МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИМЕНИ М.В. ЛОМОНОСОВА

КУРС: «ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ И СУПЕРКОМПЬЮТЕРЫ»

Рекурсивная координатная бисекция

Выполнил:

Сактаганов Нуржан (аспирант)

Дата подачи: 05.01.2018

Москва 2018

Table of Contents

Постановка задачи

Дана двумерная сетка, топологически эквивалентная индексному прямоугольнику n1xn2. Необходимо разбить сетку на k частей (доменов). Узел сетки с индексами (i, j) связан с существующими соседними по i и j узлами: (i-1, j), (i+1, j), (i, j-1), (i, j+1).

Сетка представляется в виде одмерного массива структур Point.

```
Point{
    float coord[2];
    int index;
}
 P[n1*n2];
```

Точки данной сетки имеют координаты P[i*n2+j].coord[0] = x(i, j), P[i*n2+j].coord[1] = y(i,j), где i = 0, …, n1-1, j = 0, …, n2-1. Индекс определяется соотношением P[i*n2+j].index = i*n2+j.

Для разбиения сетки нужно написать параллельную программу для вычислительной системы с распределенной памятью, обеспечивающую разбиение методом рекурсивной координатной бисекции узлов сетки на k доменов. Число вершин в доменах должно быть равно с точностью до одной вершины величине (n1*n2/k).

На выходе должны получить:

1. файл, содержащий n1*n2 строк, в каждой из которых 5 чисел: i j Xij Yij d // номера вершины в сетке, координаты вершины, номер домена [0,...,k-1]
2. число разрезанных рёбер
3. время выполнения декомпозиции (только декомпозиции, без учета формирования сетки и вывода результатов).

Метод решения

Сначала опишем последовательный алгоритм. Далее на основе последовательного алгоритма будем описывать параллельный алгоритм.

Последовательный алгоритм очень простой. На вход даются массив структур в описанном выше формате и количество доменов на которое нужно разбить. Мы сначала сортируем точки по заданной координате. После этого раскладываем k = k1 + k2, где k1 = (k + 1) / 2, k2 = k-k1. Далее делим массив в пропорции k1:k2. Меняем координату, по которой нужно сортировать массив, и рекурсивно вызываем на первой половине с параметром k := k1, и на второй половине k := k2. После чего обратно воостанавливаем координату для сортировки. Рекурсия останавливается в моменте, когда у нас k = 1. Это означает, что все точки массива принадлежат одному и тому же домену.

Параллельный алгоритм делает по сути точно то же самое что и последовательный. Будем считать, что каждый процесс запущен на отдельном узле, и что на узле находится один

процессор.

Допустим, что распределенный массив расположен на n процессорах. На каждом процессоре находится по part_size точек. И пусть имеется число k — количество доменов, на которое нужно разбить. Тогда, сначала сортируем глобальный массив по заданной координате (алгоритм сортировки Бэтчера). Далее, точно так же раскладываем k=k1+k2. Делим глобальный массив в пропорции k1:k2. Находим разбивающий элемент — первый элемент второй половинки глобального массива. Множество процессоров надо разбить на два подмножества, первое подмножество будет обрабатывать первую половинку, второе — вторую. Узел, который содержит в себе разбивающий элемент отнесем ко второму подмножеству. Поэтому, все элементы до разбивающего элемента нужно равномерно распределить по процессорам первого подножества. Причем, некоторые процессоры первого подмножества нужно дополнить фиктивными элементами, чтобы у всех было одинаковое количество элементов (это требуется для сортировки бэтчера). А в процессорах второго подмножества нужно сделать перераспределение элементов в связи с уменьшением их количества в этом множестве. После этого меняем координату для соортировки и применяем описанный алгоритм рекурсивно для двух подмножеств процессоров с соответствующими параметрами k1 и k2. После возврата из рекурсии надо не забыть воостановить координату для сортировки. Когда k или количество процессоров будет равно 1 — это означает конец рекурсии параллельного алгоритма, теперь уже на каждом узле выполним последовательный алгоритм с имеющим значением k.

Случай, когда разбивающий элемент будет на нулевом процессоре обрабатывается чуть по-другому. В этом случае нулевой процессор отнесем к первому подмножеству, а остальные — ко второму. Все элементы, начиная с разбивающего, распределим по процессорам из второго подмножества, пометив отправленные как фиктивные и добавив фиктивные по необходимости.

Подсчет числа разрезанных рёбер. Возьмем произвольный домен. Для каждой его точки проверим принадлежность каждой из четырех соседних точек данному домену. Если какой-то сосед не будет принадлежать данному домену — это означает что есть разрез ребра. Если для каждого домена посчитаем количество разрезов, и потом просуммируем — получим удвоенное количество всех разрезанных рёбер.

Будем считать, что k больше количества процессоров. Тогда не будет случая, когда один домен будет размещен на нескольких процессорах. Это означает, что каждый домен будет целиком помещаться внутри одного узла. Тогда мы можем для каждого домена локально посчитать количество разрезов. После чего легко получить количество всех разрезанных ребер.

Описание вычислительной системы

Вычислительная система представляет собой комплекс IBM Blue Gene/P. В составе имеется две стойки по 1024 четырехъядерных процессоров в каждой стойке. Пиковая производтельность — 27.9 терафлопс. Размер оперативной памяти — 2 Гб на каждый процессор. Каждое ядро модели PowerPC 450 с рабочей частотой 850 МГц, с 32 битной адресацией. Производительность ядра — 3.4 Гфлопс.

Описание коммуникационной сети:

- сеть общего назначения, объединяющие все вычислительные узлы; предназначена для операций типа «точка-точка»
- вычислительный узел имеет двунаправленные связи с шестью соседями
- пропускная способность каждого соединения — 425 MB/s (5,1 GB/s для всех 12 каналов)
- латентность (ближайший сосед):
  - 32-байтный пакет: 0,1 μs
  - 256-байтный пакет: 0,8 μs

Анализ полученных результатов

Ниже приведена таблица времен (в секундах) запусков на различном наборе сеток и процессоров (число доменов k = 256).

Сетка	1 процессор	8 процессоров	32 процессора	128 процессоров
4k x 2.5k	79.0919	11.055	3.56367	1.3142
4k x 5k	167.652	23.4848	7.44001	2.71177

Посчитаем ускорение.

Сетка	1 процессор	8 процессоров	32 процессора	128 процессоров
4k x 2.5k	1.0	7.1544	22.194	60.183
4k x 5k	1.0	7.1387	22.5338	61.8238

Посчитаем эффективность.

Сетка	1 процессор	8 процессоров	32 процессора	128 процессоров
4k x 2.5k	100%	89.43%	69.36%	48.02%
4k x 5k	100%	89.23%	70.42%	48.30%

Из таблиц видим, что программа хорошо параллелится и время выполнения продолжает сокращаться с ростом числа процессоров.

Для разнообразия сетка генерировалась псевдослучайно для каждого запуска (в точке (i, j) значение x и y — равномерное отклонение от i и j на 0.25), число разрезов тоже колеблется в некотором промежутке. Следующая таблица хорошо это демонстрирует.

Сетка	1 процессор	8 процессоров	32 процессора	128 процессоров
4k x 2.5k	118058	118124	117946	118835
4k x 5k	150875	151328	150984	154872

В качестве примера работы алгоритма на сетке 1000x1000 приведены две иллюстрации. В первом случае разбиение на 10 доменов с использованием 5 процессов. Во втором случае разбиение на 50 доменов с использованием 20 процессов.

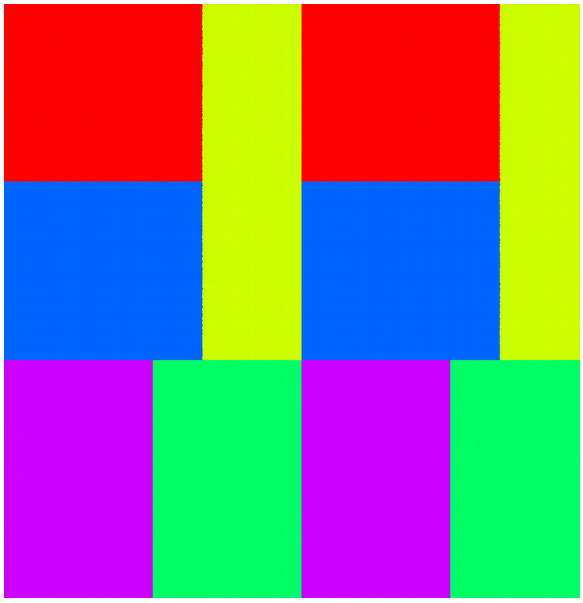

Illustration 1: 1000x1000,d=10,p=5

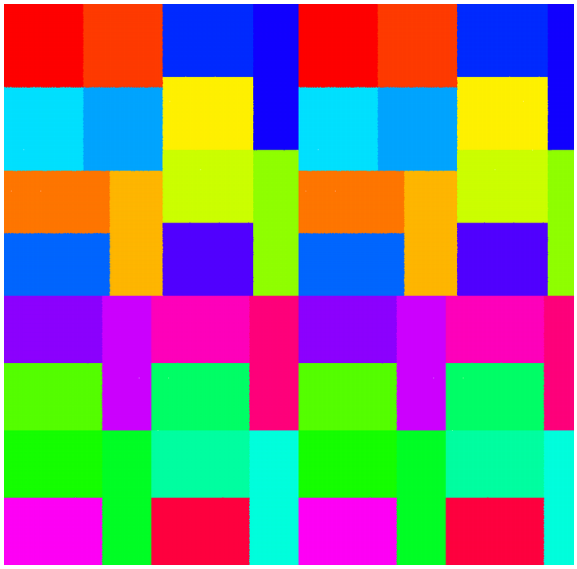

Illustration 2: 1000x1000,d=50,p=20

Приложение А. Последовательная программа

```
#pragma once
#include <algorithm>
#include "Point.hpp"

//struct represents that point with given index
//belongs to given domain
typedef struct _point_domain_t {
    Point point;
    int domain;
} point_domain_t;

/*
points - given array of points
points_domain - elements belonging
length - length of points array
k - wanted domains count
lowest_domain - lowest available domain number
*/
void decompose(
    Point *points,
    point_domain_t *points_domain,
    const int length,
    const int k,
    const int lowest_domain)
{
    if (k == 1) {
        for (int i = 0; i < length; ++i) {
            points_domain[i].domain = lowest_domain;
            points_domain[i].point = points[i];
        }
        return;
    }

    std::sort(points, points + length);

    const int k1 = (k + 1) / 2;
    const int k2 = k - k1;
    const int l1 = 1.0 * k1/k * length;
    const int l2 = length - l1;

    Point::switch_sort_way();
    decompose(points, points_domain, l1, k1, lowest_domain);
    decompose(points + l1, points_domain + l1, l2, k2, lowest_domain + k1);
    Point::switch_sort_way();
}
```

Приложение Б. Параллельная программа

```
#include <cstring>
#include "Point.hpp"
#include "decompose.hpp"
#include "parallel_sort.hpp"

//represents a 'vector' with given length (i.e. size) and ptr
typedef struct _points_domain_t {
    int k;
    int procs_num;
    int size;
    point_domain_t *info;
} points_domain_t;

points_domain_t _parallel_decompose(
    MPI::Intracomm &comm,
    MPI::Datatype &datatype,
    Point *points,
    int part_size,
    int k,
    int lowest_domain,
    int *sendcounts_buffer, // for using in Scatterv
    int *displacements_buffer // for using in Scatterv
);

int calculate_max_part_size(int procs_num, int part_size, int k);

points_domain_t parallel_decompose(
    MPI::Intracomm &comm,
    MPI::Datatype &datatype,
    Point *points,
    int part_size,
    int k,
    int lowest_domain)
{
    points_domain_t points_domain;

    const int procs_num = comm.Get_size();
    const int max_part_size = calculate_max_part_size(procs_num, part_size, k);

    Point *points_copy = new Point[max_part_size];
    memcpy(points_copy, points, part_size * sizeof(Point));

    //Now we can do everyting with this copy.
    //However, passed user's array will remain unchanged.

    int *sendcounts_buffer = new int [procs_num];
    int *displacements_buffer = new int [procs_num];

    points_domain = _parallel_decompose(
            comm, datatype, points_copy,
            part_size, k, lowest_domain,
```

```
52.            sendcounts_buffer,
53.            displacements_buffer
54.        );
55.
56.    delete [] sendcounts_buffer;
57.    delete [] displacements_buffer;
58.
59.    delete [] points_copy;
60.
61.    return points_domain;
62.};
63.
64.points_domain_t _parallel_decompose(
65.    MPI::Intracomm &comm,
66.    MPI::Datatype &datatype,
67.    Point *points,
68.    int part_size,
69.    int k,
70.    int lowest_domain,
71.    int *sendcounts_buffer, // for using in Scatterv
72.    int *displacements_buffer // for using in Scatterv
73.)
74.{
75.    const int procs_num = comm.Get_size();
76.    const int rank = comm.Get_rank();
77.
78.    points_domain_t points_domain;
79.
80.    if (procs_num == 1 || k == 1) { // there is no more processes or domains
81.        if (k > 1) {
82.            for (int i = part_size - 1; i >= 0; --i) {
83.                if (points[i].index != -1) continue;
84.                std::swap(points[i], points[part_size - 1]);
85.                --part_size;
86.            }
87.        }
88.        points_domain.k = k;
89.        points_domain.procs_num = procs_num;
90.        points_domain.size = part_size;
91.        points_domain.info = new point_domain_t[part_size];
92.        decompose(points, points_domain.info, part_size, k, lowest_domain);
93.        return points_domain;
94.    }
95.
96.    std::sort(points, points + part_size); // sort each local array
97.    parallel_sort(comm, datatype, points, part_size);
98.
99.    //now we have global sorted array of points
100.
101.     // searching global middle element's coordinates
102.     const long long global_length = part_size * procs_num;
103.     const int k1 = (k + 1) / 2;
104.     const int k2 = k - k1;
```

```
105.  const long long l1 = 1.0 * k1 / k * global_length;
106.  //const long long l2 = global_length - l1;
107.
108.  const int middle_proc = l1 / part_size;
109.  const int middle_index = l1 % part_size;
110.
111.  MPI::Cartcomm new_comm;
112.  int color; // 0 - left half, 1 - right half
113.  bool i_am_sender = rank == middle_proc;
114.  bool i_am_receiver;
115.  int send_count;
116.  int recv_procs;
117.  Point *send_points_start;
118.
119.  if (middle_proc > 0) {
120.      // we have to send to left
121.      i_am_receiver = rank < middle_proc;
122.      color = rank >= middle_proc;
123.      send_count = middle_index; // elements in position 0, ..., middle_index-1
124.      send_points_start = points; // send from begining
125.      recv_procs = middle_proc;
126.  } else { // middle_proc == 0
127.      // no way to send to left, only to right
128.      i_am_receiver = rank > middle_proc;
129.      color = rank > middle_proc;
130.      send_count = part_size - middle_index; // elements in position middle_index, ..., part_siz
      e-1
131.      send_points_start = points + middle_index; // send from middle_index and so on
132.      recv_procs = procs_num - 1;
133.  }
134.
135.  // communicate only if required
136.  if (send_count > 0) {
137.      memset(sendcounts_buffer, 0, procs_num * sizeof(int));
138.      memset(displacements_buffer, 0, procs_num * sizeof(int));
139.
140.      int offset = middle_proc == 0;
141.
142.      // init sendcounts
143.      for (int i = 0; i < recv_procs; ++i) {
144.          sendcounts_buffer[i + offset] = send_count / recv_procs;
145.      }
146.
147.      for (int i = 0; i < send_count % recv_procs; ++i) {
148.          sendcounts_buffer[i + offset] += 1;
149.      }
150.
151.      // init displacements
152.      displacements_buffer[offset] = 0;
153.      for (int i = 1; i < recv_procs; ++i) {
154.          displacements_buffer[i + offset] =
155.              displacements_buffer[i-1 + offset]
156.              + sendcounts_buffer[i-1 + offset];
```

```
157.     }
158.
159.     comm.Scatterv(
160.         send_points_start, // we send from here
161.         sendcounts_buffer,
162.         displacements_buffer,
163.         datatype,
164.         points + part_size, // we receive to this buffer
165.         sendcounts_buffer[rank], // we receive this amount of points
166.         datatype,
167.         middle_proc
168.     );
169.
170.     int receiver_part_size = 0;
171.     // in all receivers indentify phantom elements
172.     if (i_am_receiver) {
173.         const int recv_n = send_count / recv_procs + (send_count % recv_procs > 0);
174.         receiver_part_size = part_size + recv_n;
175.         const int phantom_elements = recv_n - sendcounts_buffer[rank];
176.
177.         for (int i = 0; i < phantom_elements; ++i) {
178.             points[receiver_part_size-1-i].index = -1;
179.         }
180.     }
181.
182.     const bool can_compensate = middle_proc > 0 && (procs_num - recv_procs > 1);
183.
184.     if (i_am_sender && !can_compensate) {
185.         // make sent elements phantom
186.         for (int i = 0; i < send_count; ++i) send_points_start[i].index = -1;
187.     }
188.
189.     if (can_compensate) {
190.         // compensate sent elements
191.         const int new_procs_num = procs_num - recv_procs;
192.         const long long new_length = 1L * part_size * new_procs_num - send_count;
193.         const int new_part_size = new_length / new_procs_num + (new_length % new_procs
_num > 0);
194.         const int new_middle_proc_points = part_size - send_count;
195.         const int to_compensate = new_part_size - new_middle_proc_points;
196.         const int compensators = new_procs_num - 1;
197.
198.         int *recv_counts = sendcounts_buffer;
199.         memset(recv_counts, 0, procs_num * sizeof(int));
200.         memset(displacements_buffer, 0, procs_num * sizeof(int));
201.
202.         const int offset = middle_proc + 1;
203.         for (int i = 0; i < compensators; ++i) {
204.             recv_counts[offset + i] = to_compensate / compensators;
205.         }
206.
207.         for (int i = 0; i < to_compensate % compensators; ++i) {
208.             recv_counts[offset + i] += 1;
```

```
        }

        displacements_buffer[offset] = 0;
        for (int i = 1; i < compensators; ++i) {
            displacements_buffer[offset + i] =
                displacements_buffer[offset + i-1]
                + recv_counts[offset + i-1];
        }

        if (rank == middle_proc) {
            memmove(points, points + send_count, new_middle_proc_points * sizeof(Point));
        }

        comm.Gatherv(
            points + part_size - recv_counts[rank], // from the tail
            recv_counts[rank], // how many
            datatype,
            points + new_middle_proc_points,
            recv_counts,
            displacements_buffer,
            datatype,
            middle_proc
        );

        // make redistributed as phantom
        for (int i = 0; i < recv_counts[rank]; ++i){
            points[part_size - 1 - i].index = -1;
        }

        if (rank >= middle_proc) {
            part_size = new_part_size;
        }
    }

    if (i_am_receiver) {
        part_size = receiver_part_size;
    }
}

if (color == 0) {
    k = k1;
} else { // color == 1
    k = k2;
    lowest_domain += k1;
}

new_comm = comm.Split(color, rank);

Point::switch_sort_way();
points_domain = _parallel_decompose(
    new_comm, datatype, points,
    part_size, k, lowest_domain,
```

```
        sendcounts_buffer,
        displacements_buffer
    );
    Point::switch_sort_way();

    new_comm.Free();

    return points_domain;
}

/*    Trying predict max memory amount to prevent array reallocation.
    So, algorithm is recursive. In worst case we have to redistribute
    send_n := part_size-1 elements from sender. There is always only one sender
    in each recursion step. Let's consider that the amount of receivers is recv_p.
    It means, every receiver has to add at most ceil(send_n / recv_p) elements.
    There are at least recv_p := n_procs / 2 receivers, we finally get new part_size
    for each receiver: part_size := part_size + ceil(send_n / (n_procs / 2)).
    Let's consider that we already have max recursion depth in recursion_depth variable.
    Then we can describe algorithm in pseudocode.

    def max_part_size(part_size, recursion_depth, n_procs):
        while recursion_depth > 0:
            send_n = part_size - 1
            recv_p = n_procs / 2
            part_size += ceil(send_n / recv_p)
            n_procs = ceil(n_procs / 2)
        return part_size

    Recursion depth depends from k and n_procs.
    Exactly, it equals to rounded up value of log(min(k, procs_num)):

    recursion_depth := ceil(log(min(k, n_procs), base=2)).

    But it's too roughly. So, by modelying part_size changing,
    we can get exact value of max part size.
    */
int calculate_max_part_size(int procs_num, int part_size, int k)
{
    if (procs_num == 1 || k == 1) return part_size;
    // searching global middle element's coordinates
    const long long global_length = part_size * procs_num;
    const int k1 = (k + 1) / 2;
    const int k2 = k - k1;
    const long long l1 = 1.0 * k1 / k * global_length;
    //const long long l2 = global_length - l1;

    const int middle_proc = l1 / part_size;
    const int middle_index = l1 % part_size;

    int procs1, procs2;
    int p1_size, p2_size;
    p1_size = p2_size = part_size;

```

```
314.    if (middle_proc > 0) {
315.        const int send_n = middle_index;
316.        const int recv_p = middle_proc;
317.        const int recv_n = send_n / recv_p + (send_n % recv_p > 0);
318.        p1_size += recv_n;
319.
320.        procs1 = middle_proc;
321.        procs2 = procs_num - procs1;
322.    } else {// middle_proc == 0
323.        const int send_n = part_size - middle_index;
324.        const int recv_p = procs_num - 1;
325.        const int recv_n = send_n / recv_p + (send_n % recv_p > 0);
326.        p2_size += recv_n;
327.
328.        procs1 = 1;
329.        procs2 = procs_num - 1;
330.    }
331.
332.    p1_size = calculate_max_part_size(procs1, p1_size, k1);
333.    p2_size = calculate_max_part_size(procs2, p2_size, k2);
334.
335.    part_size = p1_size > p2_size ? p1_size : p2_size;
336.    return part_size;
337.}
```