|  |  |  |
| --- | --- | --- |
| ПрИб-181 | Лабораторная работа №5 | Зачёт |
| Кащенко В. А. | Распараллеливание циклов в OpenMP. |  |

Цель работы: Изучить методы распараллеливания циклов.

Теоретические сведения:

(Конспект теоретических сведений написан в тетради)

Задание 6, программа «Сумма чисел»:

Для распараллеливания циклов используется директива #omp parallel for

Исходный код (k — количество нитей, n — количество чисел ряда 1,2,3...∞):

#include <stdio.h>

#include <omp.h>

#include <iostream>

#include <stdlib.h>

#include <iomanip>

using namespace std;

// Хранение результатов сложения

int sequental\_res, parallel\_res;

double time(bool parallel, int k, int N)

{

// Для замеров скорости

double start\_time, end\_time;

// объявим переменные результатов сложения

int res = 0;

int res\_omp = 0;

// последовательно сложим

start\_time = omp\_get\_wtime();

for (int i = 1; i <= N; ++i)

res += i;

end\_time = omp\_get\_wtime();

sequental\_res = res;

// параллельно умножим

start\_time = omp\_get\_wtime();

// используем parallel for

#pragma omp parallel for num\_threads(k) if(parallel)

for (int i = 1; i <= N; ++i)

res\_omp += i;

end\_time = omp\_get\_wtime();

parallel\_res = res\_omp;

return (end\_time-start\_time) \* 1'000'000 ; // мкс

}

// вывод на экран времени

void print\_time(int n, int k)

{

// func format is (bool parallel, int k, int N)

// Time table

cout << left << fixed << setprecision(3)

<< setw(10) << n << setw(17) << time(0, k, n) << setw(18)

<< time(0, k, n) << setw(18) << sequental\_res << setw(15) << parallel\_res << endl;

}

// точка входа

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

{

cout << "Кащенко В. А. ПрИб-181\n\"Сложение чисел\" (параллельная версия)\ n\n4 потока:\n" <<

setw(10) << "Кол-во " << setw(20) << "Послед-но, мкс " << setw(20) << "Паралл-но, мкс " << setw(22) << " Р-ат послед-но " << setw(15) << " Р-ат паралл-но " << endl;

// func format is (int k, int n)

print\_time(50, 4);

print\_time(100, 4);

print\_time(500, 4);

print\_time(1000, 4);

print\_time(5000, 4);

print\_time(15000, 4);

print\_time(30000, 4);

print\_time(50000, 4);

print\_time(100000, 4);

print\_time(1000000, 4);

cout << "\n6 потоков:\n" << setw(10) << "Кол-во " << setw(20) << "Послед-но, мкс " << setw(20) << "Паралл-но, мкс " << setw(22) << " Р-ат послед-но " << setw(15) << " Р-ат паралл-но " << endl;

print\_time(50, 6);

print\_time(100, 6);

print\_time(500, 6);

print\_time(1000, 6);

print\_time(5000, 6);

print\_time(15000, 6);

print\_time(30000, 6);

print\_time(50000, 6);

print\_time(100000, 6);

print\_time(1000000, 6);

cout << "\n12 потоков:\n" << setw(10) << "Кол-во " << setw(20) << "Послед- но, мкс " << setw(20) << "Паралл-но, мкс " << setw(22) << " Р-ат послед-но " << setw(15) << " Р-ат паралл-но " << endl;

print\_time(50, 12);

print\_time(100, 12);

print\_time(500, 12);

print\_time(1000, 12);

print\_time(5000, 12);

print\_time(15000, 12);

print\_time(30000, 12);

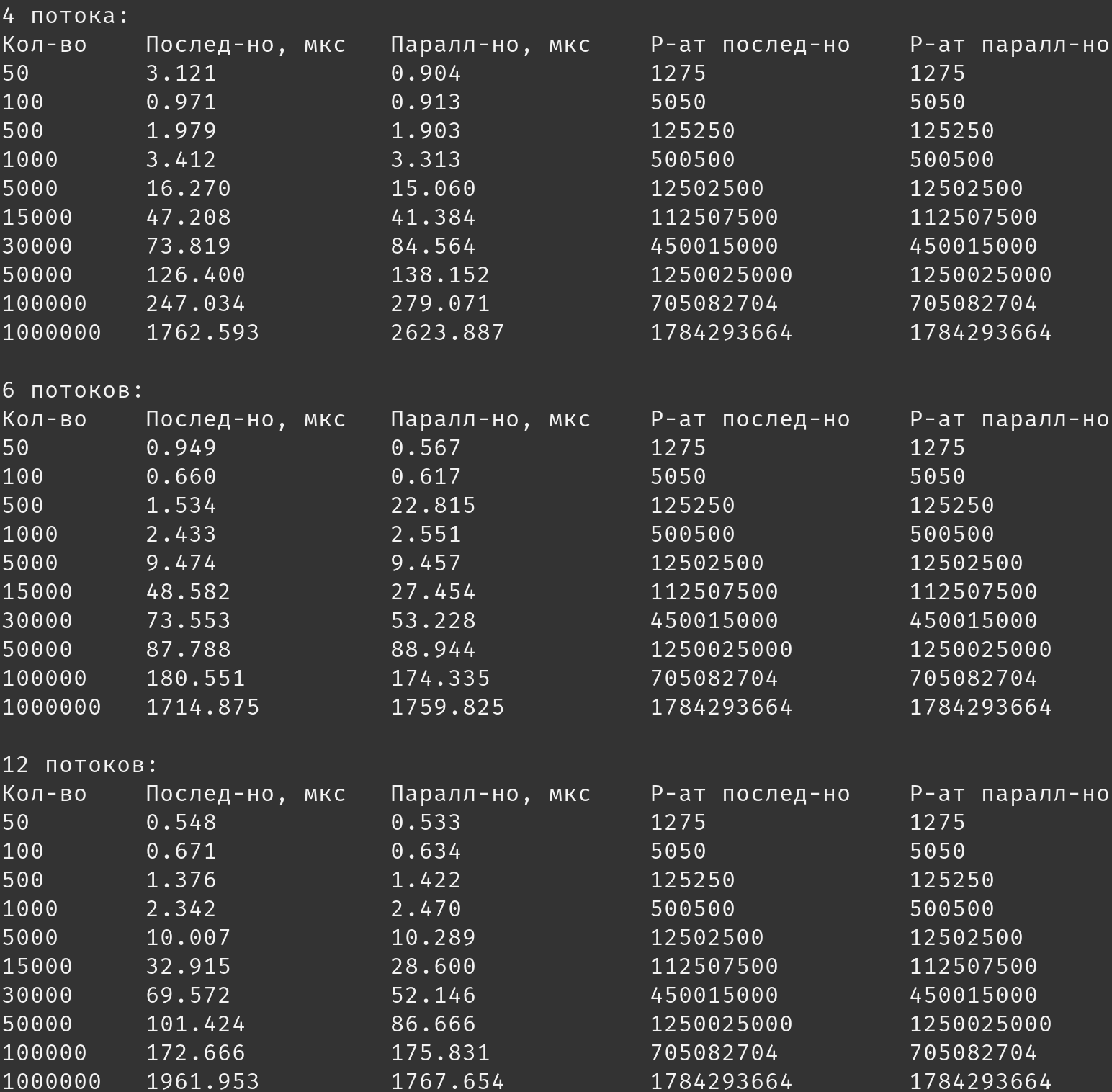
print\_time(50000, 12);

print\_time(100000, 12);

print\_time(1000000, 12);

}

Работа программы:



В общем случае получилось, что самая быстрая реализация оказалась на 6 нитях (меньше не хватило быстродействия нитей, а с большим количеством нити просто теряли свое быстродействие на организацию параллелизма). А заметна разница, по сравнению с последовательной версией, на двенадцати потоках.

Выигрыш по сравнению с последовательным выполнением виден в основном на бОльшем количестве нитей и на бОльших суммах.

Задание 7, параметр schedule:

Для распараллеливания циклов используется директива #omp parallel for

Для сравнения работы параметр schedule изменялся для замеров.

Блокировка k и N не в вводилась для получения более полной картины.

Исходный код (k — количество нитей, n — количество чисел ряда 1,2,3...∞):

#include <stdio.h>

#include <omp.h>

#include <iostream>

#include <stdlib.h>

#include <iomanip>

using namespace std;

// Хранение результатов сложения

int sequental\_res = 0, parallel\_res = 0;

double time(bool parallel, int k, int n)

{

// Для замеров скорости

double start\_time, end\_time;

// объявим переменные результатов сложения

int res = 0;

int res\_omp = 0;

// последовательно сложим

start\_time = omp\_get\_wtime();

for (int i = 1; i <= n; ++i)

res += i;

end\_time = omp\_get\_wtime();

sequental\_res = res;

// параллельно умножим

start\_time = omp\_get\_wtime();

// используем parallel for

#pragma omp parallel for num\_threads(k) reduction(+: res\_omp) if(parallel)

//#pragma omp parallel for schedule (static, 1) num\_threads(k) reduction(+: res\_omp) if(parallel)

//#pragma omp parallel for schedule (static, 2) num\_threads(k) reduction(+: res\_omp) if(parallel)

//#pragma omp parallel for schedule (dynamic) num\_threads(k) reduction(+: res\_omp) if(parallel)

//#pragma omp parallel for schedule (dynamic, 2) num\_threads(k) reduction(+: res\_omp) if(parallel)

//#pragma omp parallel for schedule (guided) num\_threads(k) reduction(+: res\_omp) if(parallel)

//#pragma omp parallel for schedule (guided, 2) num\_threads(k) reduction(+: res\_omp) if(parallel)

//#pragma omp parallel for schedule (static) num\_threads(k) reduction(+: res\_omp) if(parallel)

for (int i = 1; i <= n; ++i)

res\_omp += i;

end\_time = omp\_get\_wtime();

parallel\_res = res\_omp;

return (end\_time-start\_time) \* 1'000'000 ; // мкс

}

// вывод на экран времени

void print\_time(int n, int k)

{

// func format is (bool parallel, int k, int N)

// Time table

cout << left << fixed << setprecision(3)

<< setw(10) << n << setw(17) << time(0, k, n) << setw(18)

<< time(1, k, n) << setw(18) << sequental\_res << setw(15) << parallel\_res << endl;

}

// точка входа

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

{

cout << "Кащенко В. А. ПрИб-181\n\"Сложение чисел\" (параллельная версия)\n\n4 потока:\n" <<

setw(10) << "Кол-во " << setw(20) << "Послед-но, мкс " << setw(20) << "Паралл-но, мкс " << setw(22) <<

" Р-ат послед-но " << setw(15) << " Р-ат паралл-но " << endl;

int threads = 4;

print\_time(50, threads);

print\_time(100, threads);

print\_time(500, threads);

print\_time(1000, threads);

print\_time(5000, threads);

print\_time(15000, threads);

print\_time(30000, threads);

print\_time(50000, threads);

print\_time(100000, threads);

print\_time(1000000, threads);

cout << "\n6 потоков:\n" <<

setw(10) << "Кол-во " << setw(20) << "Послед-но, мкс " << setw(20) << "Паралл-но, мкс " << setw(22) <<

" Р-ат послед-но " << setw(15) << " Р-ат паралл-но " << endl;

threads = 6;

print\_time(50, threads);

print\_time(100, threads);

print\_time(500, threads);

print\_time(1000, threads);

print\_time(5000, threads);

print\_time(15000, threads);

print\_time(30000, threads);

print\_time(50000, threads);

print\_time(100000, threads);

print\_time(1000000, threads);

cout << "\n12 потоков:\n" <<

setw(10) << "Кол-во " << setw(20) << "Послед-но, мкс " << setw(20) << "Паралл-но, мкс " << setw(22) <<

" Р-ат послед-но " << setw(15) << " Р-ат паралл-но " << endl;

threads = 12;

print\_time(50, threads);

print\_time(100, threads);

print\_time(500, threads);

print\_time(1000, threads);

print\_time(5000, threads);

print\_time(15000, threads);

print\_time(30000, threads);

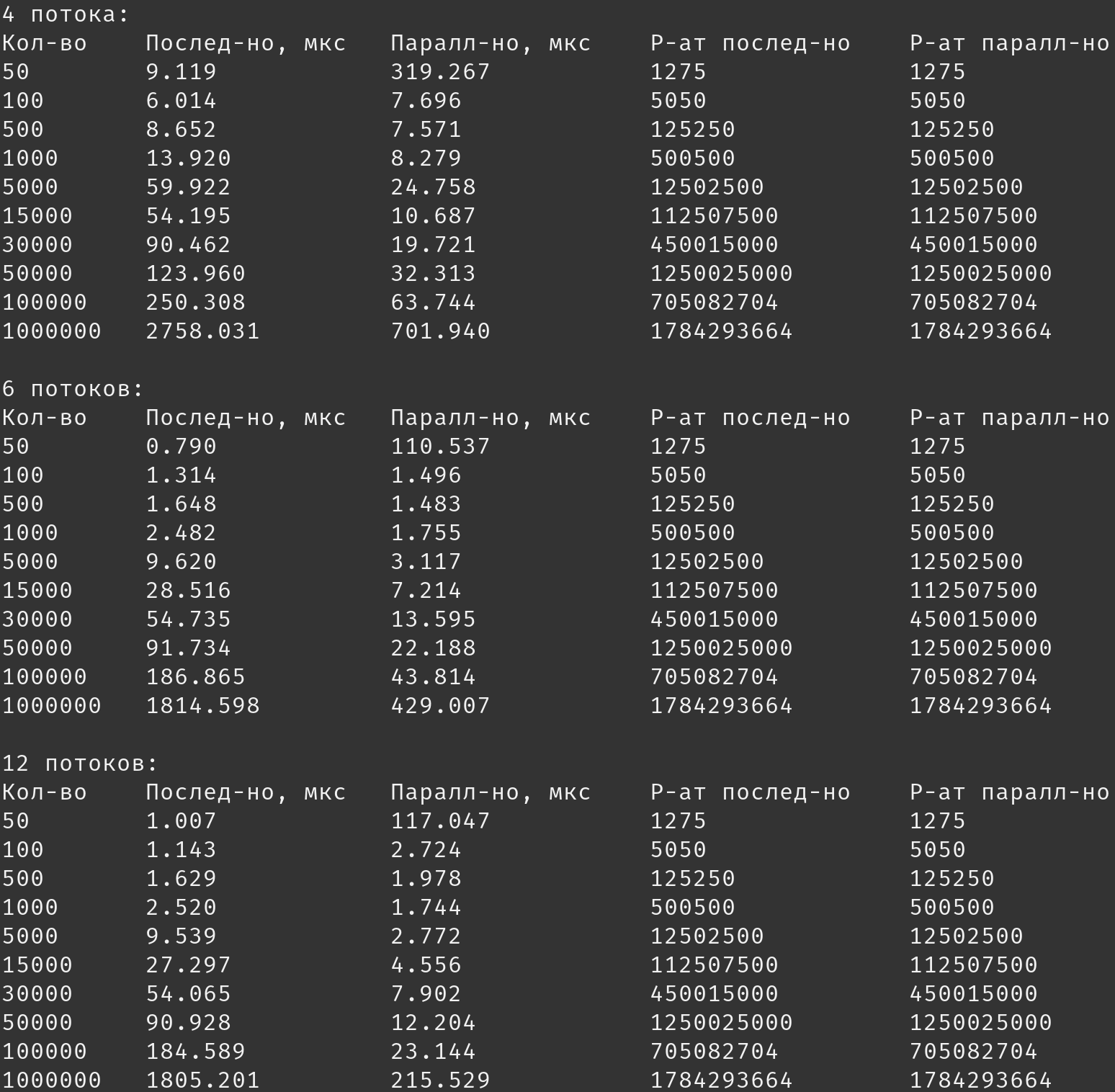
print\_time(50000, threads);

print\_time(100000, threads);

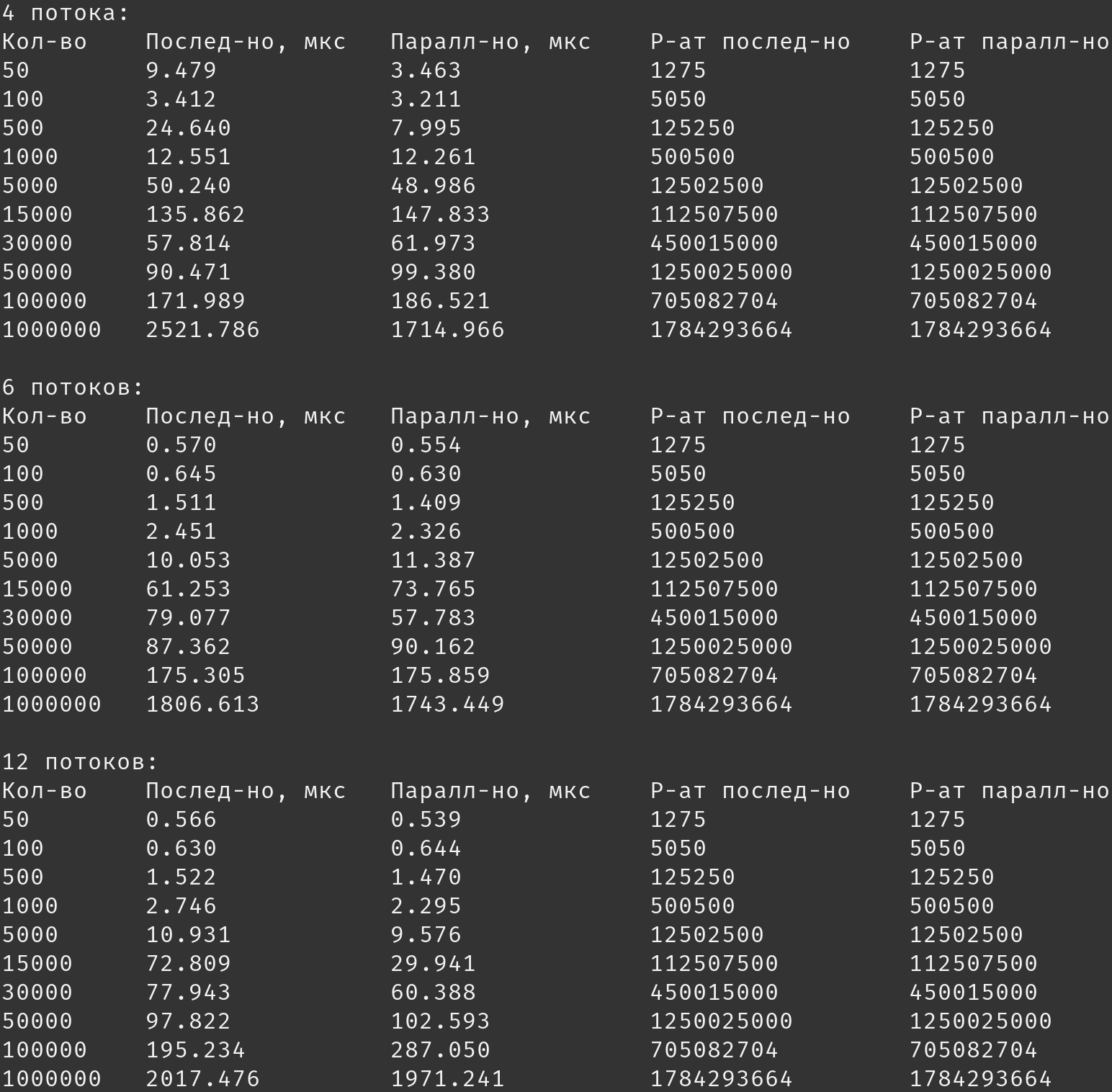
print\_time(1000000, threads);

}

Работа программы (без планирования):

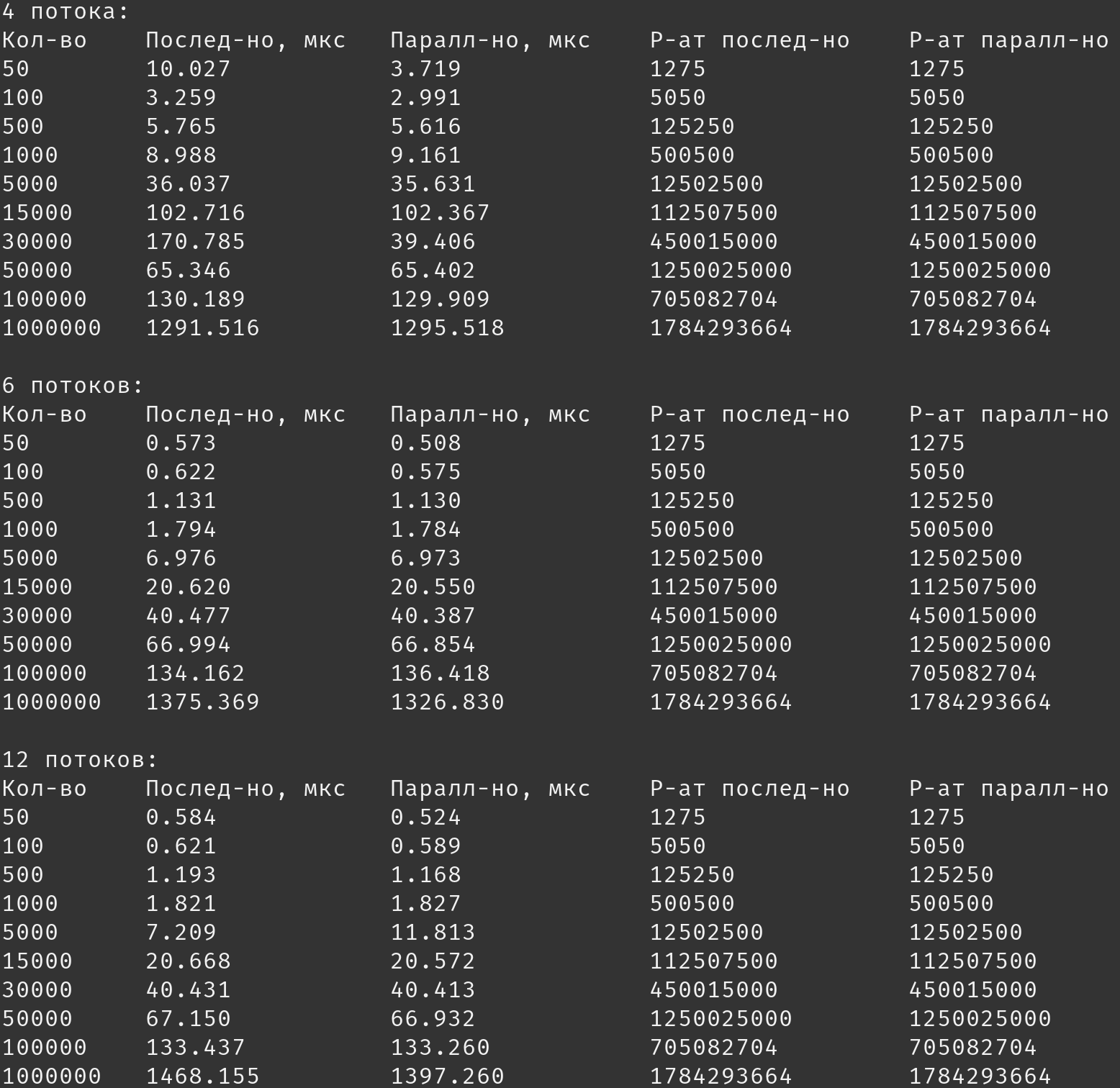
  
Данные результаты будем считать «отправной точкой исследования», с ним будем сравнивать директивы планирования. Здесь видны стабильные «просадки» скорости на маленьком числе элементов.

Работа программы (schedule (static)):

  
Для статического планирования свойственно, что во время выполнения OpenMP гарантирует, что если есть два отдельных цикла с одинаковым количеством итераций и нужно выполнить их с тем же количеством потоков, то каждый поток получит точно такой же диапазон итераций в обеих параллельных областях, используя статическое планирование.

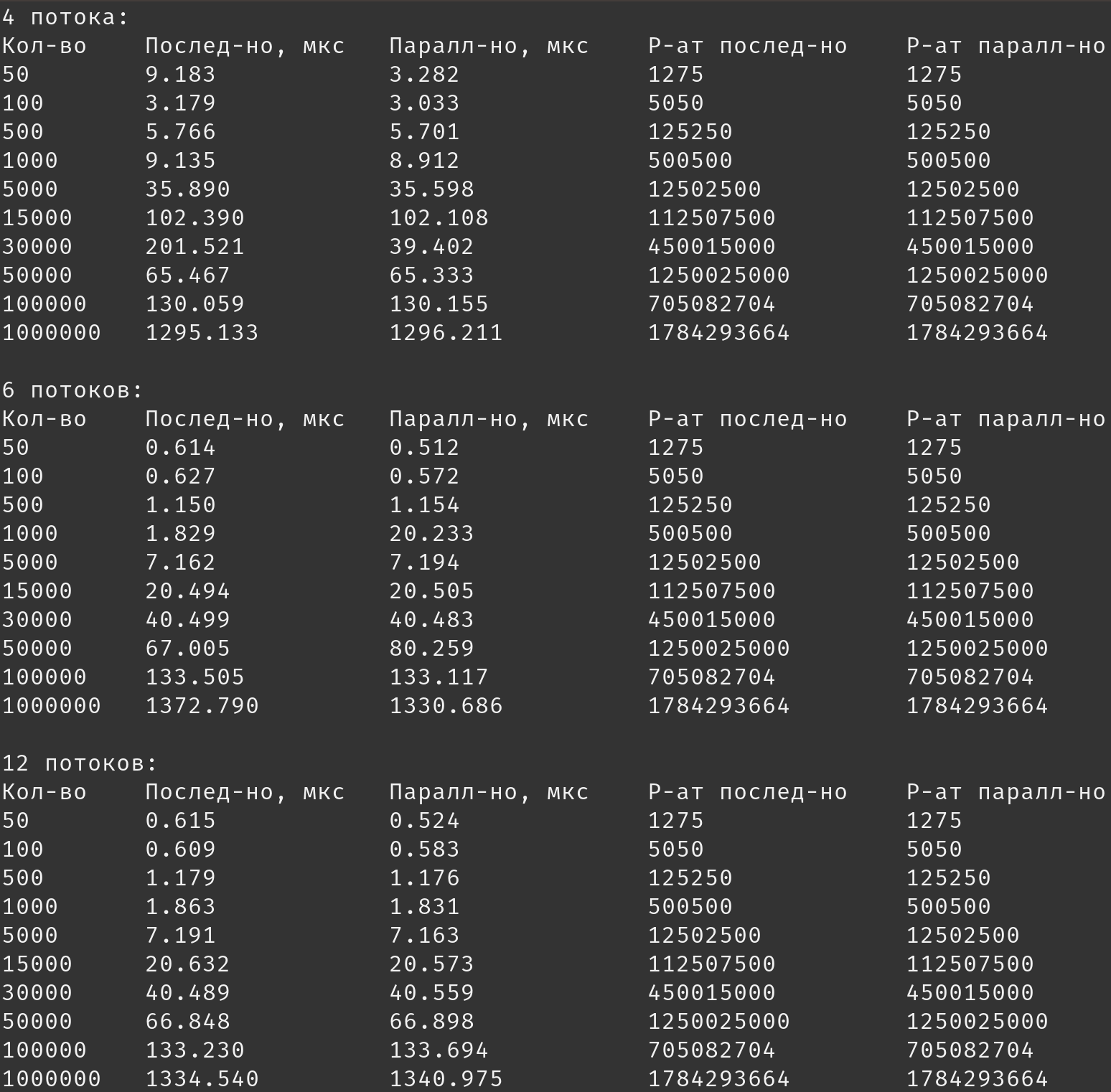
При сравнении с результатом без планирования видно, что общем случае программа стала медленнее, а значит, данная директива не подходит для этой задачи, однако, «просадок» скорости нет, как в случае без планирования.

Работа программы (schedule (static, 1)):

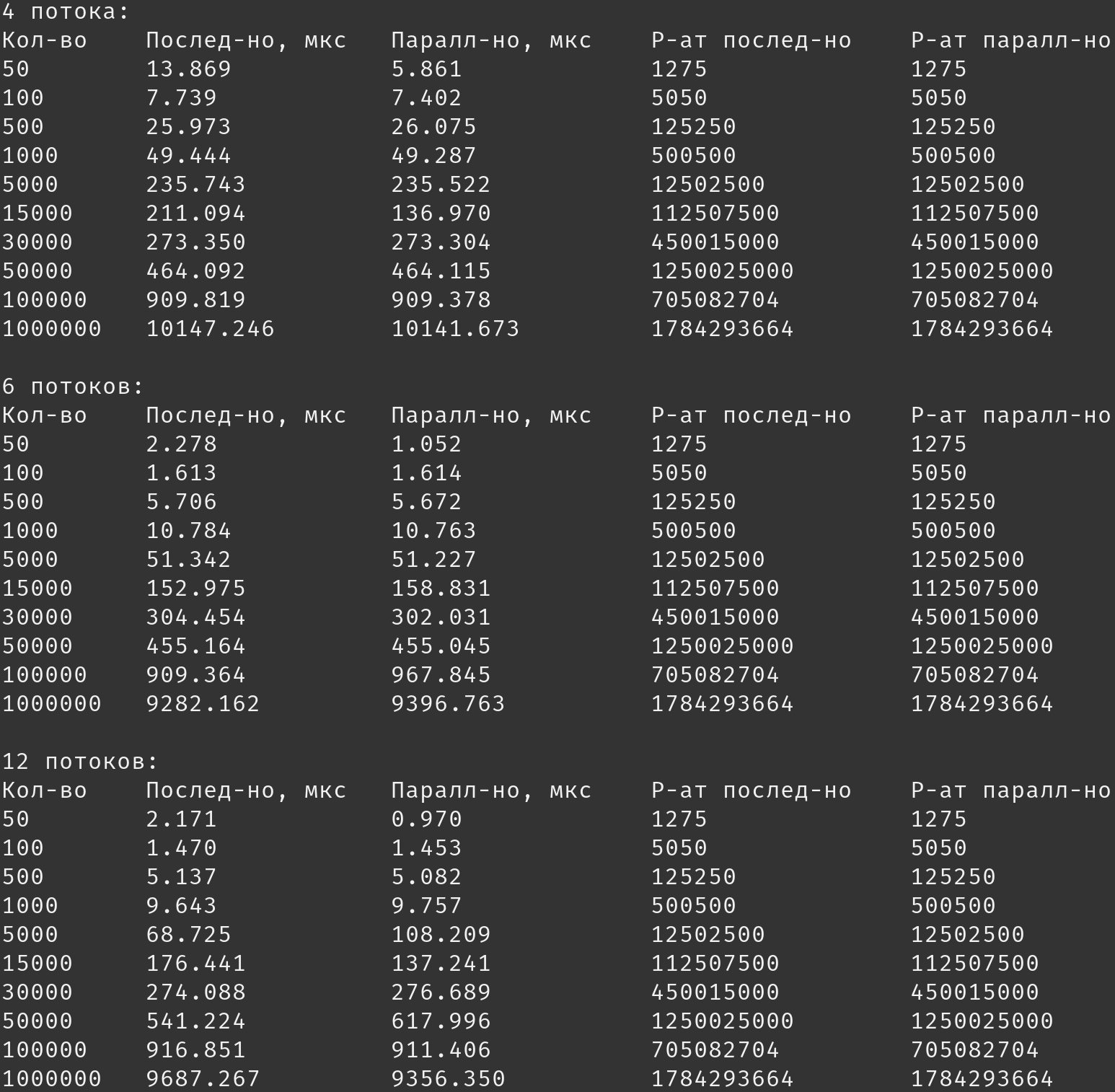
  
Данный код работает быстрее на маленьком числе итераций, но медленнее на большом. Его целесообразно использовать при небольшом числе итераций.

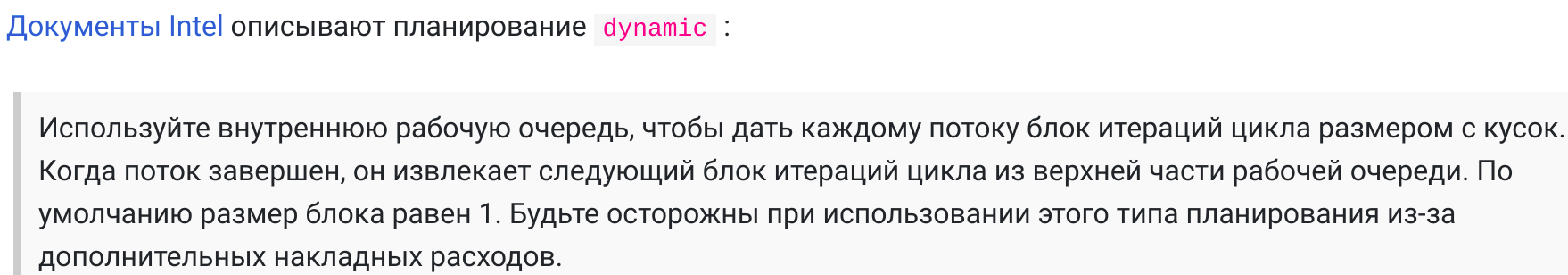
«Просадок» скорости нет, как в случае без планирования.

Работа программы (schedule (static, 2)):

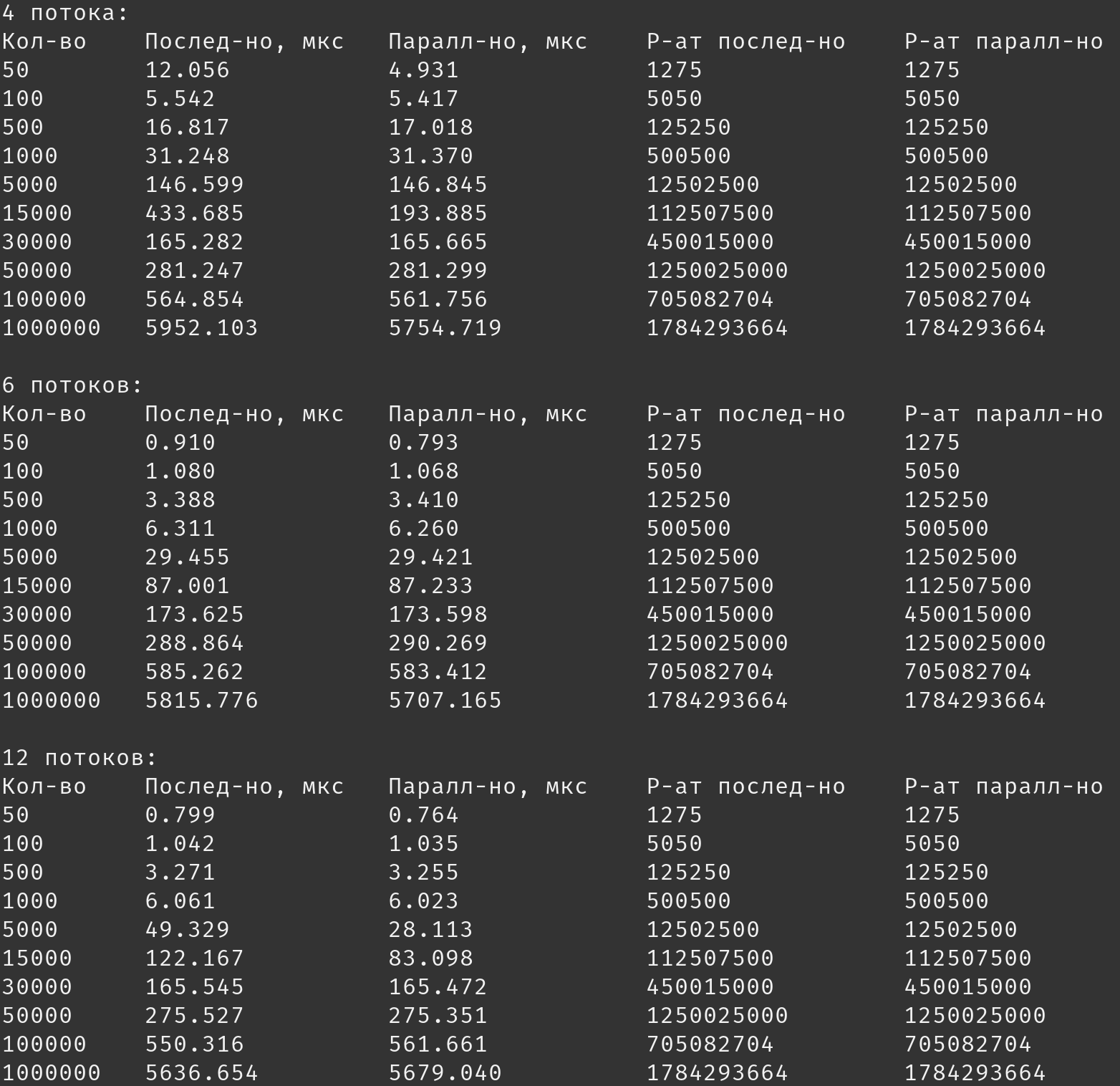
  
Данный код показывает выигрыш по сравнению со (static, 1), но, аналогично, его целесообразно использовать при небольшом числе итераций. Опять таки, «просадок» скорости на малом числе итераций нет.

Работа программы (schedule (dynamic)):

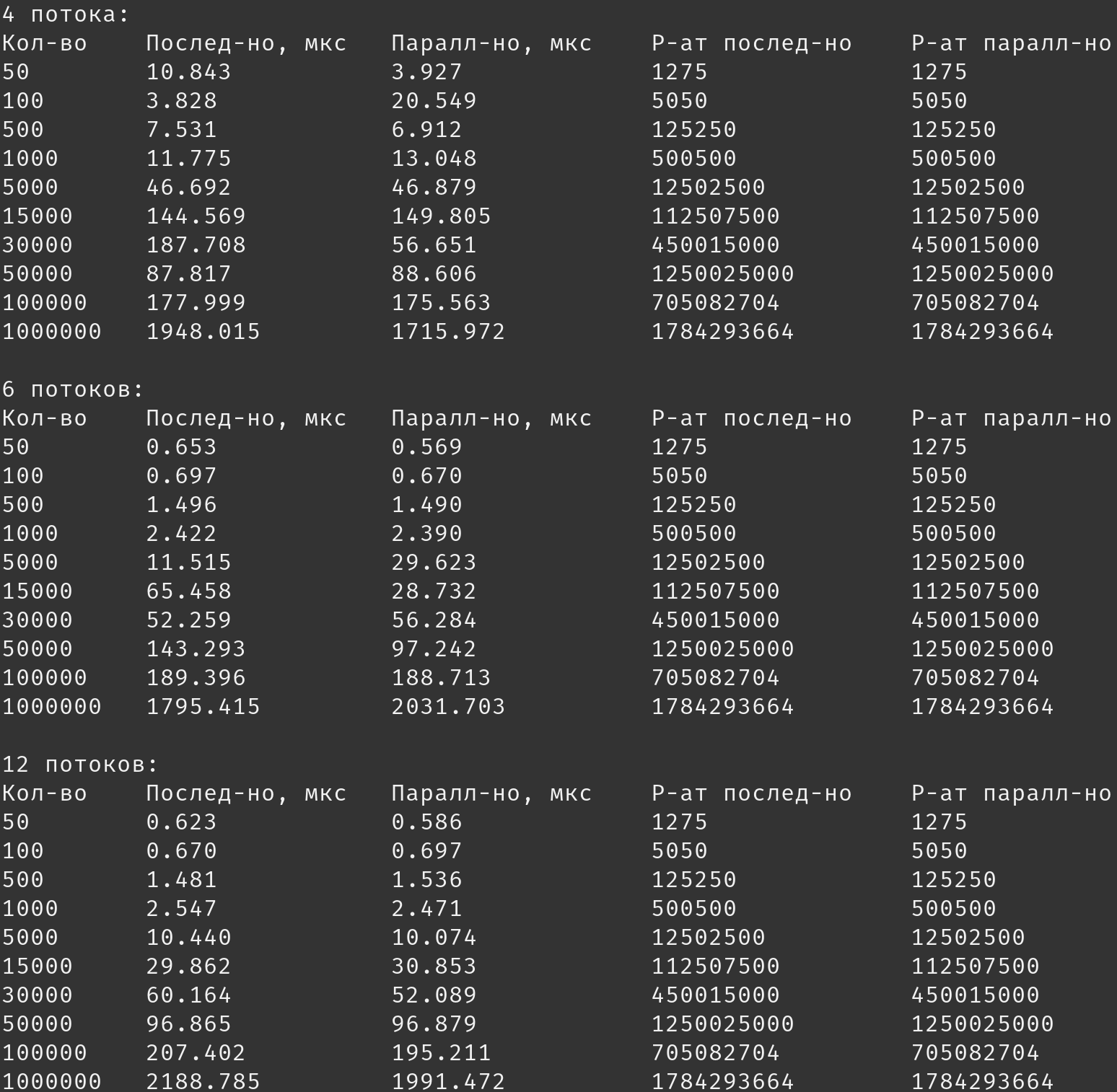
  
Здесь меняется метод планирования на динамическое.

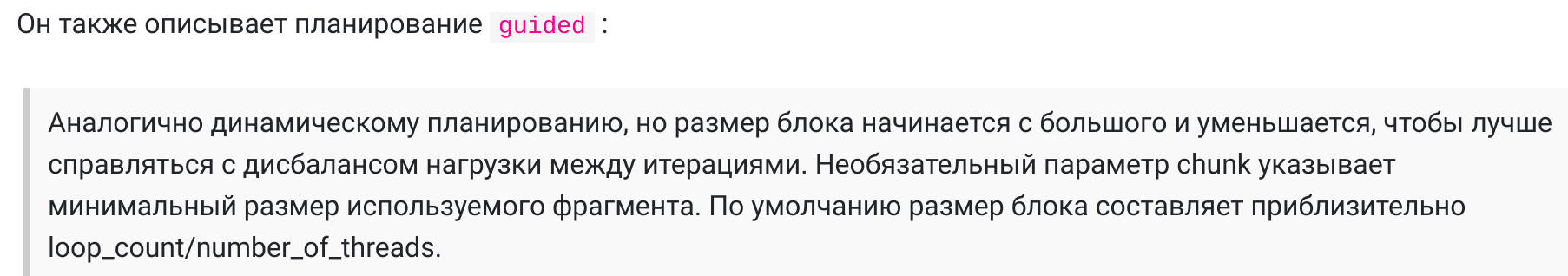
  
На небольшом числе итераций можно заметить прирост скорости, но на большем числе, получается, видны эти дополнительные накладные расходы.

Работа программы (schedule (dynamic, 2)):

  
Эта версия dynamic работает немного лучше, но совершенно аналогично видны накладные расходы с увеличением числа итераций.

Работа программы (schedule (guided)):

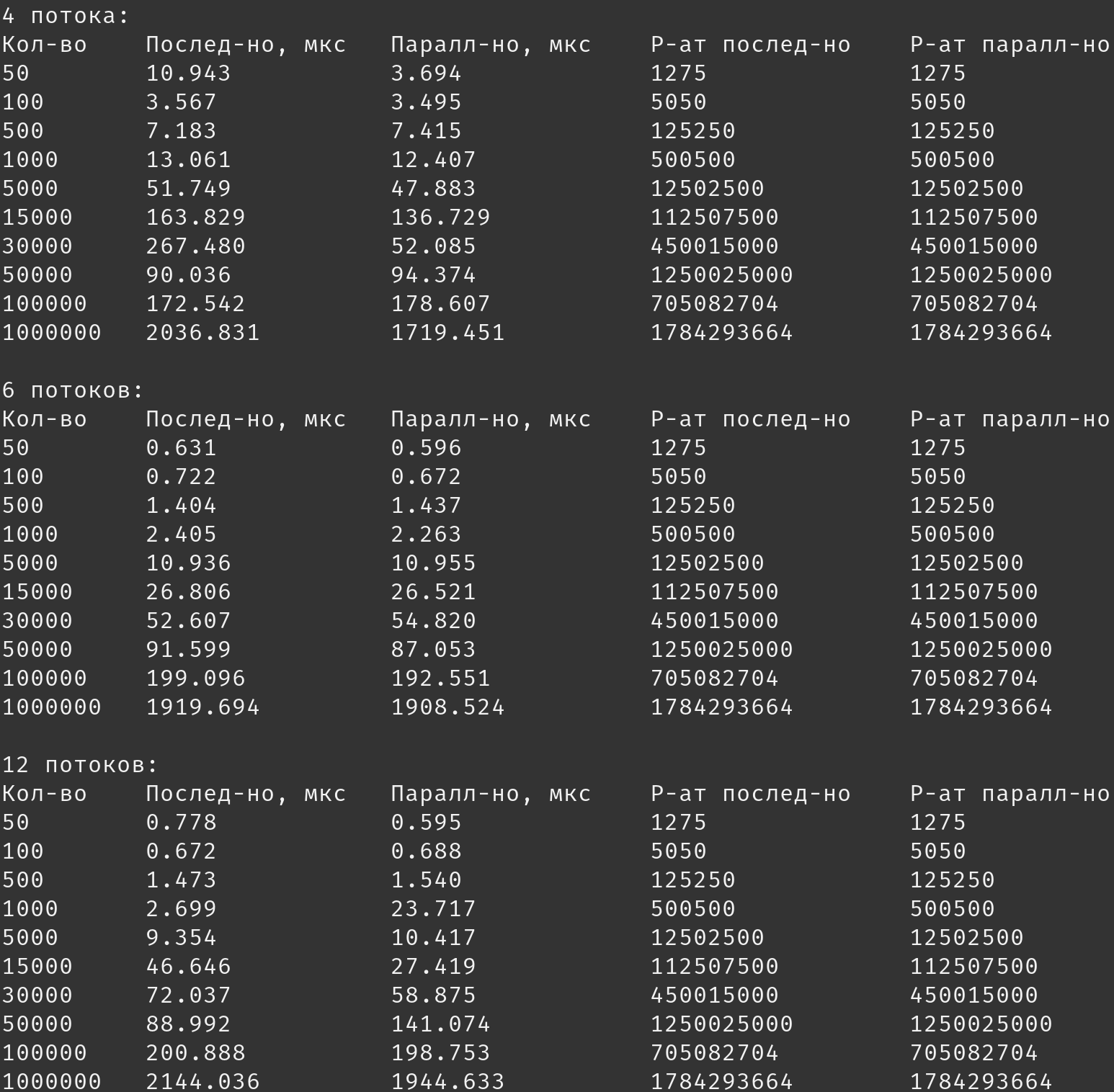
  
Здесь меняется метод планирования на управляемое.

   
Получается, размер блока при 12 потоках и самом большом количестве итераций:

~83 333, 333

По сравнению с результатами без планирования «просадок» нет, но в общем случае выполнение оказалось медленнее.

Работа программы (schedule (guided, 2)):

  
Эта версия guided работает быстрее, но в общем случае она всё равно оказалась медленнее случая без планирования, что говорит о неподходящем варианте использования данного планирования.

Задание 8, программа «Число π»:

Здесь количество итераций фиксированное (иначе не дойдём до числа π в процессе вычислений), посмотрим на изменение скорости относительно числа активных потоков.

Исходный код программы:

#include <stdio.h>

#include <omp.h>

#include <iostream>

#include <stdlib.h>

#include <iomanip>

using namespace std;

// Хранение результатов

double sequental\_res, parallel\_res;

// Часть вычисления Pi

double f(double x)

{

return (4.0 / (1.0 + x \* x));

}

double time(bool parallel, int k, int N)

{

double x = 0, pi = 0;

// Для замеров скорости

double start\_time, end\_time;

// объявим переменные результатов

double res = 0;

double res\_omp = 0;

double dx = 1.0 / (double)N;

// последовательно сложим

start\_time = omp\_get\_wtime();

for (int i = 0; i < N; ++i)

{

x = dx \* (i + 0.5);

res += f(x);

}

end\_time = omp\_get\_wtime();

sequental\_res = res \* dx;

// параллельно умножим

if (parallel)

start\_time = omp\_get\_wtime();

// используем parallel for с опцией reduction

#pragma omp parallel for private(x) shared(dx) num\_threads(k) reduction(+: res\_omp) if(parallel)

for (int i = 0; i < N; ++i)

{

x = dx \* (i + 0.5);

res\_omp += f(x);

}

if (parallel)

end\_time = omp\_get\_wtime();

parallel\_res = res\_omp \* dx;

return end\_time-start\_time;

}

// вывод на экран времени

void print\_time(int n, int k)

{

// func format is (bool parallel, int k, int N)

// Time table

cout << left << fixed << setprecision(8)

<< setw(11) << n << setw(12) << time(0, k, n) << setw(15)

<< time(1, k, n) << setw(18) << sequental\_res << setw(15) << parallel\_res << endl;

}

// точка входа

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

{

cout << "Кащенко В. А. ПрИб-181\n\"Число Pi\" (параллельная версия)\n\n" <<

setw(11) << "Кол-во " << setw(20) << " Послед-но " << setw(20) << "Паралл- но " << setw(22) <<

" Р-ат послед-но " << setw(15) << " Р-ат паралл-но " << endl;

cout << "2 потока:" << endl;

print\_time(1000000000, 2);

cout << "4 потока:" << endl;

print\_time(1000000000, 4);

cout << "6 потоков:" << endl;

print\_time(1000000000, 6);

cout << "8 потоков:" << endl;

print\_time(1000000000, 8);

cout << "10 потоков:" << endl;

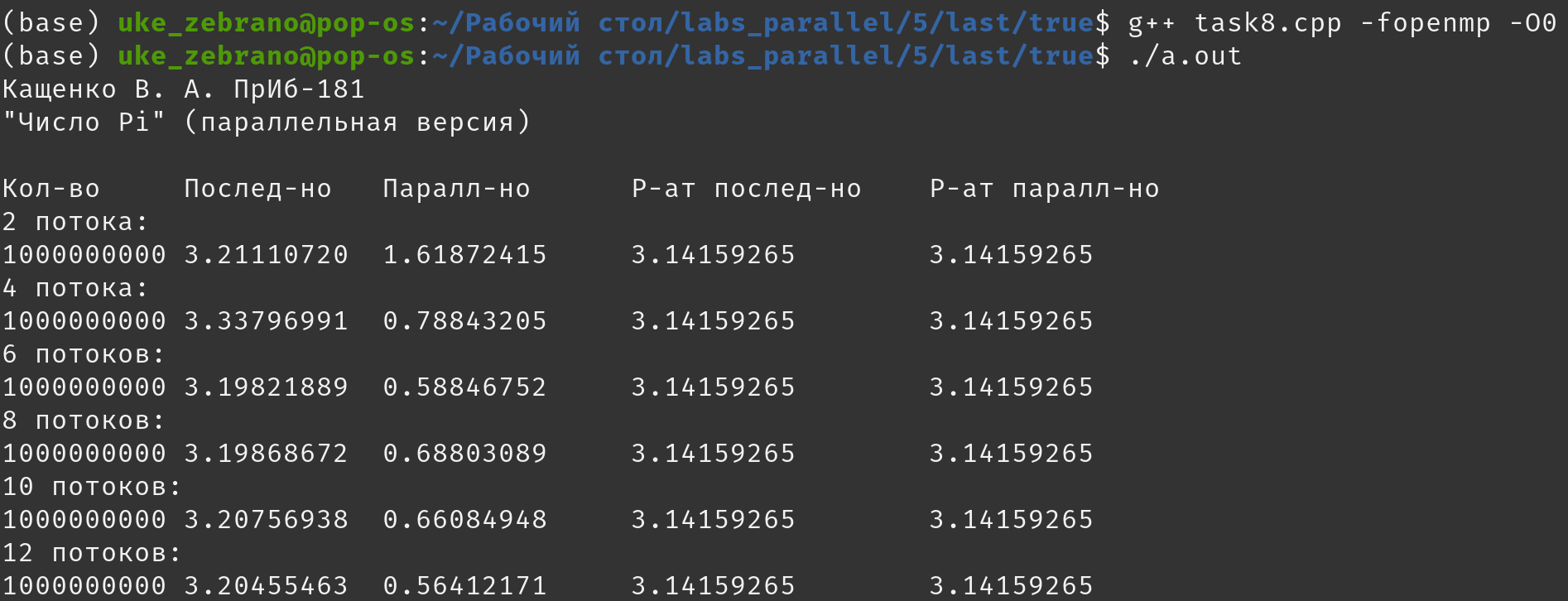
print\_time(1000000000, 10);

cout << "12 потоков:" << endl;

print\_time(1000000000, 12);

}

Работа программы:

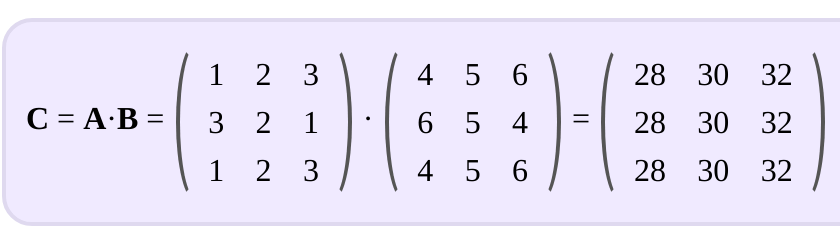


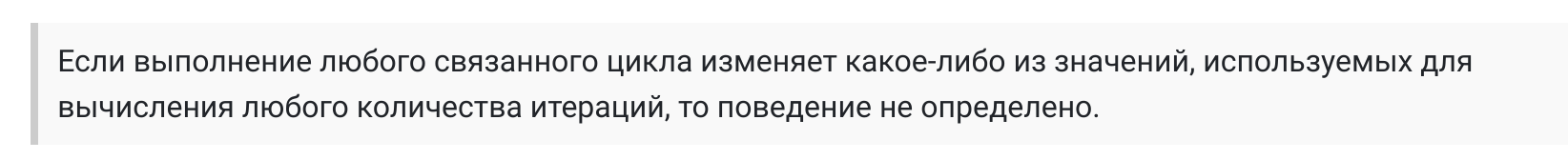
Исходя из результатов исследования, видно, что скорость растет при 2 → 6 потоках. Потом при 8→10 потоках она падает, а при 12 выдаёт самый быстрый результат.

Думаю, объяснить такое можно созданием накладных расходов при организации параллелизма на большом количестве потоков. При 12 результат должен быть более быстрым, просто накладные расходы делают своё дело.

Задание 9, программа «Матрица»:

Привычную таблицу не получается создать, так как нужно вручную вводить матрицы, ограничимся замером времени при каждом запуске программы и в качестве входных данных возьмём следующие:

  
При решении данной задачи нужно учитывать, что:

  
поэтому будем использовать collapse. (в соответствии со спецификацией OpenMP)

Исходный код программы:

#include <stdio.h>

#include <omp.h>

// матрица

double\*\* new\_matrix(int n)

{

double\*\* matrix;

matrix = new double\*[n];

for(int i = 0; i < n; ++i)

matrix[i] = new double[n];

return matrix;

}

// чтение матрицы с клавиатуры

void read\_matrix(double\*\* matrix, int n)

{

for(int i = 0; i < n; ++i)

for(int j = 0; j < n; ++j)

scanf("%lf", &matrix[i][j]);

}

// вывод матрицы

void print\_matrix(double\*\* matrix, int n)

{

for(int i = 0; i < n; ++i)

{

for(int j = 0; j < n; ++j)

printf("%lf ", matrix[i][j]);

printf("\n");

}

}

// точка входа

int main()

{

double t1, t2;

int n, k, intBool;

printf("0 - последовательно | !0 - параллельно\n");

scanf("%d", &intBool);

if (intBool)

{

printf("параллельная версия\n");

printf("потоки\n");

// количество потоков

scanf("%d", &k);

}

printf("Размер матриц\n");

// размер матриц

scanf("%d", &n);

double\*\* A = new\_matrix(n);

double\*\* B = new\_matrix(n);

double\*\* C = new\_matrix(n);

read\_matrix(A, n);

read\_matrix(B, n);

t1 = omp\_get\_wtime();

if (!intBool)

{

for (int i = 0; i < n; ++i)

{

for (int j = 0; j < n; ++j)

{

C[i][j] = 0.0;

for (int k = 0; k < n; ++k)

C[i][j] += A[i][k] \* B[k][j];

}

}

}

// основной вычислительный блок (с collapse(2) для работы со связанными циклами)

#pragma omp parallel for shared(A, B, C) num\_threads(k) collapse(2) if (intBool)

for (int i = 0; i < n; ++i)

{

for (int j = 0; j < n; ++j)

{

C[i][j] = 0.0;

for (int k = 0; k < n; ++k)

C[i][j] += A[i][k] \* B[k][j];

}

}

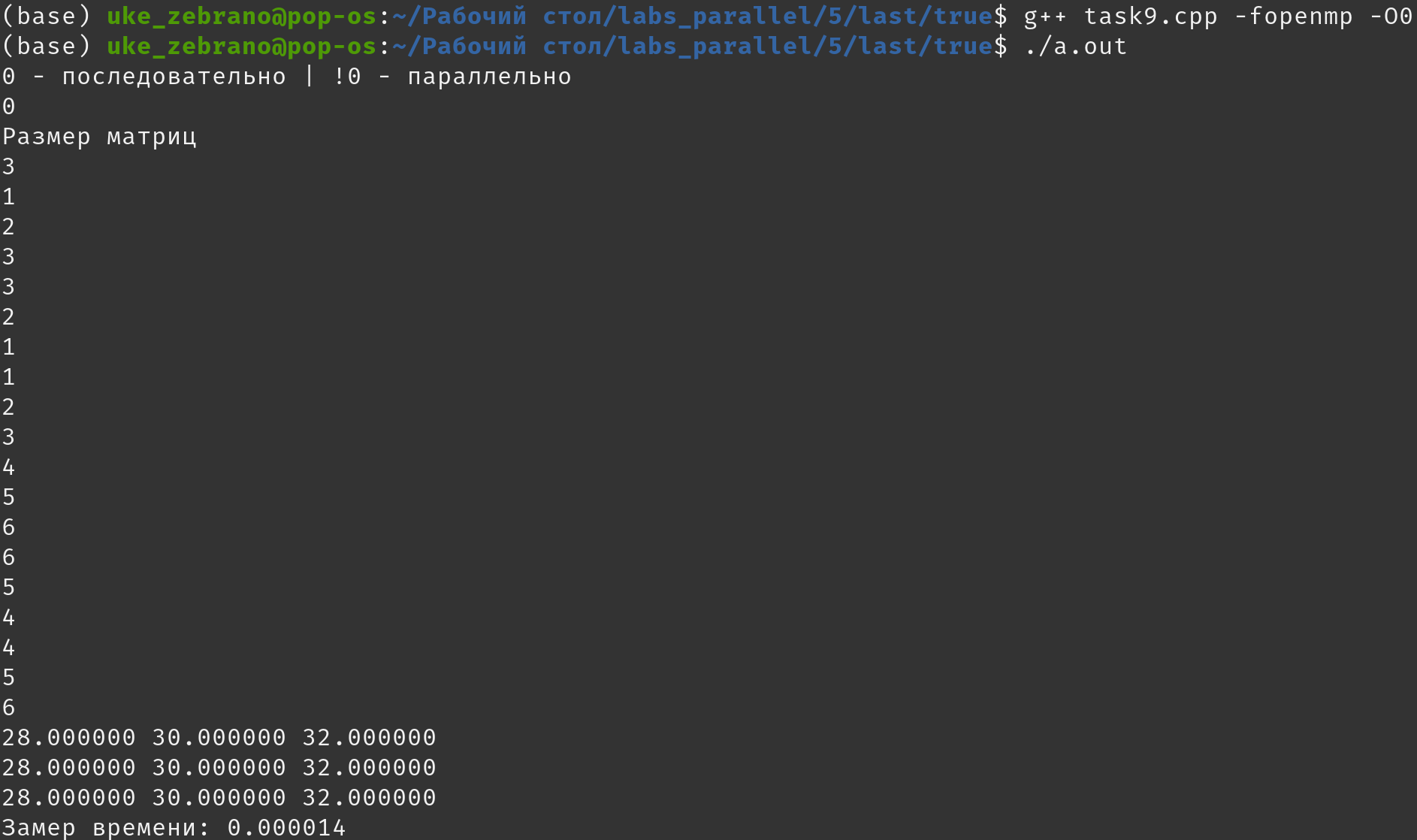
t2 = omp\_get\_wtime();

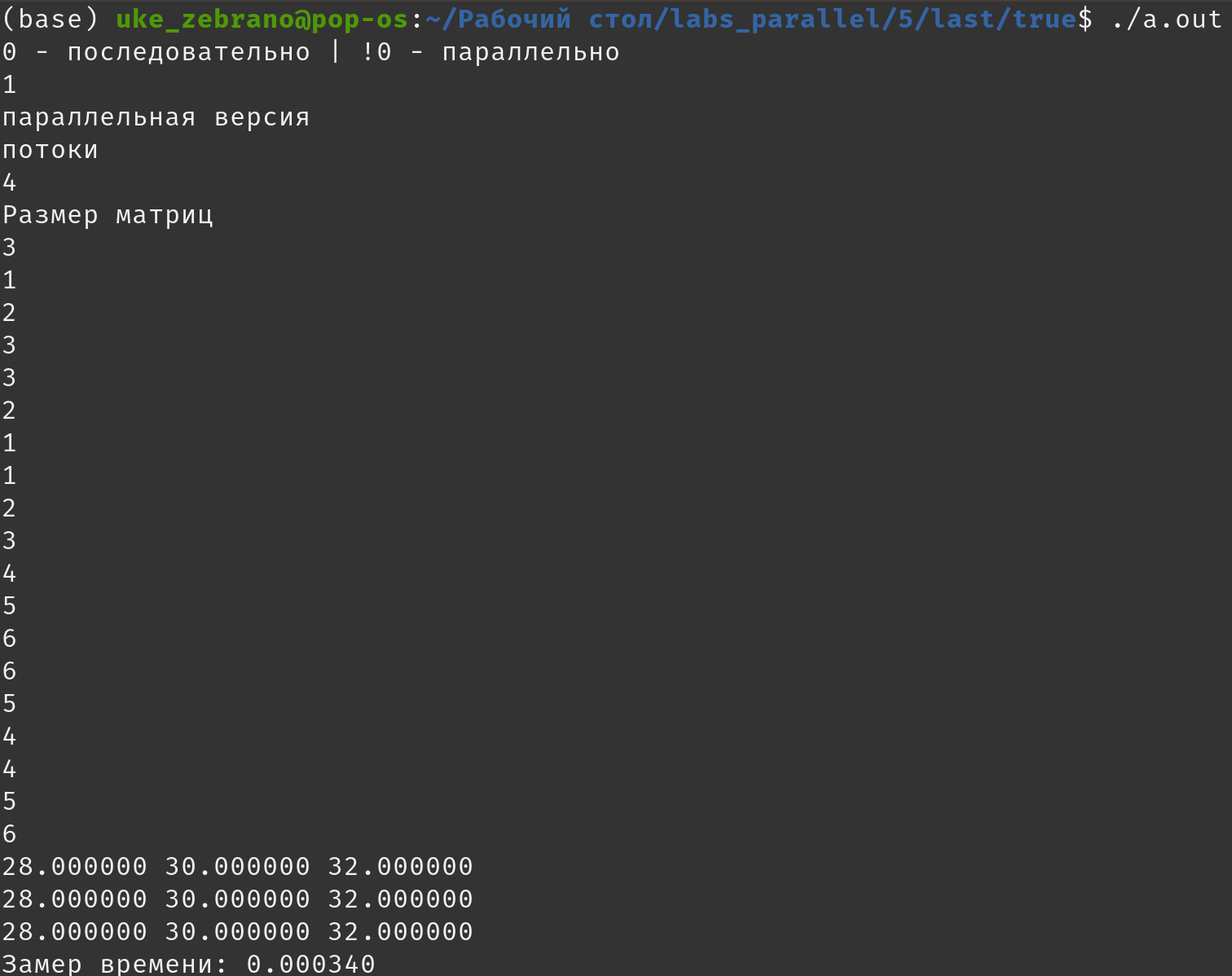
print\_matrix(C, n);

printf("Замер времени: %lf\n", t2 - t1);

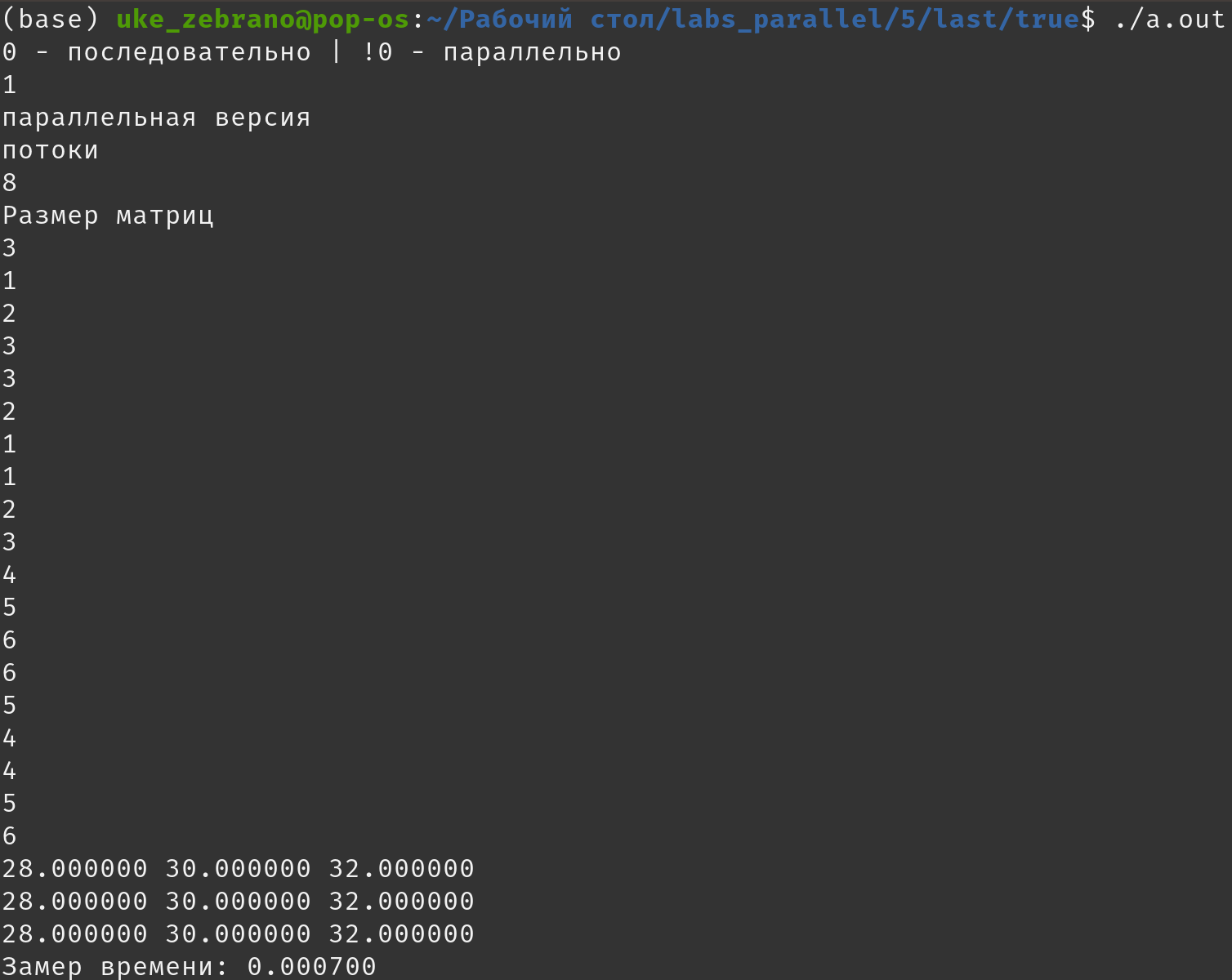
}

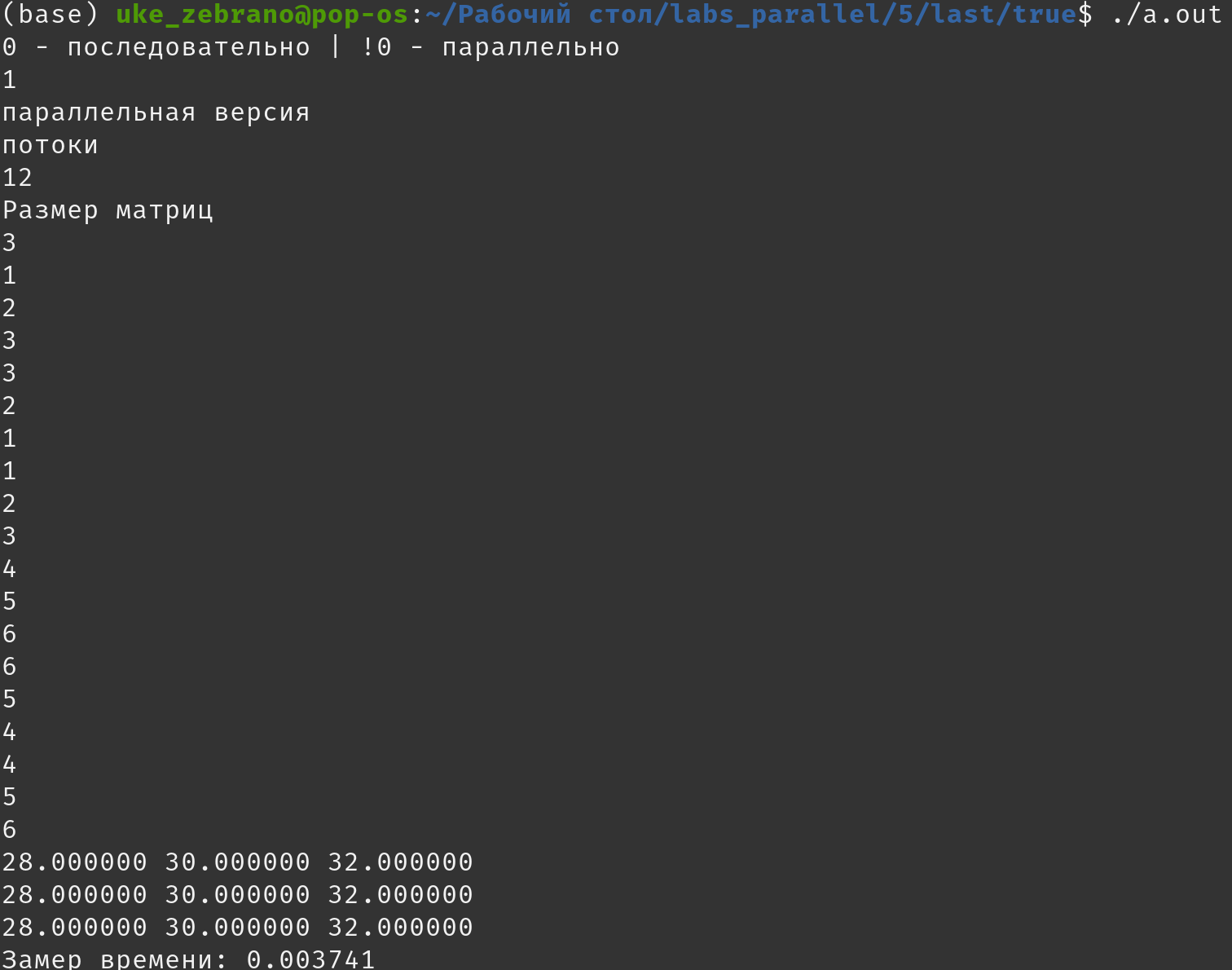
Работа программы (замеры):  
(последовательная версия)

  
(параллельная версия, 4 потока)



(параллельная версия, 8 потоков)

  
(параллельная версия, 12 потоков)



Тут видно, что последовательная версия отработала быстрее любой параллельной реализации, причем видно заметное замедление при добавлении числа потоков. Объяснить это можно тем, что:

Во-первых, размер матриц мал для получения пользы от распараллеливания.

Во-вторых, при увеличении количества потоков увеличиваются и накладные расходы на организацию параллелизма.

Исходя из всего этого можно сделать промежуточный вывод: польза от параллельного выполнения будет в случае большого количества данных. Под каждое конкретное количество данных нужно использовать соответствующее число нитей.

Вывод: Изучены методы распараллеливания циклов.