Задание 2
Численное решение задачи Дирихле. Разработка параллельной MPI-программы и исследование ее эффективности.
Вариант 2
Метод SOR

Выполнила студентка 521 группы
Коростелева Мария Викторовна

Постановка задачи. Разработать параллельную программу с использованием технологии MPI, реализующую решение 2-мерной задачи Дирихле с помощью метода SOR. Провести исследование эффективности разработанной программы на системах Blue Gene/P и «Ломоносов». Выполнить визуализацию полученного решения.

Цель. Получить навыки разработки и исследования параллельных программ с использованием технологии MPI.

Требуется.

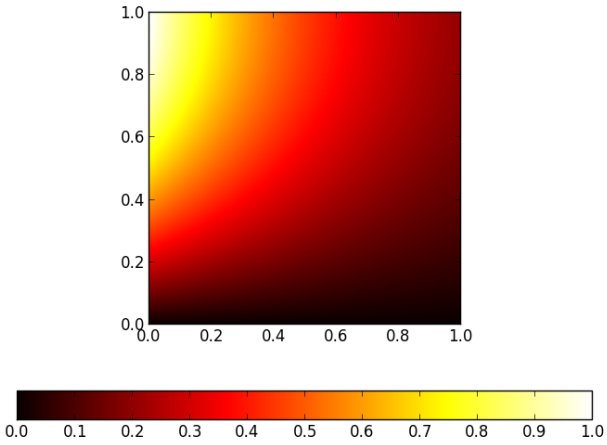
1. Разработать параллельную программе с использованием технологии MPI.
2. Исследовать время выполнения разработанной программы в зависимости от задаваемой точности, размера сетки и количества используемых процессов на вычислительных системах Blue Gene/P и «Ломоносов».

Описание метода SOR:

Метод SOR решения задачи Дирихле для уравнения Лапласа в двумерном пространстве отличается от простейшего итерационного метода Якоби тем, что на каждой итерации в качестве результата берется взвешенная сумма значения, полученного на данной итерации и значения, полученного на предыдущей итерации. При этом вычисляемое на данной итерации значение в ячейке сетки использует, когда это возможно, уже обновленные на данной итерации значения соседних ячеек, что является залогом сходимости метода. Выбор весов и краевых значений описан в лекциях.

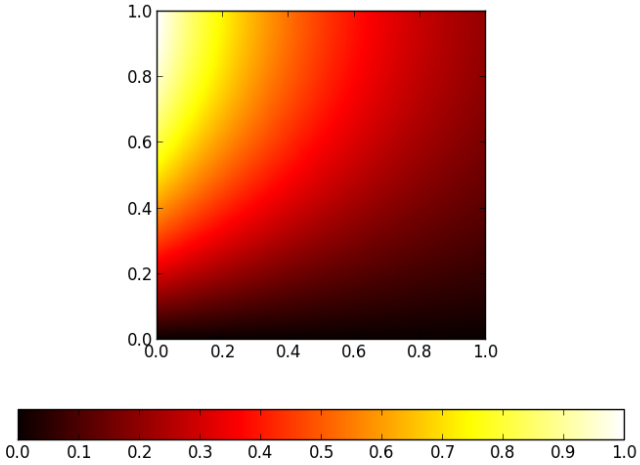
Визуализации решения:

1. Аналитическое решение, сетка 512x512

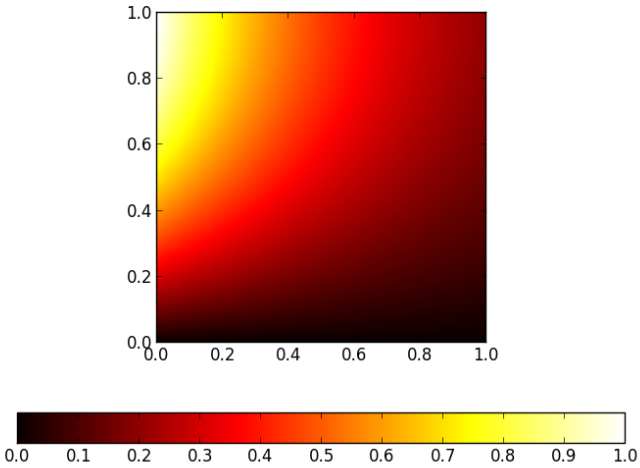


1.

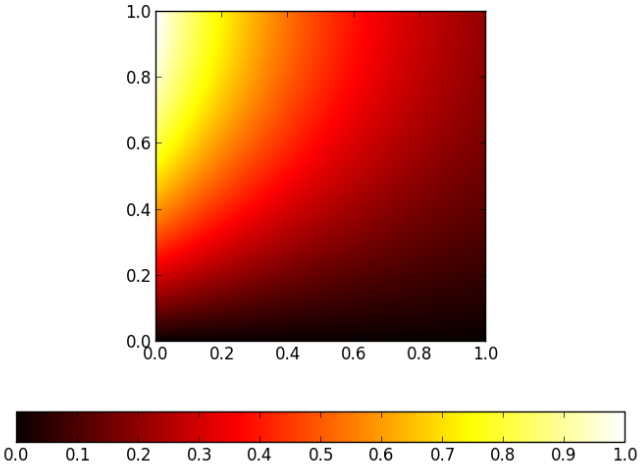
2. Сетка 512x512, точность 0.01



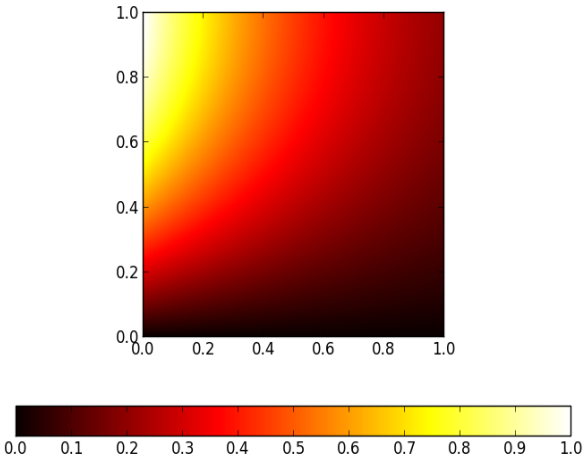
3. Сетка 512x512, точность 0.001



4. Сетка 1024x1024, точность 0.01

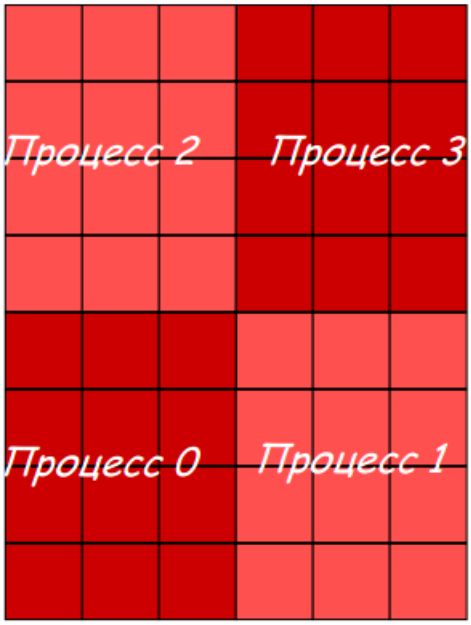


5. Сетка 1024x1024, точность 0.001



Описание параллельной программы:

Для создания параллельной программы и общения между исплняемыми на разных узлах процессами используется технология MPI. Матрица, которая итеративно обсчитывается в процессе выполения программы и поиска приближенного решения поставленной задачи, делится на блоки и каждый блок отдается на обсчет отдельному процессу:



Для этого используется виртуальная топология решетки, реализованная средствами MPI. Ускорение здесь должно достигаться за счет того, что блоки, относящиеся к одной побочной диагонали могут выполнятся параллельно. Однако побочные диагонали выполняются последовательно: для подсчета очередного блока требуются новые данные от блока слева и сверху.

I. Blue Gene/P

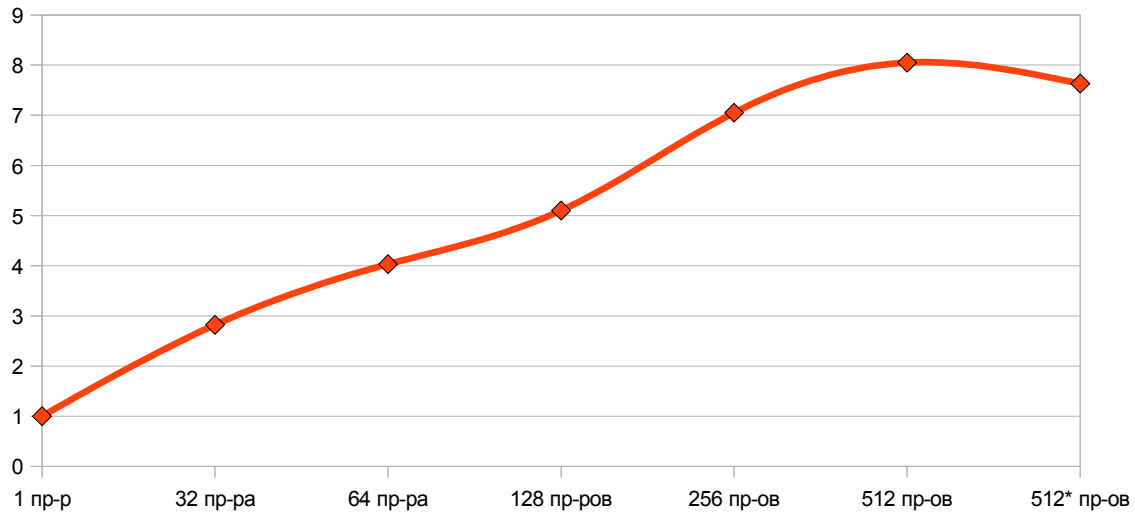
Сравнительная таблица для разработанной параллельной программы в зависимости от размера сетки, точности и количества используемых потоков на вычислительной системе Blue Gene/P :

Размер сетки	Точность	Параллельный алгоритм											
		1 процессор			32 процессора			64 процессора			128 процессоров		
		Время	Ускорение	Число итераций	Время	Ускорение	Число итераций	Время	Ускорение	Число итераций	Время	Ускорение	Число итераций
512x512	0.01	17,86	1	436	6,33	2,82	436	4,43	4,03	436	3,5	5,1	436
512x512	0.001	28,6	1	699	10,15	2,81	699	7,1	4,02	699	5,6	5,11	699
1024x1024	0.01	142,58	1	872	49,73	2,87	872	34,09	4,18	872	26,51	5,38	872
1024x1024	0.001	229,07	1	1402	79,92	2,87	1402	54,78	4,18	1402	42,58	5,38	1402

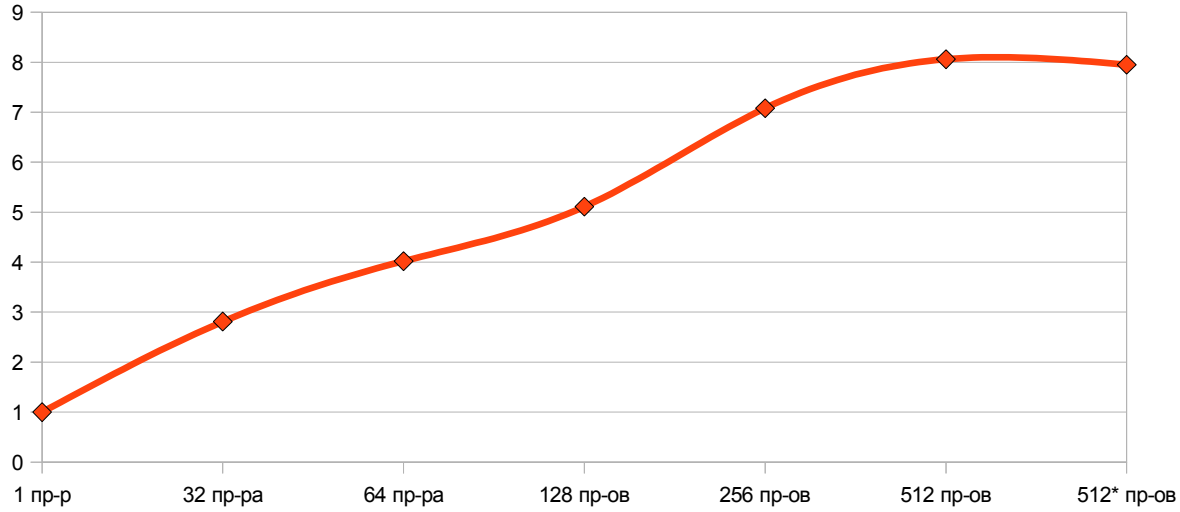
Размер сетки	Точность	Параллельный алгоритм								
		256 процессоров			512 процессоров, стандартный маппинг			512 процессоров, произвольный маппинг		
		Время	Ускорение	Число итераций	Время	Ускорение	Число итераций	Время	Ускорение	Число итераций
512x512	0.01	2,53	7,05	436	2,22	8,05	436	2,34	7,63	436
512x512	0.001	4,04	7,08	699	3,55	8,06	699	3,56	7,95	699
1024x1024	0.01	18,31	7,79	872	14,26	10	872	14,31	9,96	872
1024x1024	0.001	29,41	7,79	1402	22,9	10	1402	22,9	10	1402

Графики ускорения:

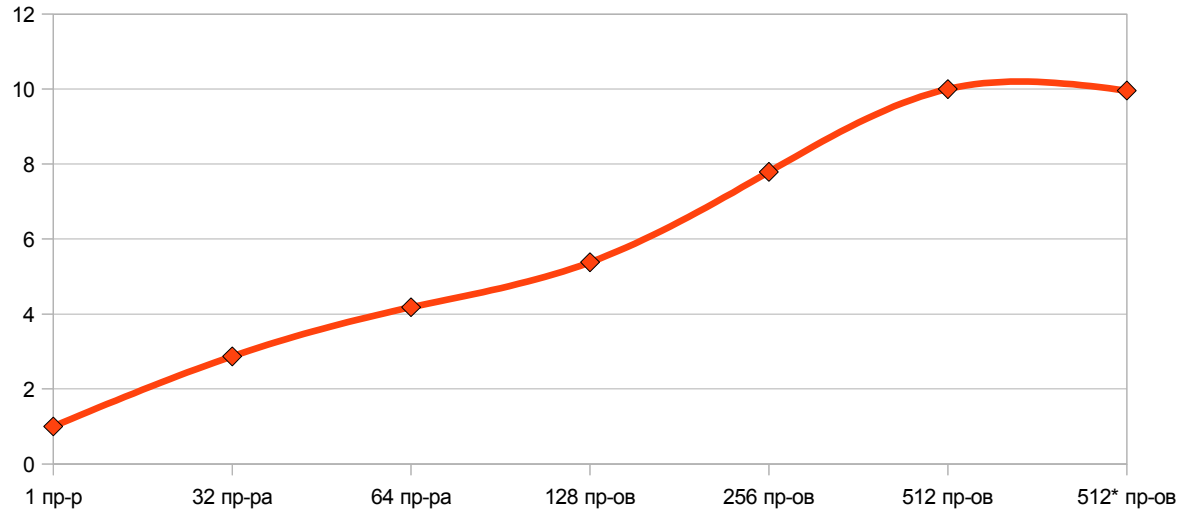
1. Сетка 512x512, точность 0.01



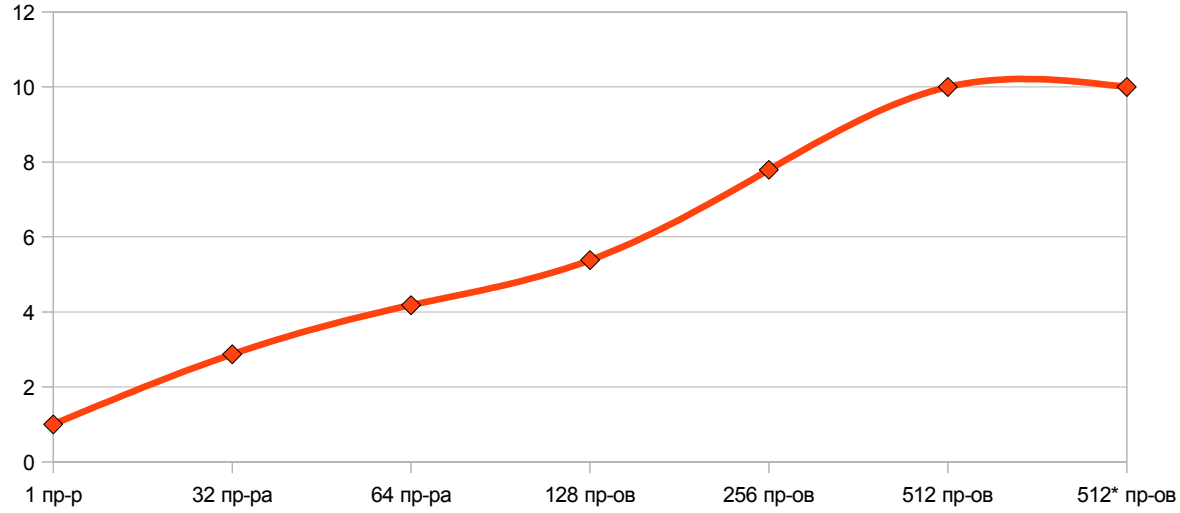
2. Сетка 512x512, точность 0.001



3. Сетка 1024x1024, точность 0.01



4. Сетка 1024x1024, точность 0.001



II. «Ломоносов»

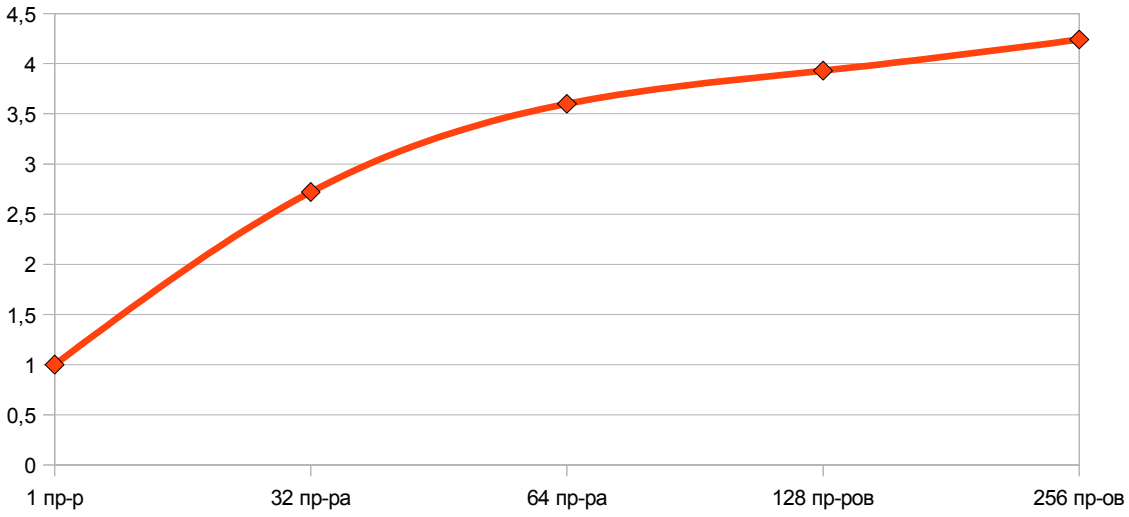
Сравнительная таблица для разработанной параллельной программы в зависимости от размера сетки, точности и количества используемых потоков на вычислительной системе «Ломоносов» :

Размер сетки	Точность	Параллельный алгоритм								
		1 процессор			32 процессора			64 процессора		
		Время	Ускорение	Число итераций	Время	Ускорение	Число итераций	Время	Ускорение	Число итераций
512x512	0.01	1,06	1	436	0,39	2,72	436	0,3	3,6	436
512x512	0.001	1,69	1	699	0,62	2,73	699	0,46	3,67	699
1024x1024	0.01	8,51	1	872	2,96	2,88	872	2,09	4,07	872
1024x1024	0.001	13,81	1	1402	4,74	2,91	1402	3,3	4,18	1402

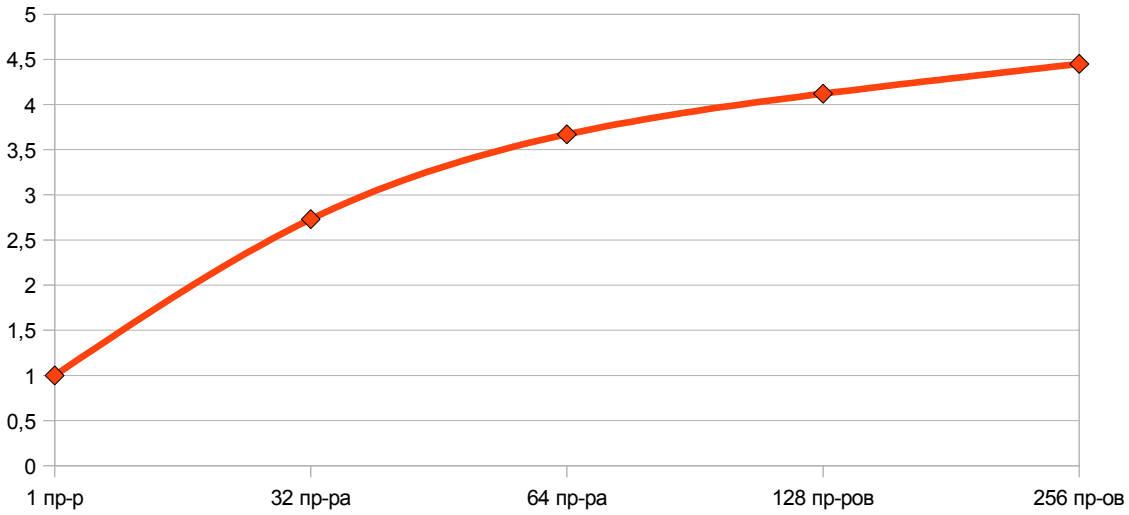
Размер сетки	Точность	Параллельный алгоритм					
		128 процессоров			256 процессоров		
		Время	Ускорение	Число итераций	Время	Ускорение	Число итераций
512x512	0.01	0,27	3,93	436	0,25	4,24	436
512x512	0.001	0,41	4,12	699	0,38	4,45	699
1024x1024	0.01	1,67	5,1	872	1,22	6,98	872
1024x1024	0.001	2,68	5,15	1402	1,92	7,19	1402

Графики ускорения:

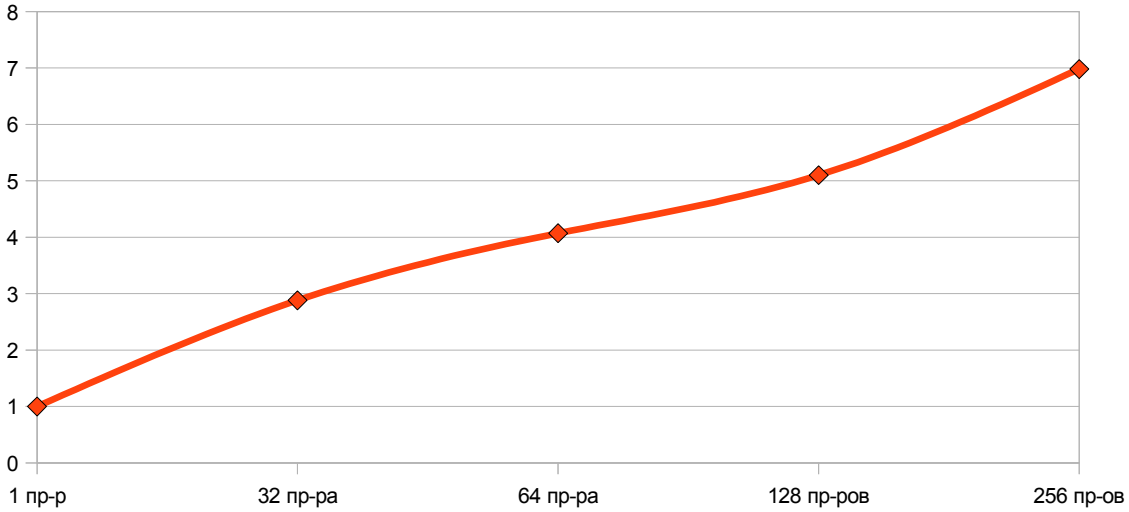
1. Сетка 512x512, точность 0.01



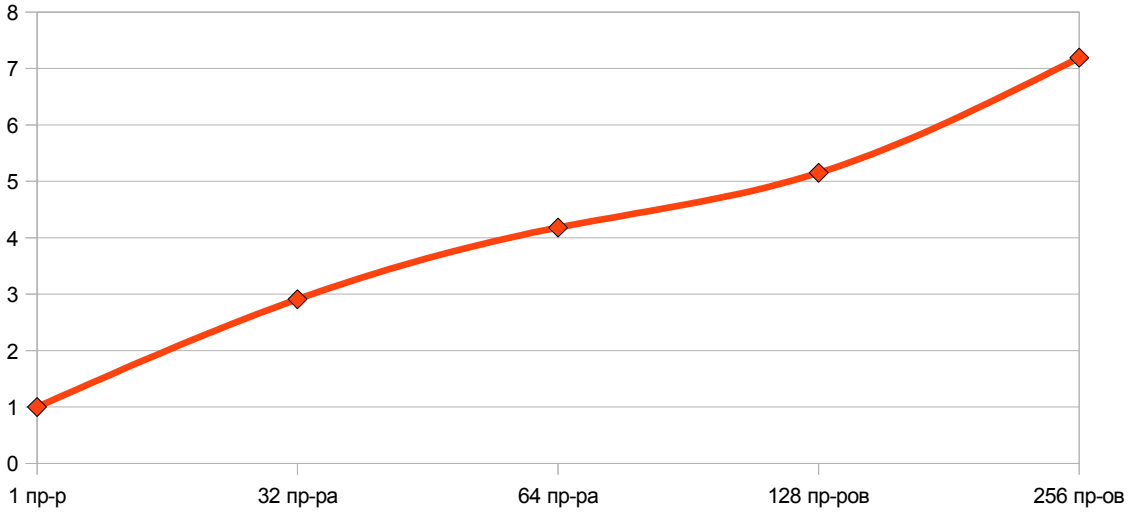
2. Сетка 512x512, точность 0.001



3. Сетка 1024x1024, точность 0.01



4. Сетка 1024x1024, точность 0.001



Выводы:

Во всех представленных вариантах запусков программы (сетки 512 и 1024, точности 0,01 и 0,001) скорость работы параллельной программы возрастает практически линейно, что логично, т.к. программе предоставляется все большее количество ресурсов при увеличении числа используемых процессоров. Ускорения в целом не очень большие, т.к. внутри распаллеленного цикла имеется существенная зависимость по данным между витками, которая требует обслуживания в виде отсылки и приема сообщений с недостающими данными —> тратятся ресурсы узла и канала связи, а так же не позволяет отдельным процессам работать полностью параллельно внутри одной итерации, как описано выше, пааллельно выполняются процессы, обрабатывающие блоки матрицы, лежащие на одной побочной диагонали и только. Побочные диагонали исполняются последовательно друг за другом.

Так же можно заметить, что ускорения для сетки размером 1024 выше, чем для сетки размером 512, это связано с более оптимальным использванием пересылок между соседними процессорами — сообщения MPI помимо значимых данных содержат большое количество служебной информации, причем наклодные расходы в рассчете на одну пересылку данных в рамках наших сеток получаются очень существенными, поэтому оптимальнее рассылать блоки побольше, что и можно заметить, сравнив ускорения для сеток 512 и 1024.

Произвольное распределение процессов по узлам вычислительного комплекса существенно не повлияло на производительность программы, хотя иногда пользовательски определенный маппинг может ускорить выполнение программы за счет расположения часто общающихся друг с другом процессов на соседних процессорах — меньшее вермя доставки собщения соседу. Топология выч.узлов системы Blue Gene/P — это трехмерный тор, программа использует 2D распределение элементов матрицы по отдельным процессам, поэтому в данном случае довольно трудно хорошо оптимизировать распределение процессов по процессорам. Вероятно, в следующем задании на решение задачи Дирихле для уравнения Лапласа в трехменом пространстве получится выгадать несколько секунд на более удачном расположении процессов.

Ускорения, полученные в результате вычислений на вычислительном комплексе «Ломоносов», хотя и ожидаемо возрастают с увеличением числа используемых процессов, оказались в целом меньше, чем результаты запусков на вычислительном комплексе BlueGene/P. Предположительно этот факт связан с более высокой вычислительной мощьностью отдельных узлов ВК «Ломоносов», в связи с чем выигрыш от параллельности вычислений не столь существеннен по сравнению с накладными расходами на загрузку ядер и передачу сообщений между отдельными процессами. Однако, этот факт не говорит однозначно против использования «Ломоносова», так как абсолютные цифры времени выполенения параллельной программы значительно выше результатов аналогичных запусков на BlueGene/P, то есть программа работает значительно быстрее.

Текст параллельной программы

Параметры, передаваемые в командной строке:

Первый параметр: m – число точек по одному измерению для задания двумерной сетки. По умолчанию – 512.

Второй параметр – точность. По умолчанию – 0.01.

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <mpi.h>
#include <math.h>
```

```
#define Max(a,b) ((a)>(b)?(a):(b))

double maxeps = 0.01;
int N = 512;
// spectral radius
int itmax = 10000;
int proc_size;
int proc_rank;
int box_size_i;
int box_size_j;
MPI_Comm topology;
double eps;
double** U = NULL;
void relax();
void init();
void verify();
void wtime();

void verify()
{
	double s;
	s = 0.;
	int i, j;
	for(i = 0; i < N+2; i++)
		for(j = 0; j < N+2; j++)
			{
				s = s + U[i][j] * (i + 1) * (j + 1) / (N * N);
			}
	printf(" S = %f\n",s);
}
void wtime(double *t)
{
	static int sec = -1;
	struct timeval tv;
	gettimeofday(&tv, (void *)0);
	if (sec < 0)
		sec = tv.tv_sec;
	*t = (tv.tv_sec - sec) + 1.0e-6*tv.tv_usec;
}

double loc_prev_om = 0;
double omega(int n)
{
   double p = 1. - (M_PI / ((N - 1)))*(M_PI / ((N - 1)));
	double om;
	if (n == 0)
		loc_prev_om = 0;
	else if (n == 1)
	{
		loc_prev_om = 1./(1. - p*p/2);
	}
	else
```

```
        {
                loc_prev_om = 1./(1. - loc_prev_om*p*p/4);
        }
        if (proc_rank == 0)
        {
                //printf("Omega is %lf with n == %d, p is %lf\n", loc_prev_om, n, p);
        }
        return loc_prev_om;
}
void output(const char* name)
{
        // write matrix to file
        FILE* grid = fopen(name, "w");
        int i, j;
        for (i = 0; i < box_size_i+2; i++)
        {
                for (j = 0; j < box_size_j+2; j++)
                {
                        fprintf(grid, "%lf ", U[i][j]);
                }
                fprintf(grid, "\n");
        }
        fclose(grid);
}
void init()
{
        int coords[2], r;
   MPI_Comm_rank(topology, &r);
        MPI_Cart_coords (topology, r, 2, coords);
        //printf( "Hello world from process %d of %d\n", proc_rank, proc_size );
        //printf( "Hello world from coords %d x %d\n", coords[0], coords[1]);
        int i, j;
        U = calloc(box_size_i + 2, sizeof(U[0]));
        for (i = 0; i < box_size_i + 2; i++)
        {
                U[i] = calloc(box_size_j + 2, sizeof(U[i][0]));
        }
   for(i=0; i< box_size_i+2; i++)
      for(j=0; j< box_size_j+2; j++)
      {
                        int ti = box_size_i*coords[0] + i;
                        int tj = box_size_j*coords[1] + j;

                        double x = ti * (1.0 / (N + 1));
                        double y = tj * (1.0 / (N + 1));

                        if(tj == 0)
        {
                                U[i][j] = sin(M_PI * x / 2);
        }
                        else if (tj == N + 1)
                        {
```

```
                        U[i][j] = sin(M_PI * x / 2) * exp(-M_PI/2);
                }
                else if (ti == N + 1)
                {
                        U[i][j] = exp(- M_PI * y / 2);
                }
                else
                        U[i][j] = 0;
                        //U[i][j] = sin(M_PI * x / 2) * exp(-M_PI * y/2);
        }
        /*
        char out[19];
        sprintf(out, "grid_init_%d.txt", proc_rank);
        output(out);
        */
}

// parameter: iteration number
void relax(int iter)
{
        int coords[2], r;
   MPI_Comm_rank(topology, &r);
        MPI_Cart_coords (topology, r, 2, coords);
        MPI_Status status;
        MPI_Request request;

   int i, j;
        // Recieve current iteration data from top and left processes
        if (coords[0] - 1 >= 0) // top
        {
                int r_top, coords_top[2];
                coords_top[0] = coords[0] - 1;
                coords_top[1] = coords[1];
                MPI_Cart_rank(topology, coords_top, &r_top);
                double* buf = (double*) malloc ((box_size_j+2) * sizeof(double));
                MPI_Recv(buf, box_size_j+2, MPI_DOUBLE, r_top, 0,
                        topology, &status);
                int count;
                MPI_Get_count (&status, MPI_DOUBLE, &count);
                if (count != box_size_j+2)
                        printf("Error while recieving from top\n");
                for (i = 0; i < box_size_j+2; i++)
                {
                        U[0][i] = buf[i];
                }
                free(buf);
        }

        if (coords[1] - 1 >= 0) // left
        {
                int r_left, coords_left[2];
                coords_left[0] = coords[0];
```

```
            coords_left[1] = coords[1] - 1;
            MPI_Cart_rank(topology, coords_left, &r_left);
            double* buf = (double*) malloc ((box_size_i+2) * sizeof(double));
            MPI_Recv(buf, box_size_i+2, MPI_DOUBLE, r_left, 0,
                    topology, &status);
            for (i = 0 ; i < box_size_i + 2; i++)
            {
                    U[i][0] = buf[i];
            }
            free(buf);
            int count;
            MPI_Get_count (&status, MPI_DOUBLE, &count);
            if (count != box_size_i+2)
                    printf("Error while recieving from left\n");
    }

    // Me
    double max_e = 0;
    double loc_eps = 0.;
    double om = omega(iter);
    double loc_eps0 = 0., loc_eps1 = 0., loc_eps2 = 0., loc_eps3 = 0.;
    for(i=1; i < box_size_i+1; i++)
for(j=1; j< box_size_j+1; j++)
{
    double U_dop = (U[i-1][j] + U[i+1][j] + U[i][j-1] + U[i][j+1])/4.;
                    double e;
                    e = U[i][j];
    U[i][j] = om * U_dop + (1. - om) * U[i][j];
                    if (fabs(e-U[i][j]) > loc_eps)
                            loc_eps = fabs(e-U[i][j]);
            }

    // Send new data to bottom and right processes
    if (coords[0] + 1 <= (N / box_size_i) - 1) // bottom
    {
            int r_b, coords_b[2];
            coords_b[0] = coords[0] + 1;
            coords_b[1] = coords[1];
            MPI_Cart_rank(topology, coords_b, &r_b);
            double* buf = (double*) malloc ((box_size_j+2) * sizeof(double));
            for (i = 0 ; i < box_size_j + 2; i++)
            {
                    buf[i] = U[box_size_i][i];
            }
            MPI_Send(buf, box_size_j+2, MPI_DOUBLE, r_b, 0,
                    topology);
            free(buf);
    }

    if (coords[1] + 1 <= (N / box_size_j) - 1) // right
    {
            int r_right, coords_r[2];
```

```
        coords_r[0] = coords[0];
        coords_r[1] = coords[1] + 1;
        MPI_Cart_rank(topology, coords_r, &r_right);
        double* buf = (double*) malloc ((box_size_i+2) * sizeof(double));
        for (i = 0 ; i < box_size_i + 2; i++)
                buf[i] = U[i][box_size_j];
        MPI_Send(buf, box_size_i+2, MPI_DOUBLE, r_right, 0,
                 topology);
        free(buf);
}

// Send data to top and left processes for future use
if (coords[0] - 1 >= 0) // top
{
        int r_top, coords_top[2];
        coords_top[0] = coords[0] - 1;
        coords_top[1] = coords[1];
        MPI_Cart_rank(topology, coords_top, &r_top);
        double* buf = (double*) malloc ((box_size_j+2) * sizeof(double));
        for(j = 0; j < box_size_j + 2; j++)
        {
                buf[j] = U[1][j];
        }
        MPI_Send(buf, box_size_j+2, MPI_DOUBLE, r_top, 0,
                 topology);
        free(buf);
}

if (coords[1] - 1 >= 0) // left
{
        int r_left, coords_left[2];
        coords_left[0] = coords[0];
        coords_left[1] = coords[1] - 1;
        MPI_Cart_rank(topology, coords_left, &r_left);
        double* buf = (double*) malloc ((box_size_i+2) * sizeof(double));
        for (i = 0 ; i < box_size_i + 2; i++)
        {
                buf[i] = U[i][1];
        }
        MPI_Send(buf, box_size_i+2, MPI_DOUBLE, r_left, 0,
                 topology);
        free(buf);
}

// Recieve data from right and bottom processes for future use
if (coords[0] + 1 <= N / box_size_i - 1) // bottom
{
        int r_b, coords_b[2];
        coords_b[0] = coords[0] + 1;
        coords_b[1] = coords[1];
        MPI_Cart_rank(topology, coords_b, &r_b);
        double* buf = (double*) malloc ((box_size_j+2) * sizeof(double));
```

```
                MPI_Recv(buf, box_size_j+2, MPI_DOUBLE, r_b, 0,
                        topology, &status);
                for (i = 0 ; i < box_size_j + 2; i++)
                {
                        U[box_size_i+1][i] = buf[i];
                }
                int count;
                MPI_Get_count (&status, MPI_DOUBLE, &count);
                if (count != box_size_j+2)
                        printf("Error while recieving from bottom\n");
                free(buf);
        }

        if (coords[1] + 1 <= N / box_size_j - 1) // right
        {
                int r_right, coords_r[2];
                coords_r[0] = coords[0];
                coords_r[1] = coords[1] + 1;
                MPI_Cart_rank(topology, coords_r, &r_right);
                double* buf = (double*) malloc ((box_size_i+2) * sizeof(double));
                MPI_Recv(buf, box_size_i+2, MPI_DOUBLE, r_right, 0,
                        topology, &status);
                for (i = 0 ; i < box_size_i + 2; i++)
                        U[i][box_size_j+1] = buf[i];
                int count;
                MPI_Get_count (&status, MPI_DOUBLE, &count);
                if (count != box_size_i+2)
                        printf("Error while recieving from right\n");
                free(buf);
        }
        MPI_Allreduce(&loc_eps, &eps, 1, MPI_DOUBLE, MPI_MAX, topology);
        /*
        printf("Proc %d, Iter %d, local eps %lf, eps %lf, max_e %lf\n", proc_rank, iter, loc_eps, eps,
max_e);
        if (proc_size == 1)
        {
                printf("Local eps 0 %lf\n", loc_eps0);
                printf("Local eps 1 %lf\n", loc_eps1);
                printf("Local eps 2 %lf\n", loc_eps2);
                printf("Local eps 3 %lf\n", loc_eps3);
        }
        */
}

void work()
{
        if ((int)sqrt(proc_size) == sqrt(proc_size))
        {
                box_size_j = sqrt(N*N / proc_size);
                box_size_i = box_size_j;
        }
        else
```

```
	{
		box_size_j = sqrt(N*N / (proc_size / 2));
		box_size_i = box_size_j / 2;

	}
	if (proc_rank == 0)
	{
		//printf("Box size is %d x %d\n", box_size_j, box_size_i);
		//printf( "Dimentions %d x %d\n", N/box_size_i, N/box_size_j );
		printf("N is %d\n", N);
		printf("Eps is %lf\n", maxeps);
		printf("Proc is %d\n", proc_size);
	}

	int dims[2], periods[2], reorder;
	dims[0] = N / box_size_i;
	dims[1] = N / box_size_j;
	periods[0] = 0;
	periods[1] = 0;
	reorder = 1;

	int err = MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &topology);
if (topology == MPI_COMM_NULL)
	{
		printf("Error while creating Cart\n");
		exit(1);
	}
	init();


	int it = 0;
	int final_it = 0;
while(1)
	{
   it++;
		eps = 0.;
   relax(it);
		//if (proc_rank == 0)
		//	printf( "ALL: it=%4i eps=%f\n", it, eps);
   if (eps < maxeps || it > itmax)
		{
			final_it = it;
			break;
		}
}
	if (proc_rank == 0)
	{
		printf("Resulting eps: %lf\n", eps);
		printf("Iterations: %d\n", final_it);
	}

	/*
```

```
        printf("Output %d\n", proc_rank);
        int coords[2], r;
    MPI_Comm_rank(topology, &r);
        MPI_Cart_coords (topology, r, 2, coords);
        printf("Coords %d x %d\n", coords[0], coords[1]);
        char out[19];
        sprintf(out, "grid_%d.txt", proc_rank);
        output(out);
        */
        //MPI_Gather(U, int sendcount, MPI_DOUBLE,
         //             void *recvbuf, (N+2)*(N+2), MPI_DOUBLE, 0, topology);
    //verify();
}


int main(int argc, char ** argv)
{
        if (argc > 1)
        {
                sscanf(argv[1], "%d", &N);
        }
        if (N <= 0)
        {
                printf("Error: grid size is too small\n");
                return 3;
        }
        if (argc > 2)
        {
                sscanf(argv[2], "%lf", &maxeps);
        }
        if (maxeps <= 0)
        {
                printf("Error: epsilon is too small\n");
                return 4;
        }

        int err = MPI_Init(&argc, &argv);
        if (err != MPI_SUCCESS)
        {
                fprintf(stdout, "Error: MPI_Init. Aborting...\n");
                return 1;
        }
    MPI_Comm_size(MPI_COMM_WORLD, &proc_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &proc_rank);

    double start, fin;
        if (proc_rank == 0)
                wtime(&start);

        work();

        if (proc_rank == 0)
```

```
    {
            wtime(&fin);
            printf("Time in seconds=%gs\n", fin - start);
        }

        err = MPI_Finalize();
        if (err)
        if (err != MPI_SUCCESS)
        {
            fprintf(stdout, "Error: MPI_Finalize. Aborting...\n");
            return 2;
        }
        int i;
        for (i = 0; i < box_size_i + 2; ++i)
        {
            free(U[i]);
        }
        free(U);
    return 0;
}
```
