Heat.o
    $(CXX) $(LDFLAGS) -o $@ $^

Params.o: Params.cpp Params.hpp
    $(CXX) $(CFLAGS) -c -o $@ $<
```

```
decompositor.o: decompositor.cpp Decompositor.hpp
    $(CXX) $(CFLAGS) -c -o $@ $<

decompositor: decompositor.o Params.o
    $(CXX) $(LDFLAGS) $(METIS) -o $@ $^

Vertex.o : Vertex.cpp Vertex.hpp Neighbour.hpp Chunk.hpp
    $(CXX) $(CFLAGS) -c -o $@ $<

Neighbour.o : Neighbour.cpp Neighbour.hpp Vertex.hpp Chunk.hpp
    $(CXX) $(CFLAGS) -c -o $@ $<

Chunk.o : Chunk.cpp Chunk.hpp Neighbour.hpp Vertex.hpp
    $(CXX) $(CFLAGS) -c -o $@ $<

Calculation.o : Calculation.cpp Chunk.hpp Neighbour.hpp Vertex.hpp
    $(CXX) $(CFLAGS) -c -o $@ $<

calculation : Calculation.o Chunk.o Neighbour.o Vertex.o Params.o
    $(CXX) $(LDFLAGS) -o $@ $^

clean:
    rm -rf matrix-test *.o
```

```cpp
#include "Manifest.hpp"
#include <stdexcept>
#include <cmath>

using namespace std;

double Manifest::calc(const position &pos)
{
    double res;
    if (step() == 1 || pos.first == 0 || pos.second == 0 ||
            pos.first >= (ssize_t)N() - 1 || pos.second >= (ssize_t)M() - 1)
    {
        res = m_conditions(step(), pos.first, pos.second);
        if (res != NAN) {
            return res;
        }
    }
    return method(pos);
}
void Manifest::on_start(unsigned step)
{
    send_recv_top();
    send_recv_bottom();
    send_recv_left();
    send_recv_right();
    m_eps = 0;
}
void Manifest::on_stop(unsigned step)
{
    double eps = m_eps;

    MPI_Allreduce(&m_eps, &eps, 1, MPI_DOUBLE, MPI_SUM, comm());
    m_eps = sqrt(eps);
}
double Manifest::eps()
{
    return m_eps;
}
void Manifest::v_next_iterate(Step::net &dst)
{
    size_t Hi = at().Hi();
    size_t Hj = at().Hj();
#pragma omp parallel
    {
#pragma omp for collapse (2)
        for (size_t i = 0; i < Hi; ++i) for (size_t j = 0; j < Hj; ++j) {
            dst[i][j] = calc(global(i,j));
        }
//#pragma omp barrier
        double eps = resid(dst);
        if (eps > m_eps) {
#pragma omp critical
            {
                if (eps > m_eps) {
                    m_eps = eps;
                }
            }
        }
    }
}
double Manifest::resid (Step::net &dst)
{
    size_t Hi = at().Hi();
    size_t Hj = at().Hj();
    double eps = 0;
#pragma omp for collapse (2)
    for (size_t i = 0; i < Hi; ++i) for (size_t j = 0; j < Hj; ++j) {
        double r1 = at(global(i,j)), r2 = dst[i][j];
        if (isnan(r2)) continue;
        eps += (r1 - r2) * (r1 - r2);
    }
    return eps;
}
```

```
#include "MPI.hpp"
#include <stdexcept>
#include <cmath>

#ifndef _MANIFEST_HPP_
#define _MANIFEST_HPP_

class EdgeCondition {
public:
    virtual double calc (size_t step, size_t i, size_t j) const = 0;
    double operator () (size_t step, size_t i, size_t j) const {
        return calc(step, i, j);
    }
};

class Manifest : public MPI_Exchanger {
public :
    template <typename ... Args>
    Manifest(const EdgeCondition &conditions, Args ... args)
        : MPI_Exchanger(args ...)
        , m_conditions(conditions)
    {}
    virtual double method(const position &pos) = 0;
    double eps();
    virtual void on_start(unsigned step);
    virtual void on_stop(unsigned step);
private:
    virtual double calc(const position &pos);
    virtual void v_next_iterate(Step::net &dst);
    double resid (Step::net &dst);
private:
    const EdgeCondition &m_conditions;
    double m_eps = 0;
};

class SplitEdgeCondition : public EdgeCondition {
public:
    SplitEdgeCondition(size_t N, size_t M = 0, double Lx = 1, double Ly = 0)
        : Hx(Lx/(M == 0 ? N : M))
        , Hy((Ly == 0 ? Lx : Ly)/N)
        , m_N(N), m_M(M)
    {}
    template <typename F>
    void zero (const F &f) {
        m_z = f;
    }
    template <typename F>
    void top (const F &f) {
        m_t = f;
    }
    template <typename F>
    void bottom (const F &f) {
        m_b = f;
    }
    template <typename F>
    void left (const F &f) {
        m_l = f;
    }
    template <typename F>
    void right (const F &f) {
        m_r = f;
    }
    virtual double calc (size_t step, size_t i, size_t j) const
    {
        if (step == 1) return m_z(j*Hx, i*Hy);
        if (j == 0) return m_l(i*Hy);
        if (j == m_M-1) return m_r(i*Hy);
        if (i == 0) return m_t(j*Hx);
        if (i == m_N-1) return m_b(j*Hx);
        return NAN;
    }
private:
    std::function<double(double, double)> m_z = [](double x, double y)
```

```
        -> double { return 0; };
    std::function<double(double)> m_t = [](double x) -> double { return NAN;};
    std::function<double(double)> m_b = [](double x) -> double { return NAN;};
    std::function<double(double)> m_l = [](double y) -> double { return NAN;};
    std::function<double(double)> m_r = [](double y) -> double { return NAN;};
    double Hx, Hy;
    size_t m_N, m_M;
};

#endif /* Manifest.hpp */
```

```
/*
 *  Matrix is the fixed size vector of fixed sized vectors.
 */


#include <vector>
#include <functional>
#include <utility>
#include <cstddef>

#ifndef _MATRIX_HPP_
#define _MATRIX_HPP_

template <typename T>
class Matrix : public std::vector< std::vector<T> > {
public:
    typedef std::vector<std::vector<T> > base_type;
    typedef std::vector<T> column;
    typedef std::vector<T> line, raw;
    typedef T value_type;
    typedef value_type &reference;
    typedef value_type *pointer;
    typedef const value_type &const_reference;
    typedef const value_type *const_pointer;
    typedef std::pair<ssize_t, ssize_t> position;

public:
    Matrix(size_t in_N = 0, size_t in_M = 0,
           const value_type &val = value_type())
        : base_type(in_N,line(in_M, val))
        , m_N(in_N)
        , m_M(in_M)
    {}
    template<typename F, typename ...Args>
    Matrix (size_t in_N, size_t in_M, const F &f, Args... args)
        : base_type(in_N,line(in_M))
        , m_N(in_N)
        , m_M(in_M)
    {
        for (size_t i = 0; i< m_N; ++i) for (size_t j = 0; j < m_M; ++j) {
            (*this)[i][j] = f(i,j, args...);
        }
    }

    size_t N() const
    {
        return m_N;
    }
    size_t M() const
    {
        return m_M;
    }

    using base_type::at;
    reference at(size_t i, size_t j)
    {
        return at(i).at(j);
    }
    const_reference at(size_t i, size_t j) const
    {
        return at(i).at(j);
    }
    reference at(const position &pos)
    {
        return at(pos.first,pos.second);
    }
    const_reference at(const position &pos) const
    {
        return at(pos.first,pos.second);
    }
    reference operator()(size_t i, size_t j)
    {
        return at(i, j);
```

```
    }
    const_reference operator()(size_t i, size_t j) const
    {
        return at(i, j);
    }
    reference operator()(const position &pos)
    {
        return at(pos);
    }
    const_reference operator()(const position &pos) const
    {
        return at(pos);
    }
    reference operator  [] (const position &pos)
    {
        return at(pos);
    }
    const_reference operator  [] (const position &pos) const
    {
        return at(pos);
    }
    void fill (const value_type &val)
    {
        line c;
        c.fill(val);
        base_type::fill(c);
    }
    line &operator[] (size_t i)
    {
        return base_type::at(i);
    }
    const line &operator[] (size_t i) const
    {
        return base_type::at(i);
    }
    column get_column(size_t j) const
    {
        column result(m_N);
        for (size_t i = 0; i < m_N; ++i) {
            result[i] = at(i,j);
        }
        return result;
    }
    template<typename F, typename ...Args>
    void foreach (const F &f, Args... args)
    {
        for (line &l : (*this)) {
            for (reference val : l) {
                f(val, args...);
            }
        }
    }
    template<typename F, typename ...Args>
    void foreach (const F &f, Args... args) const
    {
        for (const line &l : (*this)) {
            for (const_reference val : l) {
                f(val, args...);
            }
        }
    }
private:
    size_t m_N, m_M;
};

#endif /* Matrix.hpp */
```

```cpp
/*
 * Test of Matrix class
 */

#include "Matrix.hpp"
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

template<typename T>
void print(const T &ref)
{
    for (auto val : ref) {
        cout << val << ' ';
    }
}
template<typename T>
void print(const Matrix<T> &m)
{
    for (size_t i = 0; i < m.N(); ++i) {
        for (size_t j = 0; j < m.M(); ++j) {
            cout << m[i][j] << " ";
        }
        cout << endl;
    }
}

int main()
{
    #define N 10
    #define M 20

    srand(time(0));

    for (size_t m = 0; m < M; m++) {
        for (size_t n = 0; n < N; n++) {
            cout << "Matrix " << n << " x " << m << endl;
            Matrix<int> mx(n, m,
                    [](size_t, size_t) -> int { return rand() % 10; });
            cout << "Matrix\n";
            print(mx);

            cout << "T_Matrix\n";
            for (size_t j = 0; j < m; ++j) {
                print (mx.get_column(j));
                cout << endl;
            }
            if (m < M - 1 || n < N - 1) {
                cout << endl << endl;
            }
        }
    }
};
```

```cpp
#include <iostream>
#include "MPI.hpp"


MPI_Exchanger::~MPI_Exchanger()
{
    if (m_result != NULL) {
        delete m_result;
    }
}
int MPI_Exchanger::send (size_t i, const line &s, int dst, int tag)
{
    return MPI_Send(s.data(), s.size(), MPI_DOUBLE, dst, tag*100 + i, m_comm);
}
int MPI_Exchanger::recv (size_t i, line &r, int dst, int tag)
{
    return MPI_Recv(r.data(), r.size(), MPI_DOUBLE, dst, tag*100 + i, m_comm,
            MPI_STATUS_IGNORE);
}
int MPI_Exchanger::send_recv (size_t i, const line &s, int s_tag,
        line &r, int r_tag, int dst)
{
    return MPI_Sendrecv(s.data(), s.size(), MPI_DOUBLE, dst, s_tag*100 + i,
                r.data(), r.size(), MPI_DOUBLE, dst, r_tag*100 + i,
                m_comm, MPI_STATUS_IGNORE);
}

void MPI_Exchanger::send_top(size_t i)
{
    const line &s = (*this)->top();
    int dst = at(0).index_top();
    if (dst < 0 || dst == (int)at(0).index()) {
        return;
    }
    send(i, s, dst, tag_top);
}
void MPI_Exchanger::recv_top(size_t i)
{
    line &r = top(W)[i];
    int dst = at(0).index_top();
    if (dst < 0 || dst == (int)at(0).index()) {
        return;
    }
    recv(i, r, dst, tag_bottom);
}
void MPI_Exchanger::send_recv_top(size_t i)
{
    const line &s = (*this)->top();
    line &r = top(W)[i];
    int dst = at(0).index_top();
    if (dst < 0 || dst == (int)at(0).index()) {
        return;
    }
    send_recv(i, s, tag_top, r, tag_bottom, dst);
}
void MPI_Exchanger::send_bottom(size_t i)
{
    const line &s = at(0)[at(0).size() - 1 - i];
    int dst = at(0).index_bottom();
    if (dst < 0 || dst == (int)at(0).index()) {
        return;
    }
    send(i, s, dst, tag_bottom);
}
void MPI_Exchanger::recv_bottom(size_t i)
{
    line &r = bottom(W)[i];
    int dst = at(0).index_bottom();
    if (dst < 0 || dst == (int)at(0).index()) {
        return;
    }
    recv(i, r, dst, tag_top);
}
```

```
void MPI_Exchanger::send_recv_bottom(size_t i)
{
    const line &s = at(0)[at(0).size() - 1 - i];
    line &r = bottom(W)[i];
    int dst = at(0).index_bottom();
    if (dst < 0 || dst == (int)at(0).index()) {
        return;
    }
    send_recv(i, s, tag_bottom, r, tag_top, dst);
}
void MPI_Exchanger::send_left(size_t i)
{
    const line s = at(0).get_column(i);
    int dst = at(0).index_left();
    if (dst < 0 || dst == (int)at(0).index()) {
        return;
    }
    send(i, s, dst, tag_left);
}
void MPI_Exchanger::recv_left(size_t i)
{
    line &r = left(W)[i];
    int dst = at(0).index_left();
    if (dst < 0 || dst == (int)at(0).index()) {
        return;
    }
    recv(i, r, dst, tag_right);
}
void MPI_Exchanger::send_recv_left(size_t i)
{
    const line s = at(0).get_column(i);
    line &r = left(W)[i];
    int dst = at(0).index_left();
    if (dst < 0 || dst == (int)at(0).index()) {
        return;
    }
    send_recv(i, s, tag_left, r, tag_right, dst);
}
void MPI_Exchanger::send_right(size_t i)
{
    const line s = at(0).get_column(at(0).Hj() - 1 - i);
    int dst = at(0).index_right();
    if (dst < 0 || dst == (int)at(0).index()) {
        return;
    }
    send(i, s, dst, tag_right);
}
void MPI_Exchanger::recv_right(size_t i)
{
    line &r = right(W)[i];
    int dst = at(0).index_right();
    if (dst < 0 || dst == (int)at(0).index()) {
        return;
    }
    recv(i, r, dst, tag_left);
}
void MPI_Exchanger::send_recv_right(size_t i)
{
    const line s = at(0).get_column(at(0).Hj() - 1 - i);
    line &r = right(W)[i];
    int dst = at(0).index_right();
    if (dst < 0 || dst == (int)at(0).index()) {
        return;
    }
    send_recv(i, s, tag_right, r, tag_left, dst);
}
int MPI_Exchanger::index() const
{
    return at(0).index();
}
int MPI_Exchanger::runk() const
{
    return index();
```

```
}
MPI_Exchanger::matrix *MPI_Exchanger::sync_results()
{
    if (index() == 0) {
        return sync_master();
    }
    sync_slave();
    return NULL;
}
MPI_Comm MPI_Exchanger::comm() const
{
    return m_comm;
}
MPI_Exchanger::matrix *MPI_Exchanger::sync_master()
{
    if (m_result != NULL) delete m_result;
    m_result = new matrix(at(0).N(), at(0).M());
    sync_put(at(0), index());
    matrix r(at(0).Hi(), at(0).Hj());
    for (size_t i = 1; i < at(0).Nc()*at(0).Mc(); ++i) {
        if (sync_recv(r, i) < 0) return NULL;
        if (sync_put(r, i) < 0) return NULL;
    }
    return m_result;
}
int MPI_Exchanger::sync_recv(matrix &rcv, size_t index)
{
    for (size_t i = 0; i < rcv.size(); ++ i) {
        if (recv (0, rcv[i], index, tag_result) < 0) {
            return -1;
        }
    }
    return 0;
}
int MPI_Exchanger::sync_put(const matrix &rcv, size_t index)
{
    position start = at(0).pos_of(index);
    for (size_t i = 0; i< rcv.size(); ++i) {
        for (size_t j=0; j < rcv[i].size(); ++j) {
            (*m_result)[i + start.first][j + start.second] = rcv[i][j];
        }
    }
    return 0;
}
int MPI_Exchanger::sync_slave()
{
    for (size_t i = 0; i < at().size(); ++ i) {
        if (send (0, at()[i], 0, tag_result) < 0) {
            return -1;
        }
    }
    return 0;
}
```

```
#ifndef _MPI_HPP_
#define _MPI_HPP_

#include <mpi.h>
#include "Exchanger.hpp"

class MPI_Exchanger : public Exchanger {
public:
    typedef typename Exchanger::matrix matrix;
    typedef typename Exchanger::line line;
    typedef typename Exchanger::raw raw;
    typedef typename Exchanger::column column;
    typedef typename Exchanger::value_type value_type;
    typedef typename Exchanger::reference reference;
    typedef typename Exchanger::pointer pointer;
    typedef typename Exchanger::const_reference const_reference;
    typedef typename Exchanger::const_pointer const_pointer;
    typedef typename Exchanger::position position;
    typedef typename Exchanger::edge_t edge_t;

public:
    template <typename ... Args>
    MPI_Exchanger(const MPI_Comm &comm, Args ... args)
        : Exchanger(args ...)
        , m_comm(comm)
    {}
    virtual ~MPI_Exchanger();

enum {
    tag_top = 0,
    tag_bottom = 1,
    tag_left = 2,
    tag_right = 3,
    tag_result = 4,
};

public:
    void send_top(size_t i = 0);
    void recv_top(size_t i = 0);
    void send_recv_top(size_t i = 0);

    void send_bottom(size_t i = 0);
    void recv_bottom(size_t i = 0);
    void send_recv_bottom(size_t i = 0);

    void send_left(size_t i = 0);
    void recv_left(size_t i = 0);
    void send_recv_left(size_t i = 0);

    void send_right(size_t i = 0);
    void recv_right(size_t i = 0);
    void send_recv_right(size_t i = 0);
    int index() const;
    int runk() const;
    matrix *sync_results();
    MPI_Comm comm() const;
private:
    int send (size_t i, const line &s, int dst, int tag);
    int recv (size_t i, line &r, int dst, int tag);
    int send_recv (size_t i, const line &s, int s_tag, line &r, int r_tag,
            int dst);
    matrix *sync_master();
    int sync_recv(matrix &rcv, size_t index);
    int sync_put(const matrix &rcv, size_t index);
    int sync_slave();

private:
    MPI_Comm m_comm;
    matrix *m_result = NULL;

};
```

```
#endif /* MPI_Exchanger.hpp */
```

```cpp
/*
 * Test of Step class
 */

#include "Exchanger.hpp"
#include "MPI.hpp"
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <unistd.h>

using namespace std;

template <typename T> class SimpleNet;

class MPITest : public MPI_Exchanger {
public:
    template <typename ...Args>
    MPITest(Args ...args)
        : MPI_Exchanger(args...)
    {
        if (index() == 0) {
            at(0)[0][0] = 1;
        }
    }
    double calc(const position &pos);
    virtual void on_start(unsigned step);

};
double MPITest::calc(const position &pos)
{
    double result;
    result = at(pos);
    if (pos.first > 0) {
        result += at(pos.first - 1, pos.second);
    }
    if (pos.second > 0) {
        result += at(pos.first, pos.second - 1);
    }
    if (pos.first > 0 && pos.second > 0) {
        result += at(pos.first - 1, pos.second - 1);
    }
    return (long)result % 100000;

}
void MPITest::on_start(unsigned step)
{
    send_recv_top();
    send_recv_bottom();
    send_recv_left();
    send_recv_right();
}
template<typename T>
void print(const Matrix<T> &m)
{
    for (size_t i = 0; i < m.N(); ++i) {
        for (size_t j = 0; j < m.M(); ++j) {
            cout.width(5);
            cout << m[i][j] << " ";
        }
        cout << endl;
    }
}

    #define N 10
    #define M 20
    #define Hi 5
    #define Hj 10
    #define Steps 20
    #define Count (N/Hi * M/Hj)
void mpi(int &argc, char **&argv)
{
    MPI_Init(&argc, &argv);
```

```
    MPI_Comm comm = MPI_COMM_WORLD;
    int rank;
    MPI_Comm_rank(comm, &rank);
    cout << "Got runk " << rank << endl;
    MPITest mpi(comm, 1, 2, rank, N, M, Hi, Hj);
    for (int i = 0; i < Steps; ++i) {
        mpi.next();
    }
    Matrix<double> *result = mpi.sync_results();
    if (result != NULL) {
        print(*result);
    }
    MPI_Finalize();
}

int main(int argc, char *argv[])
{
    mpi(argc, argv);
    return 0;
};
```

```cpp
#include <algorithm>
#include <iostream>

#include "Neighbour.hpp"
#include "Chunk.hpp"

using namespace std;

Neighbour::Neighbour (Chunk &chunk, int rank, const Functor &zero)
    : m_chunk (chunk)
    , m_rank (rank)
    , m_zero(zero)
    , m_step(chunk.step())
    , m_snd_req(MPI_REQUEST_NULL)
    , m_rcv_req(MPI_REQUEST_NULL)
{
}
Neighbour::Neighbour (const Neighbour &other)
    : m_chunk (other.m_chunk)
    , m_rank (other.m_rank)
    , m_zero(other.m_zero)
    , m_step(other.m_step)
    , m_snd_req(MPI_REQUEST_NULL)
    , m_rcv_req(MPI_REQUEST_NULL)
{
}
Neighbour::~Neighbour()
{
}

size_t Neighbour::N() const
{
    return m_chunk.N();
}
size_t Neighbour::M() const
{
    return m_chunk.M();
}
size_t &Neighbour::step() const
{
    return m_chunk.step();
}
void Neighbour::add (size_t index, const Vertex *inner, Vertex::Direction d) const
{
    NeighborVertexSet::iterator ptr = m_vertecies.insert(
            NeighbourVertex(index, N(), M(), step(), *this)).first;

    if (ptr == m_vertecies.end()) {
        exit(1);
    }
    if (m_border.insert(inner).first == m_border.end()) {
        exit(1);
    }
    inner->set(*ptr, d);
    ptr->set(inner, Vertex::reverse(d));
    ptr->set(m_zero(ptr->x(), ptr->y(), 0));
    m_send.resize(m_border.size());
    m_receive.resize(m_vertecies.size());
}
int Neighbour::operator < (int rank) const
{
    return m_rank < rank;
}
int Neighbour::operator == (int rank) const
{
    return m_rank == rank;
}
int Neighbour::operator > (int rank) const
{
    return m_rank > rank;
}
int Neighbour::operator < (const Neighbour &other) const
{
```

```
    return m_rank < other.m_rank;
}
int Neighbour::operator == (const Neighbour &other) const
{
    return m_rank == other.m_rank;
}
int Neighbour::operator > (const Neighbour &other) const
{
    return m_rank > other.m_rank;
}
void Neighbour::finish_rcv() const
{
    size_t i = 0;
    if (m_step > 0) {
        for (NeighborVertexSet::iterator it = m_vertecies.begin();
                it != m_vertecies.end(); ++it, ++i)
        {
            it->set(m_receive[i]);
        }
    }
    MPI_Irecv(m_receive.data(), m_receive.size(),MPI_DOUBLE, m_rank, 0,
            MPI_COMM_WORLD, &m_rcv_req);
}
void Neighbour::finish_send() const
{
    size_t i = 0;
    if (m_snd_req != MPI_REQUEST_NULL) {
        MPI_Wait(&m_snd_req, MPI_STATUS_IGNORE);
    }
    for (PtrVertexSet::iterator it = m_border.begin(); it != m_border.end();
            ++it, ++i)
    {
        m_send[i] = (*it)->get(0);
    }
    MPI_Isend(m_send.data(), i, MPI_DOUBLE, m_rank, 0, MPI_COMM_WORLD,
            &m_snd_req);
}
```

```
#ifndef _NEIGHBOUR_HPP_
#define _NEIGHBOUR_HPP_

#include "Vertex.hpp"
#include "Params.hpp"

#include <map>
#include <set>
#include <vector>
#include <mpi.h>

class Chunk;
class Neighbour {
    typedef std::set<NeighbourVertex> NeighborVertexSet;
    typedef std::set<const Vertex *> PtrVertexSet;
    typedef std::vector<double> DoubleVector;
public:
    Neighbour (Chunk &chunk, int rank, const Functor &zero);
    Neighbour (const Neighbour &other);
    ~Neighbour ();
    size_t N() const;
    size_t M() const;
    size_t &step() const;

    void add (size_t index, const Vertex *inner, Vertex::Direction d) const;
    int operator < (int rank) const;
    int operator == (int rank) const;
    int operator > (int rank) const;
    int operator < (const Neighbour &other) const;
    int operator == (const Neighbour &other) const;
    int operator > (const Neighbour &other) const;

    template <typename F>
    bool try_step(const F &f) const
    {
        int flag;
        if (m_step >= step()) return false;
        if (m_step == 0) {
            finish(f);
            return true;
        }
        MPI_Test(&m_rcv_req, &flag, MPI_STATUS_IGNORE);
        if (!flag) return false;
        finish(f);
        return true;
    }
    template <typename F>
    void step(const F &f) const
    {
        if (m_step >= step()) return;
        if (m_step == 0) {
            calc(f);
            finish_send();
            return;
        }
        MPI_Wait(&m_rcv_req, MPI_STATUS_IGNORE);
        finish(f);
    }
private:
    template <typename F>
    void calc (F f) const
    {
        ++m_step;
        for (PtrVertexSet::iterator it = m_border.begin(); it != m_border.end();
                ++it)
        {
            (*it)->set(f((*it)->get(),(*it)->top()->get(), (*it)->right()->get(),
                        (*it)->bottom()->get(), (*it)->left()->get(), m_step,
                        (*it)->x(), (*it)->y()));
        }
    }
    template <typename F>
    void finish(F f) const
```

```
    {
        finish_rcv();
        calc(f);
        finish_send();
    }
    void finish_rcv() const;
    void finish_send() const;

private:
    Chunk &m_chunk;
    int m_rank;
    mutable NeighborVertexSet m_vertecies;
    mutable PtrVertexSet m_border;
    mutable DoubleVector m_send;
    mutable DoubleVector m_receive;
    const Functor &m_zero;
    mutable size_t m_step;
    mutable MPI_Request m_snd_req, m_rcv_req;
};

#endif /* Neighbour.hpp */
```

```
/*
 * Class contains the meta-information of entire net
 * and the data of current cell.
 */

#ifndef _NET_HPP_
#define _NET_HPP__

#include "Matrix.hpp"

template <typename T = double>
class Net : public Matrix <T> {
public:
    typedef Matrix <T> matrix;
    typedef typename matrix::line line;
    typedef typename matrix::raw raw;
    typedef typename matrix::column column;
    typedef typename matrix::value_type value_type;
    typedef typename matrix::reference reference;
    typedef typename matrix::pointer pointer;
    typedef typename matrix::const_reference const_reference;
    typedef typename matrix::const_pointer const_pointer;
    typedef typename matrix::position position;

public:
    Net(size_t index, size_t N, size_t M, size_t Hi, size_t Hj,
            const_reference val = value_type())
        : matrix(Hi, Hj, val)
    {
        p_init(index, N, M, Hi, Hj);
    };
    template<typename F, typename ...Args>
    Net (size_t index, size_t N, size_t M, size_t Hi, size_t Hj,
            const F &f, Args... args)
        : matrix(Hi, Hj, f, args...)
    {
        p_init(index, N, M, Hi, Hj);
    };


    size_t N() const
    {
        return m_N;
    }
    size_t M() const
    {
        return m_M;
    }
    size_t Hi() const
    {
        return matrix::N();
    }
    size_t Hj() const
    {
        return matrix::M();
    }
    size_t Nc() const
    {
        return m_Nc;
    }
    size_t Mc() const
    {
        return m_Mc;
    }
    size_t I() const
    {
        return m_I;
    }
    size_t J() const
    {
        return m_J;
    }
```

```
    position matrix_pos(size_t i, size_t j) const
    {
        return {i / Hi(), j / Hj()};
    }
    position matrix_pos(position &pos) const
    {
        return matrix_pos(pos.first, pos.second);
    }
    size_t global_index(size_t i, size_t j) const
    {
        return index(matrix_pos(i, j));
    }
    size_t global_index(const position &pos) const
    {
        return global_index(pos.first, pos.second);
    }

    size_t index() const
    {
        return m_index;
    }
    size_t index(size_t i, size_t j) const
    {
        if (i >= m_Nc) i = m_Nc - 1;
        if (j >= m_Mc) j = m_Mc - 1;
        return m_Mc * i + j;
    }
    position pos_of(size_t index) const
    {
        position pos;
        pos.first = index / m_Mc * Hi();
        pos.second = index % m_Mc * Hj();
        return pos;
    }
    size_t index(const position &pos) const
    {
        return index(pos.first, pos.second);
    }
    ssize_t index_relative(ssize_t i = 0, ssize_t j = 0) const
    {
        ssize_t l_I = m_I + i, l_J = m_J + j;
        if (l_I < 0) return -1;
        if (l_I >= (ssize_t)m_Nc) return -1;
        if (l_J < 0) return -1;
        if (l_J >= (ssize_t)m_Mc) return -1;
        return index(l_I, l_J);
    }
    ssize_t index_top(size_t i = 1) const
    {
        return index_relative(-(ssize_t)i, 0);
    }
    ssize_t index_bottom(size_t i = 1) const
    {
        return index_relative(i, 0);
    }
    ssize_t index_left(size_t j = 1) const
    {
        return index_relative(0, -(ssize_t)j);
    }
    ssize_t index_right(size_t j = 1) const
    {
        return index_relative(0, j);
    }

    position global (size_t i, size_t j) const
    {
        return {m_I * Hi() + i, m_J * Hj() + j};
    }
    position global (const position &pos) const
    {
        return global(pos.first, pos.second);
    }
    position local (size_t i, size_t j) const
```

```
    {
        return { i - m_I * Hi(), j - m_J * Hj()};
    }
    position local (const position &pos) const
    {
        return local(pos.first, pos.second);
    }

    const line &top(size_t i = 1) const {
        return this->at(i);
    }
    line &top(size_t i = 0) {
        return this->at(i);
    }
    const line &bottom(size_t i = 1) const {
        return this->at(Hi() - i);
    }
    line &bottom(size_t i = 1) {
        return this->at(Hi() - i);
    }
    column left(size_t j = 1) {
        return this->get_column(j);
    }
    column right(size_t j = 1) {
        return this->get_column(Hj() - j);
    }
private:
    void p_init(size_t index, size_t N, size_t M, size_t Hi, size_t Hj)
    {
        m_N = N + (Hi - N % Hi) % Hi; /* Make net devidable on step  */
        m_M = M + (Hj - M % Hj) % Hj;
        m_index = index;
        m_Nc = m_N / Hi;
        m_Mc = m_M / Hj;
        m_I = index / m_Mc;
        m_J = index % m_Mc;
    }
private:
    size_t m_N, m_M;
    size_t m_index;

    size_t m_Nc, m_Mc;
    size_t m_I, m_J;
};

#endif /* Net.hpp */
```

```
#include "Params.hpp"
#include <map>

using namespace std;
class __base_params : public HoleParams {
protected:
    __base_params ()
        : m_hole (NULL)
    {}
    __base_params(const Hole &h)
        : m_hole (h.copy())
    {}
    __base_params(const Hole *h)
        : m_hole (h->copy())
    {}
    ~__base_params() {
        if (m_hole != NULL)
            delete m_hole;
    }
    virtual const Hole *hole() const
    {
        return m_hole;
    }
    virtual const EdgeCondition *edge() const
    {
        return NULL;
    }
    void add(const Hole *hole) {
        Hole *tmp = m_hole;
        if (tmp == NULL) {
            m_hole = hole == NULL ? NULL : hole->copy();
            return;
        }
        m_hole = new Holes(tmp, hole);
    }
    void add(const Hole &hole) {
        return add(&hole);
    }
private:
    Hole *m_hole;
};
class Test_HoleParams : public __base_params {
public:
    Test_HoleParams() :
        __base_params(Holes{
            RectangleHole(1./4, 1./8, 3*1./8, 1./4),
            RectangleHole(1./8, 5*1./8, 1./4, 6*1./8),
            RectangleHole(5*1./8, 6*1./8, 6*1./8, 7*1./8),
            RectangleHole(6*1./8, 3*1./8, 7*1./8, 1./2),
            RectangleHole(3*1./8, 1./2, 1./2, 5*1./8),
            RectangleHole(3*1./8 - 1./16, 1./2 - 1./16, 1./2 - 1./16, 5*1./8 - 1./16
)})
    {    }
};
const static Test_HoleParams Test;

class Double_HoleParams:public __base_params {
public:
    Double_HoleParams() :
        __base_params(Holes(fHole(up), fHole(down)))
    {}
private:
    static bool down(double x, double y) {
        return y < x*x;
    }
    static bool up(double x, double y) {
        return y*y > x;
    }
};
const static Double_HoleParams Double;

class Line_HoleParams : public __base_params {
public:
```

```
    Line_HoleParams() :
        __base_params(Holes(fHole(up), fHole(down)))
        {}
private:
    static bool down(double x, double y) {
        return y < 0.5 - x;
    }
    static bool up(double x, double y) {
        return y > 1.5 - x;
    }
};
const static Line_HoleParams Line;

const HoleParams *HoleParams::get (const string &in_name)
{
#define PARAM(name) params[#name] = &name
    static map<string, const HoleParams * > params;
    if (params.empty ()){
        PARAM(Test);
        PARAM(Double);
        PARAM(Line);
    }
#undef PARAM

    map<string, const HoleParams *>::iterator it = params.find(in_name);
    if (it == params.end()) return NULL;
    return it->second;
}

#define PI (3.14159265358979)
double Functor_zero1(double x, double y, double t){
    return x*y;
}
double Functor_hole_edge1(double x, double y, double t){
    return 1;
}
double Functor_left1(double x,double y, double t){
    return 2*sin(2*PI*y);
}
double Functor_right1(double x,double y, double t){
    return 1-cos(2*PI*y);
}
double Functor_top1(double x, double y, double t){
    return 0;
}
double Functor_bottom1(double x, double y, double t){
    return 2*sin(2*PI*x);
}
double Functor_edge1(double x, double y, double t){
    return sin(PI*x) + sin(PI*y);
}
double Functor_hole1(double x, double y, double t){
    return sin(100*PI*t);
}
double Functor_foo1(double x, double y, double t){
    return sin(100*PI*t);
}

double Functor_ant(double x, double y, double t){
    return x*x + y*y + sin(100*PI*t);
}
double Functor_ant_foo(double x, double y, double t){
    return 100*PI*cos(PI*t) - 4;
}
double __zero(double x, double y, double t){
    return 0;
}
const Functor &Functor::get(const std::string &in_name)
{
#define FUNCTOR(name) functors.insert(pair<string, Functor>(#name, \
            Functor(Functor_ ## name)));
    static map<string, Functor> functors;
    if (functors.empty ()){
```

```
        FUNCTOR(zero1);
        FUNCTOR(hole_edge1);
        FUNCTOR(left1);
        FUNCTOR(right1);
        FUNCTOR(top1);
        FUNCTOR(bottom1);
        FUNCTOR(edge1);
        FUNCTOR(hole1);
        FUNCTOR(ant);
        FUNCTOR(ant_foo);
    }
    static Functor zero(__zero);
    map<string, Functor>::iterator it = functors.find(in_name);
    if (it == functors.end()) return zero;
    return it->second;
}
```

```
#ifndef _PARAMS_HPP_
#define _PARAMS_HPP_

#include "Hole.hpp"
#include "Manifest.hpp"
#include <string>

class HoleParams {
public:
    virtual const Hole *hole() const = 0;

    static const HoleParams *get(const std::string &name);

};

class Functor {
public:
typedef double (F)(double, double, double);
public:
    Functor(const F &f)
        : m_f(f)
    {}
    Functor(const Functor &f)
        : m_f(f.m_f)
    {}
    virtual double calc(double x, double y, double t) const
    {
        return m_f(x,y,t);
    }
    double operator () (double x, double y, double t) const
    {
        return calc(x,y,t);
    }
    static const Functor &get(const std::string &name);
private:
    const F &m_f;
};

#endif /* Params.hpp */
```

```cpp
#include "Step.hpp"

using namespace std;
typedef typename Step::net net;
typedef typename Step::line line;
typedef typename Step::raw raw;
typedef typename Step::column column;
typedef typename Step::value_type value_type;
typedef typename Step::reference reference;
typedef typename Step::pointer pointer;
typedef typename Step::const_reference const_reference;
typedef typename Step::const_pointer const_pointer;
typedef typename Step::position position;

Step::~Step()
{
    for (auto n : (*this)) delete n;
}

net &Step::at(size_t i)
{
    return *base_type::at(i);
}
const net &Step::at(size_t i) const
{
    return *base_type::at(i);
}
reference Step::at(size_t i, size_t j)
{
    return at().at(local(i,j));
}
const_reference Step::at(size_t i, size_t j) const
{
    return at().at(local(i,j));
}
reference Step::at(const position &pos)
{
    return at(pos.first,pos.second);
}
const_reference Step::at(const position &pos) const
{
    return at(pos.first,pos.second);
}
reference Step::operator()(size_t i, size_t j)
{
    return at(i,j);
}
const_reference Step::operator()(size_t i, size_t j) const
{
    return at(i,j);
}
reference Step::operator()(const position &pos)
{
    return at(pos);
}
const_reference Step::operator()(const position &pos) const
{
    return at(pos);
}
net &Step::operator [] (size_t i)
{
    return at(i);
}
const net &Step::operator [] (size_t i) const
{
    return at(i);
}
reference Step::operator [] (const position &pos)
{
    return at(pos);
}
const_reference Step::operator [] (const position &pos) const
{
```

```
    return at(pos);
}
net &Step::operator()(size_t i)
{
    return at(i);
}
const net &Step::operator()(size_t i) const
{
    return at(i);
}
position Step::global (size_t i, size_t j) const
{
    return at().global(i,j);
}
position Step::global (const position &pos) const
{
    return at().global(pos);
}
position Step::local (size_t i, size_t j) const
{
    return at().local(i,j);
}
position Step::local (const position &pos) const
{
    return at().local(pos);
}

Step::operator net &()
{
    return at();
}
Step::operator const net &() const
{
    return at();
}

net *Step::operator ->()
{
    return &at();
}
const net *Step::operator ->() const
{
    return &at();
}
unsigned Step::step() const
{
    return m_step;
}
unsigned Step::next()
{
    ++m_step;
    on_start(m_step);
    v_next();
    on_stop(m_step);

    return m_step;
}
void Step::v_next()
{
    v_next_swap_begin();
    v_next_iterate(*m_swap);
    v_next_swap_end();
}
void Step::v_next_iterate(net &dst)
{
    size_t Hi = at().Hi();
    size_t Hj = at().Hj();
    for (size_t i = 0; i < Hi; ++i) for (size_t j = 0; j < Hj; ++j) {
        dst[i][j] = calc(global(i,j));
    }
}
void Step::v_next_swap_begin()
{
```

```
    m_swap = base_type::at(size() - 1);
    for (size_t i = size() - 1; i > 0; --i) {
        base_type::at(i) = base_type::at(i-1);
    }
}
void Step::v_next_swap_end()
{
    base_type::at(0) = m_swap;
}
void Step::on_start(unsigned step)
{}
void Step::on_stop(unsigned step)
{}
template <typename T>
static T &step_store(vector<T> &v, size_t i)
{
    if (i >= v.size()) {
        v.resize(i+1);
    }
    return v[i];
}
line &Step::store_up(size_t i)
{
    return step_store(m_top, i);
}
line &Step::store_down(size_t i)
{
    return step_store(m_bottom, i);
}
column &Step::store_right(size_t j)
{
    return step_store(m_left, j);
}
column &Step::store_left(size_t j)
{
    return step_store(m_right, j);
}
```

```
/*
 * Class, represents the thread and responsible for running each step and
 * watching the states of old steps.
 */

#ifndef _STEP_HPP_
#define _STEP_HPP_

#include "Net.hpp"

class Step : public std::vector<Net<double> *> {
public:
    typedef Net<double> net;
    typedef std::vector<net *>base_type;
    typedef typename net::matrix matrix;
    typedef typename matrix::line line;
    typedef typename matrix::raw raw;
    typedef typename matrix::column column;
    typedef typename matrix::value_type value_type;
    typedef typename matrix::reference reference;
    typedef typename matrix::pointer pointer;
    typedef typename matrix::const_reference const_reference;
    typedef typename matrix::const_pointer const_pointer;
    typedef typename matrix::position position;

public:
    template<typename ...Args>
    Step(size_t len, Args ... args)
        : base_type(len, NULL)
    {
        for (size_t i = 0; i < len; ++i) {
            base_type::at(i) = new net(args...);
        }
    }
    virtual ~Step();

    /* Passes indexes from all over the entire net */
    virtual double calc (const position &pos) = 0;

    net &at(size_t i = 0);
    const net &at(size_t i = 0) const;

    /* This operator gets the indexes from all over the entire net */
    reference at(size_t i, size_t j);
    const_reference at(size_t i, size_t j) const;

    reference at(const position &pos);
    const_reference at(const position &pos) const;

    reference operator()(size_t i, size_t j);
    const_reference operator()(size_t i, size_t j) const;

    reference operator()(const position &pos);
    const_reference operator()(const position &pos) const;

    net &operator [] (size_t i);
    const net &operator [] (size_t i) const;

    reference operator [] (const position &pos);
    const_reference operator [] (const position &pos) const;

    net &operator()(size_t i = 0);
    const net &operator()(size_t i = 0) const;

    net *operator ->();
    const net *operator ->() const;

    position global (size_t i, size_t j) const;
    position global (const position &pos) const;
    position local (size_t i, size_t j) const;
    position local (const position &pos) const;

    operator net &();
```

```
    operator const net &() const;

    unsigned step() const;

    unsigned next();
    virtual void on_start(unsigned step);
    virtual void on_stop(unsigned step);

    line &store_up(size_t i = 0);
    line &store_down(size_t i = 0);
    column &store_right(size_t j = 0);
    column &store_left(size_t j = 0);

    size_t N() const
    {
        return at().N();
    }
    size_t M() const
    {
        return at().M();
    }
    size_t Hi() const
    {
        return at().Hi();
    }
    size_t Hj() const
    {
        return at().Hj();
    }
    size_t Nc() const
    {
        return at().Nc();
    }
    size_t Mc() const
    {
        return at().Mc();
    }
    size_t I() const
    {
        return at().I();
    }
    size_t J() const
    {
        return at().J();
    }

protected:
    virtual void v_next();
    virtual void v_next_iterate(net &dst);
    virtual void v_next_swap_begin();
    virtual void v_next_swap_end();
private:
    unsigned m_step = 0;
    std::vector<line> m_top;
    std::vector<line> m_bottom;
    std::vector<column> m_left;
    std::vector<column> m_right;
    net *m_swap = NULL;

};

#endif /* Step.hpp */
```

```cpp
/*
 * Test of Step class
 */

#include "Step.hpp"
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <unistd.h>

using namespace std;

template<typename T>
void print(const T &ref)
{
    for (auto val : ref) {
        cout << val << ' ';
    }
}
void print(const Step &m)
{
    cout << "Size: " << m.Hi() << "x" << m.Hi() << endl;
    for (size_t i = 0; i < m.Hi(); ++i) {
        for (size_t j = 0; j < m.Hj(); ++j) {
            cout << m(m.global(i,j)) << " ";
        }
        cout << endl;
    }
}

class StepTest : public Step {
public:
    template <typename ...Args>
    StepTest(Args ...args) : Step(args...)
    {}
    double calc(const position &pos)
    {
        return at(pos) + step();
    }
};

template <typename T> class SimpleNet;
class SimpleStep : public Step {
public:
    template <typename ...Args>
    SimpleStep(SimpleNet<SimpleStep> &net, Args ...args)
        : Step(args...)
        , m_net(net)
    {}
    double calc(const position &pos);
private:
    SimpleNet<SimpleStep> &m_net;
};
template <typename T>
class SimpleNet {
public:
    template <typename ...Args>
    SimpleNet(size_t N, size_t M, size_t Hi, size_t Hj, Args... args)
    {
        Net<> tmp(0, N, M, Hi, Hj);
        m_length = tmp.Nc() * tmp.Mc();
        for (size_t i = 0; i<m_length; ++i) {
            m_net.push_back(new T(*this, 2, i, N, M, Hi, Hj, args...));
        }
        m_net[0]->at(0, 0) = 1;
    }
    ~SimpleNet() {
        for (auto s : m_net) {
            delete s;
        }
    }
    void next() {
        for (int i = m_length - 1; i >= 0; --i) {
```

```
            m_net[i]->next();
        }
    }
    template<typename ...Args>
    double at(Args ... pos) const {
        size_t i = (*m_net[0])->global_index(pos...);
        return (*m_net[i]).at(pos...);
    }
    void print () const {
        for (size_t i = 0; i < m_net[0]->N(); ++i) {
            for (size_t j = 0; j < m_net[0]->M(); ++j) {
                cout.width(5);
                cout << long(at(i,j)) << " ";
            }
            cout << endl;
        }
    }
private:
    vector<T *> m_net;
    size_t m_length;
};
double SimpleStep::calc(const position &pos)
{
    double result;
    result = m_net.at(pos);
    if (pos.first > 0) {
        result += m_net.at(pos.first - 1, pos.second);
    }
    if (pos.second > 0) {
        result += m_net.at(pos.first, pos.second - 1);
    }
    if (pos.first > 0 && pos.second > 0) {
        result += m_net.at(pos.first - 1, pos.second - 1);
    }
    return (long)result % 100000;

}

    #define N 10
    #define M 20
    #define Hi 5
    #define Hj 10
    #define Steps 20
    #define Count (N/Hi * M/Hj)
static void arr()
{
    array<Step *, Count> net;

    for (size_t c = 0; c < Count; ++c) {
        net[c] = new StepTest(2, c, N, M, Hi, Hj);
    }
    for (size_t s = 0; s < Steps; ++s) {
        cout << endl << endl << "=========================================";
        cout << endl << "Step " << s << endl;
        for (size_t c = 0; c < Count; ++c) {
            net[c]->next();
            print (*net[c]);
        }
    }
    for (size_t c = 0; c < Count; ++c) {
        delete net[c];
    }


}
static void net()
{
    SimpleNet<SimpleStep> net(N, M, Hi, Hj);
    net.print();
    for (size_t s = 0; s < Steps; ++s) {
        cout << endl << endl << "=========================================";
        cout << endl << "Step " << s << endl;
        net.next();
```

```
        net.print();
    }
}
int main()
{
    arr();
    cout << endl << endl << endl;
    cout << "============================================================" << endl;
    cout << "============================================================" << endl;
    cout << "============================================================" << endl;
    net();

    return 0;

};
```

```cpp
#include "Vertex.hpp"
#include "Neighbour.hpp"

size_t Vertex::degree(const ::Neighbour *nb)
{
    size_t result = 0;
    for (size_t i = 0; i < 4; ++i) {
        Vertex *v = neighbor((Direction)i);
        if (v == NULL) continue;
        NeighbourVertex *nv = (NeighbourVertex *)v;
        if (v->m_type == Neighbour && nv->neighbour() == nb) {
            ++result;
        }
    }
    return result;
}
```

```
#ifndef _VERTEX_HPP_
#define _VERTEX_HPP_

#include <cstring>
#include <cmath>
#include <utility>

template <typename T>
static inline T square(T v) {
    return v*v;
}

class Neighbour;
class Vertex {
public:
    enum Type {
        Inner, I = Inner,
        Border, B = Border,
        Edge, E = Edge,
        Neighbour, N = Neighbour,
        Hole, H = Hole,
    };
    enum Direction {
        Top = 0, Right = 1, Bottom = 2, Left = 3,
    };
public:
    Vertex (size_t index, size_t N, size_t M, size_t &step, double val = 0,
            Type type = Inner)
        : m_index(index)
        , m_step(step)
        , m_type(type)
    {
        init(N, M);
        m_value[0] = m_value[1] = val;
    }
    Vertex (const Vertex &v)
        : m_index(v.m_index)
        , m_i(v.m_i), m_j(v.m_j)
        , m_x(v.m_x), m_y(v.m_y)
        , m_step(v.m_step)
        , m_type(v.m_type)
    {
        m_value[0] = v.m_value[0];
        m_value[1] = v.m_value[1];
        for (size_t i = 0; i < 4; ++i) {
            m_neighbors[i] = v.m_neighbors[i];
        }
    }
    virtual ~Vertex()
    {}

    virtual void set (double value, size_t i = 0) const
    {
        m_value[ (m_step + i) % 2 ] = value;
    }
    virtual double get (size_t i = 1) const
    {
        return m_value[ (m_step + m_i) % 2 ];
    }
    operator double () const
    {
        return get();
    }

    void pos (size_t &i, size_t &j) const {
        i = m_i;
        j = m_j;
    }
    void pos (double &x, double &y) const {
        x = m_x;
        y = m_y;
    }
    double x() const {
```

```
        return m_x;
    }
    double y() const {
        return m_y;
    }
    size_t i() const {
        return m_i;
    }
    size_t j() const {
        return m_j;
    }
    size_t index () const {
        return m_index;
    }
    Type type () const {
        return m_type;
    }
    static double distance (double x1, double y1, double x2 = 0, double y2 = 0)
    {
        return sqrt (square(x1-x2) + square(y1 - y2));
    }
    double distance (const Vertex &other) const
    {
        return distance (m_x, m_y, other.m_x, other.m_y);
    }
    double distance () const
    {
        return distance(m_x,m_y);
    }
    size_t step() const {
        return m_step;
    }
    bool inner() const {
        return m_type == Inner || m_type == Border;
    }

    void neighbors (Vertex *const nb[4])
    {
        neighbors (nb[Top], nb[Right], nb[Bottom], nb[Left]);
    }
    void neighbors (Vertex *t, Vertex *r, Vertex *b, Vertex *l)
    {
        set_top(t);
        set_right(r);
        set_bottom(b);
        set_left(l);
    }
    bool full() const
    {
        return (m_neighbors[0]!= NULL &&
                m_neighbors[1] != NULL &&
                m_neighbors[2]!=NULL &&
                m_neighbors[3]!= NULL);
    }
    void set(const Vertex *v, Direction d) const
    {
        m_neighbors[d] = (Vertex *)v;
    }
    void set(const Vertex &v, Direction d) const
    {
        m_neighbors[d] = (Vertex *)&v;
    }
    void set_top(Vertex *top)
    {
        m_neighbors[Top] = top;
    }
    void set_right(Vertex *right)
    {
        m_neighbors[Right] = right;
    }
    void set_bottom(Vertex *bottom)
    {
        m_neighbors[Bottom] = bottom;
```

```
    }
    void set_left(Vertex *left)
    {
        m_neighbors[Left] = left;
    }

    void set_top(Vertex &top)
    {
        m_neighbors[Top] = &top;
    }
    void set_right(Vertex &right)
    {
        m_neighbors[Right] = &right;
    }
    void set_bottom(Vertex &bottom)
    {
        m_neighbors[Bottom] = &bottom;
    }
    void set_left(Vertex &left)
    {
        m_neighbors[Left] = &left;
    }
    size_t degree(const ::Neighbour *nb);

    Vertex *neighbor(Direction d)
    {
        return m_neighbors[d];
    }
    const Vertex *neighbor(Direction d) const
    {
        return m_neighbors[d];
    }
    Vertex *top() { return neighbor(Top); }
    Vertex *right() { return neighbor(Right); }
    Vertex *bottom() { return neighbor(Bottom); }
    Vertex *left() { return neighbor(Left); }

    const Vertex *top() const { return neighbor(Top); }
    const Vertex *right() const { return neighbor(Right); }
    const Vertex *bottom() const { return neighbor(Bottom); }
    const Vertex *left() const { return neighbor(Left); }

    bool operator < (const Vertex &other) const {
        return m_index < other.m_index;
    }
    bool operator == (const Vertex &other) const {
        return m_index == other.m_index;
    }
    bool operator > (const Vertex &other) const {
        return m_index > other.m_index;
    }
    bool operator < (size_t index) const {
        return m_index < index;
    }
    bool operator == (size_t index) const {
        return m_index == index;
    }
    bool operator > (size_t index) const {
        return m_index > index;
    }

    static Direction reverse(Direction d) {
        switch(d) {
        case Top:
            return Bottom;
        case Right:
            return Left;
        case Bottom:
            return Top;
        case Left:
            return Right;
        }
        return d;
```

```
    }
    static size_t index (size_t i, size_t j, size_t N, size_t M = 0)
    {
        return j*N + i;
    }
    static size_t index (double x, double y, size_t N, size_t M)
    {
        return index ((size_t) x*N, (size_t)y*M, N, M);
    }
    static std::pair<size_t, size_t> pos (size_t index, size_t N, size_t M = 0)
    {
        return std::pair<size_t, size_t>(index % N, index /N);
    }
    static std::pair<double, double> dpos (size_t index, size_t N, size_t M)
    {
        std::pair<size_t, size_t> p = pos(index, N, M);
        return std::pair<double, double> (p.first*1./N, p.second*1./M);
    }

protected:
    void init (size_t N, size_t M)
    {
        m_i = m_index % N;
        m_j = m_index / N;
        m_x = (1./N) * m_i;
        m_y = (1./N) * m_j;
        for (size_t i = 0; i < 4; ++i) {
            m_neighbors[i] = NULL;
        }
    }
private:
    void check_type (Vertex *nb)
    {
        if (nb == NULL) return;
        if (m_type == Inner && nb->m_type == Neighbour) {
            m_type = Border;
        }
    }

private:
    const size_t m_index;
    size_t m_i, m_j;
    double m_x, m_y;
    mutable double m_value[2];
    size_t &m_step;
    mutable Vertex *m_neighbors[4];
    Type m_type;
};

template <typename F>
class CondVertex : public Vertex {
public:
    CondVertex (size_t index, size_t N, size_t M, size_t &step,
                double t, const F &f, Type type)
        : Vertex (index, N, M, step, 0, type)
        , m_F (f)
        , m_T(t)
    {}
    using Vertex::set;
    virtual void set(double, size_t i=0 ) const { };
    virtual double get(size_t i = 1) const
    {
        return m_F(x(), y(), step()*m_T);
    };
private:
    F m_F;
    double m_T;
};

class NeighbourVertex : public Vertex {
public:
    NeighbourVertex (size_t index, size_t N, size_t M, size_t &step,
                     const ::Neighbour &nb)
```

```
        : Vertex(index, N, M, step, Vertex::Neighbour)
        , m_neighbour(nb)
    {}
    NeighbourVertex (const NeighbourVertex &v)
        : Vertex(v)
        , m_neighbour(v.m_neighbour)
    {}
    const ::Neighbour *neighbour() {
        return &m_neighbour;
    }

private:
    const ::Neighbour &m_neighbour;
};

#endif /* Vertex.hpp */
```