Московский государственный университет имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра математической физики

Задание по курсу «Суперкомпьютерное моделирование и технологии»

Исследование масштабируемости разностной схемы для трехмерного гиперболического уравнения

ОТЧЁТ

Работу выполнил:
студент 601 группы
И.С. Лузин

Москва, 2017

Оглавление

Постановка задачи

В трехмерной замкнутой области

$$\Omega = [0 \le x \le L_x] \times [0 \le y \le L_y] \times [0 \le z \le L_z]$$

для $(0 < t \le T]$ требуется найти решение $u(x, y, z, t)$ уравнения в частных производных

$$\frac{\partial^2 u}{\partial t^2} = \Delta u \tag{1}$$

с начальными условиями

$$u|_{t=0} = \phi(x, y, z), \tag{2}$$

$$\frac{\partial u}{\partial t}_{t=0} = 0, \tag{3}$$

при условии, что на границах области заданы однородные граничные условия первого рода

$$u(0, y, z, t) = 0, \qquad u(L_x, y, z, t) = 0, \tag{4}$$

$$u(x, 0, z, t) = 0, \qquad u(x, L_y, z, t) = 0, \tag{5}$$

$$u(x, y, 0, t) = 0, \qquad u(x, y, L_z, t) = 0, \tag{6}$$

либо периодические граничные условия

$$u(0, y, z, t) = u(L_x, y, z, t), \qquad u_x(0, y, z, t) = u_x(L_x, y, z, t), \tag{7}$$

$$u(x, 0, z, t) = u(x, L_y, z, t), \qquad u_y(x, 0, z, t) = u_y(x, L_y, z, t), \tag{8}$$

$$u(x, y, 0, t) = u(x, y, L_z, t), \qquad u_z(x, y, 0, t) = u_z(x, y, L_z, t). \tag{9}$$

Конкретная комбинация граничных условий определяется индивидуальным вариантом задания (см. п. 5).

Для аппроксимации исходного уравнения (1) с однородными граничными условиями (4)–(6) и начальными условиями (2)–(3) воспользуемся следующей системой уравнений:

$$\frac{y^{n+1}_{ijk} - 2y^n_{ijk} + y^{n-1}_{ijk}}{\tau^2} = \Delta_h y^n, \quad (x_i, y_j, z_k) \in \omega_h, \quad n = 1, 2, \ldots, K - 1,$$

Здесь Δ_h — семиточечный разностный аналог оператора Лапласа:

$$\Delta_h y^n = \frac{y^n_{i-1,j,k} - 2y^n_{i,j,k} + y^n_{i+1,j,k}}{h^2} + \frac{y^n_{i,j-1,k} - 2y^n_{i,j,k} + y^n_{i,j+1,k}}{h^2} + \frac{y^n_{i,j,k-1} - 2y^n_{i,j,k} + y^n_{i,j,k+1}}{h^2}.$$

Для начала счета (т.е.для нахождения y^2_{ijk}) должны быть заданы значения y^0_{ijk}, y^1_{ijk}, $(x_i, y_j, z_k) \in \omega_h$. Из условия (2) имеем

$$y^0_{ijk} = \phi(x_i, y_j, z_k), \quad (x_i, y_j, z_k) \in \omega_h.$$

Простейшая замена начального условия (3) уравнением $(y^1_{ijk} - y^0_{ijk})/\tau = 0$ имеет лишь первый порядок аппроксимации по τ. Аппроксимацию второго порядка по τ и h дает разностное уравнение

$$\frac{y^1_{ijk} - y^0_{ijk}}{\tau} = \frac{\tau}{2}\Delta_h \phi(x_i, y_j, z_k), \quad (x_i, y_j, z_k) \in \omega_h.$$

$$y^1_{ijk} = y^0_{ijk} + \frac{\tau^2}{2}\Delta_h \phi(x_i, y_j, z_k)$$

Разностная аппроксимация для периодических граничных условий выглядит следующим образом

$$y^{n+1}_{0jk} = y^{n+1}_{Njk}, \qquad y^{n+1}_{1jk} = y^{n+1}_{N+1jk},$$
$$y^{n+1}_{i0k} = y^{n+1}_{iNk}, \qquad y^{n+1}_{i1k} = y^{n+1}_{iN+1k},$$
$$y^{n+1}_{ij0} = y^{n+1}_{ijN}, \qquad y^{n+1}_{ij1} = y^{n+1}_{ijN+1},$$

$i, j, k = 0, 1, \ldots, N.$

В данной работе будет рассмотрен вариант задания с комбинацией граничных условий (4), (5) и (9). В качестве начального условия будет использована функция распределения:

$$\phi(x, y, z) = \sin\frac{2\pi x}{L_x} \cdot \sin\frac{2\pi y}{L_y} \cdot \cos\frac{2\pi z}{L_z} \quad .$$

В таком случае точное решение задачи (1)—(5), (9) приобретает следующий вид:

$$u(x, y, z) = \sin\frac{2\pi x}{L_x} \cdot \sin\frac{2\pi y}{L_y} \cdot \cos\frac{2\pi z}{L_z} \cdot \cos\left(2\pi\sqrt{L_x^{-2} + L_y^{-2} + L_z^{-2}}\, t\right) \quad .$$

Программная реализация

Область, в которой требуется найти решение, представляет из себя прямоугольный параллелепипед. Первоначально эта область равномерно распределяется между MPI-процессами в виде блоков, которые так же имеют форму прямоугольного параллелепипеда. Каждому процессу соответствует его собственный единственный блок. В рамках одного процесса численное решение на соответствующем блоке инициализируется в соответствии с начальными условиями. На каждой итерации алгоритма для обеспечения возможности применения явной разностной схемы поддерживаются значения на последнем и предпоследнем шаге. После выполнения очередного шага значения на границе блока передаются процессам, соответствующим топологически соседним блокам. Таким образом осуществляется обмен данными между процессами. Отклонение численного решения от точного на каждом блоке определяется отдельно и передаётся процессу, осуществляющему обобщение и вывод значения ошибки. Ниже приведён исходный код программы:

```
#include <algorithm>
#include <cmath>
#include <iostream>
#include <mpi.h>
#include <numeric>
#include <omp.h>
#include <stdlib.h>

struct block_t {
  int column, row, plane;
  int width, height, depth;
  int left, right, bottom, top, back, front;
  template <typename _Tp>
  inline double mean_absolute_error(double *, _Tp, double, double,
double, double, double, double, double);
};

inline int idx(int, int, int, int, int);
inline int idx(int, int, int, int, int, int);
inline double phi(double, double, double, double, double, double);
inline double sol(double, double, double, double, double, double,
double);

int main(int argc, char *argv[]) {
  double Lx = atof(argv[1]), Ly = atof(argv[2]), Lz = atof(argv[3]), T
= atof(argv[4]);
  int N = atoi(argv[5]), K = atoi(argv[6]);
  double hx = Lx / (N - 1), hy = Ly / (N - 1), hz = Lz / (N - 1), tau =
T / K;
  MPI_Init(&argc, &argv);
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Barrier(MPI_COMM_WORLD);
  double time = MPI_Wtime();
  int columns = 1, rows = 1, planes = 1;
  if (!rank) {
```

```
    for (int dim = 0, Np = size; Np > 1; ++dim, Np >>= 1) {
      if (dim > 2) {
        dim = 0;
      }
      if (dim < 1) {
        columns <<= 1;
      } else if (dim < 2) {
        rows <<= 1;
      } else {
        planes <<= 1;
      }
    }
  }
  MPI_Bcast(&columns, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&rows, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&planes, 1, MPI_INT, 0, MPI_COMM_WORLD);
  block_t block = {
    rank % columns,
    rank % (columns * rows) / columns,
    rank / (columns * rows),
    N / columns,
    N / rows,
    N / planes,
    idx(block.column - 1, block.row, block.plane, columns, rows,
planes),
    idx(block.column + 1, block.row, block.plane, columns, rows,
planes),
    idx(block.column, block.row - 1, block.plane, columns, rows,
planes),
    idx(block.column, block.row + 1, block.plane, columns, rows,
planes),
    idx(block.column, block.row, block.plane - 1, columns, rows,
planes),
    idx(block.column, block.row, block.plane + 1, columns, rows,
planes)
  };
  if (block.plane == 0) {
    block.back = idx(block.column, block.row, planes - 1, columns,
rows, planes);
  }
  if (block.plane == planes - 1) {
    block.front = idx(block.column, block.row, 0, columns, rows,
planes);
  }
  double *next = new double[(block.width + 2) * (block.height + 2) *
(block.depth + 2)];
  double *self = new double[(block.width + 2) * (block.height + 2) *
(block.depth + 2)];
  double *prev = new double[(block.width + 2) * (block.height + 2) *
(block.depth + 2)];
  double *scores = new double[size];
  int input_left_size = block.height * block.depth;
  double *input_left = new double[input_left_size];
  int input_right_size =block.height * block.depth;
  double *input_right = new double[input_right_size];
  int output_left_size = block.height * block.depth;
```

```
  double *output_left = new double[output_left_size];
  int output_right_size = block.height * block.depth;
  double *output_right = new double[output_right_size];
  int input_bottom_size = block.depth * block.width;
  double *input_bottom = new double[input_bottom_size];
  int input_top_size = block.width * block.depth;
  double *input_top = new double[input_top_size];
  int output_bottom_size = block.depth * block.width;
  double *output_bottom = new double[output_bottom_size];
  int output_top_size = block.width * block.depth;
  double *output_top = new double[output_top_size];
  int input_back_size = block.width * block.height;
  double *input_back = new double[input_back_size];
  int input_front_size = block.width * block.height;
  double *input_front = new double[input_front_size];
  int output_back_size =  block.height * block.width;
  double *output_back = new double[output_back_size];
  int output_front_size = block.width * block.height;
  double *output_front = new double[output_front_size];
  for (int n = 0; n <= std::min(K, 20); ++n) {
   if (n == 1) {
//#pragma omp parallel for
     for (int k = 1; k <= block.depth; ++k) {
       for (int j = 1; j <= block.height; ++j) {
         for (int i = 1; i <= block.width; ++i) {
           double Uxx = (self[idx(i - 1, j, k, block.width + 2,
block.height + 2)] - 2 * self[idx(i, j, k, block.width + 2,
block.height + 2)] + self[idx(i + 1, j, k, block.width + 2,
block.height + 2)]) / (hx * hx);
           double Uyy = (self[idx(i, j - 1, k, block.width + 2,
block.height + 2)] - 2 * self[idx(i, j, k, block.width + 2,
block.height + 2)] + self[idx(i, j + 1, k, block.width + 2,
block.height + 2)]) / (hy * hy);
           double Uzz = (self[idx(i, j, k - 1, block.width + 2,
block.height + 2)] - 2 * self[idx(i, j, k, block.width + 2,
block.height + 2)] + self[idx(i, j, k + 1, block.width + 2,
block.height + 2)]) / (hz * hz);
           next[idx(i, j, k, block.width + 2, block.height + 2)] =
self[idx(i, j, k, block.width + 2, block.height + 2)] + 0.5 * pow(tau,
2) * (Uxx + Uyy + Uzz);
         }
       }
     }
   } else if (n) {
//#pragma omp parallel for
     for (int k = 1; k <= block.depth; ++k) {
       for (int j = 1; j <= block.height; ++j) {
         for (int i = 1; i <= block.width; ++i) {
           double Uxx = (self[idx(i - 1, j, k, block.width + 2,
block.height + 2)] - 2 * self[idx(i, j, k, block.width + 2,
block.height + 2)] + self[idx(i + 1, j, k, block.width + 2,
block.height + 2)]) / (hx * hx);
           double Uyy = (self[idx(i, j - 1, k, block.width + 2,
block.height + 2)] - 2 * self[idx(i, j, k, block.width + 2,
block.height + 2)] + self[idx(i, j + 1, k, block.width + 2,
block.height + 2)]) / (hy * hy);
```

```
          double Uzz = (self[idx(i, j, k - 1, block.width + 2,
block.height + 2)] - 2 * self[idx(i, j, k, block.width + 2,
block.height + 2)] + self[idx(i, j, k + 1, block.width + 2,
block.height + 2)]) / (hz * hz);
          next[idx(i, j, k, block.width + 2, block.height + 2)] = 2 *
self[idx(i, j, k, block.width + 2, block.height + 2)] - prev[idx(i, j,
k, block.width + 2, block.height + 2)] + pow(tau, 2) * (Uxx + Uyy +
Uzz);
        }
      }
    }
  } else {
//#pragma omp parallel for
    for (int k = 1; k <= block.depth; ++k) {
      for (int j = 1; j <= block.height; ++j) {
        for (int i = 1; i <= block.width; ++i) {
          next[idx(i, j, k, block.width + 2, block.height + 2)] =
phi((block.column * block.width + i - 1) * hx, (block.row *
block.height + j - 1) * hy, (block.plane * block.depth + k - 1) * hz,
Lx, Ly, Lz);
        }
      }
    }
  }
  if (block.column == 0) {
//#pragma omp parallel for
    for (int k = 1; k <= block.depth; ++k) {
      for (int j = 1; j <= block.height; ++j) {
        next[idx(1, j, k, block.width + 2, block.height + 2)] = 0;
      }
    }
  }
  if (block.column == columns - 1) {
//#pragma omp parallel for
    for (int k = 1; k <= block.depth; ++k) {
      for (int j = 1; j <= block.height; ++j) {
        next[idx(block.width, j, k, block.width + 2, block.height + 2)]
= 0;
      }
    }
  }
  if (block.row == 0) {
//#pragma omp parallel for
    for (int k = 1; k <= block.depth; ++k) {
      for (int i = 1; i <= block.width; ++i) {
        next[idx(i, 1, k, block.width + 2, block.height + 2)] = 0;
      }
    }
  }
  if (block.row == rows - 1) {
//#pragma omp parallel for
    for (int k = 1; k <= block.depth; ++k) {
      for (int i = 1; i <= block.width; ++i) {
        next[idx(i, block.height, k, block.width + 2, block.height +
2)] = 0;
      }
```

```
      }
    }
//#pragma omp parallel for
    for (int k = 1; k <= block.depth; ++k) {
      for (int j = 1; j <= block.height; ++j) {
        output_left[idx(0, j - 1, k - 1, 1, block.height)] = next[idx(1,
j, k, block.width + 2, block.height + 2)];
        output_right[idx(0, j - 1, k - 1, 1, block.height)] =
next[idx(block.width, j, k, block.width + 2, block.height + 2)];
      }
    }
//#pragma omp parallel for
    for (int k = 1; k <= block.depth; ++k) {
      for (int i = 1; i <= block.width; ++i) {
        output_bottom[idx(i - 1, 0, k - 1, block.width, 1)] =
next[idx(i, 1, k, block.width + 2, block.height + 2)];
        output_top[idx(i - 1, 0, k - 1, block.width, 1)] = next[idx(i,
block.height, k, block.width + 2, block.height + 2)];
      }
    }
//#pragma omp parallel for
    for (int j = 1; j <= block.height; ++j) {
      for (int i = 1; i <= block.width; ++i) {
        if (block.plane) {
          output_back[idx(i - 1, j - 1, 0, block.width, 1)] = next[idx(i,
j, 1, block.width + 2, block.height + 2)];
        } else {
          output_back[idx(i - 1, j - 1, 0, block.width, 1)] = next[idx(i,
j, 2, block.width + 2, block.height + 2)];
        }
        if (block.plane != planes - 1) {
          output_front[idx(i - 1, j - 1, 0, block.width, 1)] =
next[idx(i, j, block.depth, block.width + 2, block.height + 2)];
        } else {
          output_front[idx(i - 1, j - 1, 0, block.width, 1)] =
next[idx(i, j, block.depth - 1, block.width + 2, block.height + 2)];
        }
      }
    }
    MPI_Request input_left_request, input_right_request,
input_bottom_request, input_top_request, input_back_request,
input_front_request;
    MPI_Request output_left_request, output_right_request,
output_bottom_request, output_top_request, output_back_request,
output_front_request;
    if (0 <= block.left && block.left < size) {
      MPI_Irecv(input_left, input_left_size, MPI_DOUBLE, block.left, 1,
MPI_COMM_WORLD, &input_left_request);
      MPI_Isend(output_left, output_left_size, MPI_DOUBLE, block.left,
1, MPI_COMM_WORLD, &output_left_request);
    }
    if (0 <= block.right && block.right < size) {
      MPI_Irecv(input_right, input_right_size, MPI_DOUBLE, block.right,
1, MPI_COMM_WORLD, &input_right_request);
      MPI_Isend(output_right, output_right_size, MPI_DOUBLE,
block.right, 1, MPI_COMM_WORLD, &output_right_request);
```

```
    }
    if (0 <= block.bottom && block.bottom < size) {
      MPI_Irecv(input_bottom, input_bottom_size, MPI_DOUBLE,
block.bottom, 1, MPI_COMM_WORLD, &input_bottom_request);
      MPI_Isend(output_bottom, output_bottom_size, MPI_DOUBLE,
block.bottom, 1, MPI_COMM_WORLD, &output_bottom_request);
    }
    if (0 <= block.top && block.top < size) {
      MPI_Irecv(input_top, input_top_size, MPI_DOUBLE, block.top, 1,
MPI_COMM_WORLD, &input_top_request);
      MPI_Isend(output_top, output_top_size, MPI_DOUBLE, block.top, 1,
MPI_COMM_WORLD, &output_top_request);
    }
    if (0 <= block.back && block.back < size) {
      MPI_Irecv(input_back, input_back_size, MPI_DOUBLE, block.back, 1,
MPI_COMM_WORLD, &input_back_request);
      MPI_Isend(output_back, output_back_size, MPI_DOUBLE, block.back,
1, MPI_COMM_WORLD, &output_back_request);
    }
    if (0 <= block.front && block.front < size) {
      MPI_Irecv(input_front, input_front_size, MPI_DOUBLE, block.front,
1, MPI_COMM_WORLD, &input_front_request);
      MPI_Isend(output_front, output_front_size, MPI_DOUBLE,
block.front, 1, MPI_COMM_WORLD, &output_front_request);
    }
//#pragma omp barrier
    MPI_Status status;
    if (0 <= block.left && block.left < size) {
      MPI_Wait(&input_left_request, &status);
      MPI_Wait(&output_left_request, &status);
    }
    if (0 <= block.right && block.right < size) {
      MPI_Wait(&input_right_request, &status);
      MPI_Wait(&output_right_request, &status);
    }
    if (0 <= block.bottom && block.bottom < size) {
      MPI_Wait(&input_bottom_request, &status);
      MPI_Wait(&output_bottom_request, &status);
    }
    if (0 <= block.top && block.top < size) {
      MPI_Wait(&input_top_request, &status);
      MPI_Wait(&output_top_request, &status);
    }
    if (0 <= block.back && block.back < size) {
      MPI_Wait(&input_back_request, &status);
      MPI_Wait(&output_back_request, &status);
    }
    if (0 <= block.front && block.front < size) {
      MPI_Wait(&input_front_request, &status);
      MPI_Wait(&output_front_request, &status);
    }
//#pragma omp parallel for
    for (int k = 1; k <= block.depth; ++k) {
      for (int j = 1; j <= block.height; ++j) {
        next[idx(0, j, k, block.width + 2, block.height + 2)] =
input_left[idx(0, j - 1, k - 1, 1, block.height)];
```

```
      next[idx(block.width + 1, j, k, block.width + 2, block.height +
2)] = input_right[idx(0, j - 1, k - 1, 1, block.height)];
     }
    }
//#pragma omp parallel for
    for (int k = 1; k <= block.depth; ++k) {
     for (int i = 1; i <= block.width; ++i) {
      next[idx(i, 0, k, block.width + 2, block.height + 2)] =
input_bottom[idx(i - 1, 0, k - 1, block.width, 1)];
      next[idx(i, block.height + 1, k, block.width + 2, block.height +
2)] = input_top[idx(i - 1, 0, k - 1, block.width, 1)];
     }
    }
//#pragma omp parallel for
    for (int j = 1; j <= block.height; ++j) {
     for (int i = 1; i <= block.width; ++i) {
      next[idx(i, j, 0, block.width + 2, block.height + 2)] =
input_back[idx(i - 1, j - 1, 0, block.width, 1)];
      next[idx(i, j, block.depth + 1, block.width + 2, block.height +
2)] = input_front[idx(i - 1, j - 1, 0, block.width, 1)];
     }
    }
//#pragma omp parallel for
    for (int i = 0; i < (block.width + 2) * (block.height + 2) *
(block.depth + 2); ++i) {
     prev[i] = self[i];
     self[i] = next[i];
    }
    double score = block.mean_absolute_error(self, sol, hx, hy, hz, Lx,
Ly, Lz, tau * n);
    MPI_Gather(&score, 1, MPI_DOUBLE, scores, 1, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
    if (!rank) {
     score = std::accumulate(scores, scores + size, 0.) / size;
     std::cout << "-*- time: " << tau * n << ", score: " << score << "
-*-" << std::endl;
    }
//#pragma omp barrier
    MPI_Barrier(MPI_COMM_WORLD);
  }
  MPI_Barrier(MPI_COMM_WORLD);
  delete [] output_front;
  delete [] output_back;
  delete [] input_front;
  delete [] input_back;
  delete [] output_top;
  delete [] output_bottom;
  delete [] input_top;
  delete [] input_bottom;
  delete [] output_right;
  delete [] output_left;
  delete [] input_right;
  delete [] input_left;
  delete [] scores;
  delete [] prev;
  delete [] self;
```

```
  delete [] next;
  if (!rank) {
    std::cout << "-*- Wtime: " << MPI_Wtime() - time << ", Np: " <<
size << ", N: " << N << " -*-" << std::endl;
  }
  MPI_Finalize();
  return EXIT_SUCCESS;
}

template <typename _Tp>
double block_t::mean_absolute_error(double *self, _Tp func, double hx,
double hy, double hz, double Lx, double Ly, double Lz, double t)
{
  double score = 0;
  for (int k = 1; k <= depth; ++k) {
    for (int j = 1; j <= height; ++j) {
      for (int i = 1; i <= width; ++i) {
        score += self[idx(i, j, k, width + 2, height + 2)] -
func((column * width + i - 1) * hx, (row * height + j - 1) * hy,
(plane * depth + k - 1) * hz, Lx, Ly, Lz, t);
      }
    }
  }
  score /= width * height * depth;
  return score;
}

int idx(int i, int j, int k, int columns, int rows) {
  return i + (j + k * rows) * columns;
}

int idx(int i, int j, int k, int columns, int rows, int planes) {
  if (0 > i || i >= columns || 0 > j || j >= rows || 0 > k || k >=
planes) {
    return -1;
  }
  return idx(i, j, k, columns, rows);
}

double phi(double x, double y, double z, double Lx, double Ly, double
Lz) {
  return sin(2 * M_PI * x / Lx) * sin(2 * M_PI * y / Ly) * cos(2 * M_PI
* z / Lz);
}

double sol(double x, double y, double z, double Lx, double Ly, double
Lz, double t) {
  return phi(x, y, z, Lx, Ly, Lz) * cos(2 * M_PI * sqrt(pow(Lx, -2) +
pow(Ly, -2) + pow(Lz, -2)) * t);
}
```

Результаты расчётов

В таблицах 1 и 2 приведены результаты расчётов. Ускорение S вычислялось относительно времени решения T на ближайшем меньшем числе процессов N_p.

Таблица 1: Исследование параллельных характеристик на ПВС «Ломоносов»

Число процессов N_p	*Число точек* N^3	*Время решения* T	*Ускорение* S
8	128^3	1,43	1
16	128^3	0,73	1,98
32	128^3	0,36	2
64	128^3	0,22	1,68
128	128^3	0,16	1,38
8	256^3	11,09	1
16	256^3	5,63	1,97
32	256^3	2,86	1,97
64	256^3	1,45	1,98
128	256^3	0,78	1,85
8	512^3	87,92	1
16	512^3	44,42	1,98
32	512^3	22,41	1,98
64	512^3	11,62	1,93
128	512^3	5,77	2,01

Таблица 2: Исследование параллельных характеристик на ПВС Blue Gene/P

Число процессов N_p	*Число точек* N^3	*Время решения* T	*Ускорение* S
128	512^3	32,84	1
256	512^3	16,68	1,97
512	512^3	8,37	1,99
128	1024^3	255,59	1
256	1024^3	129,31	1,98
512	1024^3	64,96	1,99
128	1536^3	857,51	1
256	1536^3	431,43	1,99
512	1536^3	193,54	2,23

Приведённые выше результаты расчётов показывают достаточную хорошую масштабируемость алгоритма. И правда, реализация блочного разбиения между процессами обладает большим ресурсом параллелизма, поскольку в

этом случае предполагается меньше межпроцессорных коммуникаций, по сравнению с ленточным. Ниже приведены результаты расчётов с использованием открытого стандарта OpenMP для распараллеливания программы в рамках каждого из MPI-процессов.

Таблица 3: Исследование параллельных характеристик гибридной программы

Число процессов N_p	*Число точек* N^3	*Время решения* T'	*Ускорение* S'	$\frac{T}{T'}$
128	512^3	29,39	1	1,12
256	512^3	14,94	1,97	1,12
512	512^3	7,53	1,98	1,11
128	1024^3	229,58	1	1,11
256	1024^3	115,62	1,99	1,12
512	1024^3	58,02	1,99	1,12
128	1536^3	770,84	1	1,11
256	1536^3	386,79	1,99	1,12
512	1536^3	173,58	2,23	1,11

Список литературы

1. IBM Blue Gene/P. — http://hpc.cmc.msu.ru
2. Суперкомпьютер «Ломоносов». — http://hpc.cmc.msu.ru
3. IBM eServer pSeries 690 Regatta. — http://www.regatta.cmc.msu.ru
4. Самарский А.А., Гулин А.В. Численные методы. — М.: Наука. Гл. ред. физ-мат. лит., 1989.