**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МОЭВМ**

отчет

**по лабораторной работе №4**

**по дисциплине «Параллельные алгоритмы»**

Тема: Потоки в UNIX

|  |  |  |
| --- | --- | --- |
| Студент гр. 6304 |  | Ковынев М.В. |
| Преподаватель |  | Митяков А.В. |

Санкт-Петербург

2018

**Цель работы.**

Знакомство с концепцией потоков. POSIX API для потоков pthread. Алгоритм параллельной чёт-нечётной сортировки.

**Задание.**

Реализовать программу параллельной сортировки массива на основе потоков методом чёт-нечётной перестановки. Необходимо задействовать миниум 4 потока.

Начальная сортировка частей данных осущетсвляется с помощью быстрой сортировки.

Сравнить время выполнения параллельной версии с последовательной(быстрая сортировка) на разном объеме входных данных.

Сформулировать выводы об условиях применимости данного параллельного метода.

**Основные теоретические положения.**

Поточность — это создание и управление множеством исполняемых элементов внутри одного процесса. Процесс содержит один или несколько потоков, которые существуют в одном адресном пространстве процесса.

Преимущества потоков – это параллелизм, переключение контекста, экономия памяти.

POSIX Threads – единый стандарт UNIX потоков. API находится в <pthread.h>: включает в себя системные вызовы для создания, уничтожения соединения и отсоединения потоков и синхронизации.

**Ход работы.**

В алгоритм сортировки вводятся два разных правила выполнения итераций метода – в зависимости от чётности или нечётности номера итерации сортировки для обработки выбираются элементы с чётными или нечётными номерами, сравнение выделяемых значений всегда осуществляется с их правыми соседями.

При количестве используемых потоков p меньше чем количество элементов n в сортируемом массиве p<n, каждый поток содержит блоки данных размера n/p. Соответственно на чётных итерациях потоки с чётными номерами осуществляют сортировку слиянием с правым соседом, а на нечётных с нечётными номерами также со своими правыми соседями.

Пример работы для 4 потоков:

Рисунок 1 – Пример работы алгоритма

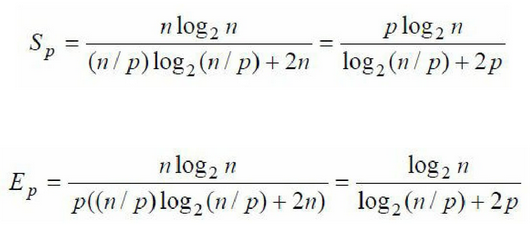
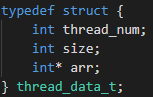
Анализ эффективности алгоритма:

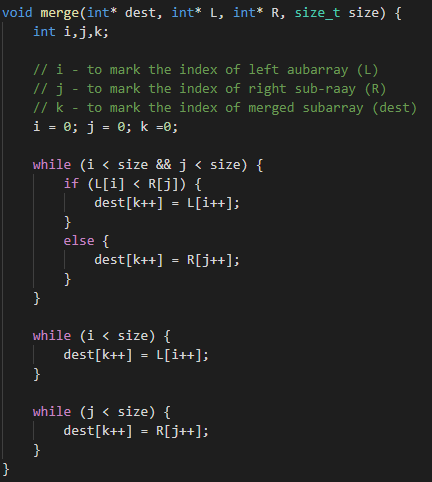
Рисунок 2 – Характеристики алгоритма

1. Определим структуру для передачи данных в каждый поток:

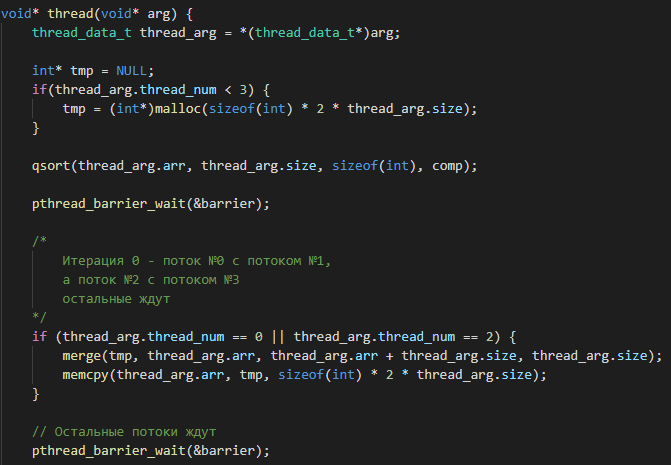
Рисунок 3 – Данные в потоке

Каждый поток знает свой номер, размер части данных за которую он отвечает и указатель на начало части данных.

1. Определим функцию слияния двух частей данных в одну:

Рисунок 4 – Функция слияния частей данных

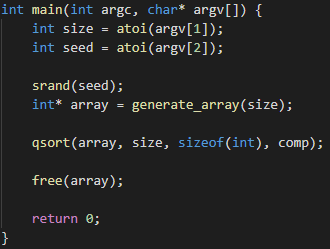
1. Определим функцию работы потока:

Рисунок 5 – Функция работы потока

Каждый поток получает на вход структуру данных содержащую необходимую информацию для работы потока. Далее каждый поток выполняет начальную сортировку своей части данных с помощью алгоритма быстрой сортировки и ждет все остальные потоки.

После, зависимости от своего номера и итерации каждый поток совершает слияние со своими правым соседом или просто ждёт остальные потоки.

1. Составим для сравнения последовательную версию сортировки. Будем использовать алгоритм быстрой сортировки для всего массива.

Рисунок 6 – Последовательная версия программы для сравнения

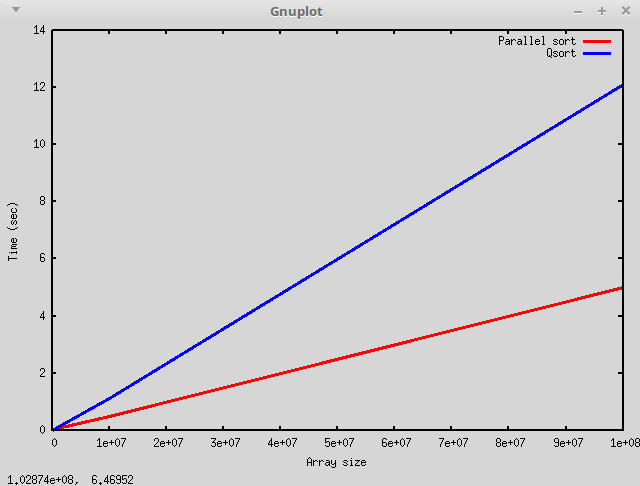
1. Составим программу для сравнения времени работы последовательной и параллельной сортировки и визуализации полученных результатов.

Рисунок 7 – Сравнения времени работы

1. Как видно из графика время работы последовательной быстрой сортировки растёт быстрее чем время работы параллельной версии, причём параллельная программа работает в два раза быстрее на массиве размером 100000000 элементов
2. Исходный код параллельной программы приведен в приложении - А, последовательной в приложении - Б, а программы для измерения времени в приложении - В.

**Выводы**

Параллельный алгоритм чёт-нечётной сортировки работает заметно быстрее чем последовательный алгоритм быстрой сортировки, однако сложнее в реализации и требует наличия нескольких ядер в процессоре и большого объема входных данных.

Приложение А

parallel.C

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

#include <string.h>

#define THREAD\_NUM 4

int comp (const void\* x, const void\* y) {

return \*((int\*)x) - \*((int\*)y);

}

int\* generate\_array(size\_t size) {

int\* array = (int\*)malloc(sizeof(int) \* size);

for (int i = 0; i < size; i++) {

array[i] = -99 + rand() % 199;

}

return array;

}

void merge(int\* dest, int\* L, int\* R, size\_t size) {

int i,j,k;

// i - to mark the index of left aubarray (L)

// j - to mark the index of right sub-raay (R)

// k - to mark the index of merged subarray (dest)

i = 0; j = 0; k =0;

while (i < size && j < size) {

if (L[i] < R[j]) {

dest[k++] = L[i++];

}

else {

dest[k++] = R[j++];

}

}

while (i < size) {

dest[k++] = L[i++];

}

while (j < size) {

dest[k++] = R[j++];

}

}

typedef struct {

int thread\_num;

int size;

int\* arr;

} thread\_data\_t;

pthread\_barrier\_t barrier;

void\* thread(void\* arg) {

thread\_data\_t thread\_arg = \*(thread\_data\_t\*)arg;

int\* tmp = NULL;

if(thread\_arg.thread\_num < 3) {

tmp = (int\*)malloc(sizeof(int) \* 2 \* thread\_arg.size);

}

qsort(thread\_arg.arr, thread\_arg.size, sizeof(int), comp);

pthread\_barrier\_wait(&barrier);

/\*

Итерация 0 - поток №0 с потоком №1,

а поток №2 с потоком №3

остальные ждут

\*/

if (thread\_arg.thread\_num == 0 || thread\_arg.thread\_num == 2) {

merge(tmp, thread\_arg.arr, thread\_arg.arr + thread\_arg.size, thread\_arg.size);

memcpy(thread\_arg.arr, tmp, sizeof(int) \* 2 \* thread\_arg.size);

}

// Остальные потоки ждут

pthread\_barrier\_wait(&barrier);

/\*

Итерация 1 - поток №1 с потоком №2,

остальные ждут

\*/

if (thread\_arg.thread\_num == 1) {

merge(tmp, thread\_arg.arr, thread\_arg.arr + thread\_arg.size, thread\_arg.size);

memcpy(thread\_arg.arr, tmp, sizeof(int) \* 2 \* thread\_arg.size);

}

// Остальные потоки ждут

pthread\_barrier\_wait(&barrier);

/\*

Итерация 2 - поток №0 с потоком №1,

а поток №2 с потоком №3

остальные ждут

\*/

if (thread\_arg.thread\_num == 0 || thread\_arg.thread\_num == 2) {

merge(tmp, thread\_arg.arr, thread\_arg.arr + thread\_arg.size, thread\_arg.size);

memcpy(thread\_arg.arr, tmp, sizeof(int) \* 2 \* thread\_arg.size);

}

// Остальные потоки ждут

pthread\_barrier\_wait(&barrier);

/\*

Итерация 3 - поток №1 с потоком №2,

остальные ждут

\*/

if (thread\_arg.thread\_num == 1) {

merge(tmp, thread\_arg.arr, thread\_arg.arr + thread\_arg.size, thread\_arg.size);

memcpy(thread\_arg.arr, tmp, sizeof(int) \* 2 \* thread\_arg.size);

}

// Остальные потоки ждут

pthread\_barrier\_wait(&barrier);

if (tmp) {

free(tmp);

}

return NULL;

}

int main(int argc, char\* argv[]) {

pthread\_barrier\_init(&barrier, NULL, THREAD\_NUM);

pthread\_t ids[THREAD\_NUM];

thread\_data\_t args[THREAD\_NUM];

int size = atoi(argv[1]);

int seed = atoi(argv[2]);

srand(seed);

int\* array = generate\_array(size);

int size\_per\_thread = size / THREAD\_NUM;

for (int i = 0; i < THREAD\_NUM; i++) {

args[i].thread\_num = i;

args[i].size = size\_per\_thread;

args[i].arr = array + i \* size\_per\_thread;

pthread\_create(&ids[i], NULL, thread, &args[i]);

}

for (int i=0; i < THREAD\_NUM; i++) {

pthread\_join(ids[i], NULL);

}

free(array);

return 0;

}

Приложение Б

qsort.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

#include <string.h>

int\* generate\_array(size\_t size) {

int\* array = (int\*)malloc(sizeof(int) \* size);

for (int i = 0; i < size; i++) {

array[i] = -9 + rand() % 19; // [-9, 9]

}

return array;

}

void print\_array(int\* array, size\_t size) {

for (int i = 0; i < size - 1; i++) {

printf("%d, ", array[i]);

}

printf("%d\n", array[size - 1]);

}

int comp (const void\* a, const void\* b) {

return \*((int\*)a) - \*((int\*)b);

}

int main(int argc, char\* argv[]) {

int size = atoi(argv[1]);

int seed = atoi(argv[2]);

srand(seed);

int\* array = generate\_array(size);

qsort(array, size, sizeof(int), comp);

free(array);

return 0;

}Приложение В

**RUN.SH**

#!/bin/bash

gcc $1 -pthread -o merge

gcc $2 -o qsort

echo -e "# Array size\tTime" > plot1.txt

echo -e "# Array size\tTime" > plot2.txt

for (( i=5; i<=8; i++))

do

current\_data=`dc -e "10 ${i} ^ p"`

echo -n "Sort ${current\_data} array. Parallel sort. Time: "

(time ./merge "${current\_data}" 123) &> file.txt

echo -ne "${current\_data}\t" >> plot1.txt

time=`head -2 file.txt | grep -o -P '(?<=m).\*(?=s)'`

echo "${time}"

echo "${time}" >> plot1.txt

echo -n "Sort ${current\_data} array. QSort. Time: "

(time ./qsort "${current\_data}" 123) &> file.txt

echo -ne "${current\_data}\t" >> plot2.txt

time=`head -2 file.txt | grep -o -P '(?<=m).\*(?=s)'`

echo "${time}"

echo "${time}" >> plot2.txt

done

rm -f file.txt merge qsort

gnuplot -e "set ylabel \"Time (sec)\"; set xlabel \

\"Array size\"; plot \"plot1.txt\" \

using 1:2 with lines lt rgb \"red\" lw 3 t \"Parallel sort\", \

\"plot2.txt\" using 1:2 with lines lt rgb \"blue\" lw 3 t \"Qsort\"; pause -1"