Gauss-Seidel

Algoritmos Paralelos

Luís Neto (a77763) João Alves (a77070)

Universidade do Minho,
5 de Setembro de 2019

1 Introdução

Um problema comum na física computacional é determinar o estado de equilíbrio da difusão de calor numa superfície, neste caso quadrangular, na qual é aplicada calor nas suas extremidades.

Neste caso de estudo, será utilizada a equação de Poisson para calcular aproximação do estado de equilíbrio. Como consequência será necessário resolver um sistema de equações lineares que possui a forma:

$$Ax = b$$

Existem vários métodos para resolver o sistema de equações linear e, neste caso, será utilizado um método iterativo de **Gauss-Seidel**.

Para além disto, será implementado um algoritmo com o esquema **Red-Black**, o que permitirá comparar o numero de iterações até ao estado de equilíbrio e implementar um algoritmo distribuído, para que seja possível melhorar e comparar temporalmente a aproximação.

2 Descrição do problema

Tal como foi referido anteriormente, o problema subjacente a este trabalho é a aproximação do estado de equilíbrio de uma superfície quadrangular, à qual é aplicada calor nas suas extremidades.

De forma a ser mais compreensível ao leitor, consideremos que a figura seguinte representa a placa.

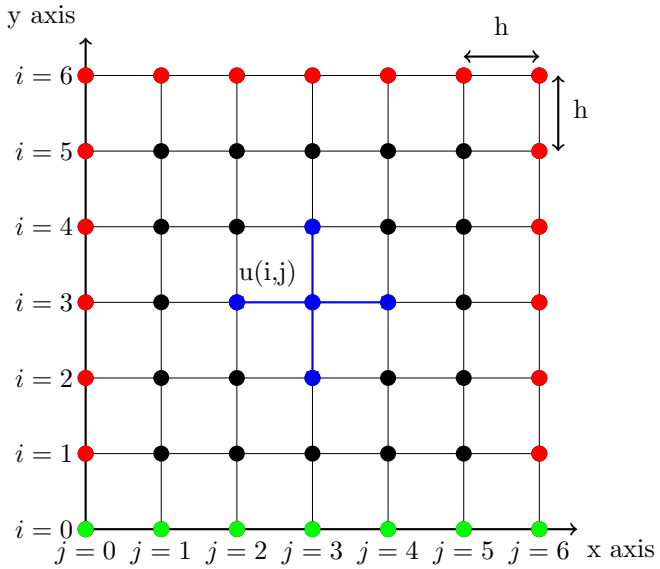

Figura 1: Representação do problema de difusão de calor no plano cartesiano

Como se pode observar na figura 1, a placa pode ser *vista* como uma malha de pontos. De notar que estes se encontram a uma distância h, onde $h = \frac{1}{N}$ e N o numero de pontos ou incógnitas.

Para este problema também irão ser consideradas as seguintes condições de fronteira:

- Pontos a vermelho: 100^{o} Celsius
- Pontos a verde: 0^{o} Celsius

Sem excluir, os pontos preto que se encontram a uma temperatura de 50^{o} Celsius.

Posto isto, o estado de equilíbrio pode ser aproximado através da equação de Poisson:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad \text{seja } eq - I$$

Descritizando $eq - I$ utilizando o esquema de diferenças centradas:

$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2} = 0 \quad \text{seja } eq - II$$

Como $\Delta x^2 = \Delta y^2$ vem que:

$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j} + u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{2\Delta x^2} = 0 \quad \text{seja } eq - III$$

Resolvendo $eq - III$ em ordem a $u_{i,j}$ temos:

$$u_{i,j} = \frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}}{4}$$

Esta equação pode ser traduzida a nível computacional como um **5 point-stencil**, que está assinalado a azul na figura 1. É também importante referir que o facto de ser utilizado esquema das diferenças centradas é introduzido um erro de truncatura, neste caso, $O\left(h^2\right)$. Este erro advém do uso de uma parte finita serie de Taylor na qual assenta o esquema, todavia, este erro pode ser desprezado quando h é suficientemente pequeno. Finalmente, a aproximação envolve encontrar o valor de N^2 incógnitas e resolver N^2 equações.

2.1 Método iterativo Gauss-Seidel

Na álgebra linear numérica, o método iterativo de Gauss-Seidel permite resolver sistema de equações lineares, este método é também muito semelhante ao método de Jacobi.

A computação da solução x^{k+1} utiliza os elementos de x^{k+1} cujo resultado já tenha sido calculado e os elementos de x^k, de notar que, como existe dependências no calculo das equações este método não pode ser paralelizado, contudo, se o esquema **Red-Black** for utilizado é possível chegar a uma versão paralela.

A nível computacional, isto permite que apenas seja utilizado uma *vector* ou *matriz* para armazenar a aproximação, o que se torna vantajoso quando o tamanho da malha é elevado. Para além disto, este método é um método *matrix-free*, isto é, não necessário armazenar a matriz de coeficientes para resolver o sistema, o que, mais uma vez, se torna vantajoso quando o tamanho da malha é elevado.

Finalmente, este método possui como condição de convergência a seguinte inequação:

$$max(x^{k+1} - x^k) \leq h^2$$

Esta condição foi escolhida, uma vez que, não existe muita relevância em calcular uma solução do sistema com muitos algarismos correctos, quando estes mesmos algarismos já estão afectados pelo erro de truncatura, introduzido pelo método de diferenças finitas.

2.2 Esquema Red-Black

O esquema Red-Black quando aplicado ao método de Gauss-Seidel permite seja executado em paralelo. Mais uma vez, com o intuito de ser mais perceptível ao leitor, foram criadas as seguintes imagem para que seja possível compreender o que este esquema faz.

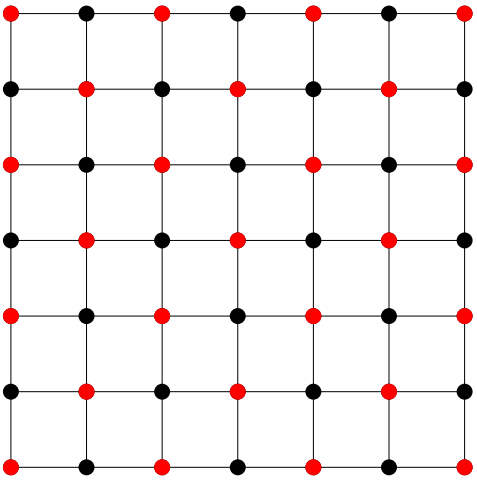

Figura 2: Malha antes do calculo do estado vermelho

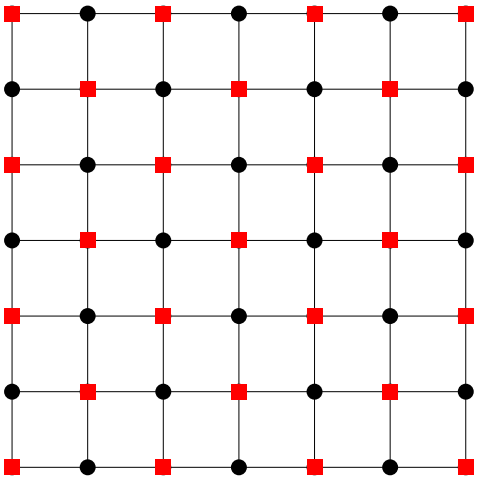

Figura 3: Malha após do calculo do estado vermelho

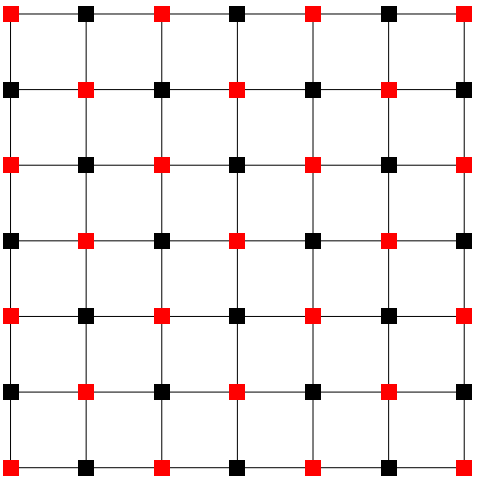

Figura 4: Malha após do calculo do estado preto

Com observação das figuras 2, 3 e 4 é possível identificar dois estados, o vermelho e o preto. Estes estados indicam quais os pontos que podem ser processados de forma simultânea, uma vez que, para efectuar o calculo dos pontos vermelhos estes apenas dependem de pontos pretos e vice-versa, sendo assim, possível implementar um uma versão paralela do método.

3 Implementação MPI

A computação paralela pode ser vista como um paradigma, em que a execução de certas operações ou tarefas é feita em simultâneo [1].

Os principais paradigmas utilizados, quando se desenvolvem aplicações são paralelas, são **memória partilhada** e **memória distribuída**. A principal vantagem do primeiro, reside no facto de existir um espaço de endereçamento partilhado, que contem todos os dados necessários, para a execução de um programa.

Memória distribuída encontra-se normalmente associada com arquiteturas multi-processador, onde cada um possui o seu espaço de endereçamento de memória privado. Por isso, qualquer tarefa computacional a realizar opera com os dados locais e, quando necessário, esta deve comunicar com os processadores remotos, de maneira a obter os dados requeridos que não existem no seu espaço de memória privado, através de uma forma de interconexão.

Pode-se então concluir que a principal vantagem associada à computação com memória distribuída encontra-se na exclusão de *race conditions*, sendo que o principal desafio destes sistemas consiste na forma como os dados e as tarefas são distribuídas pelos respetivos processadores.

Com isto em mente e com o intuito de modelar uma implementação com memória distribuída, para o problema em questão, seguiu-se a metodologia de **Foster** [2], para desenvolver aplicações **MPI**.

Em computação paralela, o *Message Passing Interface* (**MPI**) fornece uma infraestrutura, que permite a processadores, nodos ou clusters trocarem informações necessárias, para a computação de um problema.

A metodologia de **Foster** passa por quatro fases distintas, sendo que cada uma será analisada nas próximas secções.

Contudo, é possível, neste momento, identificar que o objetivo passa por processar um conjunto de dados, neste caso uma matriz, ao qual é aplicada uma mesma operação. Uma vez que se trata de um processo iterativo, o valor de cada ponto é calculado com base em valores de iterações anteriores.

3.1 Partição do Problema e dos Dados a Processar

Posto isto, a primeira fase consiste em identificar as oportunidades de paralelismo, existentes no algoritmo em estudo. Deste modo, decidiu-se decompor o domínio de dados, para que este pudesse ser processado em paralelo.

Antes de mais, é possível verificar que a placa descrita no problema pode ser modelada como uma matriz, onde cada elemento corresponde a um ponto da placa e o seu valor à temperatura nesse ponto.

Com isto em mente, considerar-se-á cada tarefa como um elemento da matriz, originando assim um número elevado de tarefas de grão fino, que serão executadas em paralelo.

3.2 Identificação da Comunicação Necessária

Estas tarefas geradas, no entanto, não podem executar de forma independente, uma vez que necessitam de dados associados a outras. Por isso, é necessário identificar o tipo de comunicação adequado, para este algoritmo.

Sendo este um algoritmo iterativo, cada tarefa deve, para cada iteração, comunicar com outras tarefas, para obter os valores das iterações anteriores. Desta forma, é possível ver que a comunicação é local, porque requer que cada uma transmita esses valores a um pequeno conjunto de tarefas vizinhas.

Todavia, uma tarefa e as suas vizinhas formam uma estrutura regular, podendo-se dizer estruturada. Contudo, é necessário que haja conhecimento global do valor da condição de verificação, de modo a que todo o sistema converja para um estado uniforme. Deste modo, foi necessário implementar também um mecanismo de comunicação global, que garante o conhecimento deste valor, por parte de todos os intervenientes.

E por último, pode-se concluir que a comunicação deve ser estática, uma vez que os parceiros com que cada tarefa troca mensagens, não se alteram durante a execução do algoritmo.

3.3 Aglomeração de Computação e de Comunicação

Depois de decompor o domínio de dados, em tarefas de baixa granularidade, e determinado o tipo de comunicação necessário, para tornar a execução do algoritmo eficiente, é essencial aglomerar tarefas e/ou comunicação.

A aglomeração de comunicação pode ser efetuada juntando várias mensagens, por forma a reduzir os custos de comunicação. A junção de computação, pode ser feita aglomerando vários tarefas, de modo a eliminar a necessidade de comunicação remota entre elas.

No caso do problema em estudo, a utilização de um ponto da matriz por tarefa origina um grão demasiado fino, para a generalidade das arquiteturas, tornando-se então conveniente aglomerar várias tarefas. Visto isto, decidiu-se que seriam aglomeradas várias linhas, constituídas por elementos da matriz, aumentando assim o número de operações por tarefa e reduzindo o número de mensagens por cada iteração.

3.4 Mapeamento das Tarefas no Sistema

Neste última fase, é necessário especificar onde cada tarefa irá ser executada, com intuito de minimizar o tempo de execução.

Uma vez que se trata de um problema com paralelismo de dados, uma estrutura regular e com tarefas de dimensão idêntica, o mapeamento é linear. Por isso, pode ser feita uma partição regular da matriz, pelos nodos do sistema.

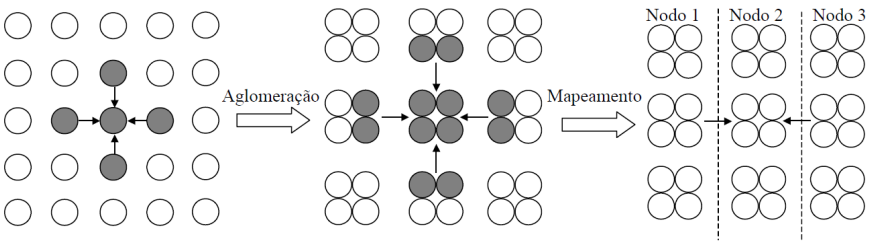

Figura 5:

4 Análise de Resultados

Com a implementação MPI foram retirados resultados para um número variável de equações (N^2), uma tolerância ($\frac{1}{N^2}$), onde N pertencente ao intervalo, e com um número de processos variável entre 1 e 4. Os resultados seguintes demonstram os tempos de execução, numero de iterações e tempo por iteração do algoritmo

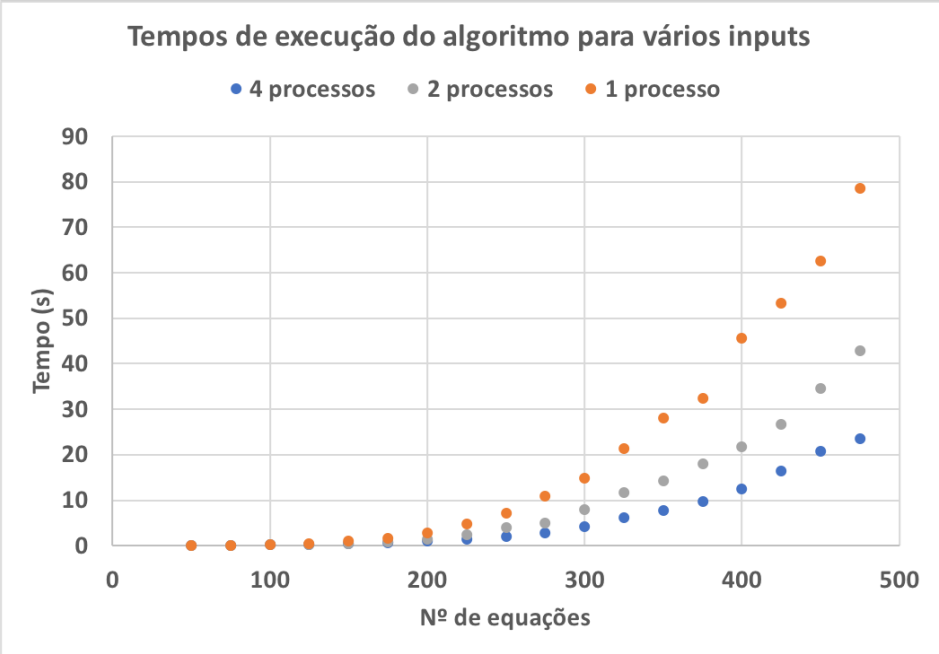

Figura 6: Tempos de execução para um numero de equações e processos variável

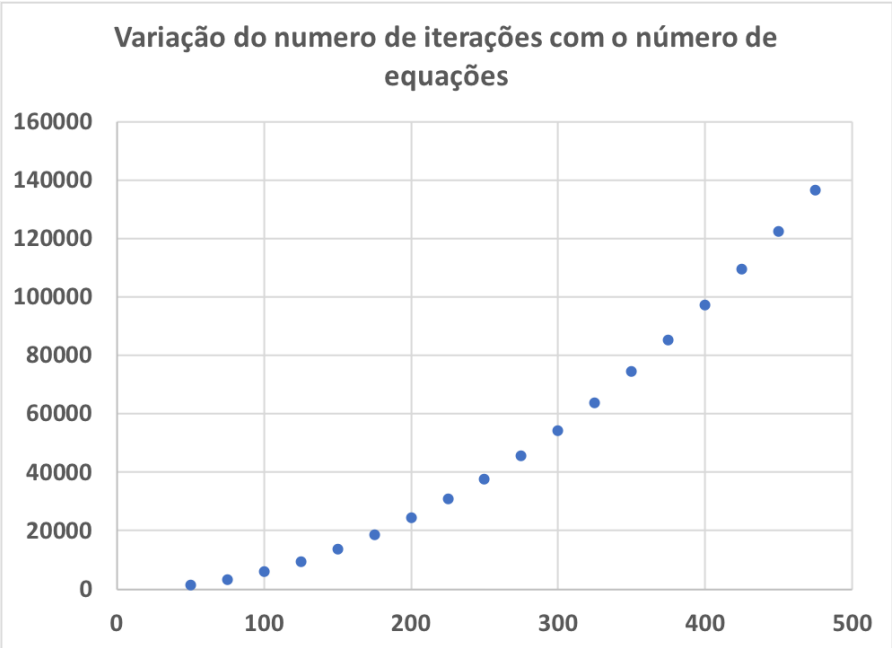

Figura 7: Numero de iterações para um numero de equações variável

Através destes resultados é possível observar que o tempo de execução e número de iterações crescem de forma exponencial em relação ao número de equações. De seguida é possível verificar que a utilização de mais processos melhora os tempos de execução, no entanto, o número de iterações mantém-se semelhante entre as execuções.

Referências

[1] G. S. Almasi e A. Gottlieb. *Highly Parallel Computing*. Benjamin-Cummings Publishing Co, 1989.

[2] Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

Apêndice A Código fonte

A.1 GS.hpp

```
#ifndef _GS_
#define _GS_
#include <iostream>
#include <vector>
#include <mpi.h>
#include <cmath>
#include <algorithm>
using std::cout;
using std::endl;
using std::vector;
using std::begin;
using std::end;
#define LOWER 0
#define LEFT 1
#define RIGHT 2
#define ABOVE 3
void printvector(vector<double>m, int row, int col);
void init_vector(vector<double>& mat, double*
                 boundary_temp, double interior_temp, int size);
int steady_state(vector<double>& u,double tolerance, int row, int col);
int mpi_steady_state(vector<double>& u,  double tolerance, int row, int col,
                     int id, int procs);
#endif
```

A.2 GS.cpp

```
#include "GS.hpp"
void init_vector(vector<double>& res, double* boundary_temp,
                 double interior_temp, int size)
{
    vector<double> mat(size*size);
    for(int i = 0; i < size; ++i){
        mat[i] = boundary_temp[LOWER];
        mat[i*size] = i < size-1 ? boundary_temp[LEFT]: boundary_temp[ABOVE];
        mat[i*size+size-1] = boundary_temp[RIGHT];
        mat[(size-1)*size+i] = boundary_temp[ABOVE];
    }
    for(int i = 1; i < size-1; ++i){
        for(int j = 1; j < size-1; ++j){
            mat[i*size+j] = interior_temp;
        }
    }
    res.swap(mat);
}

void printvector(vector<double> m, int row, int col)
```

```
{
    for(int i=0; i<row ;++i)
    {
        for (int j=0; j < col;++j)
        {
            cout<<m[col*i+j]<<";";
        }
        cout<<endl;
    }
}



void red_state(vector<double>& u, int row, int col, double& max)
{
    bool flag = true;
    int r = 0;
    int b = 0;
    double tmp = 0.0;
    for (int i=1; i < row-1; ++i){
        for (int j=1; j < col-1; ++j){
            if(flag){
                tmp = u[i*col+j];
                u[i*col+j]=
                            (
                                u[(i-1)*col+j]  +
                                u[i*col+j-1]    +
                                u[i*col+j+1]    +
                                u[(i+1)*col+j]
                            ) *0.25;
                tmp = fabs(u[i*col+j] - tmp);;
                max = max < tmp ? tmp : max;
                r++;
            }
            else{
                b++;
            }
            flag = !flag;
        }
        if(r==b){
            flag = !flag;
        }
        r = b = 0;
    }
}

void black_state(vector<double>& u, int row, int col,  double& max)
{
    bool flag = false;
    int r = 0;
```

```
    int b = 0;
    double tmp = 0.0;
    for (int i=1; i < row-1; ++i){
        for (int j=1; j < col-1; ++j){
            if(flag){
                tmp = u[i*col+j];
                u[i*col+j]=
                            (
                                u[(i-1)*col+j]  +
                                u[i*col+j-1]    +
                                u[i*col+j+1]    +
                                u[(i+1)*col+j]
                            ) *0.25;
                tmp = fabs(u[i*col+j] - tmp);
                max = max < tmp ? tmp : max;
                b++;
            }
            else{
                r++;
            }
            flag = !flag;
        }
        if(r==b){
            flag = !flag;
        }
        r = b = 0;
    }
}

int steady_state(vector<double>& u,double tolerance, int row, int col)
{
    int iteration = 0;
    double diff = 0;
    vector<double> temp(row*col);
    do{
        diff = 0;
        red_state(u,row,col,diff);
        black_state(u,row,col,diff);
        ++iteration;
    }while(diff>tolerance);

    return iteration;

}




void mpi_ss_comm(
    int id,
```

```
    vector<double>& u_local,
    int row,
    int col)
{
    int procs;
    vector<double> u_cpy_snd(col);
    vector<double> u_cpy_fst(col);

    MPI_Comm_size(MPI_COMM_WORLD, &procs);
    if (id != 0){
        MPI_Request w_req_fst;
        std::copy(
            u_local.data() + col,
            u_local.data() + 2*col,
            u_cpy_fst.data());
        MPI_Isend(u_cpy_fst.data(), col, MPI_DOUBLE, id-1, 0,
                  MPI_COMM_WORLD, &w_req_fst);
    }
    if (id + 1 != procs) {
        MPI_Request w_req_snd;
        std::copy(
            u_local.data() + u_local.size() - col*2,
            u_local.data() + u_local.size() - col,
            u_cpy_snd.data());
        MPI_Isend(u_cpy_snd.data(), col, MPI_DOUBLE, id+1, 0,
                  MPI_COMM_WORLD, &w_req_snd);
    }

    if(id != 0)
    {
        MPI_Status recv_stat;
        MPI_Recv(
            u_local.data(), col, MPI_DOUBLE,
            id-1, 0, MPI_COMM_WORLD, &recv_stat);
    }
    if(id+1 != procs)
    {
        MPI_Status recv_stat;
        MPI_Recv(
            u_local.data() + u_local.size() - col, col, MPI_DOUBLE,
            id+1, 0, MPI_COMM_WORLD, &recv_stat);
    }
}


void mpi_ss_error(double local_diff, double & diff)
{
    MPI_Allreduce(&local_diff, &diff, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
}
```

```
void mpi_ss_iteration(bool pattern, vector<double>& u_local, int row, int col,
                      double& max)
{
    if (pattern) {
        red_state(u_local,row,col, max);
    }
    else{
        black_state(u_local,row,col,max);
    }
}

void mpi_ss_init(
    vector<int> dis,
    vector<int> count,
    vector<double>& u,
    vector<double>& u_local,
    int size){
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Scatterv(
        u.data(), count.data(), dis.data(), MPI_DOUBLE,
        u_local.data(), size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        }




void mpi_gather_result(int id, int local_row, int col, vector<int> dis,
                       vector<int> count, vector<double> u_local, vector<double>& u){
    if(id == 0)
    {
        std::transform(dis.begin(),   dis.end(),   dis.begin(),
                       [col](int& d) {return d + col;});
        std::transform(count.begin(), count.end(), count.begin(),
                       [col](int& c) {return c - 2*col;});
    }
    MPI_Gatherv(
        u_local.data() + col, (local_row-2)*col, MPI_DOUBLE,
        u.data(), count.data(), dis.data(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
}



int mpi_steady_state(vector<double>& u, double tolerance, int row,
                     int col, int id, int procs)
{
    int work = round((row-2)/procs);
    vector<int> dis;
    vector<int> count;
    int iteration=0;
```

```
    double diff = 0;
    double local_diff = 0;
    bool terminate = false;
    bool first = procs*work % 2 ? true : false;
    bool second = !first;
    int local_row;
    for (int p = 0; p < procs; ++p)
    {
        dis.push_back(p * work * col);
        count.push_back((work + 2)*col);
        local_row = (work + 2);
    }

    if ((count[procs-1] + dis[procs-1]) != row){
        count[procs-1] = (row*col - dis[procs-1]);
        if(id == procs-1)
            local_row = (row*col - dis[procs-1])/col;
    }
    vector<double> u_local(count[id]);
    vector<double> w_local(count[id]);



    mpi_ss_init(dis, count, u, u_local, count[id]);
    do{

        local_diff = 0.0;
        mpi_ss_iteration(first, u_local, local_row, col, local_diff);
        mpi_ss_comm(id, u_local,local_row, col);
        mpi_ss_iteration(second, u_local, local_row, col, local_diff);
        mpi_ss_error(local_diff, diff);
        if(diff > tolerance){
            mpi_ss_comm(id, u_local, local_row, col);
            ++iteration;
        }
        else
        {
            break;
        }
    }while(true);

    mpi_gather_result(id, local_row, col, dis, count, u_local, u);

    return iteration;
}
```

A.3 main.cpp

```
#include <cstdlib>
#include <ctime>
#include <iomanip>
#include <iostream>
#include <mpi.h>
#include <algorithm>
#include <vector>
#include <cmath>
#include <chrono>
#include "GS.hpp"
using std::cout;
using std::endl;
using std::vector;




int main(int argc, char* argv[]){
    int print = 0;
    if(argc < 2){
        return -1;
    }
    if(argc == 3){
        print = atoi(argv[2]);
    }
    double boundary_temp[4] = {100,0,0,0};
    double interior_temp = 50;
    int size = atoi(argv[1]);
    double tolerance = 1/(powf(size,2));
    vector<double> u(size*size);
    vector<double> w(size*size);
    int id;
    int ierr;
    int procs;

    std::chrono::time_point<std::chrono::high_resolution_clock> start, end;
    ierr = MPI_Init ( &argc, &argv );
    ierr = MPI_Comm_size ( MPI_COMM_WORLD, &procs );
    ierr = MPI_Comm_rank ( MPI_COMM_WORLD, &id );
    if(id == 0){
        init_vector(u, boundary_temp, interior_temp, size);
        start = std::chrono::high_resolution_clock::now();
    }
    int iteration = mpi_steady_state(u,tolerance,size,size,id,procs);
    if(id==0){
        end = std::chrono::high_resolution_clock::now();
        auto millis = std::chrono::duration_cast<std::chrono::milliseconds>(end-start).cou
```

```
        if(print){
            printvector(u,size,size);
        }else{
            cout << "iteration:"<<iteration<<" millisec:"<< millis<<endl;
        }
    }
    MPI_Finalize ();
    return 0;
}
```