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 измеются 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 xls с ключом −qsmp = omp. Для распараллеливания использовались директивы OpneMP вида

#pragma omp parallel for shared(. .) private(. .)

в функциях kernel_jacobi_1d() и init_array(). Остальные функции не требуют больших вычислительных мощностей.

Код программы

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

double bench_t_start, bench_t_end;

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;
  #pragma omp parallel for private(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");
}
```

```
64
65  static
66  void kernel_jacobi_1d(int tsteps,
67                        int n,
68                        float A[ n],
69                        float B[ n])
70  {
71    int t, i;
72    for (t = 0; t < tsteps; t++)
73      {
74        #pragma omp parallel for shared(B) private(i)
75        for (i = 1; i < n - 1; i++)
76          B[i] = 0.33333 * (A[i-1] + A[i] + A[i + 1]);
77
78        #pragma omp parallel for shared(A) private(i)
79        for (i = 1; i < n - 1; i++)
80          A[i] = 0.33333 * (B[i-1] + B[i] + B[i + 1]);
81      }
82  }
83
84
85  int main(int argc, char** argv)
86  {
87    int n = N;
88    int tsteps = TSTEPS;
89    float (*A)[n]; A = (float(*)[n])malloc ((n) * sizeof(float));;
90    float (*B)[n]; B = (float(*)[n])malloc ((n) * sizeof(float));;
91
92    init_array (n, *A, *B);
93
94    bench_timer_start();;
95
96    kernel_jacobi_1d(tsteps, n, *A, *B);
97
98    bench_timer_stop();;
99    bench_timer_print();;
100
101   if (argc > 42 && ! strcmp(argv[0], "")) print_array(n, *A);
102
103   free((void*)A);;
104   free((void*)B);;
105
106   return 0;
107 }
```

Листинг 1: jacobi-1d.c

```c
#ifndef _JACOBI_1D_H
#define _JACOBI_1D_H
# if !defined(MINI_DATASET) && !defined(SMALL_DATASET) && \
!defined(MEDIUM_DATASET) && !defined(LARGE_DATASET) && !defined(EXTRALARGE_DATASET)
#define LARGE_DATASET
# endif
# if !defined(TSTEPS) && !defined(N)
# ifdef MINI_DATASET
#define TSTEPS 20
#define N 30
# endif
# ifdef SMALL_DATASET
#define TSTEPS 40
#define N 120
# endif
# ifdef MEDIUM_DATASET
#define TSTEPS 100
#define N 400
# endif
# ifdef LARGE_DATASET
#define TSTEPS 500
#define N 2000
# endif
# ifdef EXTRALARGE_DATASET
#define TSTEPS 1000
#define N 4000
# endif
#endif
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>
#endif
```

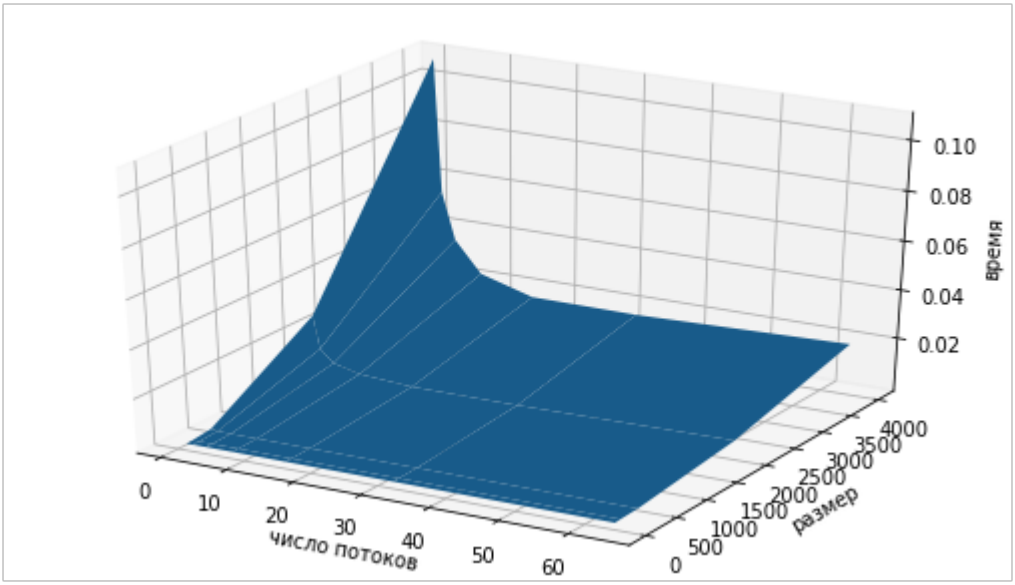
Листинг 2: jacobi-1d.h

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	0.000031	0.000143	0.001095	0.027197	0.108559
2	0.000042	0.000155	0.000608	0.013811	0.054844
4	0.000075	0.000182	0.000471	0.008670	0.035674
8	0.000107	0.000200	0.000493	0.005976	0.023009
16	0.000102	0.000175	0.000473	0.004471	0.016704
32	0.000132	0.000237	0.000549	0.004515	0.016487
64	0.000103	0.000278	0.000551	0.005802	0.019038



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 для создания потоков используются дополнительные ресурсы. Кроме того распараллеленная функция завершает своё выполнение только тогда, когда завершит своё выполнение последний поток.

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.

Вывод

В ходе работы был реализован алгоритм Якоби. Было выполнено распараллеливание программы, реализующей этот алгоритм. Также была исследована масштабируемость полученной параллельной программы.