Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа № 5 по дисциплине "Параллельные вычисления"

Выполнили:

Айтуганов Д. А.
Чебыкин И. Б.

Группа: P42111, P42101

Проверяющий: Балакшин П. В.

Санкт-Петербург, 2019

Цель работы

Взять в качестве исходной OpenMP-программу из ЛР-5, в кото- рой распараллелены все этапы вычисления. Убедиться, что в этой программе корректно реализован одновременный доступ к общей переменной, используемой для вывода в консоль процента завершения программы.

Изменить исходную программу так, чтобы вместо OpenMP-директив применялся стандарт «POSIX Threads»

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>
#include <pthread.h>
#include <assert.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]);
}

typedef struct PercentArgs {
    int* percent;
    pthread_mutex_t mtx;
} PercentArgs;


typedef struct Ctx {
    double *m1;
    double *m2;
    int j;
    int threadnum;
    double *shared;
    pthread_mutex_t mtx;
    double *results;
    int chunk_size;
    int N;
} Ctx;

typedef void (*iter_fn)(Ctx*);

typedef enum Schedule {
    Static,
} Schedule;

typedef struct Chunk {
    int low;
```

```
    int high;
    int is_used;
    int is_processed;
    pthread_mutex_t mtx;
} Chunk;

typedef struct WorkerArgs {
    int threadnum;
    int N, k;
    Ctx ctx;
    iter_fn fn;
    int chunk_size;
    Chunk *chunks;
    int chunks_count;
} WorkerArgs;

void *static_worker(void *args) {
    WorkerArgs* sargs = (WorkerArgs*)args;
    int low = (sargs->N * sargs->threadnum) / sargs->k;
    int high = (sargs->N * (sargs->threadnum + 1)) / sargs->k;
    sargs->ctx.threadnum = sargs->threadnum;
    sargs->ctx.chunk_size = sargs->chunk_size;
    sargs->ctx.N = sargs->N;
    unsigned int j;
    for (j = low; j < high; j+=sargs->chunk_size) {
        sargs->ctx.j = j;
        sargs->fn(&sargs->ctx);
    }
    return NULL;
}

void *dynamic_worker(void *args) {
    WorkerArgs* sargs = (WorkerArgs*)args;
    while(1) {
        int processed_cnt = 0;
        for(int i = 0; i < sargs->chunks_count; i++) {
            Chunk *ch = &sargs->chunks[i];
            pthread_mutex_lock(&ch->mtx);
            if (ch->is_processed) {
                processed_cnt++;
                pthread_mutex_unlock(&ch->mtx);
                continue;
            }
            if (ch->is_used) {
                pthread_mutex_unlock(&ch->mtx);
                continue;
            }
            ch->is_used = 1;
            pthread_mutex_unlock(&ch->mtx);
            unsigned int j;
            for (j = ch->low; j < ch->high; j++) {
                sargs->ctx.j = j;
                sargs->fn(&sargs->ctx);
            }
            pthread_mutex_lock(&ch->mtx);
            ch->is_processed = 1;
```

```
            pthread_mutex_unlock(&ch->mtx);
            processed_cnt++;
        }
        if (processed_cnt >= sargs->chunks_count) {
            break;
        }
    }
    return NULL;
}

int dyn_init_threads(WorkerArgs** args, pthread_t **threads, Chunk** chunks,
                int N, int threads_amount, int chunk_size, Ctx ctx, iter_fn fn) {
    *threads = malloc(threads_amount * sizeof(pthread_t));
    int rc = 0;
    *args = malloc(threads_amount * sizeof(WorkerArgs));
    WorkerArgs *args_arr = *args;
    pthread_t *threads_arr = *threads;
    *chunks = malloc(threads_amount * sizeof(Chunk));
    Chunk* chunks_arr = *chunks;
    for (int i = 0; i < threads_amount; i++) {
        int low = (N * i) / threads_amount;
        int high = (N * (i + 1)) / threads_amount;
        Chunk ch = {low, high, 0, 0, PTHREAD_MUTEX_INITIALIZER};
        chunks_arr[i] = ch;
    }
    for (int i = 0; i < threads_amount; i++) {
        WorkerArgs *cur_args = &args_arr[i];
        cur_args->threadnum = i;
        cur_args->N = N;
        cur_args->k = threads_amount;
        cur_args->ctx = ctx;
        cur_args->fn = fn;
        cur_args->chunk_size = chunk_size;
        cur_args->chunks_count = threads_amount;
        cur_args->chunks = chunks_arr;
        rc = pthread_create(&threads_arr[i], NULL, dynamic_worker, cur_args);
        if (rc != 0) {
            return rc;
        }
    }
    return 0;
}


int init_threads(WorkerArgs** args, pthread_t **threads, int N,
                int threads_amount, int chunk_size, Ctx ctx, iter_fn fn) {
    *threads = malloc(threads_amount * sizeof(pthread_t));
    int rc = 0;
    *args = malloc(threads_amount * sizeof(WorkerArgs));
    WorkerArgs *args_arr = *args;
    pthread_t *threads_arr = *threads;
    for (int i = 0; i < threads_amount; i++) {
        WorkerArgs *cur_args = &args_arr[i];
        cur_args->threadnum = i;
        cur_args->N = N;
        cur_args->k = threads_amount;
```

```
        cur_args->ctx = ctx;
        cur_args->fn = fn;
        cur_args->chunk_size = chunk_size;
        rc = pthread_create(&threads_arr[i], NULL, static_worker, cur_args);
        if (rc != 0) {
            return rc;
        }
    }
    return 0;
}

int dyn_join_threads(WorkerArgs* args, pthread_t *threads, Chunk *chunks, int threads_amount) {
    int rc = 0;
    for (int i = 0; i < threads_amount; i++) {
        rc = pthread_join(threads[i], NULL);
        if (rc != 0) {
            return rc;
        }
    }
    free(threads);
    free(args);
    free(chunks);
    return 0;
}

int join_threads(WorkerArgs* args, pthread_t *threads, int threads_amount) {
    int rc = 0;
    for (int i = 0; i < threads_amount; i++) {
        rc = pthread_join(threads[i], NULL);
        if (rc != 0) {
            return rc;
        }
    }
    free(threads);
    free(args);
    return 0;
}

void map_step1(Ctx* ctx) {
    ctx->m1[ctx->j] = lab_coth(sqrt(ctx->m1[ctx->j]));
}

void map_step2(Ctx* ctx) {
    ctx->m2[ctx->j] = lab_abs(lab_cot(ctx->m2[ctx->j]));
}

void merge_step1(Ctx* ctx) {
    ctx->m2[ctx->j] = lab_max(ctx->m1[ctx->j], ctx->m2[ctx->j]);
}

void sort_step1(Ctx* ctx) {
    ctx->m2[ctx->j] = lab_max(ctx->m1[ctx->j], ctx->m2[ctx->j]);
    if (ctx->j + ctx->chunk_size > ctx->N) {
        int new_chunk_size = ctx->N - ctx->j;
        bogo_sort(ctx->m2, ctx->j, ctx->j + new_chunk_size);
    } else {
```

```
        bogo_sort(ctx->m2, ctx->j, ctx->j + ctx->chunk_size);
    }
}


void reduce_step1(Ctx* ctx) {
    if (ctx->m2[ctx->j] == 0) {
        return;
    }
    pthread_mutex_lock(&ctx->mtx);
    double local_min = *ctx->shared;
    pthread_mutex_unlock(&ctx->mtx);

    local_min = lab_min(local_min, ctx->m2[ctx->j]);

    pthread_mutex_lock(&ctx->mtx);
    if (local_min < *ctx->shared) {
        *ctx->shared = local_min;
    }
    pthread_mutex_unlock(&ctx->mtx);
}

void reduce_step2(Ctx* ctx) {
    if (((long)(ctx->m2[ctx->j] / *ctx->shared)) & ~(1)) {
        ctx->results[ctx->threadnum] += sin(ctx->m2[ctx->j]);
    }
}

void *calc_percent(void *args) {
    PercentArgs *pargs = (PercentArgs*)args;
    pthread_mutex_lock(&pargs->mtx);
    int percent = *pargs->percent;
    pthread_mutex_unlock(&pargs->mtx);

    for(;percent!=300;) {
        printf("%lf\n", (percent / 300.0) * 100.0);
        fflush(stdout);
        sleep(1);

        pthread_mutex_lock(&pargs->mtx);
        percent = *pargs->percent;
        pthread_mutex_unlock(&pargs->mtx);
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        return -1;
    }
    omp_set_nested(1);
    int N;
    int percent = 0;
    double reduced_sum = 0.0;
    double begin, end, delta_ms, parallel_ms = 0;
    double map_step1_ms = 0.0, map_step2_ms = 0.0, merge_step1_ms = 0.0;
```

```
double sort_step1_ms = 0.0, reduce_step1_ms = 0.0, reduce_step2_ms = 0.0;
pthread_t percent_thread;

//const int k = omp_get_num_procs();
const int k = sysconf(_SC_NPROCESSORS_ONLN);
PercentArgs pargs = {&percent, PTHREAD_MUTEX_INITIALIZER};
int rc = pthread_create(&percent_thread, NULL, calc_percent, &pargs);
assert(rc == 0);

N = atoi(argv[1]);

double * m1 = malloc(sizeof(double) * N);
double * m2 = malloc(sizeof(double) * N / 2);
double * m2_merged = malloc(N / 2 * sizeof(double));
double * results = malloc(k * sizeof(double));

//int old_threads = omp_get_num_threads();

begin = omp_get_wtime();
unsigned int i;
for (i = 0; i < c_experiments; i++) {
    memset(results, 0, k);
    Ctx ctx = {m1, m2, 0, 0, NULL, PTHREAD_MUTEX_INITIALIZER, results, 0, 0};
    // 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);
    pthread_mutex_lock(&pargs.mtx);
    percent++;
    pthread_mutex_unlock(&pargs.mtx);
    // 2. Map: coth(sqrt(M1[j])) ; M2[j] = abs(cot(M2[j]))
    unsigned int j;
    double sec_begin = omp_get_wtime();

    pthread_t *threads = NULL;
    WorkerArgs *args = NULL;
    Chunk *chunks = NULL;
    dyn_init_threads(&args, &threads, &chunks, N, k, 1, ctx, map_step1);
    dyn_join_threads(args, threads, chunks, k);

    double sec_end = omp_get_wtime() - sec_begin;
    map_step1_ms += sec_end;
    parallel_ms += sec_end;
    //puts("M1 coth");
    //print_array(m1, N);
    sec_begin = omp_get_wtime();

    init_threads(&args, &threads, N / 2, k, 1, ctx, map_step2);
    join_threads(args, threads, k);

    sec_end = omp_get_wtime() - sec_begin;
    parallel_ms += sec_end;
    map_step2_ms += sec_end;
```

```
pthread_mutex_lock(&pargs.mtx);
percent++;
pthread_mutex_unlock(&pargs.mtx);
//puts("M2 abs cot");
//print_array(m2, N / 2);
// 3. Merge: M2[j] = max(M1[j], M2[j]) , j e N/2
sec_begin = omp_get_wtime();

init_threads(&args, &threads, N / 2, k, 1, ctx, merge_step1);
join_threads(args, threads, k);

sec_end = omp_get_wtime() - sec_begin;
parallel_ms += sec_end;
merge_step1_ms += sec_end;

pthread_mutex_lock(&pargs.mtx);
percent++;
pthread_mutex_unlock(&pargs.mtx);
//puts("max of M1 M2");
//print_array(m2, N / 2);
// 4. Sort: gnome_sort(M2, N/2)
const int chunk_size = ceil(((N / 2.0) / (double)k));
sec_begin = omp_get_wtime();

init_threads(&args, &threads, N / 2, k, chunk_size, ctx, sort_step1);
join_threads(args, threads, k);

sec_end = omp_get_wtime() - sec_begin;
parallel_ms += sec_end;
sort_step1_ms += sec_end;

memset(m2_merged, 0, N / 2);
merge(m2_merged, m2, chunk_size, N / 2);

pthread_mutex_lock(&pargs.mtx);
percent++;
pthread_mutex_unlock(&pargs.mtx);
// 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;
ctx.shared = &min_non_zero;
ctx.m2 = m2_merged;

sec_begin = omp_get_wtime();

init_threads(&args, &threads, N / 2, k, 1, ctx, reduce_step1);
join_threads(args, threads, k);

min_non_zero = *ctx.shared;
sec_end = omp_get_wtime() - sec_begin;
parallel_ms += sec_end;
reduce_step1_ms += sec_end;

pthread_mutex_lock(&pargs.mtx);
```

```
        percent++;
        pthread_mutex_unlock(&pargs.mtx);
        //printf("Min non zero: %f\n", min_non_zero);
        sec_begin = omp_get_wtime();

        init_threads(&args, &threads, N / 2, k, 1, ctx, reduce_step2);
        join_threads(args, threads, k);

        for (j = 0; j < k; j++) {
            reduced_sum += results[j];
        }
        sec_end = omp_get_wtime() - sec_begin;
        parallel_ms += sec_end;
        reduce_step2_ms += sec_end;

        pthread_mutex_lock(&pargs.mtx);
        percent++;
        pthread_mutex_unlock(&pargs.mtx);
        //printf("Sum: %e\n", reduced_sum);
    }
    end = omp_get_wtime();
    delta_ms = end - begin;
    printResults(N, delta_ms, reduced_sum / c_experiments);
    printf("%lf\n", 1000 * parallel_ms);
    printf("%lf\n", 1000 * map_step1_ms);
    printf("%lf\n", 1000 * map_step2_ms);
    printf("%lf\n", 1000 * merge_step1_ms);
    printf("%lf\n", 1000 * sort_step1_ms);
    printf("%lf\n", 1000 * reduce_step1_ms);
    printf("%lf\n", 1000 * reduce_step2_ms);

    rc = pthread_join(percent_thread, NULL);
    assert(rc == 0);
    free(m1);
    free(m2);
    free(m2_merged);
    free(results);

    return 0;
}
```

Результаты

OpenMP

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

N	seq(N)	p(N)
19	2393	20.316079
20	21513	21.677666

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

.

pthreads

N	seq(N)	p(N)
14	25	149.51189
15	30	158.00416
16	177	160.36467
17	177	159.20362
18	2401	159.13963
19	2393	165.20000
20	21513	159.75579

Время работы параллельных участков:

N	p(N)
14	148.32282
15	156.00416
16	158.46567
17	157.80382
18	158.23863
19	164.10320

N	p(N)
20	158.90577

N	k
14	0.8920469
15	0.8873421
16	0.8881582
17	0.8912074
18	0.9443383
19	0.9333607
20	0.9246792

.

.

Выводы

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