Университет ИТМО

Лабораторная работа №2
Вариант 3

Митрофанов А.А
группа P4115

2018г

Лабораторная работа №2

Описание:
В рамках данной лабораторной работы предлагается ознакомится с алгоритмами распределения нагрузки между потоками обработки данных и оценить их эффективность на базе решения построенного в первой лабораторной работе.

Задание :
Разработать консольное приложение для обработки большого потока данных из внешнего источника (файла) с применением алгоритма распределения данных. В рамках данной работы необходимо реализовать три алгоритма: Round-Robin, Least Loaded, Predictive. Для алгоритма «Predictive» необходимо самостоятельно выбрать функцию оценки сложности данных и предоставить обоснование выбора. В качестве аргументов запуска приложения должны передаваться следующие параметры: номер способа распределения данных, путь до файла с исходными данными

Round-Robin - задачи равномерно распределяются по всем нодам
Least Loaded – новая задачи посылается на менее загруженную ноду
Predictive – Делается предсказание, на какой ноде задача отработается быстрее, и посылает на предсказанную ноду задачу. Как функцию оценки брал среднее время работы над одной таской * количество тасок в очереди.
Код:

```
class block_write_queue{
   std::queue<std::pair<int,int>> data;
   std::mutex w_lock;
   int counter=0;

public:
   bool done = false;
   bool have_tasks = false;
   block_write_queue() = default;
   std::pair<int,int>* pop(){


      std::pair<int,int> *e = new std::pair<int,int>(data.front());
      w_lock.lock();
      data.pop();
      have_tasks = not data.empty();
      w_lock.unlock();
      return e;
   };

   void push(std::pair<int,int> e){
      w_lock.lock();
      data.push(e);
      have_tasks = true;
      w_lock.unlock();
```

```
        counter++;
    }
    int get_processed_count(){
        return counter;
    }
};


void calculator(block_write_queue * q, std::vector<matrix*> *data_ptr, matrix * out_matrix, int shape, int
thr_id) {
    std::vector<matrix*>& data =  *data_ptr;
    std::chrono::duration<double> helpfull_time;
    std::chrono::duration<double> useless_time;
    std::cout << "Thread "+ std::to_string(thr_id) + " started.\n";
    auto hard_start_time = std::chrono::system_clock::now();
    while (true){
        if (q->have_tasks) {
            std::pair<int, int> *task = q->pop();
            if (not task)
                continue;
            auto start = std::chrono::system_clock::now();
            matrix result_matrix = matrix(shape, shape);
            matrix *xm = data[task->first];
            matrix *ym = data[task->second];
            for (size_t i = 0; i < shape; i++)
                for (size_t j = 0; j < shape; j++) {
                    int c = 0;
                    for (size_t g = 0; g < shape; g++) {
                        int x = xm->data[i * shape + g];
                        int y = ym->data[g * shape + j];
                        c += x * y;
                    }
                    result_matrix.data[i * shape + j] = c;
                }

            for (size_t i = 0; i < shape; i++)
                for (size_t j = 0; j < shape; j++)
                    out_matrix->data[i * shape + j] += result_matrix.data[i * shape + j];

        helpfull_time += (std::chrono::system_clock::now() - start);
        } else if (q->done){
            std::chrono::duration<double> thread_live_time = std::chrono::system_clock::now()-hard_start_time;
            std::chrono::duration<double> useless_time = thread_live_time - helpfull_time;
            std::cout << "Thread "+ std::to_string(thr_id) + " done.\nwork time "
            + std::to_string(helpfull_time.count()) + "sec. ("
            + std::to_string(helpfull_time.count()/(thread_live_time.count()) * 100) + "%)\n"
            + "useless time: " + std::to_string(useless_time.count()) + " sec. ("
            + std::to_string(useless_time.count()/(thread_live_time.count()) * 100) + "%)\n";
            return;
        }
        else {
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
        }
    }
}
```

```
void round_robin(int shape, int thread_num, std::string fname, matrix &out_matrix){
    std::vector<matrix*> all_matrix;

    std::vector<std::thread> thrs;
    std::vector<block_write_queue*> tasks_queues;
    std::vector<matrix*> out_matrix_for_thrs;
    for (size_t i =0; i<thread_num; i++){
        block_write_queue *tasks = new block_write_queue;
        matrix *out_m = new matrix(shape, shape);
        tasks_queues.push_back(tasks);
        out_matrix_for_thrs.push_back(out_m);
        thrs.push_back(std::thread(calculator, tasks, &all_matrix, out_m, shape, i));
    }

    std::ifstream file(fname);
    int cur_process_id=0;
    while ( not file.eof()) {
        matrix *A = new matrix(shape, shape);
        for (size_t i = 0; i < shape; i++)
            for (size_t j = 0; j < shape; j++)
                file >> A->data[i * shape + j];

        auto matrix_id = all_matrix.size();
        all_matrix.push_back(A);
        for (size_t i = 0; i < matrix_id; i++) {
            tasks_queues[cur_process_id]->push(std::pair<int, int>(i, matrix_id));
            cur_process_id = (cur_process_id + 1) % thread_num;
            tasks_queues[cur_process_id]->push(std::pair<int, int>(matrix_id, i));
            cur_process_id = (cur_process_id + 1) % thread_num;
        }
    }
    file.close();
    for ( size_t i = 0 ;i <thread_num ; i++)
        tasks_queues[i]->done = true;

    for (size_t i =0; i<thread_num; i++) {
        thrs[i].join();
        for (size_t k = 0; k < shape; k++)
            for (size_t l = 0; l < shape; l++)
                out_matrix.data[k * shape + l] += out_matrix_for_thrs[i]->data[k * shape + l];
    }
    std::cout << "TASKS:" <<std::endl;
    for (size_t i =0; i < thread_num; i++)
        std::cout << "Thread " << i << " processed " <<tasks_queues[i]->get_processed_count() << std::endl;
}

    int get_least_loaded_id(std::vector<block_write_queue *> &tasks_queues, int thread_num) {
        int min_load = tasks_queues[0]->get_loaded();
        int id = 0;
        for (size_t i = 1; i < thread_num; i++)
            if (min_load > tasks_queues[i]->get_loaded()) {
                min_load = tasks_queues[i]->get_loaded();
                id = i;
            }
        return id;
    }
```

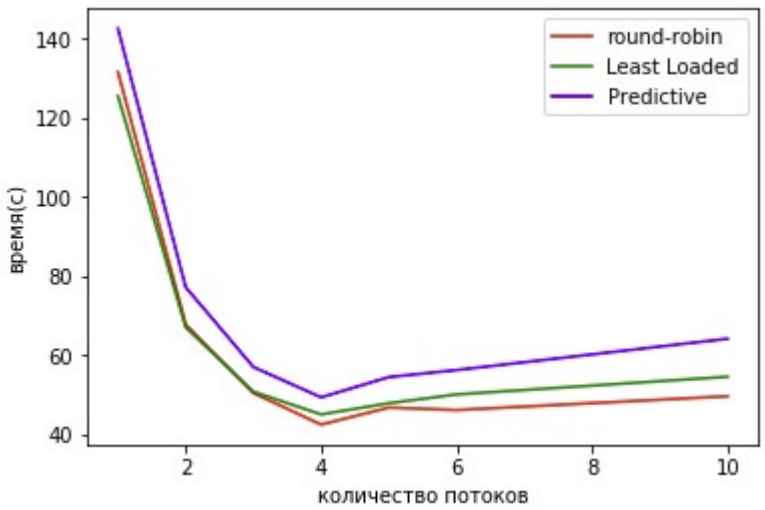
```
    void least_loaded(int shape, int thread_num, std::string fname, matrix &out_matrix) {
        std::vector<matrix *> all_matrix;

        std::vector<std::thread> thrs;
        std::vector<block_write_queue *> tasks_queues;
        std::vector<matrix *> out_matrix_for_thrs;
        for (size_t i = 0; i < thread_num; i++) {
            block_write_queue *tasks = new block_write_queue;
            matrix *out_m = new matrix(shape, shape);
            tasks_queues.push_back(tasks);
            out_matrix_for_thrs.push_back(out_m);
            thrs.push_back(std::thread(calculator, tasks, &all_matrix, out_m, shape, i));
        }

        std::ifstream file(fname);
        int cur_process_id = 0;
        while (not file.eof()) {
            matrix *A = new matrix(shape, shape);
            for (size_t i = 0; i < shape; i++)
                for (size_t j = 0; j < shape; j++)
                    file >> A->data[i * shape + j];

            auto matrix_id = all_matrix.size();
            all_matrix.push_back(A);
            for (size_t i = 0; i < matrix_id; i++) {
                cur_process_id = get_least_loaded_id(tasks_queues, thread_num);
                tasks_queues[cur_process_id]->push(std::pair<int, int>(i, matrix_id));
                cur_process_id = get_least_loaded_id(tasks_queues, thread_num);
                tasks_queues[cur_process_id]->push(std::pair<int, int>(matrix_id, i));
            }
        }
        file.close();
        for (size_t i = 0; i < thread_num; i++)
            tasks_queues[i]->done = true;

        for (size_t i = 0; i < thread_num; i++) {
            thrs[i].join();
            for (size_t k = 0; k < shape; k++)
                for (size_t l = 0; l < shape; l++)
                    out_matrix.data[k * shape + l] += out_matrix_for_thrs[i]->data[k * shape + l];
        }

        std::cout << "TASKS:" << std::endl;
        for (size_t i = 0; i < thread_num; i++)
            std::cout << "Thread " << i << " processed " << tasks_queues[i]->get_processed_count() <<
std::endl;
    }
    int get_predict_id(std::vector<block_write_queue *> &tasks_queues, int thread_num) {
        double min_predict = tasks_queues[0]->get_predict();
        int id = 0;
        for (size_t i = 1; i < thread_num; i++)
            if (min_predict > tasks_queues[i]->get_predict()) {
                min_predict = tasks_queues[i]->get_predict();
                id = i;
            }
        return id;
    }
```

```
void predictive_loaded(int shape, int thread_num, std::string fname, matrix &out_matrix) {
    std::vector<matrix *> all_matrix;

    std::vector<std::thread> thrs;
    std::vector<block_write_queue *> tasks_queues;
    std::vector<matrix *> out_matrix_for_thrs;
    for (size_t i = 0; i < thread_num; i++) {
        block_write_queue *tasks = new block_write_queue;
        matrix *out_m = new matrix(shape, shape);
        tasks_queues.push_back(tasks);
        out_matrix_for_thrs.push_back(out_m);
        thrs.push_back(std::thread(calculator, tasks, &all_matrix, out_m, shape, i));
    }

    std::ifstream file(fname);
    int cur_process_id = 0;
    while (not file.eof()) {
        matrix *A = new matrix(shape, shape);
        for (size_t i = 0; i < shape; i++)
            for (size_t j = 0; j < shape; j++)
                file >> A->data[i * shape + j];

        auto matrix_id = all_matrix.size();
        all_matrix.push_back(A);
        for (size_t i = 0; i < matrix_id; i++) {
            cur_process_id = get_predict_id(tasks_queues, thread_num);
            tasks_queues[cur_process_id]->push(std::pair<int, int>(i, matrix_id));
            cur_process_id = get_predict_id(tasks_queues, thread_num);
            tasks_queues[cur_process_id]->push(std::pair<int, int>(matrix_id, i));
        }
    }
    file.close();
    for (size_t i = 0; i < thread_num; i++)
        tasks_queues[i]->done = true;

    for (size_t i = 0; i < thread_num; i++) {
        thrs[i].join();
        for (size_t k = 0; k < shape; k++)
            for (size_t l = 0; l < shape; l++)
                out_matrix.data[k * shape + l] += out_matrix_for_thrs[i]->data[k * shape + l];
    }

    std::cout << "TASKS:" << std::endl;
    double sum_counters = 0;
    for (size_t i = 0; i < thread_num; i++)
        sum_counters += tasks_queues[i]->get_processed_count();

    for (size_t i = 0; i < thread_num; i++)
        std::cout << "Thread " << i << " processed " << tasks_queues[i]->get_processed_count() <<
        "( " << tasks_queues[i]->get_processed_count()/sum_counters *100 << "% )" << std::endl;
}
```

Полностью весь код можно найти на github:
https://github.com/medbar/study/tree/master/1sem/dis/1lab_cpp

Зависимость времени работы от количества потоков:

type	thr	time
0	1	131.5720
0	2	67.7667
0	3	50.4948
0	4	42.5160
0	5	46.8270
0	6	46.2060
0	10	49.6810
1	1	125.4930
1	2	67.0909
1	3	50.8284
1	4	45.0983
1	5	47.9192
1	6	50.1465
1	10	54.6073
2	1	142.5500
2	2	77.2238
2	3	57.0549
2	4	49.4036
2	5	54.5231
2	6	56.3022
2	10	64.2074



Вывод: В рамках данной лабораторной работы были изучены алгоритмы распределения нагрузки между потоками обработки данных и была проведена оценка их эффективность. Лучше всего показал себя самый простой round-robin так как в условии задания все таски одинаковой сложности и вычислительные ноды тоже были одинаковые, в таких условиях сложные алгоритмы балансировки не нужны.