

Параллельное и распределённое программирование

Отчет

Лабораторная работа № 2

Автор:
Поташев Н. А.

Руководитель:
Рудалев А. В.

28 мая 2018 г.



1 Постановка задачи

Целью данной лабораторной работы было сравнение параллельной и последовательной реализации алгоритма решения краевой задачи Дирихле для уравнения Пуассона.

$$\begin{cases} \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y), & (x,y) \in D \\ u(x,y) = g(x,y), & (x,y) \in D^0 \end{cases}$$

В качестве области задания D функции $u(x,y)$ был использован единичный квадрат

$$D = \{(x,y) \in D : 0 \leq x \leq 1\}.$$

Во время выполнения работы был рассмотрен следующий пример задачи Дирихле: найти стационарное распределение температуры в квадратной пластине со стороной 1, описываемое уравнением Лапласа(частный случай, когда $f(x,y)=0$) с краевыми условиями вида:

$$\begin{cases} f(x,y) = 0, & (x,y) \in D, \\ 0, & \text{if } x = 0, \\ \sin(y), & \text{if } x = 1, \\ 0, & \text{if } y = 0, \\ \sin(x), & \text{if } y = 1. \end{cases}$$

Список вариантов для решения задачи Дирихле(вариант 15).¹

2 Выполнение работы

В ходе выполнения данной работы были получены:

- программа, реализующая параллельное и последовательное решения задачи Дирихле;
- python-скрипт, реализующий построение графиков на основе полученных данных;
- bash-скрипт для запуска программы на ПК;

¹<http://old.exponenta.ru/educat/systemat/hanova/lab/LR7/LR7.asp>.

- sbatch-скрипт для запуска программы на кластере;
- результаты расчетов программы на кластере и ПК(в формате csv-файлов).

2.1 Алгоритмы

Был реализован сначала последовательный алгоритм решения задачи Дирихле методом сеток, а после были приняты попытки реализовать параллельную версию алгоритма см. Листинг 1-4.

```

1 ...
2 double max;
3 do {
4     max = 0;
5     for (size_t i = 1; i < size + 1; ++i)
6         for (size_t j = 1; j < size + 1; ++j) {
7             double u0 = u(i, j);
8             double temp = 0.25 * (u(i-1, j) + u(i+1, j) + u(i, j-1) + u(i, j+1) -
9                 h*h*F(i - 1, j - 1));
10            u(i, j) = temp;
11            double d = std::fabs(temp - u0);
12
13            if (d > max) {
14                max = d;
15            }
16        }
17    } while (max > eps);
18 ...

```

Листинг 1: реализация последовательного алгоритма

Для представления u был выбран ранее реализованный класс Matrix.

```

1 ...
2 double max;
3 do {
4     max = 0;
5     #pragma omp parallel for shared(u) //worst
6     for (size_t i = 1; i < size + 1; ++i)
7         for (size_t j = 1; j < size + 1; ++j) {
8             double u0 = u(i, j);
9             double temp = 0.25 * (u(i-1, j) + u(i+1, j) + u(i, j-1) + u(i, j+1) -
10                 h*h*F(i - 1, j - 1));
11            u(i, j) = temp;
12            double d = std::fabs(temp - u0);
13
14            if (d > max) {
15                max = d;
16            }
17        }
18    } while (max > eps);
19 ...

```

Листинг 2: неудачная реализация параллельного алгоритма(пример data race)

```

1 ...
2 omp_lock_t lock;
3 omp_init_lock(&lock);
4
5 do {
6
7     max = 0;
8     #pragma omp parallel for shared(u,size,max) private(i,j,temp,d,dm)
9     for (i = 1; i < size + 1; ++i)
10    {
11        dm = 0;
12        for (j = 1; j < size + 1; ++j) {
13
14            temp = u(i, j);
15            un(i, j) = 0.25 * (u(i-1, j) + u(i+1, j) + u(i, j-1) + u(i, j+1) - h*h*
16                           F(i-1, j-1));
17            d = std::fabs(temp - un(i, j));
18
19            if (dm < d) {
20                dm = d;
21            }
22
23            omp_set_lock(&lock);
24            if (max < dm) {
25                max = dm;
26            }
27            omp_unset_lock(&lock);
28        }
29        for (i = 1; i < size + 1; ++i){
30
31            for (j = 1; j < size + 1; ++j) {
32
33                u(i, j)=un(i, j);
34
35            }
36        }
37
38    } while (max > eps);
39    ...

```

Листинг 3: реализация медленного параллельного алгоритма

Данный алгоритм обладал плохой сходимостью и был очень медленным. Поэтому было решено игнорировать эту реализацию.

```

1 ...
2 std::vector<double> mx(size+1);
3 do
4 {
5
6     for (size_t k = 1; k < size+1; k++) {
7         mx[k] = 0;
8         #pragma omp parallel for shared(u_mat,k,mx) private(i,j,u0,d)
9         for (i = 1; i < k+1; i++) {
10            j = k + 1 - i;

```

```

11     u0 = u_mat(i, j);
12     u_mat(i, j) = 0.25 * (u_mat(i-1, j) + u_mat(i+1, j) + u_mat(i, j-1) +
13         u_mat(i, j+1) - h*h*f_mat(i-1, j-1));
14     d = std::fabs(u_mat(i, j) - u0);
15     if (d > mx[i]) mx[i] = d;
16 }
17 for (size_t k = size-1; k > 0; k--) {
18 #pragma omp parallel for shared(u_mat,k,mx) private(i,j,u0,d)
19     for (i = size-k+1; i < size+1; i++) {
20         j = 2*size - k - i + 1;
21         u0 = u_mat(i, j);
22         u_mat(i, j) = 0.25 * (u_mat(i-1, j) + u_mat(i+1, j) + u_mat(i, j-1) +
23             u_mat(i, j+1) - h*h*f_mat(i-1, j-1));
24         d = std::fabs(u_mat(i, j) - u0);
25         if (d > mx[i]) mx[i] = d;
26     }
27     max = 0;
28     for (size_t i = 1; i < size+1; i++) {
29         if (mx[i] > max) max = mx[i];
30     }
31 } while (max > eps);
32 ...

```

Листинг 4: реализация параллельного алгоритма

Именно этот алгоритм был выбран для решения задачи, так как он обладал такой же сходимостью, что и последовательный вариант, при этом был более эффективным за счет параллельных участков кода.

2.2 Результаты расчетов

Результатом выполнения программы был текстовой csv-файл с полученной поверхностью и временем работы программы. Были выполнены расчеты на ПК и кластере, также был нарисован график поверхности см Рис. 1, графики эффективности, времени работы, ускорения программы при параллельной реализации см Рис. 2-3.

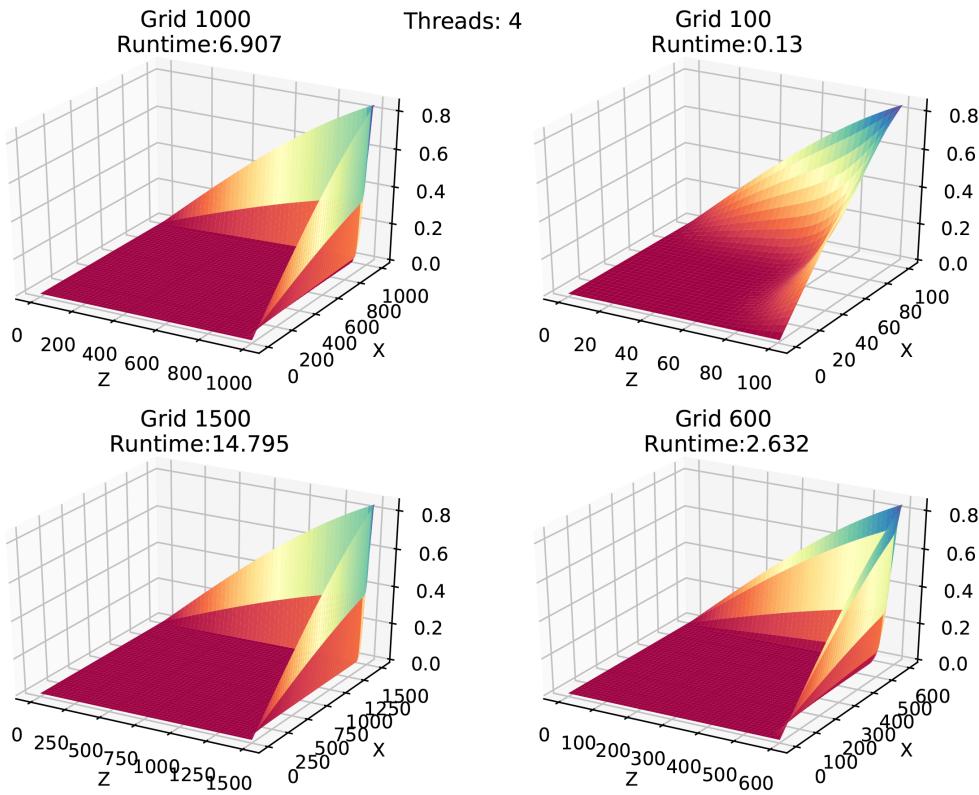


Рис. 1: результат работы программы - поверхность

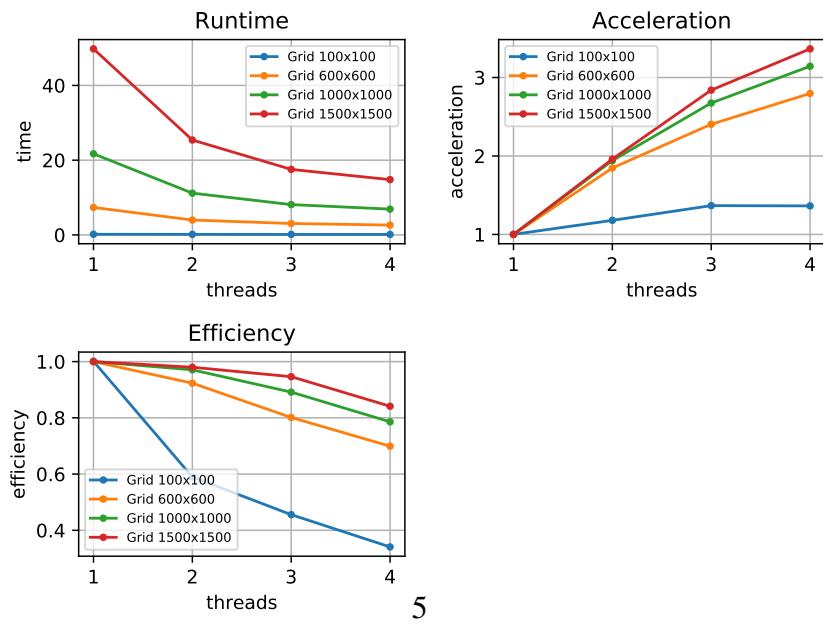


Рис. 2: графики эффективности работы на ПК

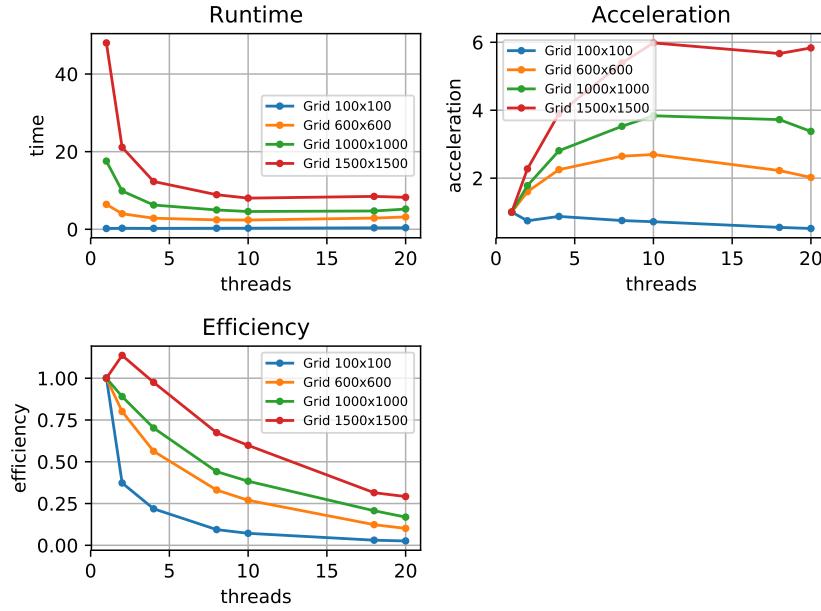


Рис. 3: графики эффективности работы на кластере

3 Вывод

По рис. 2-3 можно сделать вывод, что параллельный алгоритм решения задачи Дирихле более эффективный, чем последовательный. Реализованный волновой алгоритм не является самым оптимальным, так как не полностью использует поистине огромный потенциал кэша процессора. Проблемным местом алгоритма является то, что мы итеративно просчитываем диагонали матрицы, и расчет первого элемента

$$u_{1,1}$$

нельзя распараллелить, так как он один. В будущем можно модифицировать алгоритм следующими способами:

- блочное разделение диагоналей;
- использование очереди;
- расчет диагоналей матрицы от концов к середине(или наоборот).