МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В.Ломоносова
Факультет вычислительной математики и кибернетики

Компьютерный практикум по учебному курсу «Суперкомпьютеры и параллельная обработка данных»

Разработка параллельной версии программы для одномерного алгоритма Якоби

ОТЧЕТ

о выполненном задании

студента 328 учебной группы факультета ВМК МГУ
Боброва Игоря Валерьевича

гор. Москва
2022 год

Постановка задачи

Даны два массива A и B длины N. На каждой итерации элементам матрицы присваивается значение, равное трети от суммы их соседей из другого массива(i-му элементу из массива B присваивается треть от суммы A i-1-ого, A i-ого, A i+1-ого. Для i-ого элемента из A аналогично). Сперва изменяются элементы из B, а потом из A. Происходит это TSTEPS раз.

Описание программы

Данная программа позволяет применить алгоритм Якоби к двум массивам. На первом этапе работы программа создает два массива и заполняет их значениями (i+2)/n и (i+3)/n для элементов A и B соответственно. Числа в массивах небольшие, чтобы во время выполнения не происходило переполнений. На следующем этапе работы программы происходит выполнение алгоритма Якоби и измерение времени выполнения. На заключительном этапе происходит вывод времени работы программы в стандартный поток вывода.

Компиляция программы происходила с помощью компилятора mpicc. Для межпроцессного взаимодействия использовались специальные команды

```
MPI_Send(
        void* data,
        int count,
        MPI_Datatype datatype,
        int destination,
        int tag,
        MPI_Comm communicator)
```

Данная команда отправляет сообщение с тегом tag, содержащее count×sizeof(datatype) байт, процессу с уникальным идентификатором destination.

```
MPI_Recv(
        void* data,
        int count,
        MPI_Datatype datatype,
        int source,
        int tag,
        MPI_Comm communicator,
        MPI_Status* status)
```

Данная команда получает сообщение с тегом tag, содержащее count×sizeof(datatype) байт, от процесса с уникальным идентификатором source.

При тестировании корректности реализации алгоритма использовалась вспомогательная функция

void print_array(int n, float A[n])

Функция печатает в стандартный поток вывода n элементов массива A.

Код программы

```c
/* Include benchmark-specific header. */
#include "jacobi-1d.h"
#include <mpi.h>

double bench_t_start, bench_t_end;
int size, rank;

static
double rtclock()
{
    struct timeval Tp;
    int stat;
    stat = gettimeofday (&Tp, NULL);
    if (stat != 0)
      printf ("Error return from gettimeofday: %d", stat);
    return (Tp.tv_sec + Tp.tv_usec * 1.0e-6);
}

void bench_timer_start()
{
  bench_t_start = rtclock ();
}

void bench_timer_stop()
{
  bench_t_end = rtclock ();
}

void bench_timer_print()
{
  printf ("Time in seconds = %0.6lf\n", bench_t_end - bench_t_start);
}


static
void init_array (int n,
    float A[ n],
    float B[ n])
{
  int i;
  for (i = 0; i < n; i++)
      {
        A[i] = ((float) i+ 2) / n;
        B[i] = ((float) i+ 3) / n;
      }
}

static
void print_array(int n,
    float A[ n])

{
  int i;

  fprintf(stderr, "==BEGIN DUMP_ARRAYS==\n");
  fprintf(stderr, "begin dump: %s", "A");
  for (i = 0; i < n; i++)
    {
      if (i % 20 == 0) fprintf(stderr, "\n");
      fprintf(stderr, "%0.2f ", A[i]);
    }
  fprintf(stderr, "\nend   dump: %s\n", "A");
  fprintf(stderr, "==END   DUMP_ARRAYS==\n");
}
```

```
65
66  static
67  void kernel_jacobi_1d(int tsteps,
68                        int n,
69                        float A[ n],
70                        float B[ n])
71  {
72    int t, i, right_rank = rank + 1, left_rank = rank - 1, count, ibeg, iend;
73
74    MPI_Status status;
75    MPI_Request req;
76
77    count = n / size;
78
79    ibeg = rank * count + 1;
80    if (rank != size - 1) {
81      iend = (rank + 1) * count;
82    }else{
83      iend = n - 1;
84    }
85
86    if (rank < size) {
87      for (t = 0; t < tsteps; t++){
88        /*Массив B*/
89        for (i = ibeg; i < iend; i++)
90          B[i] = 0.33333 * (A[i - 1] + A[i] + A[i + 1]);
91        if (rank != size - 1)
92          B[iend] = 0.33333 * (A[iend - 1] + A[iend] + A[iend + 1]);
93        if (rank == 0) {
94          if (size != 1) {
95            MPI_Send(&B[iend], 1, MPI_DOUBLE, right_rank, 0, MPI_COMM_WORLD);
96            MPI_Recv(&B[iend + 1], 1, MPI_DOUBLE, right_rank, 0, MPI_COMM_WORLD, &status);
97          }
98        }else if (rank == size - 1) {
99          MPI_Send(&B[ibeg], 1, MPI_DOUBLE, left_rank, 0, MPI_COMM_WORLD);
100         MPI_Recv(&B[ibeg - 1], 1, MPI_DOUBLE, left_rank, 0, MPI_COMM_WORLD, &status);
101       }else {
102         MPI_Send(&B[iend], 1, MPI_DOUBLE, right_rank, 0, MPI_COMM_WORLD);
103         MPI_Send(&B[ibeg], 1, MPI_DOUBLE, left_rank, 0, MPI_COMM_WORLD);
104         MPI_Recv(&B[iend + 1], 1, MPI_DOUBLE, right_rank, 0, MPI_COMM_WORLD, &status);
105         MPI_Recv(&B[ibeg - 1], 1, MPI_DOUBLE, left_rank, 0, MPI_COMM_WORLD, &status);
106       }
107
108       /* Массив A*/
109       for (i = ibeg; i < iend; i++)
110         A[i] = 0.33333 * (B[i-1] + B[i] + B[i + 1]);
111       if (rank != size - 1)
112         A[iend] = 0.33333 * (B[iend-1] + B[iend] + B[iend + 1]);
113       if (rank == 0) {
114         if (size != 1) {
115           MPI_Send(&A[iend], 1, MPI_DOUBLE, right_rank, 0, MPI_COMM_WORLD);
116           MPI_Recv(&A[iend + 1], 1, MPI_DOUBLE, right_rank, 0, MPI_COMM_WORLD, &status);
117         }
118       }else if (rank == size - 1) {
119         MPI_Send(&A[ibeg], 1, MPI_DOUBLE, left_rank, 0, MPI_COMM_WORLD);
120         MPI_Recv(&A[ibeg - 1], 1, MPI_DOUBLE, left_rank, 0, MPI_COMM_WORLD, &status);
121       }else {
122         MPI_Send(&A[iend], 1, MPI_DOUBLE, right_rank, 0, MPI_COMM_WORLD);
123         MPI_Send(&A[ibeg], 1, MPI_DOUBLE, left_rank, 0, MPI_COMM_WORLD);
124         MPI_Recv(&A[iend + 1], 1, MPI_DOUBLE, right_rank, 0, MPI_COMM_WORLD, &status);
125         MPI_Recv(&A[ibeg - 1], 1, MPI_DOUBLE, left_rank, 0, MPI_COMM_WORLD, &status);
126       }
127     }
128   }
129
```

```
130   if (size != 1 && rank < size) {
131     if (rank != size - 1) {
132       MPI_Send(&B[ibeg], iend - ibeg + 1, MPI_DOUBLE, size - 1 , 0, MPI_COMM_WORLD);
133     }else {
134       int i;
135       for (i = 0; i < size - 1; i++) {
136         MPI_Recv(&B[i * count + 1], count, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &status);
137       }
138     }
139   }
140
141   MPI_Barrier(MPI_COMM_WORLD);
142
143   /*Собираем новые данные на нити size - 1*/
144   if (size != 1 && rank < size) {
145     if (rank != size - 1) {
146       MPI_Send(&A[ibeg], iend - ibeg + 1, MPI_DOUBLE, size - 1 , 0, MPI_COMM_WORLD);
147     }else {
148       int i;
149       for (i = 0; i < size - 1; i++) {
150         MPI_Recv(&A[i * count + 1], count, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &status);
151       }
152     }
153   }
154 }
155
156
157 int main(int argc, char** argv)
158 {
159   int n = N;
160   int tsteps = TSTEPS;
161   float (*A)[n];
162   float (*B)[n];
163
164   MPI_Init(&argc, &argv);
165   MPI_Comm_size(MPI_COMM_WORLD, &size);
166   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
167
168   if (size > n){
169     size = n-3;
170   }
171
172   if (rank == size - 1) {
173     printf("n=%d tsteps=%d threads=%d\n", n, tsteps, size);
174   }
175
176   A = (float(*)[n])malloc ((n) * sizeof(float));
177   B = (float(*)[n])malloc ((n) * sizeof(float));
178
179   init_array (n, *A, *B);
180
181   MPI_Barrier(MPI_COMM_WORLD);
182
183   if (rank == size - 1) {
184     bench_timer_start();
185   }
186
187   kernel_jacobi_1d(tsteps, n, *A, *B);
188
189   MPI_Barrier(MPI_COMM_WORLD);
190
191   if (rank == size - 1) {
192     bench_timer_stop();
193     bench_timer_print();
194
195     (*print_array(n, *A);
```

```
194
195       /*print_array(n, *A);
196       print_array(n, *B);*/
197     }
198
199     free((void*)A);;
200     free((void*)B);;
201
202     MPI_Finalize();
203     return 0;
204   }
205
```

Листинг 1: jacobi-1d.c

```
1   #ifndef _JACOBI_1D_H
2   #define _JACOBI_1D_H
3   # if !defined(MINI_DATASET) && !defined(SMALL_DATASET) \
4   && !defined(MEDIUM_DATASET) && !defined(LARGE_DATASET) && !defined(EXTRALARGE_DATASET)
5   #define MINI_DATASET
6   # endif
7   # if !defined(TSTEPS) && !defined(N)
8   # ifdef MINI_DATASET
9   #define TSTEPS 20
10  #define N 30
11  # endif
12  # ifdef SMALL_DATASET
13  #define TSTEPS 40
14  #define N 120
15  # endif
16  # ifdef MEDIUM_DATASET
17  #define TSTEPS 100
18  #define N 400
19  # endif
20  # ifdef LARGE_DATASET
21  #define TSTEPS 500
22  #define N 2000
23  # endif
24  # ifdef EXTRALARGE_DATASET
25  #define TSTEPS 1000
26  #define N 4000
27  # endif
28  #endif
29  #include <stdio.h>
30  #include <unistd.h>
31  #include <string.h>
32  #include <math.h>
33  #include <stdlib.h>
34  #include <math.h>
35  #include <time.h>
36  #include <sys/time.h>
37  #endif
```

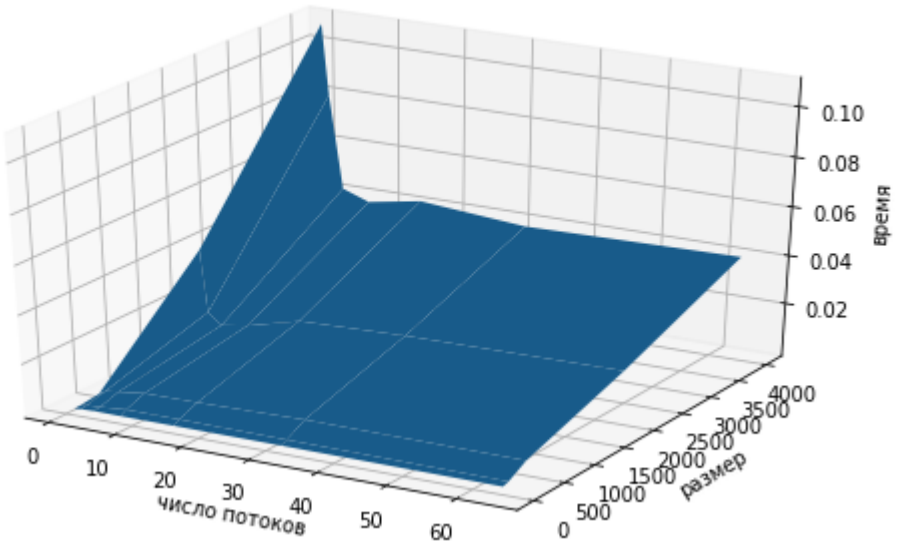
Листинг 2: jacobi-1d.h

Исследование масштабируемости

Для исследования масштабируемости распараллеленной программы были рассмотрены различные размеры массивов и числа итераций алгоритма. Для каждой пары чисел – размера массива и числа итераций – было произведено соответствующее число итераций алгоритма и было измерено время выполнения программы на различном числе потоков. Для каждого случая производилось от трёх до десяти запусков программы. В качестве результата выбиралось наименьшее полученное время. Результаты измерений были сгруппированы и записаны в таблицу. На основе этой таблицы была построена трехмерная гистограмма зависимостей между размером, временем и числом потоков. Также был проведен анализ полученных результатов.

Dataset (итерации, размер)	Mini (20, 30)	Small (40, 120)	Medium (100, 400)	Large (500, 2000)	Extralarge (1000, 4000)

Ядра	Время				
1	0.000021	0.000143	0.001115	0.040217	0.109209
2	0.000272	0.000725	0.002718	0.014581	0.080754
4	0.000485	0.000942	0.003401	0.009960	0.042764
8	0.000437	0.001890	0.004823	0.011666	0.038849
16	0.000512	0.001895	0.005033	0.017511	0.039704
32	0.000552	0.001677	0.006039	0.019045	0.039177
64	0.000583	0.001368	0.005861	0.020192	0.040928



Анализ

Таким образом, распараллеливание алгоритма Якоби обеспечивает значительное увеличение скорости выполнения программы даже на небольших объёмах данных.

Зависимость времени выполнения от числа потоков при использовании небольшого количества ядер является почти линейной. Однако с увеличением числа ядер зависимость перестает быть линейной. Это объясняется тем, что на отправку и получение необходимых данных затрачивается некоторое время. Кроме того распараллеленная функция завершает своё выполнение только тогда, когда завершит своё выполнение последний процесс.

В ходе работы было также обнаружено, что производительность растет только до определенного момента. При дальнейшем увеличении числа потоков время начинает увеличиваться. Причем чем меньше размер матрицы и блоков, тем меньшее количество ядер является наиболее оптимальным.

Время работы MPI-версии значительно больше, чем у OpenMP версии. Это объясняется, во-первых, накладными расходами, связанными с отправкой и ожиданием данных, и, во-вторых, занятостью и сложностью выделения большого числа процессов на Polus.

Кроме того, время, затраченное на разработку OpenMP версии, оказалось значительно ниже времени, затраченного на разработку MPI версии. Это связано с усложнением в MPI версии логики программы, которую

программисту необходимо реализовать самому, в отличие от помощи компилятора в OpenMP версии.

Вывод

В ходе работы был реализован алгоритм Якоби. Было выполнено распараллеливание программы с помощью технологии MPI. Также была исследована масштабируемость полученной параллельной программы. Кроме того был произведён сравнительный анализ OpenMP и MPI версий.