Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа № 4 по дисциплине ”Параллельные вычисления”

Выполнили:

Айтуганов Д. А.
Чебыкин И. Б.

Группа: P42111, P42101

Проверяющий: Балакшин П. В.

Санкт-Петербург, 2019

Цель работы

1. В программе, полученной в результате выполнения ЛР-3, так изменить этап Generate, чтобы генерируемый набор случайных чисел не зависел от количества потоков, выполняющих программу. Например, на каждой итерации i перед вызовом rand_r можно вызывать функцию srand(f(i)), где f – произвольно выбранная функция. Можно придумать и использовать любой другой способ.
2. Заменить вызовы функции gettimeofday на omp_get_wtime.
3. Распараллелить вычисления на этапе Sort, для чего выполнить сор- тировку в два этапа: • Отсортировать первую и вторую половину массива в двух независимых нитях (можно использовать OpenMP-директиву ”parallel sections”); • Объединить отсортированные половины в единый массив.
4. Написать функцию, которая один раз в секунду выводит в консоль сообщение о текущем проценте завершения работы программы. Указанную функцию необходимо запустить в отдельном потоке, параллельно работающем с основным вычислительным циклом.
5. Обеспечить прямую совместимость (forward compatibility) напи- санной параллельной программы. Для этого все вызываемые функ- ции вида «omp_*» можно условно переопределить в препроцессор- ных директивах, например, так:
6. Провести эксперименты, варьируя N от min(N 2 x , N 1) до N 2 , где значения N 1 и N 2 взять из ЛР-1, а N x – это такое значение N , 81при котором накладные расходы на распараллеливание превыша- ют выигрыш от распараллеливания.

Конфигурация

Процессор

```
CPU(s):                          16
Thread(s) per core:              1
Core(s) per socket:              8
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       AuthenticAMD
Model name:                      AMD Ryzen 7 1700 Eight-Core Processor
CPU MHz:                         2645.861
CPU max MHz:                     3000.0000
CPU min MHz:                     1550.0000
```

RAM: 32 GB

Компиляторы

```
gcc (GCC) 9.1.0
```

Исходный код

```
#include <float.h>
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>

#ifdef _OPENMP
   #include ”omp.h”

   void printResults(int N, double delta_ms, double X) {
      printf(”N = %d. milliseconds passed: %lf\n”, N, 1000 * delta_ms);
      printf(”N = %d. X=%e\n”, N, X);
   }
#else
   int omp_get_wtime(){
      struct timeval T11;
      gettimeofday(&T11, NULL);
      return T11.tv_sec * 1000 + T11.tv_usec/1000;
   }

   void printResults(int N, double delta_ms, double X) {
      printf(”N = %d. milliseconds passed: %lf\n”, N, delta_ms);
      printf(”N = %d. X=%e\n”, N, X);
   }

   int omp_get_thread_num() {
      return 1;
   }

   int omp_set_nested(int value) {
      return 0;
   }

   int omp_get_num_procs() {
      return 1;
   }

   int omp_get_num_threads() {
      return 1;
   }
   void omp_set_num_threads(int n) {
   }
#endif

#define SCHEDULE schedule(static, 2)

const static int c_a = 567;
const static int c_experiments = 50;

void generate(unsigned int seed, double *p, unsigned int N,
```

```
              unsigned int min, unsigned int max) {
    unsigned int i;
    srand(time(NULL));
    for (i = 0; i < N; i++) {
        p[i] = (rand_r(&seed) % max) + min;
    }
}

void lab_swap(double * lhs, double * rhs) {
    double tmp = *lhs;
    *lhs = *rhs;
    *rhs = tmp;
}

int correct(double *arr, int start, int n) {
    if (n < 2) {
        return 1;
    }
    for(int i = start; i < n - 1; i++) {
        if (arr[i] > arr[i + 1]) {
            return 0;
        }
    }
    return 1;
}

void shuffle(double *arr, int start, int n) {
    int i;
    for (i = start; i < n; i++) {
        lab_swap(&arr[i], &arr[(rand() % n)]);
    }
}

void bogo_sort(double *arr, int start, int n) {
    while (!correct(arr, start, n))
        shuffle(arr, start, n);
}

void merge(double *out_arr, double *arr, int chunk_size, int n) {
    int cnt = n / chunk_size;
    if (n % chunk_size != 0) {
        cnt++;
    }
    int *idx = calloc(1, cnt * sizeof(int));
    for (int i = 0; i < n; i++) {
        int min_j = -1;
        double min = FLT_MAX;
        for(int j = 0; j < cnt; j++) {
            int chunk = chunk_size;
            if (chunk * j + idx[j] > n) {
                chunk = (n - 1 - idx[j]) / j;
            }
            if (chunk * j + idx[j] == n) {
                continue;
            }
            if (idx[j] < chunk) {
```

```
                double val = arr[chunk * j + idx[j]];
                if (min_j == -1 || val < min) {
                    min_j = j;
                    min = val;
                }
            }
        }
        out_arr[i] = min;
        idx[min_j]++;
    }
    for (int i = 0; i < n; i++) {
        out_arr[i] = arr[i];
    }
    free(idx);

}

double lab_abs(double v) {
    if (v < 0) {
        return -v;
    }
    return v;
}

double lab_min(double lhs, double rhs) {
    return lhs > rhs ? rhs : lhs;
}

double lab_max(double lhs, double rhs) {
    return lhs < rhs ? rhs : lhs;
}

double lab_cot(double val) {
    return cos(val) / sin(val);
}

double lab_coth(double val) {
    return cosh(val) / sinh(val);
}

void print_array(double *p, unsigned int N) {
    unsigned int i = 0;
    for (i = 0; i < N - 1; i++) {
        printf("%f ", p[i]);
    }
    printf("%f\n", p[N - 1]);
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        return -1;
    }
    omp_set_nested(1);
    int N;
    int num = 0, percent = 0;
    double reduced_sum = 0.0;
```

```
 double begin, end, delta_ms;

 const int k = omp_get_num_procs();

#pragma omp parallel default(none) num_threads(2) private(num) shared(N, k, argv, begin, end, delta_ms, reduced_sum, pe
 {
     num = omp_get_thread_num();
     fflush(stdout);
     if (num == 0) {
         for(;percent!=300;) {
             printf("%lf\n", (percent / 300.0) * 100.0);
             fflush(stdout);
             sleep(1);
         }
     } else {
         N = atoi(argv[1]);

         double * m1 = malloc(sizeof(double) * N);
         double * m2 = malloc(sizeof(double) * N / 2);

         begin = omp_get_wtime();
         unsigned int i;
         for (i = 0; i < c_experiments; i++) {
             // 1. Generate: M1 of N elements, M2 of N/2 elements
             generate(i, m1, N, 1, c_a);
             //puts("M1");
             //print_array(m1, N);
             generate(i, m2, N / 2, c_a, 10 * c_a);
             //puts("M2");
             //print_array(m2, N / 2);
             percent++;
             // 2. Map: coth(sqrt(M1[j])) ; M2[j] = abs(cot(M2[j]))
             unsigned int j;
             #pragma omp parallel for default(none) private(j) shared(m1, N) SCHEDULE
             for (j = 0; j < N; j++) {
                 m1[j] = lab_coth(sqrt(m1[j]));
             }
             //puts("M1 coth");
             //print_array(m1, N);
             #pragma omp parallel for default(none) private(j) shared(m2, N) SCHEDULE
             for (j = 0; j < N / 2; j++) {
                 m2[j] = lab_abs(lab_cot(m2[j]));
             }
             percent++;
             //puts("M2 abs cot");
             //print_array(m2, N / 2);
             // 3. Merge: M2[j] = max(M1[j], M2[j]) , j e N/2
             #pragma omp parallel for default(none) private(j) shared(m1, m2, N) SCHEDULE
             for (j = 0; j < N / 2; j++) {
                 m2[j] = lab_max(m1[j], m2[j]);
             }
             percent++;
             //puts("max of M1 M2");
             //print_array(m2, N / 2);
             // 4. Sort: gnome_sort(M2, N/2)
             const int old_threads = omp_get_num_threads();
```

```
            const int chunk_size = ceil(((N / 2.0) / (double)k));
            omp_set_num_threads(k);
            #pragma omp parallel default(none) private(j) shared(m2, k, chunk_size, N)
            {
                int threadnum = omp_get_thread_num();
                int low = (N * threadnum) / k;
                int high = (N * (threadnum + 1)) / k;
                for(j = low; j < high; j += chunk_size) {
                    if (j + chunk_size > N / 2) {
                        int new_chunk_size = (N / 2) - j;
                        bogo_sort(m2, j, j + new_chunk_size);
                    } else {
                        bogo_sort(m2, j, j + chunk_size);
                    }
                }
            }
            omp_set_num_threads(old_threads);

            double m2_merged[N/2];
            merge(m2_merged, m2, chunk_size, N / 2);
            percent++;
            // 5. Reduce: 1. min_non_zero(M2)
            //            2. if (((long)(M2[j] / min_non_zero)) & ~(1))
            //                   sum += sin(M2[j])
            double min_non_zero = DBL_MAX;
        #pragma omp parallel for default(none) private(j) shared(m2_merged, N) reduction(min:min_non_zero) SCHEDULE
            for (j = 0; j < N / 2; j++) {
                if (m2_merged[j] != 0) {
                    min_non_zero = lab_min(min_non_zero, m2_merged[j]);
                }
            }
            percent++;
            //printf("Min non zero: %f\n", min_non_zero);
        #pragma omp parallel for default(none) private(j) shared(m2_merged, N, min_non_zero) reduction(+:reduced_sum) 
            for (j = 0; j < N / 2; j++) {
                if (((long)(m2_merged[j] / min_non_zero)) & ~(1)) {
                    reduced_sum += sin(m2_merged[j]);
                }
            }
            percent++;
            //printf("Sum: %e\n", reduced_sum);
         }
         end = omp_get_wtime();
         delta_ms = end - begin;
       }
    }
    printResults(N, delta_ms, reduced_sum / c_experiments);

    return 0;
}
```

Результаты

N	seq(N)	p(N)
14	25	18.364857
15	30	19.751233
16	177	17.995902
17	177	27.695776
18	2401	22.627936
19	2393	20.316079
20	21513	21.677666

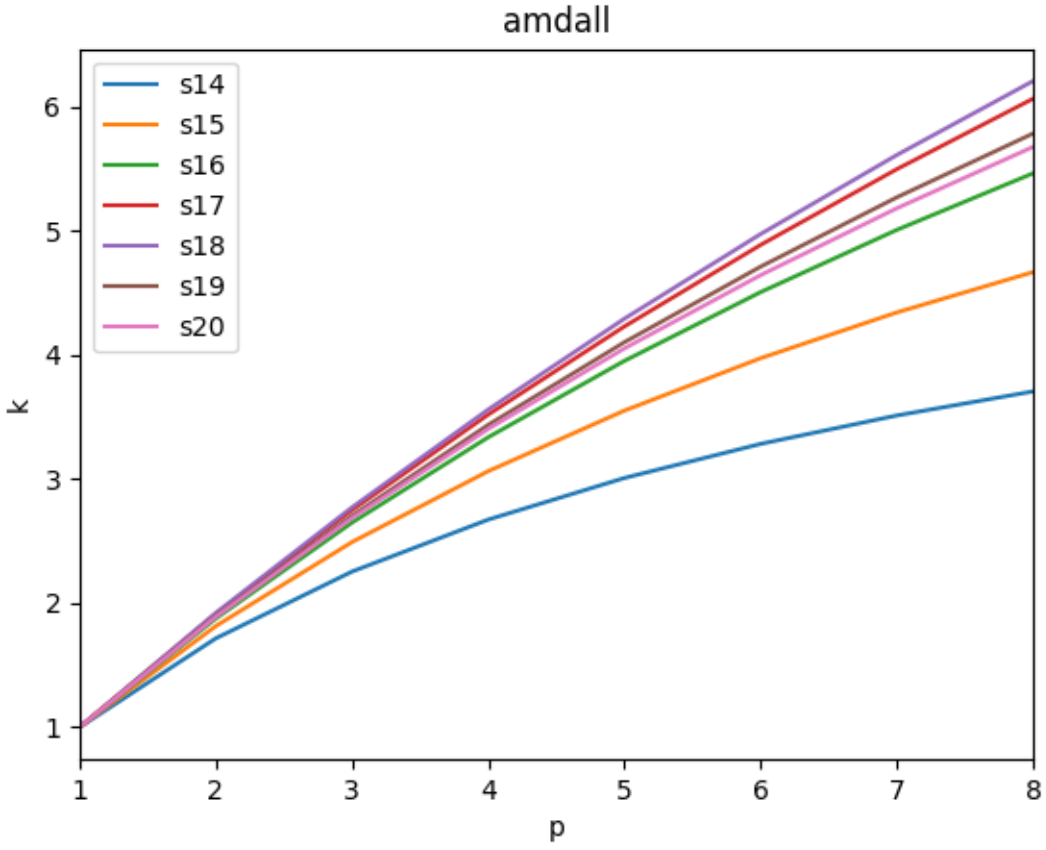
Доверительный интервал:

(6.1357968000000005, 5.374573883629832, 6.897019716370169)

Время работы параллельных участков:

N	p(N)
14	15.322097
15	17.734203
16	16.800604
17	26.430789
18	21.692247
19	19.202428
20	20.407868

N	k
14	0.834316
15	0.897878
16	0.933579
17	0.954325
18	0.958648
19	0.945183
20	0.941423



.

Выводы

После выполнения лабораторной работы можно сказать, что распараллеливание сортировки сильно ускорило работу программы.