**МИНОБРНАУКИ РОССИИ**

**федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«Балтийский государственный технический университет «ВОЕНМЕХ» им. Д.Ф. Устинова»**

**(БГТУ «ВОЕНМЕХ» им. Д.Ф. Устинова)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кафедра |  | О7 |  | Программная инженерия |
|  |  | шифр |  | наименование кафедры, по которой выполняется работа |
| Дисциплина |  | Параллельные вычисления | | |
|  |  | наименование дисциплины | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ПРАКТИЧЕСКАЯ РАБОТА | | | | |
|  | 2 |  |
|  | номер задания (при наличии) |  |
| ИСПОЛЬЗОВАНИЕ БИБЛИОТЕКИ OPENMP | | | |
|  | | | |

при наличии указать тему учебно-практической работы и (или) номер варианта

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ОБУЧАЮЩИЙСЯ** | | | | | |
| группы | | | |  | О726Б |
|  |  | Махов Н.М. | | | |
| подпись |  | фамилия и инициалы | | | |
|  | | | | |
| дата сдачи | | | | |
| **ПРОВЕРИЛ** | | | | | |
| Преподаватель | | | | | | |
| ученая степень, ученое звание, должность | | | | | | |
|  |  | Зимина Д.В. | | | | |
| подпись |  | фамилия и инициалы | | | | |
| Оценка / балльная оценка | | |  | | |
|  | | | | |
| дата проверки | | | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Санкт-Петербург | | | | | | |
|  |  |  | 20 | 25 | г. |  |

Часть 1

1-4 Текст программы:

#include <stdio.h>  
#include <stdlib.h>  
#include <sys/time.h>  
#include <time.h>  
#include <math.h>  
#ifdef \_OPENMP  
#include <omp.h>  
#endif  
int main(int argc, char\* argv[]) {  
#ifdef \_OPENMP  
printf("openmp is init/n");  
#endif  
 int i, N, A;  
 struct timeval T1, T2;  
 long delta\_ms;  
 N = atoi(argv[1]);  
 A = atoi(argv[2]);  
  
 gettimeofday(&T1, NULL);  
  
 //Этап 1  
 double \*M1 = malloc(sizeof(double)\* N);  
 double \*M2 = malloc(sizeof(double)\* N/2);  
 unsigned int seed = time(NULL);  
  
#ifdef \_OPENMP  
 omp\_set\_num\_threads(atoi(argv[3]));  
#endif  
 for (i=0; i<100; i++) {  
 srand (i);  
  
 // Заполнение M1  
 for (int i = 0; i < N; i++) {  
 M1[i] = 1.0 + (A - 1.0) \* ((double)rand\_r(&seed) / RAND\_MAX);  
 }  
 // Заполнение M2  
 for (int i = 0; i < N / 2; i++) {  
 M2[i] = A + (10 \* A - A) \* ((double)rand\_r(&seed) / RAND\_MAX);  
 }  
#ifdef \_OPENMP  
#pragma omp parallel for default(none) private(i) shared(N, M1)  
#endif  
 // Этап 2: Map (гиперболический синус в квадрат)  
 for (int i = 0; i < N; i++) {  
 M1[i] = pow(sinh(M1[i]), 2);  
 }  
 // Создаем копию M2  
 double M2\_copy[N / 2];  
#ifdef \_OPENMP  
#pragma omp parallel for default(none) private(i) shared(N, M2\_copy, M2)  
#endif  
 for (int i = 0; i < N / 2; i++) {  
 M2\_copy[i] = M2[i];  
 }  
#ifdef \_OPENMP  
#pragma omp parallel for default(none) private(i) shared(N, M2, M2\_copy)  
#endif  
 // Этап 2: Map для M2 (кумулятивная сумма + sqrt(e \* сумма))  
 for (int i = 0; i < N / 2; i++) {  
 double prev = (i == 0) ? 0 : M2\_copy[i - 1]; // Предыдущий элемент или 0  
 M2[i] = sqrt(M\_E \* (M2\_copy[i] + prev));  
 }  
#ifdef \_OPENMP  
#pragma omp parallel for default(none) private(i) shared(M2, N, M1)  
#endif  
 // Этап 3: Merge (умножение элементов M2 и M1 с одинаковыми индексами)  
 for (int i = 0; i < N / 2; i++) {  
 M2[i] = M2[i] \* M1[i]; // Умножение  
 }  
 // Этап 4: Sort (сортировка вставками для M2)  
 for (int i = 1; i < N / 2; i++) {  
 double key = M2[i];  
 int j = i - 1;  
  
 // Сдвиг элементов массива, которые больше ключа  
 while (j >= 0 && M2[j] > key) {  
 M2[j + 1] = M2[j];  
 j = j - 1;  
 }  
  
 M2[j + 1] = key;  
 }  
 // Этап 5: Reduce (расчет суммы синусов для определенных элементов)  
 double min\_value = M2[0];  
#ifdef \_OPENMP  
#pragma omp parallel for default(none) private(i) shared(min\_value, N, M2)  
#endif  
 // Поиск минимального ненулевого элемента в M2  
 for (int i = 1; i < N / 2; i++) {  
 if (M2[i] != 0 && M2[i] < min\_value) {  
 min\_value = M2[i];  
 }  
 }  
  
 // Расчет суммы синусов  
 double X = 0.0;  
#ifdef \_OPENMP  
#pragma omp parallel for default(none) private(i) shared(X, N, min\_value, M2)  
#endif  
 for (int i = 0; i < N / 2; i++) {  
 // Проверка, делится ли M2[i] на минимальный элемент с четной целой частью  
 if ((int)(M2[i] / min\_value) % 2 == 0) {  
 X += sin(M2[i]);  
 }  
 }  
 }  
 //printf("\nCalculated value X: %lf\n", support\_buf);  
 free(M1); free(M2);  
 gettimeofday(&T2, NULL);  
 delta\_ms = (T2.tv\_sec - T1.tv\_sec) \* 1000 +  
 (T2.tv\_usec - T1.tv\_usec) / 1000;  
 printf ("\nN=%d. Milliseconds passed: %ld\n", N, delta\_ms);  
 return 0;  
}

Bash для запуска и компиляции:

N1=650

N2=36000

step=(36000-650)/10

gcc -O3 -Wall -Werror -o lab2-seq lab2.c -lm

gcc -O3 -Wall -Werror -fopenmp lab2.c -o lab2-par-1 -lm

gcc -O3 -Wall -Werror -fopenmp lab2.c -o lab2-par-2 -lm

gcc -O3 -Wall -Werror -fopenmp lab2.c -o lab2-par-4 -lm

gcc -O3 -Wall -Werror -fopenmp lab2.c -o lab2-par-6 -lm

for (( i=N1; i < N2+1; i+=step ))

do

./lab2-seq $i 20

done

echo -e "\n\nnow parallel:"

for N in 1 2 4 6

do

echo "$N thread(s)"

for (( i=N1; i < N2+1; i+=step ))

do

./lab2-par-$N $i 20 $N

done

done

exit 0

Bash для повторного тестирования

#!/bin/bash

gcc -Wall -Werror -o lab1fun lab1.c -lm

gcc -O3 -Wall -Werror -lm -floop-parallelize-all -ftree-parallelize-loops=2 lab1.c -o lab1-par-2-fun -lm

gcc -O3 -Wall -Werror -lm -floop-parallelize-all -ftree-parallelize-loops=4 lab1.c -o lab1-par-4-fun -lm

gcc -O3 -Wall -Werror -lm -floop-parallelize-all -ftree-parallelize-loops=10 lab1.c -o lab1-par-10-fun -lm

echo -e '\nsequential computation:'

./lab1fun 100000 20

./lab1fun 200000 20

./lab1fun 400000 20

./lab1fun 800000 20

./lab1fun 1000000 20

./lab1fun 1300000 20

./lab1fun 1600000 20

echo -e '\nconcurrent computing:'

echo 'threads 2:'

./lab1-par-2-fun 100000 20

./lab1-par-2-fun 200000 20

./lab1-par-2-fun 400000 20

./lab1-par-2-fun 800000 20

./lab1-par-2-fun 1000000 20

./lab1-par-2-fun 1300000 20

./lab1-par-2-fun 1600000 20

echo 'threads 4:'

./lab1-par-4-fun 100000 20

./lab1-par-4-fun 200000 20

./lab1-par-4-fun 400000 20

./lab1-par-4-fun 800000 20

./lab1-par-4-fun 1000000 20

./lab1-par-4-fun 1300000 20

./lab1-par-4-fun 1600000 20

echo 'threads 10:'

./lab1-par-10-fun 100000 20

./lab1-par-10-fun 200000 20

./lab1-par-10-fun 400000 20

./lab1-par-10-fun 800000 20

./lab1-par-10-fun 1000000 20

./lab1-par-10-fun 1300000 20

./lab1-par-10-fun 1600000 20

exit 0

Анализ данных

В программе добавлено подключение библиотеки OpenMP, реализована обратная совместимость, при отсутствии модификатора -fopenmp. Из-за сортировки вставками, распараллеливание вновь не удалось, так как при достижении цикла сортировки, поток, который в него попадает задерживает все остальные, как итог получаем результат хуже чем последовательном, так как часть ресурсов требуется на поддержание многопоточности.

Ниже приведена таблица затрат времени на вычисление при разном N: таблица 1 - результаты автоматического распараллеливания, таблица 2 - результаты распараллеливания с помощью библиотеки OpenMP. Диаграмма 1 соответствует таблице 1.

Таблица 1 - Результаты автоматического распараллеливания (Из практической 1).

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| time \ N | 650 | 4185 | 7720 | 11255 | 14790 | 18325 | 21860 | 25395 | 28930 | 32465 | 36000 |
| 1 поток | 4 | 84 | 256 | 532 | 907 | 1384 | 1962 | 2642 | 3401 | 4275 | 5294 |
| 2 потока | 4 | 91 | 265 | 540 | 918 | 1395 | 1967 | 2648 | 3479 | 4386 | 5417 |
| 4 потока | 3 | 96 | 269 | 545 | 927 | 1400 | 1980 | 2676 | 3458 | 4380 | 5502 |
| 6 потоков | 4 | 96 | 272 | 549 | 934 | 1406 | 2095 | 2733 | 3458 | 4353 | 5351 |
| seq | 11 | 111 | 260 | 535 | 912 | 1388 | 1964 | 2636 | 3416 | 4297 | 5294 |

Таблица 2 - Результаты распараллеливания c помощью библиотеки OpenMP.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 650 | 4185 | 7720 | 11255 | 14790 | 18325 | 21860 | 25395 | 28930 | 32465 | 36000 |
| Последовательно | 3 | 86 | 262 | 533 | 907 | 1379 | 1953 | 2624 | 3403 | 4276 | 5262 |
| 1 поток | 4 | 88 | 264 | 540 | 915 | 1389 | 1970 | 2647 | 3467 | 4298 | 5293 |
| 2 потока | 4 | 93 | 262 | 540 | 916 | 1393 | 1963 | 2632 | 3406 | 4289 | 5279 |
| 4 потока | 13 | 98 | 265 | 541 | 922 | 1394 | 1966 | 2634 | 3429 | 4281 | 5439 |
| 6 потоков | 94 | 163 | 338 | 611 | 988 | 1534 | 2074 | 2740 | 4058 | 4480 | 5552 |

Диаграмма 1 - Результат автоматического распараллеливания

Диаграмма 2 - Результат распараллеливания с помощью OpenMP

В рамках эксперимента исключим цикл сортировки.

Пересоберем файлы с помощью:

gcc -O3 -Wall -Werror -o lab2fun lab2.c -lm | gcc -O3 -Wall -Werror -fopenmp lab2.c -o lab2-par-1-fun -lm | gcc -O3 -Wall -Werror -fopenmp lab2.c -o lab2-par-2-fun -lm | gcc -O3 -Wall -Werror -fopenmp lab2.c -o lab2-par-4-fun -lm | gcc -O3 -Wall -Werror -fopenmp lab2.c -o lab2-par-10-fun -lm

Bash:

GNU nano 7.2 test\_no\_sort.sh

#!/bin/bash

echo -e '\nsequential computation:'

./lab2fun 100000 20

./lab2fun 200000 20

./lab2fun 400000 20

./lab2fun 800000 20

./lab2fun 1000000 20

./lab2fun 1300000 20

./lab2fun 1600000 20

echo -e '\nconcurrent computing:'

echo 'threads 2:'

./lab2-par-2-fun 100000 20 2

./lab2-par-2-fun 200000 20 2

./lab2-par-2-fun 400000 20 2

./lab2-par-2-fun 800000 20 2

./lab2-par-2-fun 1000000 20 2

./lab2-par-2-fun 1300000 20 2

./lab2-par-2-fun 1600000 20 2

echo 'threads 4:'

./lab2-par-4-fun 100000 20 4

./lab2-par-4-fun 200000 20 4

./lab2-par-4-fun 400000 20 4

./lab2-par-4-fun 800000 20 4

./lab2-par-4-fun 1000000 20 4

./lab2-par-4-fun 1300000 20 4

./lab2-par-4-fun 1600000 20 4

echo 'threads 10:'

./lab2-par-10-fun 100000 20 10

./lab2-par-10-fun 200000 20 10

./lab2-par-10-fun 400000 20 10

./lab2-par-10-fun 800000 20 10

./lab2-par-10-fun 1000000 20 10

exit 0

Таблица 3 - Результат при автоматическом распараллеливании

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| time \ N | 100000 | 200000 | 400000 | 800000 | 1000000 | 1300000 | 1600000 |
| Последовательно | 198 | 395 | 784 | 1567 | 1957 | 2542 | 3139 |
| 2 потока | 210 | 403 | 801 | 1599 | 1995 | 2595 | 3196 |
| 4 потока | 218 | 410 | 804 | 1595 | 1994 | 2600 | 3257 |
| 10 потоков | 224 | 419 | 822 | 1605 | 2034 | 2601 | 3209 |

Таблица 4 - Результат при распараллеливании при помощи OpenMP

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| time \ N | 100000 | 200000 | 400000 | 800000 | 1000000 | 1300000 | 1600000 |
| Последовательно | 222 | 393 | 789 | 1580 | 1965 | 2556 | 3149 |
| 2 потока | 187 | 364 | 703 | 1401 | 1754 | 2307 | 2822 |
| 4 потока | 164 | 277 | 548 | 1070 | 1334 | 1720 | 2144 |
| 10 потоков | 258 | 396 | 668 | 1210 | 1491 | 1916 | 2349 |

Диаграмма 1 - Зависимость скорости вычисления от размера массива для разного количества потоков при автоматическом распараллеливании без сортировки

Диаграмма 2 - Зависимость скорости вычисления от размера массива для разного количества потоков при использовании OpenMP без сортировки

Как можно увидеть распараллеливание программы с помощью OpenMP дало результат, это происходит по той причине, что попадая в цикл сортировки поток вычисляет его и замораживает активность других потоков, как итог даже при использовании 4 потоков результат эквивалентен последовательному вычислению, потому что большую часть времени работает именно один поток, и весь временной выигрыш полученный другими потокам просто нивелируется. Доказательство на рисунке 1.

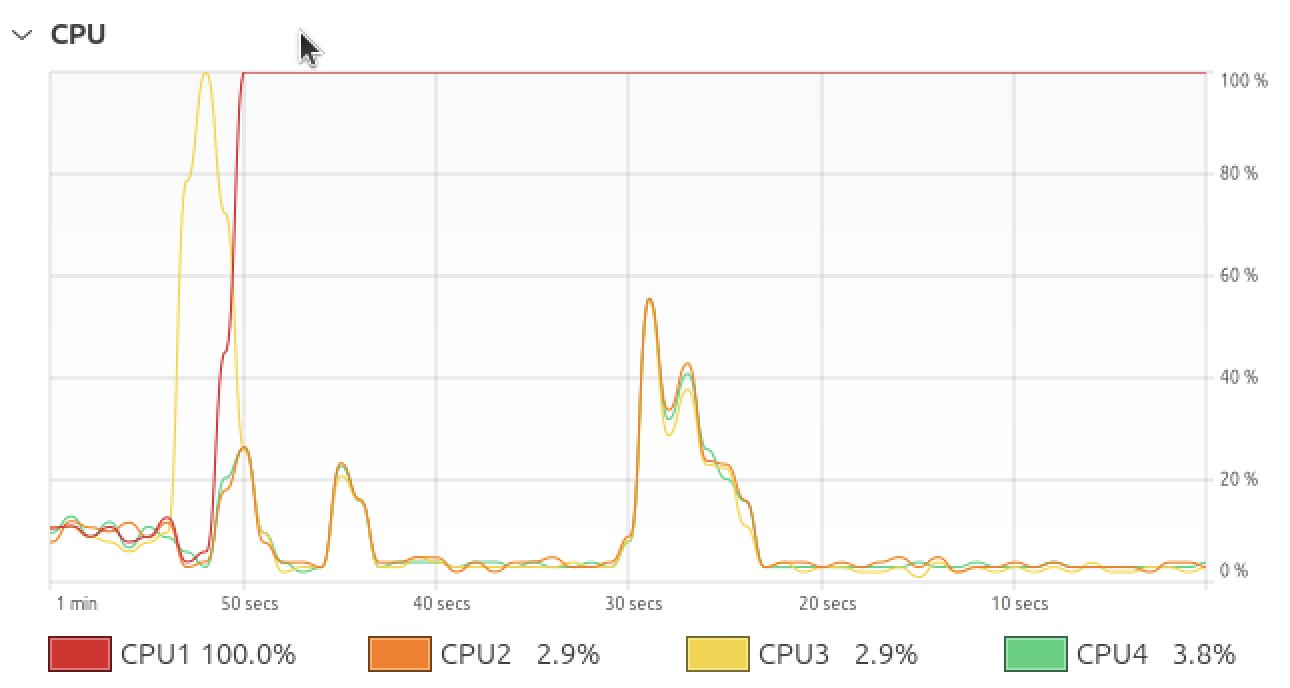


Рисунок 1 - Выполнение программы с О(n2) сложностью при распараллеливании (ядро 1 загружено на 100%)

Как только сортировка была удалена из программы (этап 5 пропущен) «узкое горлышко» программы было устранено и теперь многопоточность полноценно работает. Стоит отметить, что использование 2-ух, 4-ех поток дало лучший результат, в том время как 10 потоков, лишь немного быстрее чем последовательно, это связано с тем, что для поддержания потоков требуется ресурс.

Также можно заметить, что квадратическая зависимость сменилась на линейную, это доказывает, что именно сортировка вставками, которая имеет O(n2) сложность вычисления, оказывала серьезное влияние на время выполнения программы.

Добавим schedule: static, dynamic, guided.

…

char\* schedule\_type = argv[4];  
 int chunk\_size = atoi(argv[5]);  
  
#ifdef \_OPENMP  
 omp\_sched\_t schedule\_kind;  
  
 omp\_set\_num\_threads(atoi(argv[3]));  
  
 if (strcmp(schedule\_type, "static") == 0) {  
 schedule\_kind = omp\_sched\_static;  
 } else if (strcmp(schedule\_type, "dynamic") == 0) {  
 schedule\_kind = omp\_sched\_dynamic;  
 } else if (strcmp(schedule\_type, "guided") == 0) {  
 schedule\_kind = omp\_sched\_guided;  
 } else {  
 fprintf(stderr, "Ошибка: Неверный тип schedule: %s\n", schedule\_type);  
 return 1;  
 }  
 omp\_set\_schedule(schedule\_kind, chunk\_size);  
#endif

…

Bash для вывода

#!/bin/bash

gcc -O3 -Wall -Werror -fopenmp program.c -o program -lm

N1=650

N2=36000

step=$(( (N2 - N1) / 10 ))

A=20

schedules=("static" "dynamic" "guided")

threads=(1 2 4 10)

chunk\_sizes=(1 2 4 16)

echo "N,Threads,Schedule,Chunk,Time(ms)" > results.txt

for sched in "${schedules[@]}"; do

for t in "${threads[@]}"; do

for chunk in "${chunk\_sizes[@]}"; do

for ((N=N1; N <= N2; N+=step)); do

echo "Testing N=$N, Threads=$t, Schedule=$sched, Chunk=$chunk"

output=$(./program $N $A $t $sched $chunk)

time\_ms=$(echo "$output" | grep "Milliseconds passed" | awk '{print $4}')

echo "$N,$t,$sched,$chunk,$time\_ms" >> results.txt

done

done

done

done

echo "Результаты в results.txt"

Таблица 5 - Время выполнения программы для static schedule

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Threads** | **Chunk** | **650** | **4185** | **7720** | **11255** | **14790** | **18325** | **21860** | **25395** | **28930** | **32465** | **36000** |
| 1 | 1 | 5 | 85 | 262 | 544 | 919 | 1390 | 1980 | 2635 | 3401 | 4287 | 5285 |
| 1 | 2 | 4 | 85 | 258 | 535 | 910 | 1385 | 1960 | 2627 | 3407 | 4281 | 5273 |
| 1 | 4 | 4 | 86 | 274 | 537 | 909 | 1383 | 1961 | 2624 | 3408 | 4282 | 5260 |
| 1 | 16 | 4 | 85 | 260 | 538 | 910 | 1379 | 1959 | 2630 | 3404 | 4280 | 5260 |
| 2 | 1 | 7 | 94 | 271 | 550 | 928 | 1403 | 1978 | 2652 | 3429 | 4303 | 5295 |
| 2 | 2 | 5 | 92 | 267 | 547 | 924 | 1395 | 1970 | 2642 | 3422 | 4366 | 5277 |
| 2 | 4 | 4 | 91 | 266 | 543 | 923 | 1390 | 1971 | 2646 | 3413 | 4298 | 5301 |
| 2 | 16 | 4 | 91 | 265 | 544 | 925 | 1394 | 1985 | 2677 | 3424 | 4302 | 5294 |
| 4 | 1 | 10 | 107 | 278 | 555 | 937 | 1422 | 2006 | 2666 | 3454 | 4329 | 5312 |
| 4 | 2 | 9 | 107 | 272 | 554 | 936 | 1413 | 1985 | 2667 | 3440 | 4323 | 5301 |
| 4 | 4 | 10 | 113 | 271 | 554 | 931 | 1410 | 1982 | 2655 | 3439 | 4312 | 5320 |
| 4 | 16 | 9 | 107 | 268 | 549 | 930 | 1408 | 1982 | 2644 | 3426 | 4309 | 5295 |
| 10 | 1 | 101 | 174 | 354 | 630 | 1009 | 1490 | 2065 | 2740 | 3518 | 4402 | 5410 |
| 10 | 2 | 99 | 173 | 350 | 629 | 1004 | 1478 | 2061 | 2755 | 3544 | 4410 | 5417 |
| 10 | 4 | 101 | 178 | 349 | 631 | 1004 | 1483 | 2117 | 2738 | 3513 | 4398 | 5377 |
| 10 | 16 | 95 | 169 | 342 | 616 | 995 | 1469 | 2049 | 2718 | 3507 | 4378 | 5368 |

Таблица 6 - Время выполнения программы для dynamic schedule

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Threads** | **Chunk** | **650** | **4185** | **7720** | **11255** | **14790** | **18325** | **21860** | **25395** | **28930** | **32465** | **36000** |
| 1 | 1 | 6 | 98 | 291 | 581 | 972 | 1462 | 2050 | 2749 | 3541 | 4428 | 5436 |
| 1 | 2 | 5 | 89 | 270 | 556 | 938 | 1422 | 2002 | 2683 | 3470 | 4375 | 5347 |
| 1 | 4 | 4 | 85 | 264 | 543 | 923 | 1399 | 1983 | 2664 | 3441 | 4558 | 5379 |
| 1 | 16 | 5 | 88 | 266 | 546 | 929 | 1418 | 1990 | 2673 | 3455 | 4340 | 5330 |
| 2 | 1 | 14 | 134 | 346 | 662 | 1087 | 1595 | 2218 | 2933 | 3752 | 4679 | 5727 |
| 2 | 2 | 12 | 110 | 304 | 600 | 994 | 1488 | 2083 | 2789 | 3592 | 4492 | 5520 |
| 2 | 4 | 7 | 103 | 284 | 575 | 958 | 1446 | 2039 | 2729 | 3517 | 4400 | 5415 |
| 2 | 16 | 5 | 93 | 270 | 550 | 933 | 1409 | 1988 | 2668 | 3461 | 4340 | 5338 |
| 4 | 1 | 17 | 141 | 345 | 654 | 1075 | 1589 | 2206 | 2921 | 3745 | 4675 | 5706 |
| 4 | 2 | 12 | 120 | 308 | 599 | 999 | 1531 | 2080 | 2785 | 3587 | 4485 | 5509 |
| 4 | 4 | 12 | 123 | 289 | 569 | 963 | 1451 | 2038 | 2723 | 3521 | 4411 | 5483 |
| 4 | 16 | 9 | 112 | 275 | 553 | 947 | 1419 | 2003 | 2685 | 3472 | 4399 | 5360 |
| 10 | 1 | 92 | 189 | 381 | 692 | 1106 | 1626 | 2254 | 2955 | 3779 | 4708 | 5747 |
| 10 | 2 | 98 | 176 | 362 | 645 | 1031 | 1530 | 2118 | 2824 | 3616 | 4529 | 5539 |
| 10 | 4 | 84 | 179 | 350 | 629 | 1008 | 1503 | 2082 | 2774 | 3568 | 4451 | 5465 |
| 10 | 16 | 101 | 173 | 350 | 628 | 1002 | 1488 | 2063 | 2752 | 3534 | 4438 | 5432 |

Таблица 7 - Время выполнения программы для guided schedule

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Threads** | **Chunk** | **650** | **4185** | **7720** | **11255** | **14790** | **18325** | **21860** | **25395** | **28930** | **32465** | **36000** |
| 1 | 1 | 4 | 86 | 264 | 544 | 928 | 1409 | 2031 | 2711 | 3493 | 4346 | 5349 |
| 1 | 2 | 4 | 83 | 261 | 543 | 922 | 1403 | 1985 | 2669 | 3456 | 4336 | 5339 |
| 1 | 4 | 4 | 84 | 262 | 542 | 925 | 1402 | 1983 | 2668 | 3452 | 4335 | 5334 |
| 1 | 16 | 4 | 84 | 261 | 543 | 921 | 1400 | 1981 | 2661 | 3447 | 4361 | 5672 |
| 2 | 1 | 6 | 93 | 269 | 546 | 926 | 1406 | 1989 | 2675 | 3453 | 4351 | 5340 |
| 2 | 2 | 4 | 93 | 267 | 546 | 928 | 1402 | 2006 | 2671 | 3452 | 4344 | 5353 |
| 2 | 4 | 5 | 91 | 266 | 546 | 923 | 1408 | 1984 | 2667 | 3452 | 4338 | 5328 |
| 2 | 16 | 4 | 91 | 266 | 544 | 924 | 1407 | 1985 | 2659 | 3455 | 4338 | 5336 |
| 4 | 1 | 10 | 104 | 272 | 549 | 938 | 1412 | 1993 | 2671 | 3459 | 4336 | 5351 |
| 4 | 2 | 10 | 106 | 271 | 549 | 932 | 1411 | 1988 | 2674 | 3460 | 4340 | 5372 |
| 4 | 4 | 9 | 106 | 270 | 551 | 933 | 1410 | 1992 | 2724 | 3490 | 4340 | 5339 |
| 4 | 16 | 10 | 108 | 270 | 551 | 940 | 1413 | 2000 | 2675 | 3473 | 4418 | 5352 |
| 10 | 1 | 101 | 170 | 345 | 624 | 998 | 1479 | 2062 | 2739 | 3524 | 4472 | 5462 |
| 10 | 2 | 102 | 171 | 348 | 619 | 1007 | 1478 | 2066 | 2809 | 3546 | 4415 | 5462 |
| 10 | 4 | 103 | 170 | 346 | 621 | 999 | 1483 | 2059 | 2749 | 3526 | 4431 | 5421 |
| 10 | 16 | 103 | 173 | 346 | 620 | 1000 | 1484 | 2056 | 2740 | 3524 | 4406 | 5414 |

Использование schedule не дало выигрыша времени выполнения программы, для получения более точных данных нужно исключить сортировку или оптимизировать ее.

**Часть 2 Задание 1 - 3**

Сортировка вставками была распараллелена с помощью omp sections

Код измененного 4 этапа, сортировка массива M2 вставками:  
…

// Этап 4: Sort (сортировка вставками для M2) // распараллеленная  
#ifdef \_OPENMP  
 #pragma omp parallel sections  
 {  
#pragma omp section  
 {  
 // (M2[0:N/4])  
 for (int i = 1; i < N / 4; i++) {  
 double key = M2[i];  
 int j = i - 1;  
 while (j >= 0 && M2[j] > key) {  
 M2[j + 1] = M2[j];  
 j = j - 1;  
 }  
 M2[j + 1] = key;  
 }  
 }  
#pragma omp section  
 {  
 // (M2[N/4:N/2])  
 for (int i = N / 4; i < N / 2; i++) {  
 double key = M2[i];  
 int j = i - 1;  
 while (j >= N / 4 && M2[j] > key) {  
 M2[j + 1] = M2[j];  
 j = j - 1;  
 }  
 M2[j + 1] = key;  
 }  
 }  
 }  
 // Однопоточное слияние  
 double merged[N / 2];  
 int left = 0, right = N / 4, k = 0;  
  
 while (left < N / 4 && right < N / 2) {  
 if (M2[left] < M2[right])  
 merged[k++] = M2[left++];  
 else  
 merged[k++] = M2[right++];  
 }  
 while (left < N / 4)  
 merged[k++] = M2[left++];  
 while (right < N / 2)  
 merged[k++] = M2[right++];  
 memcpy(M2, merged, sizeof(double) \* (N / 2)); // Копируем обратно  
#endif

…

Директивы и переменные для подсчета времени были удалены и заменены:

Код программы:

double start\_time, end\_time;

…

#ifdef \_OPENMP  
 start\_time = omp\_get\_wtime();  
#endif

…

основной цикл программы

…

ifdef \_OPENMP  
 end\_time = omp\_get\_wtime();  
 printf("\nN=%d. Milliseconds passed: %lf\n", N, (end\_time - start\_time) \* 1000);  
#endif

…

После внесенных изменений, время выполнения программы значительно уменьшилось. Результаты представлены в таблице 8-10.

Таблица 8 - Время выполнения программы для static schedule

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Threads** | **Chunk** | **650** | **4185** | **7720** | **11255** | **14790** | **18325** | **21860** | **25395** | **28930** | **32465** | **36000** |
| 1 | 1 | 4 | 52 | 146 | 288 | 478 | 721 | 1013 | 1351 | 1746 | 2186 | 2672 |
| 1 | 2 | 4 | 54 | 144 | 288 | 479 | 721 | 1015 | 1352 | 1923 | 2271 | 2687 |
| 1 | 4 | 3 | 50 | 143 | 286 | 479 | 723 | 1071 | 1389 | 1848 | 2435 | 2737 |
| 1 | 16 | 3 | 51 | 145 | 289 | 481 | 729 | 1024 | 1399 | 1757 | 2194 | 2690 |
| 2 | 1 | 4 | 38 | 91 | 171 | 285 | 418 | 582 | 796 | 1227 | 1211 | 1472 |
| 2 | 2 | 3 | 34 | 88 | 168 | 275 | 408 | 567 | 749 | 962 | 1194 | 1456 |
| 2 | 4 | 3 | 35 | 87 | 165 | 275 | 405 | 562 | 747 | 954 | 1193 | 1451 |
| 2 | 16 | 3 | 33 | 86 | 161 | 274 | 406 | 558 | 742 | 951 | 1186 | 1447 |
| 4 | 1 | 12 | 61 | 103 | 177 | 290 | 421 | 578 | 763 | 971 | 1212 | 1473 |
| 4 | 2 | 8 | 63 | 100 | 175 | 289 | 418 | 577 | 759 | 968 | 1201 | 1465 |
| 4 | 4 | 9 | 50 | 99 | 170 | 285 | 408 | 565 | 747 | 960 | 1194 | 1460 |
| 4 | 16 | 13 | 46 | 99 | 168 | 278 | 411 | 563 | 743 | 955 | 1185 | 1446 |
| 10 | 1 | 112 | 140 | 190 | 271 | 376 | 507 | 662 | 845 | 1051 | 1293 | 1547 |
| 10 | 2 | 109 | 144 | 186 | 265 | 369 | 499 | 656 | 834 | 1045 | 1278 | 1541 |
| 10 | 4 | 113 | 140 | 185 | 264 | 368 | 494 | 649 | 830 | 1037 | 1272 | 1533 |
| 10 | 16 | 111 | 136 | 186 | 260 | 363 | 492 | 646 | 827 | 1031 | 1265 | 1528 |

Таблица 9 - Время выполнения программы для dynamic schedule

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Threads** | **Chunk** | **650** | **4185** | **7720** | **11255** | **14790** | **18325** | **21860** | **25395** | **28930** | **32465** | **36000** |
| 1 | 1 | 6 | 69 | 175 | 333 | 544 | 804 | 1111 | 1474 | 1884 | 2346 | 2861 |
| 1 | 2 | 5 | 57 | 156 | 306 | 508 | 758 | 1062 | 1411 | 1817 | 2265 | 2774 |
| 1 | 4 | 4 | 56 | 150 | 299 | 497 | 742 | 1042 | 1391 | 1790 | 2239 | 2741 |
| 1 | 16 | 4 | 55 | 149 | 295 | 490 | 739 | 1035 | 1386 | 1783 | 2230 | 2734 |
| 2 | 1 | 12 | 80 | 171 | 288 | 439 | 596 | 796 | 1015 | 1255 | 1531 | 1829 |
| 2 | 2 | 8 | 54 | 125 | 220 | 348 | 492 | 668 | 869 | 1092 | 1347 | 1619 |
| 2 | 4 | 5 | 46 | 102 | 192 | 303 | 445 | 612 | 806 | 1022 | 1261 | 1534 |
| 2 | 16 | 3 | 35 | 90 | 167 | 277 | 407 | 566 | 753 | 959 | 1196 | 1462 |
| 4 | 1 | 15 | 96 | 178 | 285 | 426 | 592 | 778 | 1000 | 1244 | 1522 | 1810 |
| 4 | 2 | 10 | 76 | 134 | 226 | 344 | 491 | 663 | 859 | 1088 | 1333 | 1613 |
| 4 | 4 | 9 | 59 | 110 | 195 | 307 | 446 | 609 | 798 | 1016 | 1254 | 1527 |
| 4 | 16 | 7 | 49 | 97 | 174 | 285 | 412 | 569 | 754 | 959 | 1190 | 1462 |
| 10 | 1 | 118 | 149 | 219 | 323 | 459 | 625 | 815 | 1032 | 1274 | 1545 | 1844 |
| 10 | 2 | 115 | 138 | 192 | 280 | 391 | 534 | 697 | 895 | 1115 | 1369 | 1661 |
| 10 | 4 | 116 | 136 | 187 | 264 | 370 | 502 | 666 | 847 | 1064 | 1296 | 1565 |
| 10 | 16 | 113 | 135 | 182 | 259 | 362 | 491 | 645 | 824 | 1032 | 1268 | 1527 |

Таблица 10 - Время выполнения программы для guided schedule

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Threads** | **Chunk** | **650** | **4185** | **7720** | **11255** | **14790** | **18325** | **21860** | **25395** | **28930** | **32465** | **36000** |
| 1 | 1 | 4 | 52 | 146 | 292 | 486 | 733 | 1033 | 1377 | 1774 | 2225 | 2720 |
| 1 | 2 | 4 | 54 | 146 | 291 | 486 | 733 | 1030 | 1379 | 1773 | 2222 | 2721 |
| 1 | 4 | 4 | 54 | 147 | 291 | 487 | 733 | 1031 | 1377 | 1777 | 2222 | 2725 |
| 1 | 16 | 3 | 53 | 145 | 291 | 486 | 733 | 1029 | 1376 | 1775 | 2232 | 2723 |
| 2 | 1 | 3 | 33 | 84 | 160 | 268 | 397 | 556 | 736 | 943 | 1181 | 1435 |
| 2 | 2 | 3 | 31 | 83 | 162 | 268 | 397 | 553 | 734 | 944 | 1176 | 1453 |
| 2 | 4 | 3 | 33 | 84 | 160 | 269 | 398 | 553 | 734 | 943 | 1178 | 1440 |
| 2 | 16 | 3 | 33 | 85 | 162 | 270 | 397 | 553 | 734 | 947 | 1179 | 1435 |
| 4 | 1 | 9 | 45 | 95 | 167 | 276 | 407 | 560 | 744 | 948 | 1182 | 1443 |
| 4 | 2 | 8 | 50 | 93 | 167 | 278 | 404 | 560 | 743 | 950 | 1180 | 1440 |
| 4 | 4 | 8 | 59 | 95 | 167 | 275 | 405 | 562 | 741 | 946 | 1183 | 1442 |
| 4 | 16 | 10 | 46 | 95 | 170 | 274 | 406 | 561 | 739 | 947 | 1180 | 1440 |
| 10 | 1 | 114 | 134 | 180 | 259 | 356 | 487 | 642 | 817 | 1025 | 1260 | 1516 |
| 10 | 2 | 113 | 132 | 185 | 256 | 359 | 536 | 659 | 815 | 1022 | 1254 | 1512 |
| 10 | 4 | 111 | 138 | 223 | 275 | 458 | 624 | 646 | 831 | 1028 | 1266 | 1520 |
| 10 | 16 | 120 | 135 | 178 | 260 | 358 | 486 | 640 | 820 | 1029 | 1261 | 1521 |

После исключения "узкого места программа" в виде сортировки среднее время выполнения для static изменилось с 1910 мс до 676 мс, что довольно существенно.

Диаграмма 3 - Зависимость времени выполнения от размера массива до распараллеливания сортировки

Диаграмма 4 - Зависимость времени выполнения от размера массива после распараллеливания сортировки

По построенным диаграммам видно, насколько важно использовать параллельные вычисления при работе с программами, которые требуют обработку большого количества данных. Распараллеливание процессов, позволило свести О(n2) к O(n) алгоритмической сложности.

Наилучшую эффективность показал guided schedule, время выполнения программы с данным параметром в среднем ниже, чем у static и dynamic.

Среднее выполнение static до распараллеливания сортировки: 1920 мс.

Средние значения по таблице: guided = 668, static = 676, dynamic = 730.

Средние значения при N=650: static = 32,1, guided = 32,5, dynamic =34,4.

Средние значения при N=36000: guided = 1780, static = 1787, dynamic = 1910.

Как итог можно вынести, что для малого количество итераций контроль потоков не обязателен и static будет выполняться быстрее. Но для среднего и большого лучше всего выполнил guided.

Вычисление в 1 поток дает малую эффективность, 2 и 4 потока показали лучший результат со всеми типами schedule, использование 4 потоков вместо 2 не дало двухкратное увеличение, значения времени выполнения схожи.

Применение 10 потоков их поддержание и имитация тратит много ресурсов. Особенно это заметно на малом размере массивов, когда время выполнения программы увеличивается в 10 раз. На больших размерах N разница минимальная, но все равно в пользу меньшего количества потоков.

Использование большого размера чанка, большего чем число потоков тоже ведет к увеличению времени выполнения программы. Оптимальным числом является половина от числа потоков или равное числу потоков.

Задание 4

Функция вывода прогресса имеет вид:

void print\_progress(int \*progress) {  
 while (\*progress < 100) {  
 printf("Progress: %d%%\n", \*progress);  
 sleep(1);  
 }  
 printf("Progress: 100%%\n");  
}  
#endif

Работа функции в отдельном потоке обеспечивается за счет:

#ifdef \_OPENMP  
 int progress = 0;  
#pragma omp parallel sections  
 {  
#pragma omp section  
 {  
 print\_progress(&progress);  
 }  
#pragma omp section  
 {  
#endif  
 for (i=0; i<100; i++) {

#ifdef \_OPENMP  
 #pragma omp atomic write  
 progress = i;  
#endif

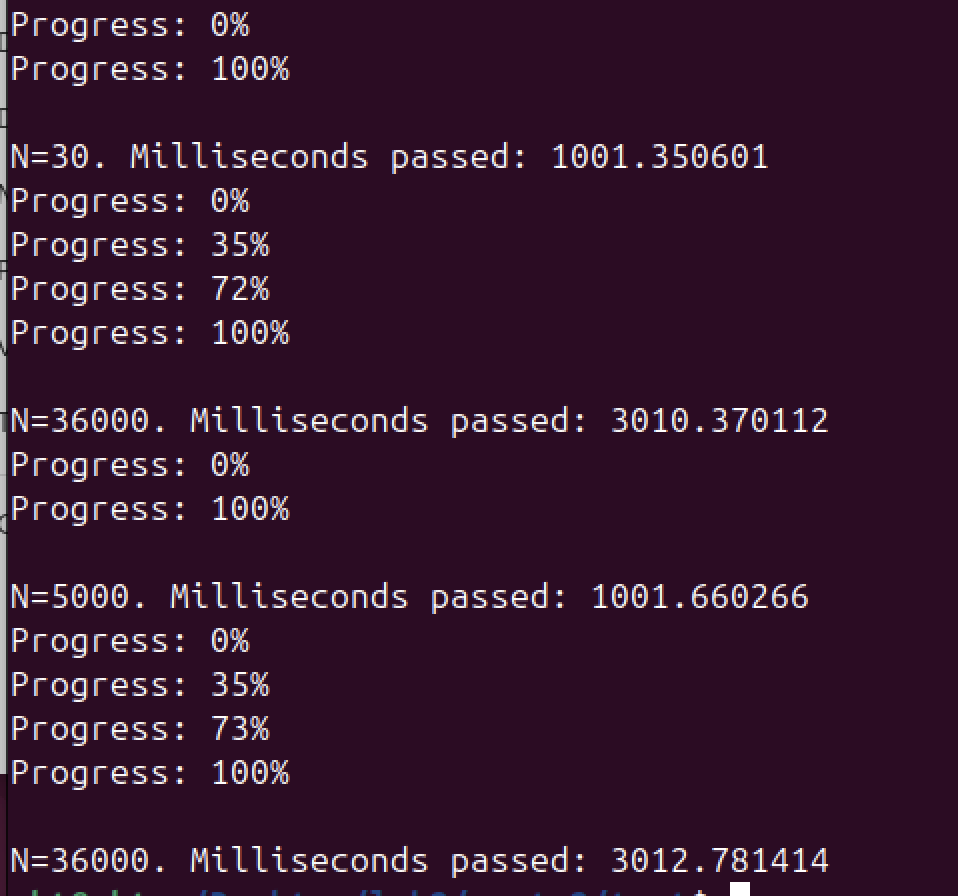
…

}

#ifdef \_OPENMP  
}  
progress = 100;  
}  
#endif

return 0;

}



**Задание 5**

Прямую совместимость можно реализовать через конструкцию

#ifdef

…

#else

…

#endif

Код программы:

#include <stdio.h>  
#include <stdlib.h>  
#include <sys/time.h>  
#include <time.h>  
#include <math.h>  
#include <string.h>  
#ifdef \_OPENMP  
#include <omp.h>  
#endif  
int main(int argc, char\* argv[]) {  
  
 int i, N, A;  
 N = atoi(argv[1]);  
 A = atoi(argv[2]);  
 char\* schedule\_type = argv[4];  
 int chunk\_size = atoi(argv[5]);  
  
#ifdef \_OPENMP  
 omp\_sched\_t schedule\_kind;  
  
 omp\_set\_num\_threads(atoi(argv[3]));  
  
 if (strcmp(schedule\_type, "static") == 0) {  
 schedule\_kind = omp\_sched\_static;  
 } else if (strcmp(schedule\_type, "dynamic") == 0) {  
 schedule\_kind = omp\_sched\_dynamic;  
 } else if (strcmp(schedule\_type, "guided") == 0) {  
 schedule\_kind = omp\_sched\_guided;  
 } else {  
 fprintf(stderr, "Ошибка: Неверный тип schedule: %s\n", schedule\_type);  
 return 1;  
 }  
 omp\_set\_schedule(schedule\_kind, chunk\_size);  
#endif  
#ifdef \_OPENMP  
 double start\_time, end\_time;  
 start\_time = omp\_get\_wtime();  
#else  
 printf("Программа выполняется последовательно");  
 struct timeval T1, T2;  
 long delta\_ms;  
 gettimeofday(&T1, NULL);  
#endif  
 //Этап 1  
 double \*M1 = malloc(sizeof(double)\* N);  
 double \*M2 = malloc(sizeof(double)\* N/2);  
 unsigned int seed = time(NULL);  
  
 for (i=0; i<100; i++) {  
 srand (i);  
  
 // Заполнение M1  
 for (int i = 0; i < N; i++) {  
 M1[i] = 1.0 + (A - 1.0) \* ((double)rand\_r(&seed) / RAND\_MAX);  
 }  
 // Заполнение M2  
 for (int i = 0; i < N / 2; i++) {  
 M2[i] = A + (10 \* A - A) \* ((double)rand\_r(&seed) / RAND\_MAX);  
 }  
#ifdef \_OPENMP  
#pragma omp parallel for schedule(runtime) default(none) private(i) shared(N, M1)  
#endif  
 // Этап 2: Map (гиперболический синус в квадрат)  
 for (int i = 0; i < N; i++) {  
 M1[i] = pow(sinh(M1[i]), 2);  
 }  
 // Создаем копию M2  
 double M2\_copy[N / 2];  
#ifdef \_OPENMP  
#pragma omp parallel for schedule(runtime) default(none) private(i) shared(N, M2\_copy, M2)  
#endif  
 for (int i = 0; i < N / 2; i++) {  
 M2\_copy[i] = M2[i];  
 }  
#ifdef \_OPENMP  
#pragma omp parallel for schedule(runtime) default(none) private(i) shared(N, M2, M2\_copy)  
#endif  
 // Этап 2: Map для M2 (кумулятивная сумма + sqrt(e \* сумма))  
 for (int i = 0; i < N / 2; i++) {  
 double prev = (i == 0) ? 0 : M2\_copy[i - 1]; // Предыдущий элемент или 0  
 M2[i] = sqrt(M\_E \* (M2\_copy[i] + prev));  
 }  
#ifdef \_OPENMP  
#pragma omp parallel for schedule(runtime) default(none) private(i) shared(M2, N, M1)  
#endif  
 // Этап 3: Merge (умножение элементов M2 и M1 с одинаковыми индексами)  
 for (int i = 0; i < N / 2; i++) {  
 M2[i] = M2[i] \* M1[i]; // Умножение  
 }  
 // Этап 4: Sort (сортировка вставками для M2) // распараллеленная  
#ifdef \_OPENMP  
 #pragma omp parallel sections  
 {  
#pragma omp section  
 {  
 // (M2[0:N/4])  
 for (int i = 1; i < N / 4; i++) {  
 double key = M2[i];  
 int j = i - 1;  
 while (j >= 0 && M2[j] > key) {  
 M2[j + 1] = M2[j];  
 j = j - 1;  
 }  
 M2[j + 1] = key;  
 }  
 }  
#pragma omp section  
 {  
 // (M2[N/4:N/2])  
 for (int i = N / 4; i < N / 2; i++) {  
 double key = M2[i];  
 int j = i - 1;  
 while (j >= N / 4 && M2[j] > key) {  
 M2[j + 1] = M2[j];  
 j = j - 1;  
 }  
 M2[j + 1] = key;  
 }  
 }  
 }  
 // Однопоточное слияние  
 double merged[N / 2];  
 int left = 0, right = N / 4, k = 0;  
  
 while (left < N / 4 && right < N / 2) {  
 if (M2[left] < M2[right])  
 merged[k++] = M2[left++];  
 else  
 merged[k++] = M2[right++];  
 }  
 while (left < N / 4)  
 merged[k++] = M2[left++];  
 while (right < N / 2)  
 merged[k++] = M2[right++];  
  
 memcpy(M2, merged, sizeof(double) \* (N / 2)); // Копируем  
#else  
 // Этап 4: Sort (сортировка вставками для M2)  
 for (int i = 1; i < N / 2; i++) {  
 double key = M2[i];  
 int j = i - 1;  
  
 // Сдвиг элементов массива, которые больше ключа  
 while (j >= 0 && M2[j] > key) {  
 M2[j + 1] = M2[j];  
 j = j - 1;  
 }  
  
 M2[j + 1] = key;  
 }  
#endif  
 // Этап 5: Reduce (расчет суммы синусов для определенных элементов)  
 double min\_value = M2[0];  
#ifdef \_OPENMP  
#pragma omp parallel for schedule(runtime) default(none) private(i) shared(min\_value, N, M2)  
#endif  
 // Поиск минимального ненулевого элемента в M2  
 for (int i = 1; i < N / 2; i++) {  
 if (M2[i] != 0 && M2[i] < min\_value) {  
 min\_value = M2[i];  
 }  
 }  
  
 // Расчет суммы синусов  
 double X = 0.0;  
#ifdef \_OPENMP  
#pragma omp parallel for schedule(runtime) default(none) private(i) shared(X, N, min\_value, M2)  
#endif  
 for (int i = 0; i < N / 2; i++) {  
 // Проверка, делится ли M2[i] на минимальный элемент с четной целой частью  
 if ((int)(M2[i] / min\_value) % 2 == 0) {  
 X += sin(M2[i]);  
 }  
 }  
 }  
 free(M1); free(M2);  
#ifdef \_OPENMP  
 end\_time = omp\_get\_wtime();  
 printf("\nN=%d. Milliseconds passed: %lf\n", N, (end\_time - start\_time) \* 1000);  
#else  
 gettimeofday(&T2, NULL);  
 delta\_ms = (T2.tv\_sec - T1.tv\_sec) \* 1000 +  
 (T2.tv\_usec - T1.tv\_usec) / 1000;  
 printf ("\nN=%d. Milliseconds passed: %ld\n", N, delta\_ms);  
#endif  
 return 0;  
}