|  |  |  |
| --- | --- | --- |
| ПрИб-181 | Лабораторная работа №4 | Зачёт |
| Кащенко В. А. | Распределение работы |  |

Цель работы: Изучить распределение работы между имеющимися нитями.

Теоретические сведения:

(Конспект теоретических данных написан в тетради)

Задания (контрольные вопросы):

1. Вопрос: «Могут ли функции omp\_get\_thread\_num() и omp\_get\_num\_threads() вернуть одинаковые значения на нескольких нитях одной параллельной области?»

Ответ: ...\_num() не может, так как возвращает номер текущей нити, а ...\_threads() только одинаковые значения и могут вывести (общее количество потоков, задействованных в области).

2. Вопрос: «Можно ли распределить между нитями итерации цикла без использова-

ния директивы for?»

Ответ: Существуют способы и помимо for. Для распределения работ можно использовать параллельные секции sections и конструкцию single.

В цикле for по умолчанию барьером для потоков является конец цикла. Все потоки достигнув конца цикла дожидаются тех, кто еще не завершился, после чего основная нить продолжает выполняться дальше. Используя условие nowait для цикла можно разрешить основной нити не дожидаться завершения дочерних нитей.

Если нужно сделать действия, которые не являются итерациями цикла, то применяется секции sections. Для разрешения не ждать синхронизации можно использовать nowait.

Конструкция singngle используется, если действия должна выполнить одна нить.

3. Вопрос: «Можно ли одной директивой распределить между нитями итерации

сразу нескольких циклов?»

Ответ: Нет, на нити распределяется один цикл, но можно сразу запустить выполнение второго с распределением без задержек.

4. Вопрос: «Возможно ли, что при статическом распределении итераций цикла ни-

тям достанется разное количество итераций?»

Ответ: Такое возможно. Чтобы избежать подобного, можно использовать #pragma omp for schedule для распределения

5. Вопрос: «Могут ли при повторном запуске программы итерации распределяемого цикла достаться другим нитям? Если да, то при каких способах распределения итераций?»

Ответ: Да, могут, порядок нитей не определен, при всех способах распределения.

6. Вопрос: «Для чего может быть полезно указывать параметр chunk при способе

распределения итераций guided?»

Ответ: Это может быть полезно, если итерации имеет смысл выполнять одной нити несколько раз.

7. Вопрос: «Можно ли реализовать параллельные секции без использования директив sections и section?»

Ответ: \_Помимо параллельных секций есть ещё parallel construct (параллельный фрагмент, блок программы, управляемый директивой parallel. Именно параллельные фрагменты, совместно с параллельными областями, представляют параллельно-выполняемую часть программы) и параллельная область patallel region (параллельно выполняемые участки программного кода, динамически-возникающие в результате вызова функций из параллельных фрагментов).

8. Вопрос: «Как при выходе из параллельных секций разослать значение некоторой

локальной переменной всем нитям, выполняющим данную параллельную область?»

Ответ: Возможно использование lastprivate для этих целей, здесь происходит работа по переносу данных с последних витках.

9. Вопрос: «В каких случаях может пригодиться механизм задач?»

Ответ: Для выделения отдельной независимой задачи. (task) Задача может выполняться немедленно после создания или быть отложенной.

10. Вопрос(задание): «Напишите параллельную программу, реализующую скалярное произведение двух векторов.»

Ответ:

#include <stdio.h>

#include <omp.h>

#include <iostream>

#include <vector>

#include <random>

#include <algorithm>

#include <iterator>

#include <stdlib.h>

#include <iomanip>

using namespace std;

// Создание векторов для скалярного произведения

vector<int> randVec(size\_t size)

{

vector<int> v(size);

// генератор true-random-number

random\_device r;

// используем лямда-функцию generate()

// & фиксирует ссылку на локальный объект, чтобы видеть актуальное значение

generate(v.begin(), v.end(), [&] {return r();});

return v;

}

// Хранение результатов сложения

int sequental\_res, parallel\_res;

// замеры

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

{

double start\_time, end\_time;

// создание векторов

vector<int> v(randVec(n));

vector<int> v1(randVec(n));

// объявим переменные результатов произведения

int res = 0;

int res\_omp = 0;

// последовательно умножим

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

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

{

res += v[i] \* v1[i];

}

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

sequental\_res = res;

// параллельно умножим

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

// используем parallel for с опцией reduction (суммирование в res\_omp)

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

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

{

res\_omp += v[i] \* v1[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 threads)

{

// Time table

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

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

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

}

// точка входа

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

{

cout << "Кащенко В. А. ПрИб-181\nСкалярное произведение векторов (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;

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);

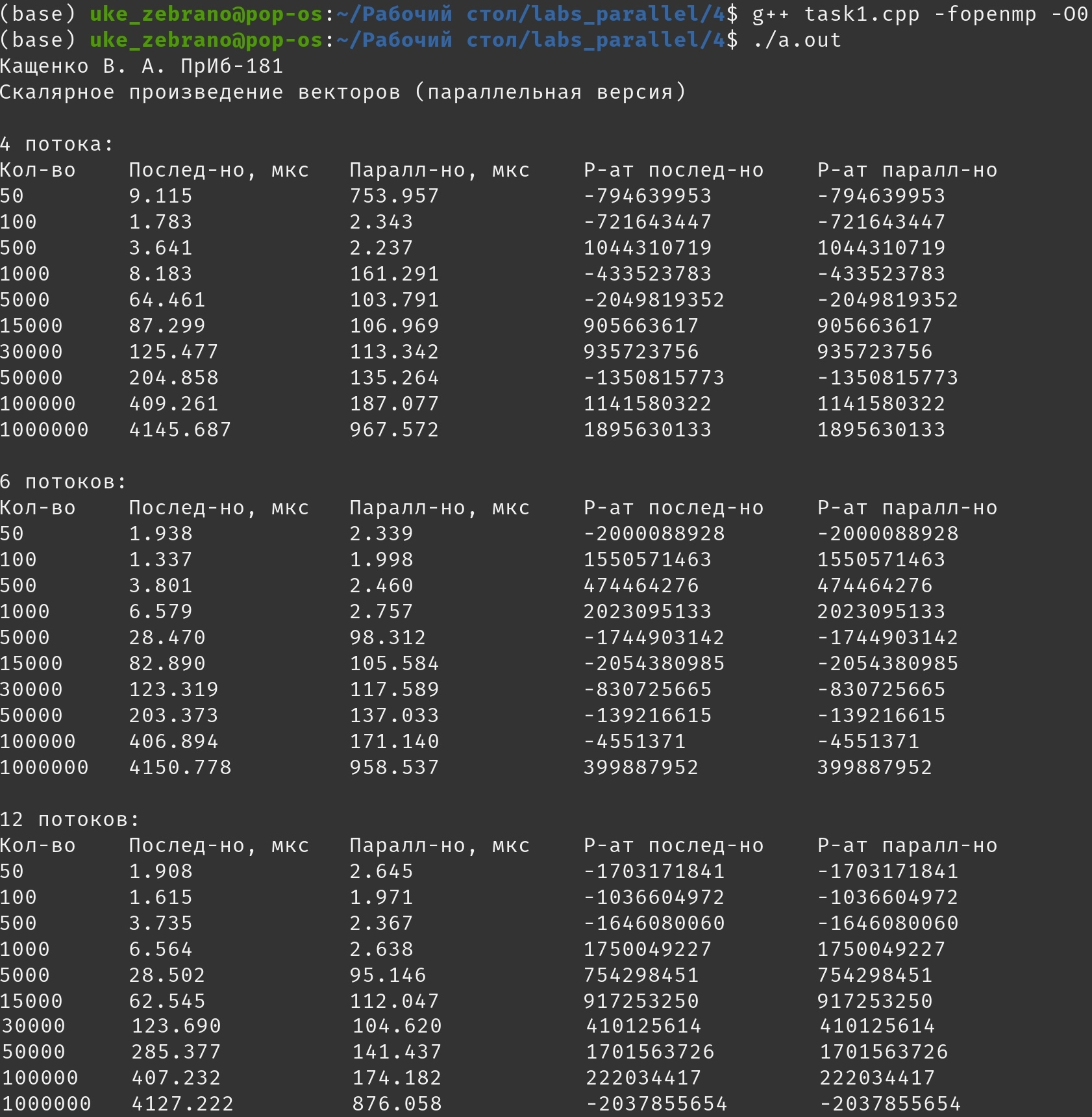
print\_time(50000, 12);

print\_time(100000, 12);

print\_time(1000000, 12);

}

Результат:

  
Пояснение:

- На экране терминала видно, что при малом числе итераций «распараллеливать» программу не имеет смысла.

- Выигрыш в производительности виден только начиная от 500 элементов массива. - Самая неэффективная конфигурация оказалась в 4 потока. Самая эффективная конфигурация оказалась в 12 потоков. (виден выигрыш параллельных вычислений)

11. Вопрос(задание): «Напишите параллельную программу, реализующую поиск максималь-

ного значения вектора.»

Ответ:

// time table

#include <stdio.h>

#include <omp.h>

#include <vector>

#include <random>

#include <algorithm>

#include <iterator>

#include <iostream>

#include <stdlib.h>

#include <iomanip>

using namespace std;

// Создание векторов для поиска

vector<int> randVec(size\_t size)

{

vector<int> v(size);

// генератор true-random-number

random\_device r;

// используем лямда-функцию generate()

// & фиксирует ссылку на локальный объект, чтобы видеть актуальное значение

generate(v.begin(), v.end(), [&] {return r();});

return v;

}

// Хранение результатов сложения

int sequental\_res, parallel\_res;

// замеры

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

{

double start\_time, end\_time;

// i - итератор

int i;

// создание векторов

vector<int> v(randVec(n));

//начальные значения векторов

int count = v[0];

int count1 = v[0];

// последовательный поиск

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

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

{

if (v[i] > count)

count = v[i];

}

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

sequental\_res = count;

// параллельный поиск

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

// общее - v; локальные - i, n

#pragma omp parallel shared(v) private(i, n)

{

/\* номер текущей нити \*/

n = omp\_get\_thread\_num();

// параллельный поиск

#pragma omp for

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

{

if (v[i] > count1)

count1 = v[i];

}

}

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

parallel\_res = count1;

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

}

// вывод на экран времени

void print\_time(int n, int threads)

{

// Time table

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

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

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

}

// точка входа

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

{

cout << "Кащенко В. А. ПрИб-181\nСкалярное произведение векторов (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;

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);

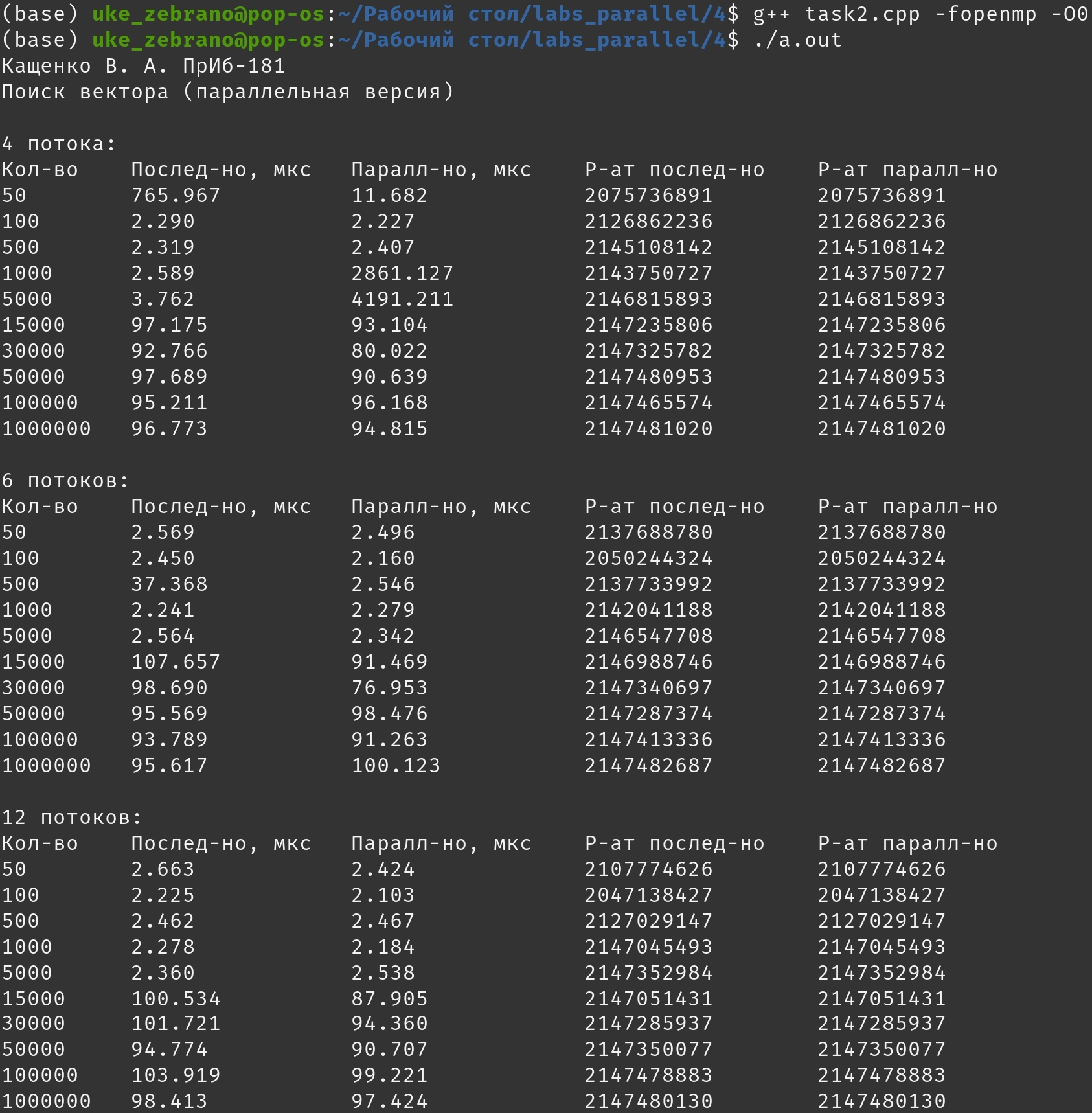
print\_time(50000, 12);

print\_time(100000, 12);

print\_time(1000000, 12);

}

Результат:

Пояснение:

- На экране терминала виден результат поиска векторов и время, которое показывает эффективность работы именно 12 потоков.

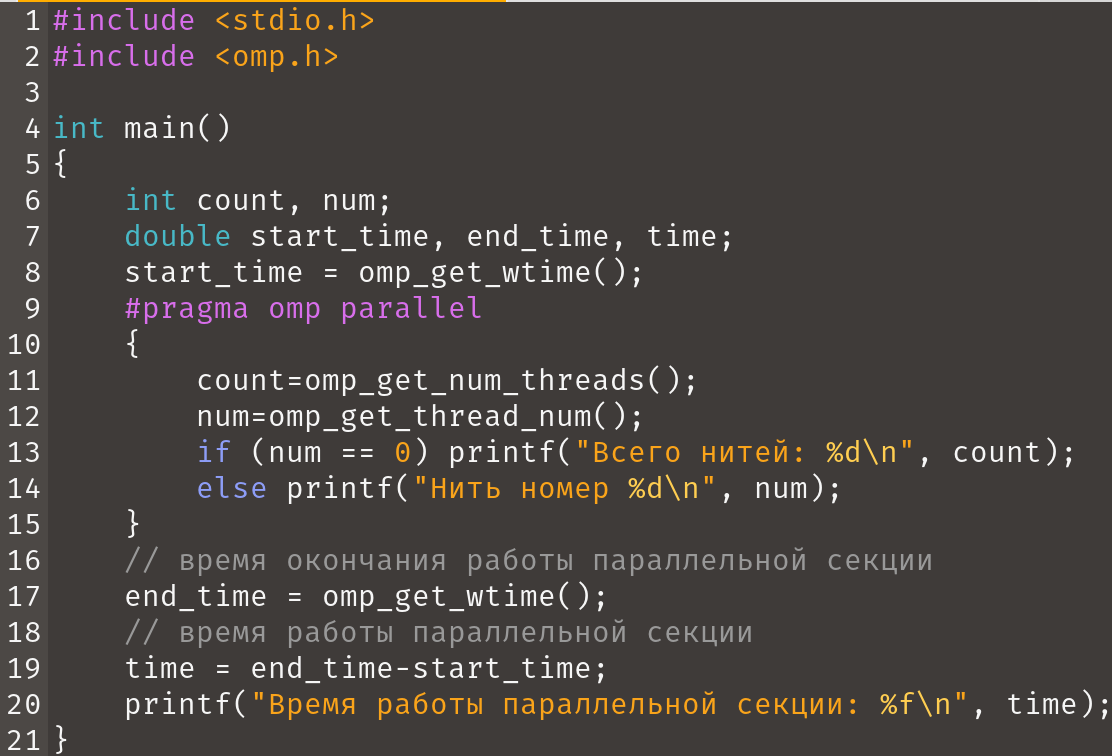
- Особого выигрыша в производительности не видно. Подобного рода задачи можно не распараллеливать вообще.

- Сильная разница видна только при 4-ех потоках. Подобная оценка появляется и при повторных замерах, что говорит о том, что это самая неэффективная конфигурация.

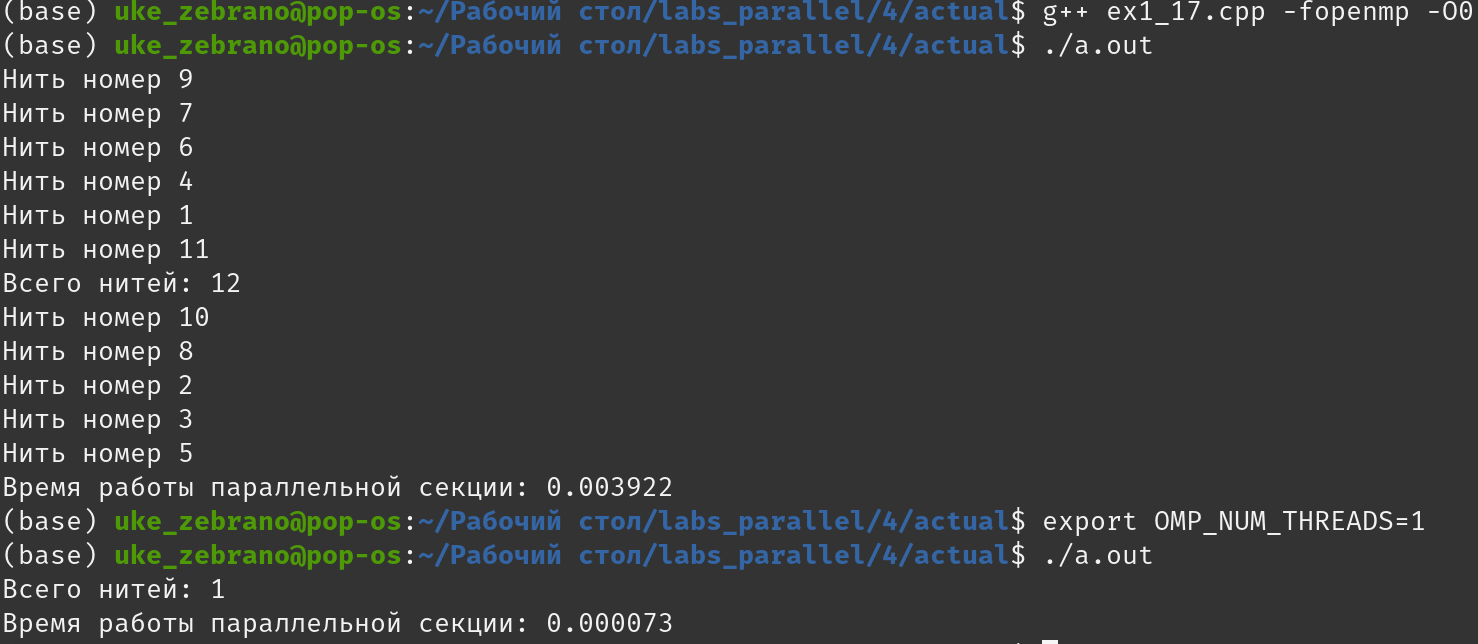
- Критически сильной разницы нет, можно использовать последовательные вычисления при решении подобного рода задач.

Практика (примеры из методички):

Пример 1 (в методичке обозначается как пример 17) демонстрация работы функций omp\_get\_num\_threads() и omp\_get\_thread\_num():

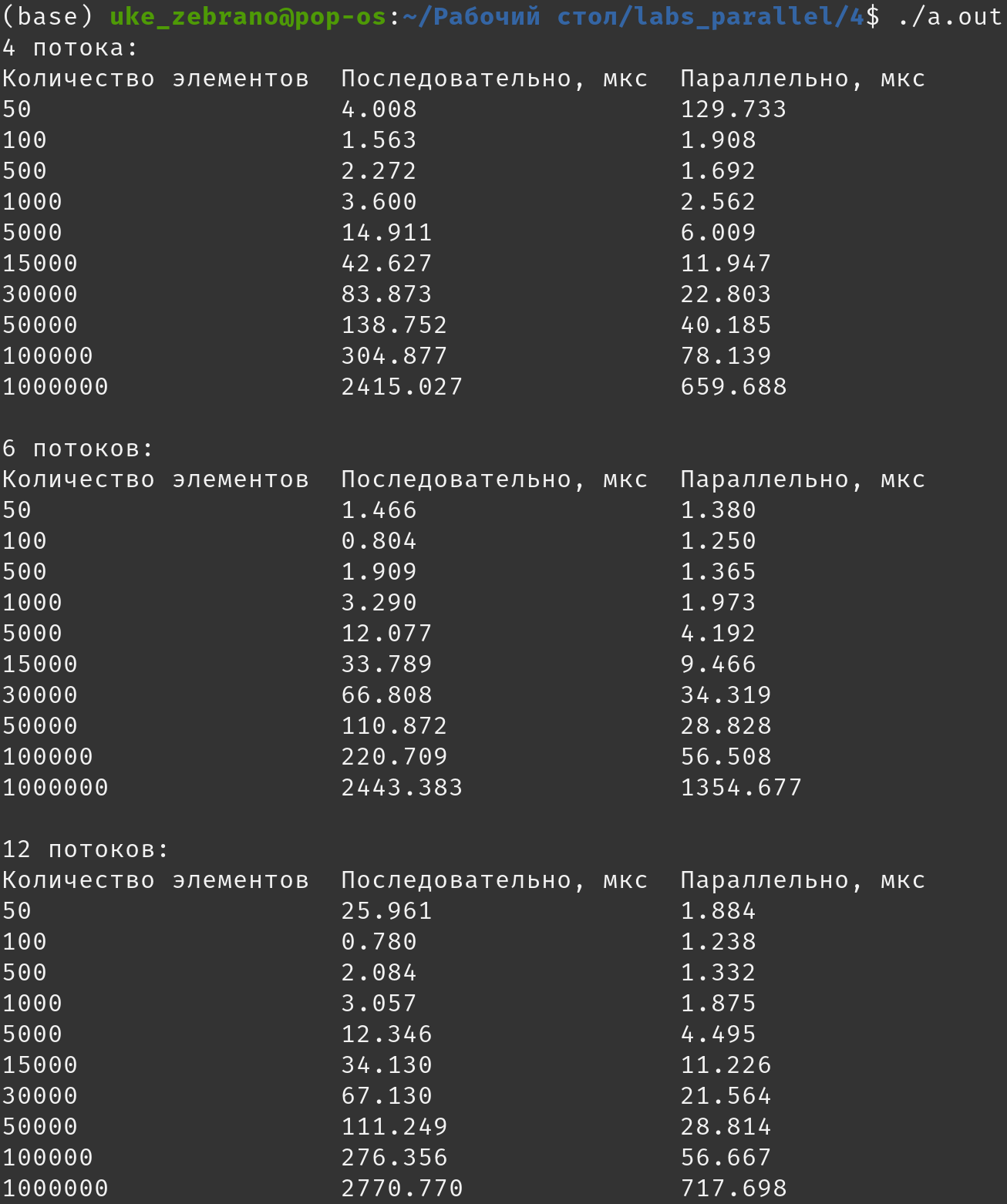


Предполагаемое поведение программы: Master-нить (0) выведет количество нитей, а остальные нити выведут свой номер. (порядок не определен) Можно заметить это в консоли:



Мы видим время работы 12-ти нитей. Затем количество потоков снижается до одной нити и мы видим совершенно логичный результат — последовательная версия программы отработала быстрее. Это легко объясняется тем, что работа 12-ти потоков с printf() занимает больше времени, чем работа одной единственной нити-Master.

Пример 2 (в методичке обозначается как пример 18) Директива for:



Код программы (ex2\_18.cpp):

#include <stdio.h>

#include <omp.h>

#include <malloc.h>

#include <stdlib.h>

#include <iostream>

#include <iomanip>

using namespace std;

// замеры

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

{

if (threads <= omp\_get\_max\_threads())

{

omp\_set\_num\_threads(threads);

}

double start\_time, end\_time;

int \*A = static\_cast<int \*>(malloc(n \* sizeof(int)));

int \*B = static\_cast<int \*>(malloc(n \* sizeof(int)));

int \*C = static\_cast<int \*>(malloc(n \* sizeof(int)));

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

{

A[i] = i;

B[i] = 2 \* i;

C[i] = 0;

}

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

#pragma omp parallel for shared(A, B, C) if(parallel)

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

C[i] = A[i] \* B[i];

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

free(A);

free(B);

free(C);

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

}

// вывод на экран времени

void print\_time(int n, int threads)

{

cout << left << fixed << setprecision(3) << setw(22) << n << setw(22) << time(0, n, threads) << setw(16) << time(1, n, threads) << setw(16) << endl;

}

// точка входа

int main()

{

#ifdef \_OPENMP

cout << "4 потока:\n";

cout << setw(22) << "Количество элементов " << setw(13) << "Последовательно, мкс " << setw(16) << "Параллельно, мкс" << end l;

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";

cout << setw(22) << "Количество элементов " << setw(13) << "Последовательно, мкс " << setw(16) << "Параллельно, мкс" << 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";

cout << setw(22) << "Количество элементов " << setw(13) << "Последовательно, мкс " << setw(16) << "Параллельно, мкс" << 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);

#else

printf ("Последовательная версия, демонстрация параллелизма невозможна\n");

#endif

}

Предполагаемое поведение программы:

Инициализация трех массивов A, B, C (общие для параллельной области + логическая переменная, определяющая параллельность/последовательность). Каждая нить присвоит переменной n номер. Далее цикл for с распределением итераций. На каждой i-ой итерации сложение i-ых элементов массивов A и B в i-ый элемент массива C. Для работы с большим количеством элементов (для замеров времени) использовался malloc.

Вывод по работе данной программы:

- Малое число элементов распараллелить дольше (50 против 100, к примеру), и выигрыша от параллелизма мы не получим. (потери производительности для организации параллелизма). На четырёх потоках это заметно очень сильно. 6 и 12 потоков тоже теряют в скорости, но не так сильно;

- на большом размере массива скорость параллелизма падает (это ожидаемо, различие на порядок достаточно велико);

- среднее время параллелизма (мкс) для четырех потоков: 95,42 мс

для шести потоков: 149,34 мс

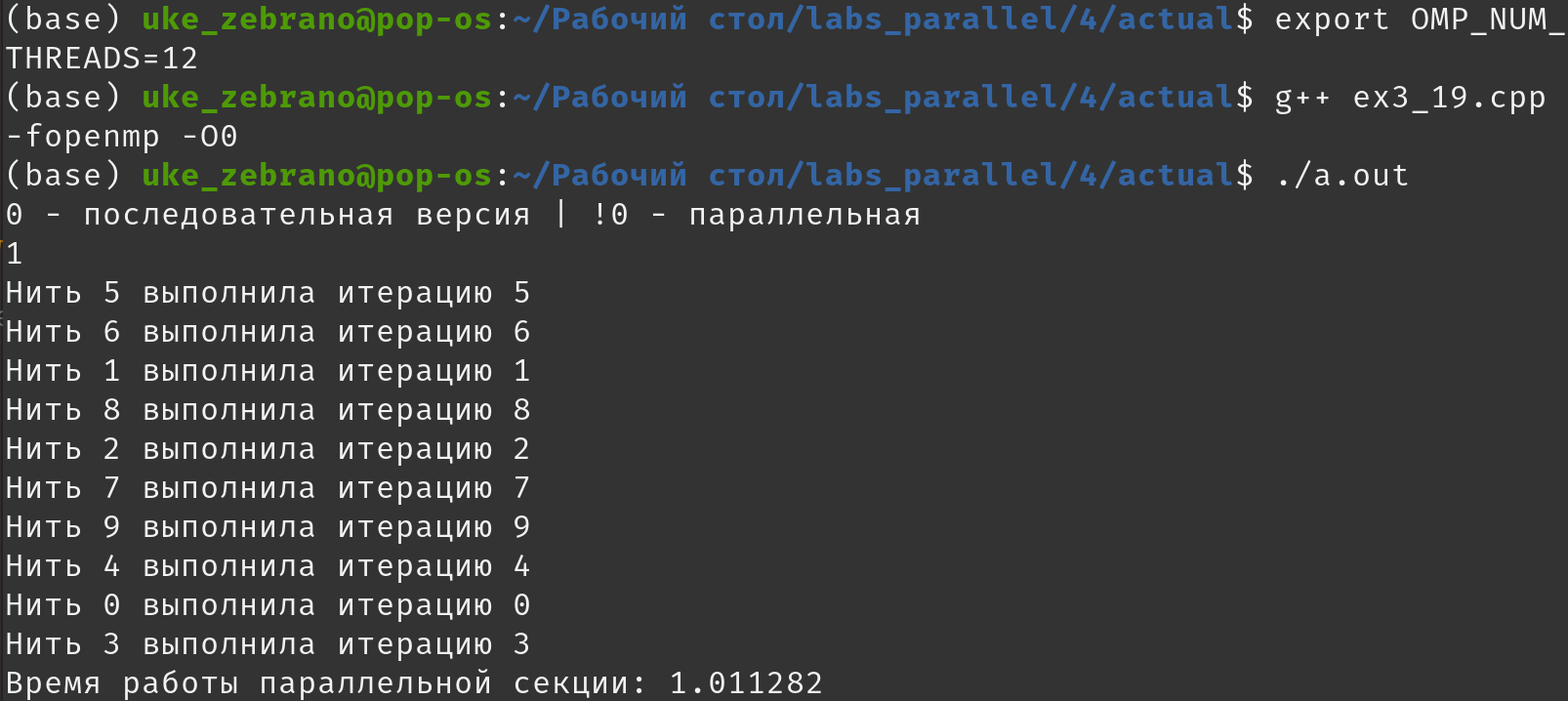
для двенадцати потоков: 84,63 мс

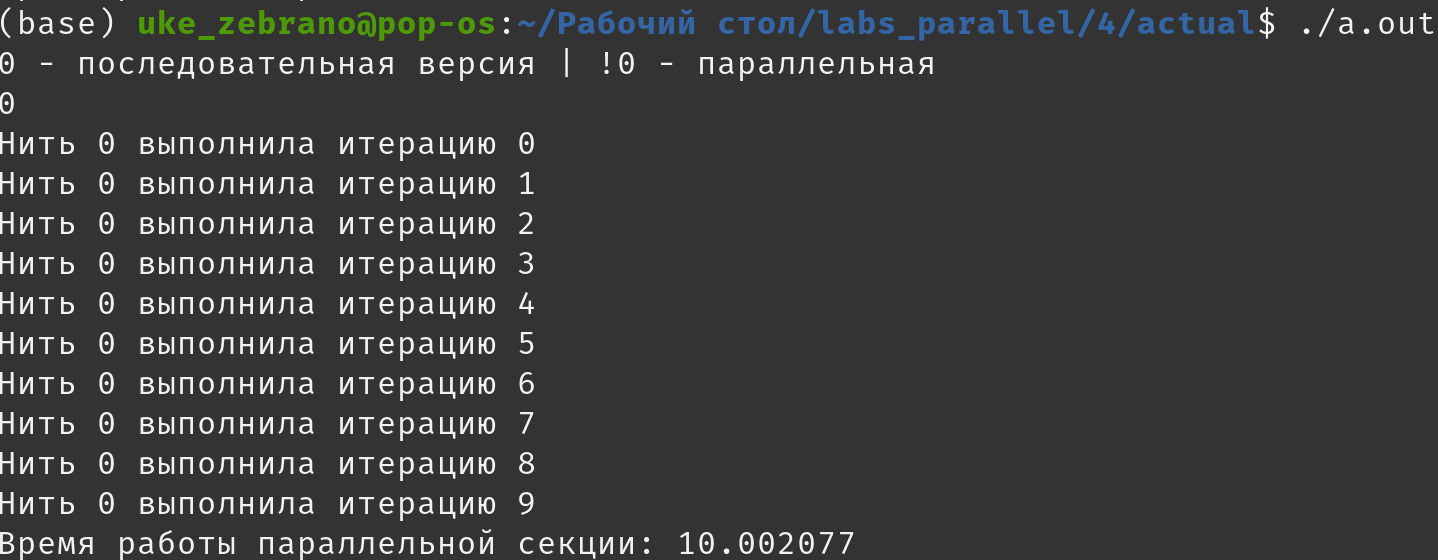
Это говорит нам о том, что не всегда увеличение количества потоков даст выигрыш в скорости. 6 потоков — самая неэффективная реализация параллелизма.

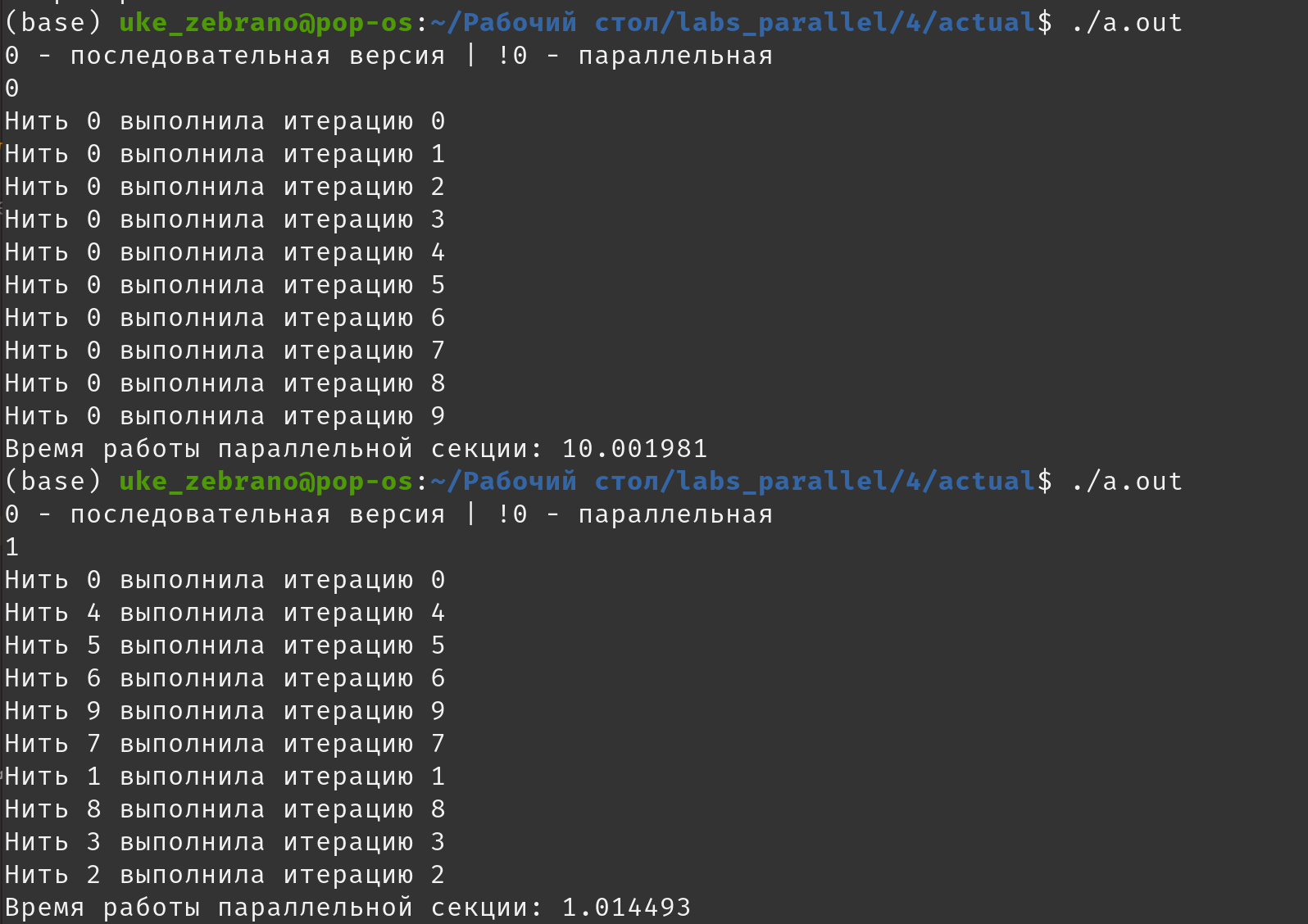
Пример 3 (в методичке обозначается как пример 19) Опция schedule

Обзор работы директив:

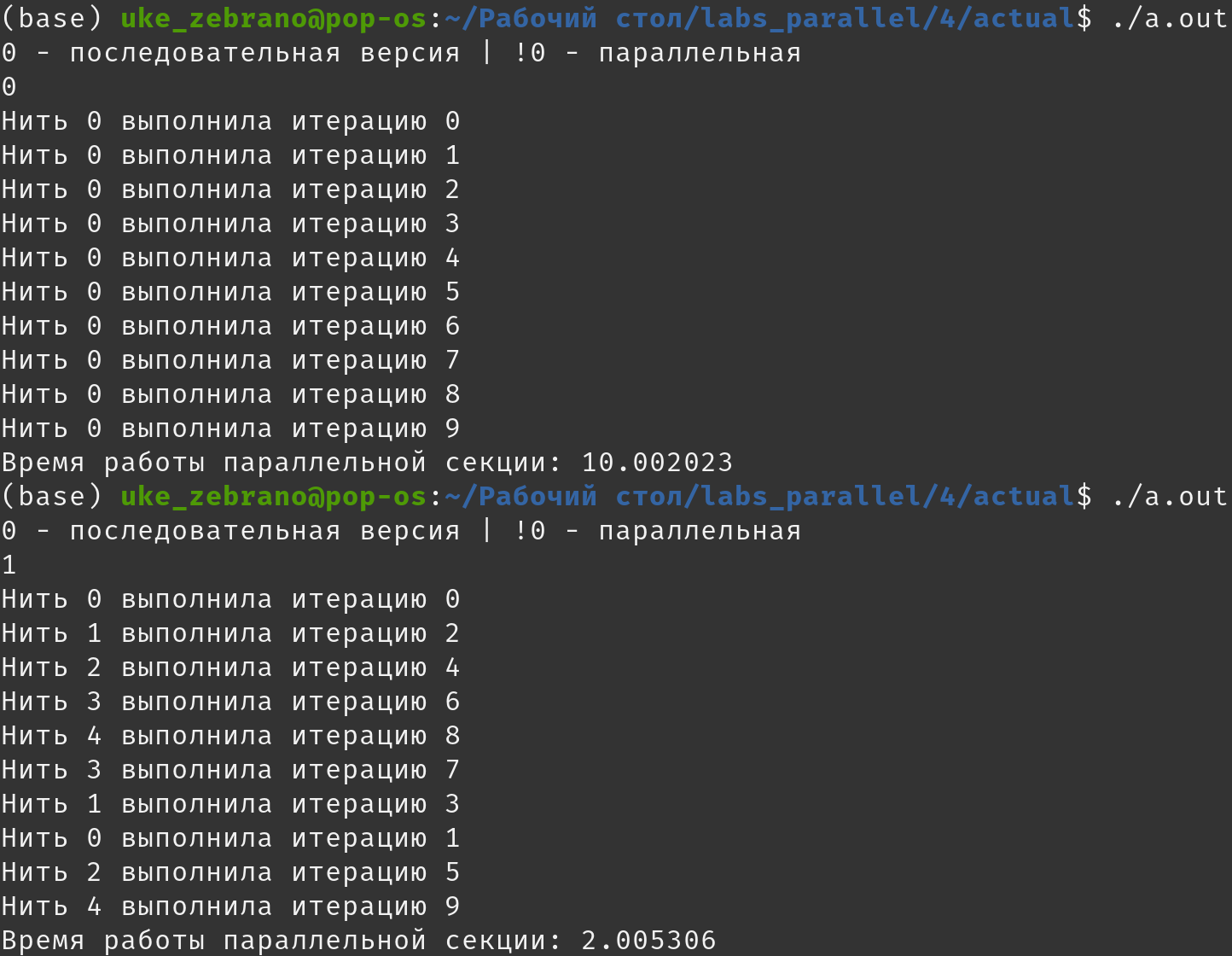
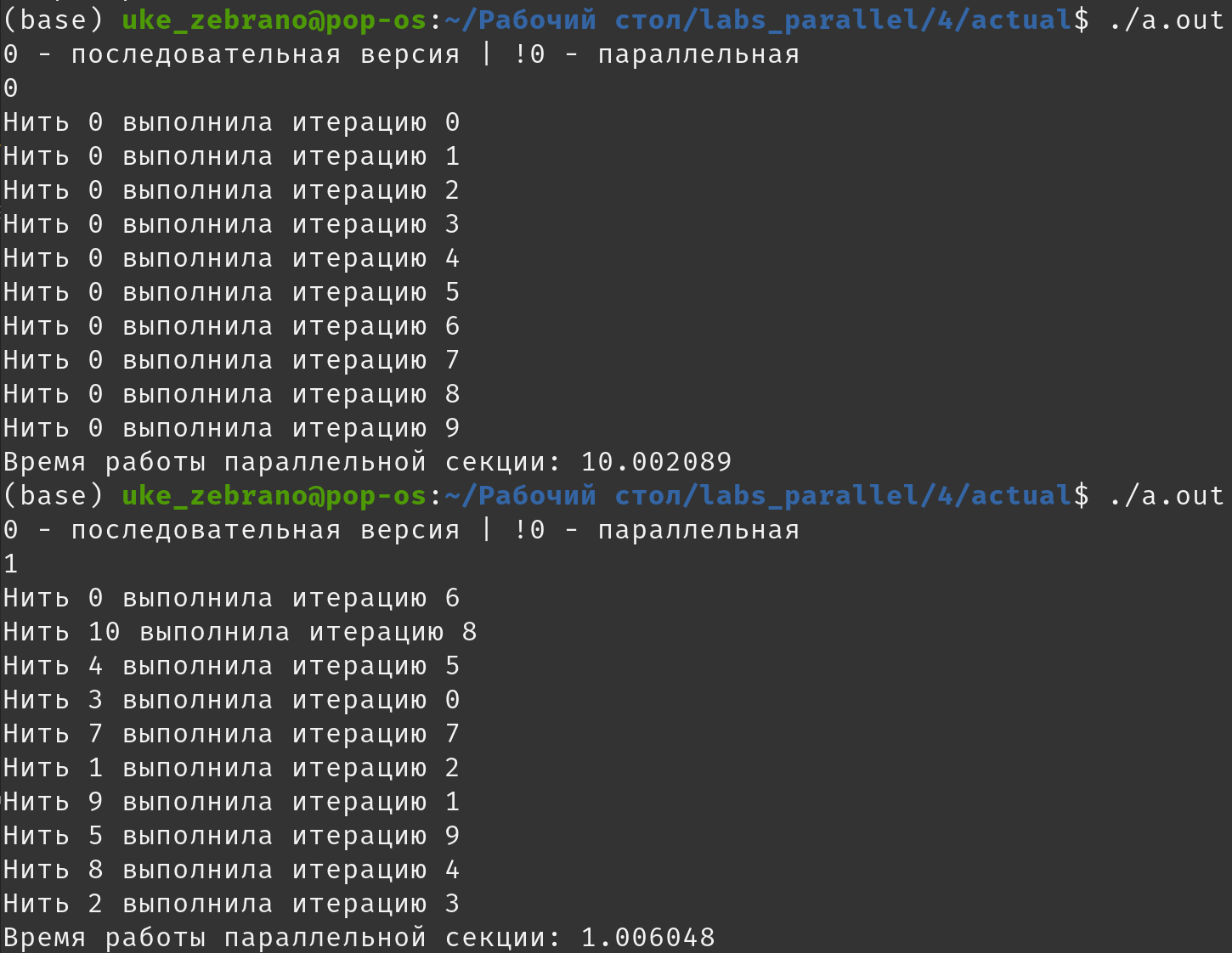
#pragma omp for schedule (static)



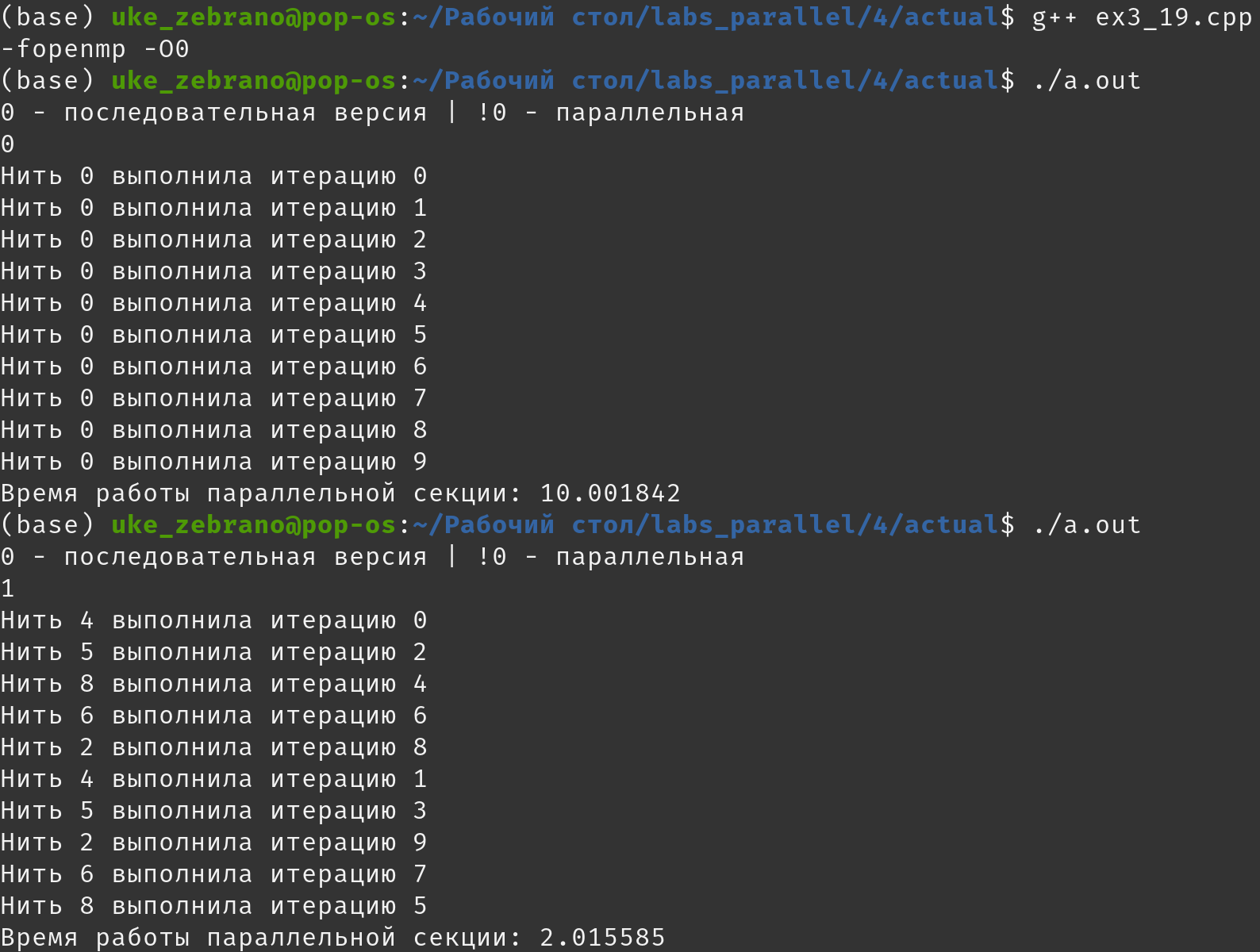
  
#pragma omp for schedule (static, 1)



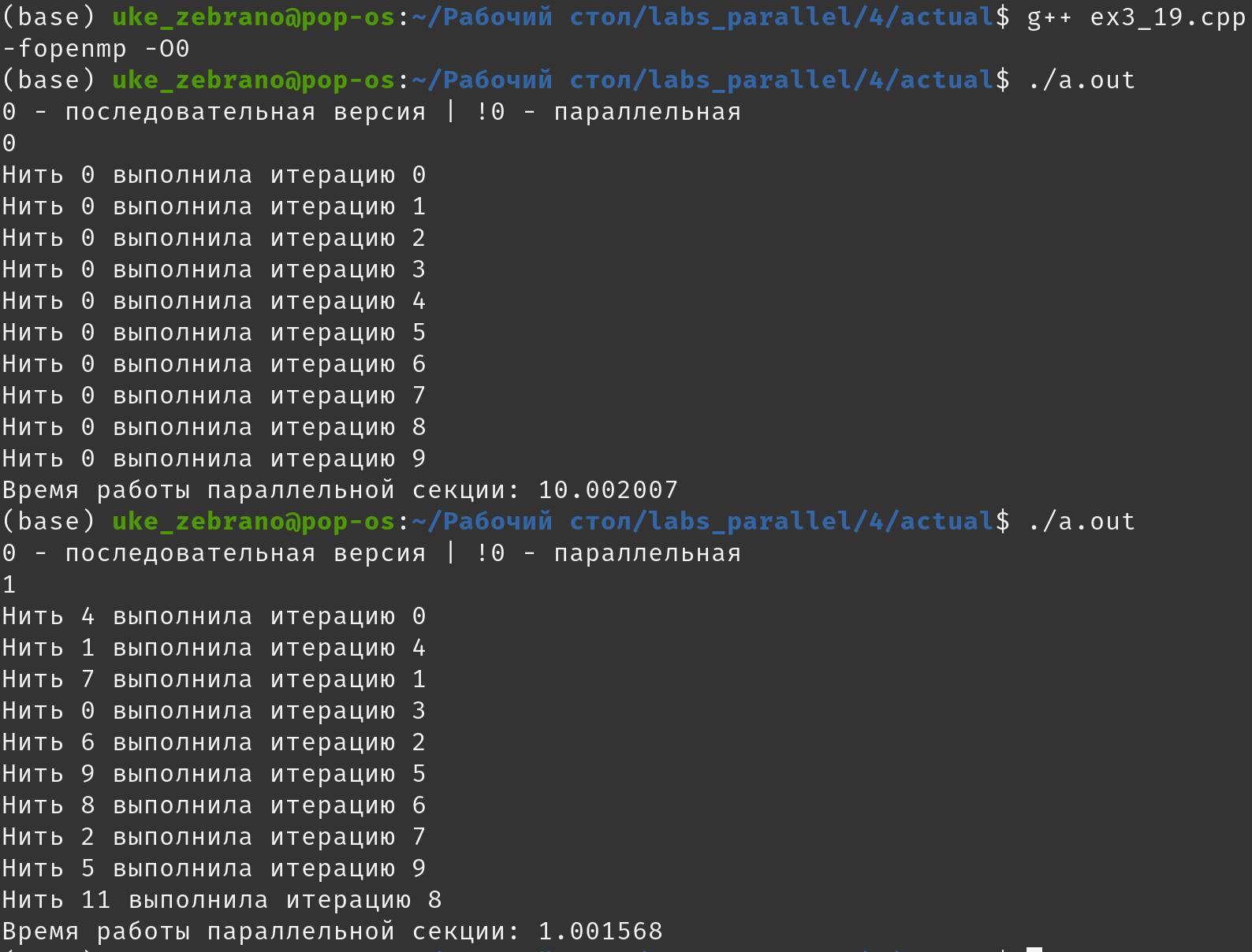
#pragma omp for schedule(static, 2)

  
#pragma omp for schedule (dynamic)

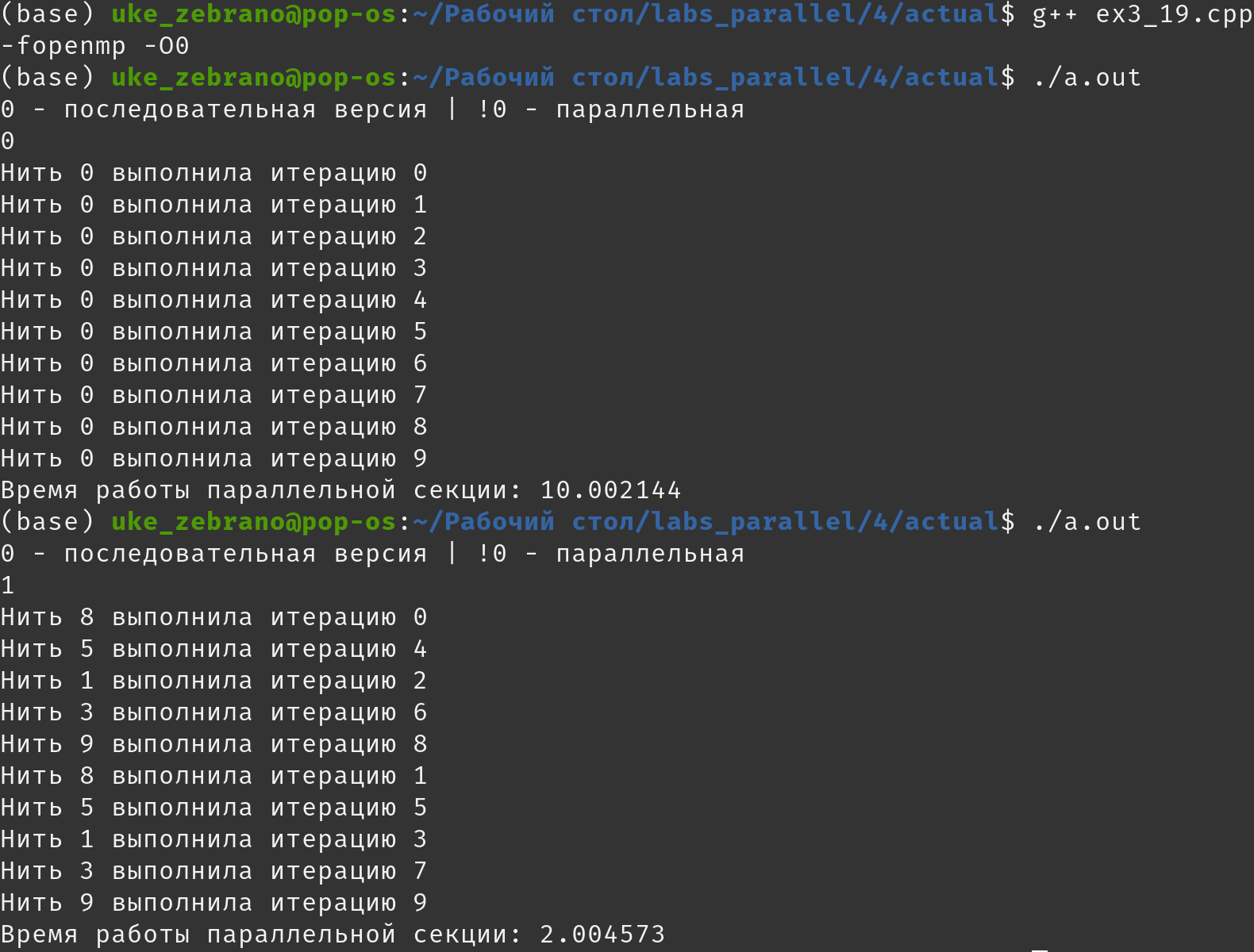
#pragma omp for schedule (dynamic, 2)



#pragma omp for schedule (guided)



#pragma omp for schedule (guided, 2)

  
В параллельной области выполняется цикл, итерации которого распределяются между существующими нитями. На каждой итерации будет напечатано, какая нить выполнила данную итерацию. В тело цикла вставлена также задержка, имитирующая некоторые

вычисления. Код можно посмотреть ниже. Здесь последовательные версии ожидаемо медленнее, чем параллельные, и скорость их всех в районе 10 секунд. Можно сказать, что директивы schedule не влияют на последовательные вычисления.

Параллельные же версии показали различный результат.

Расписание определяет, как итерации цикла распределяются между потоками. Выбор правильного расписания может сильно повлиять на скорость работы приложения.

Static — в начале цикла решается, какой поток будет работать со значениями на конкретной итерации.

Dynamic — нить для вычислений следующей итерации выбирается «на лету», что может быть полезно, если вычисления занимают разное количество нитей.

Static – итерации делятся на блоки по размер итераций и статически разделяются между потоками (в начале цикла);

Dynamic – распределение итерационных блоков осуществляется динамически (по умолчанию размер=1) Нить для вычислений следующей итерации выбирается «на лету», что может быть полезно, если вычисления занимают разное количество нитей.

Guided – размер итерационного блока уменьшается экспоненциально при каждом распределении; размер определяет минимальный размер блока (по умолчанию размер (chunk) =1)

Отлия #pragma omp for schedule (static), #pragma omp for schedule (static, 1), #pragma omp for schedule (static, 2)

Тут всё просто, числа(chunk) — это количество итераций, получаемое конкретной нитью.

Если значение chunk не указано, то всё множество итераций делится на непрерывные куски примерно одинакового размера (конкретный способ зависит от реализации).

Видно, что чем больше итераций выполняет одна нить,тем дольше выполняется программа. Без указания chunk на текущей машине побеждает по скорости.

#pragma omp for schedule (dynamic) и #pragma omp for schedule (dynamic, 2)

Динамическое распределение итераций, но при указании chunk одна нить выполняет chunk итераций.

Здесь видно логичное уменьшение времени работы при chunk равном 2.

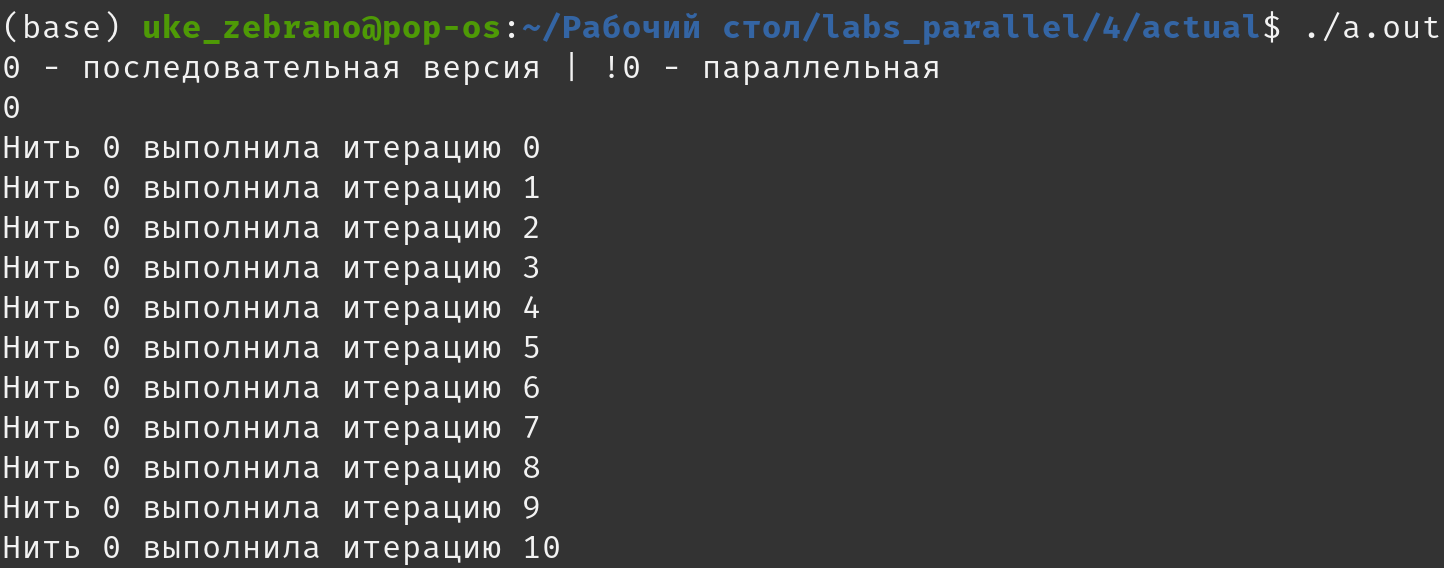
#pragma omp for schedule (guided) и #pragma omp for schedule (guided, 2)

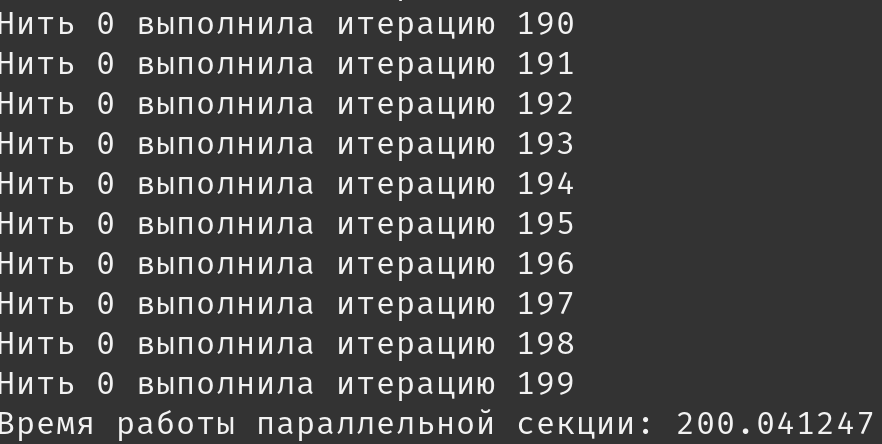
Здесь порции уменьшаются до величины chunk. При значении chunk = 2 программа работает медленнее.

Пример 4 (в методичке обозначается как пример 20) Опция schedule

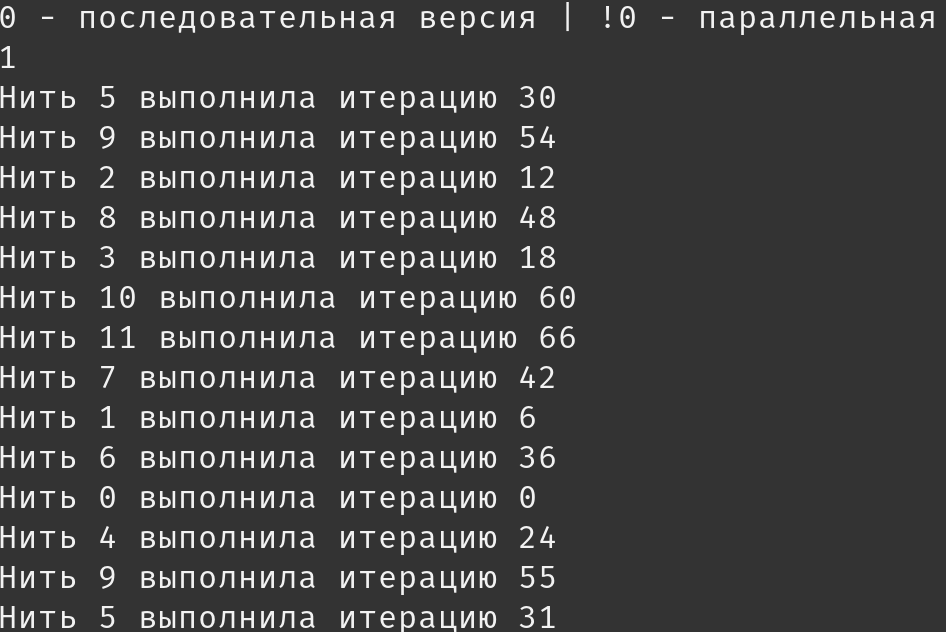
#pragma omp for schedule (static, 6)

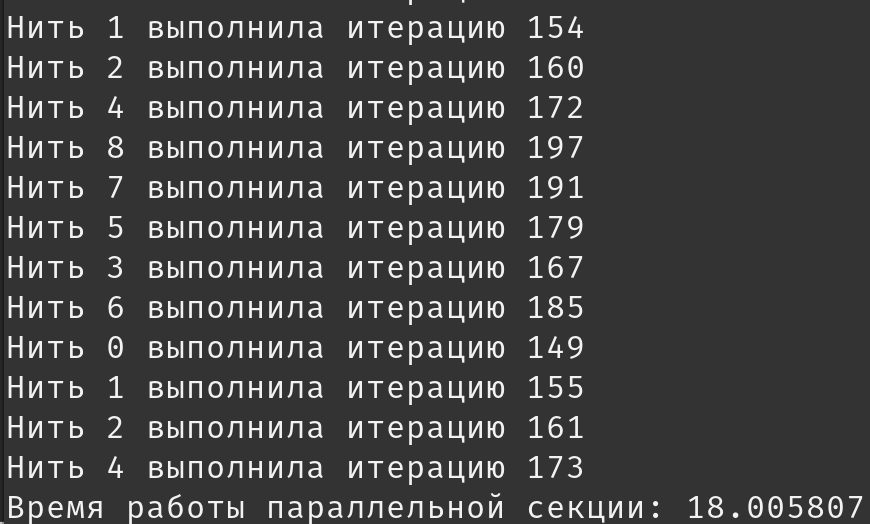
Последовательная версия

(\*\*\*)



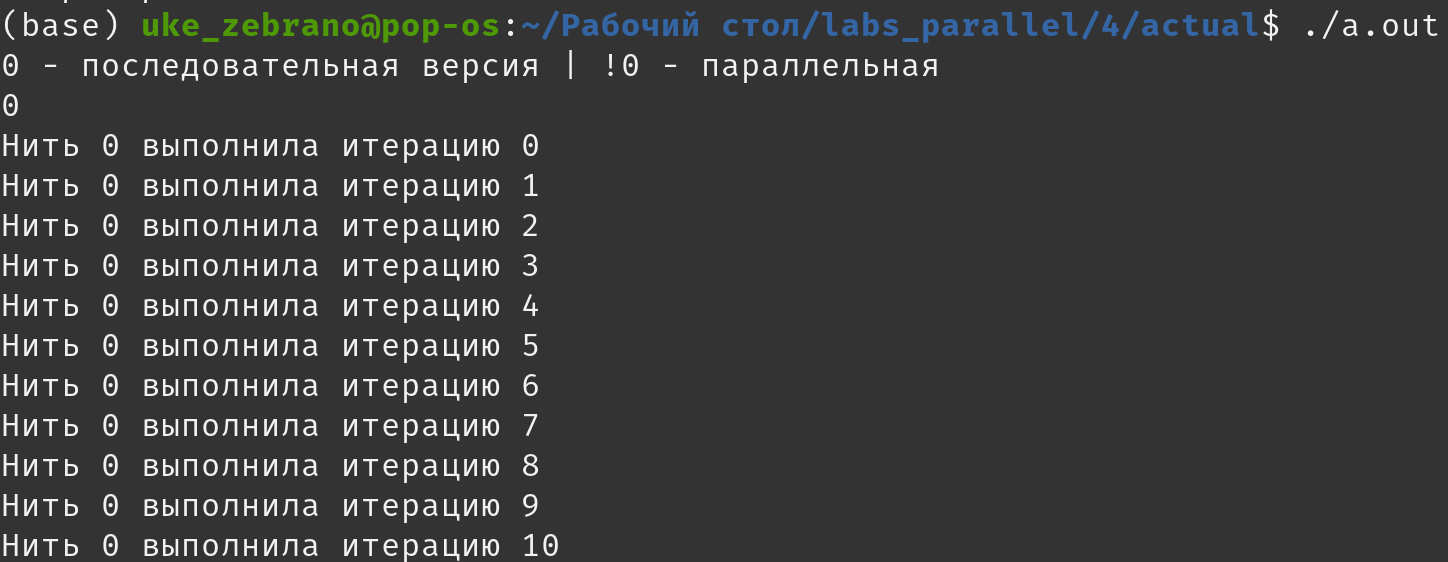
Параллельная версия

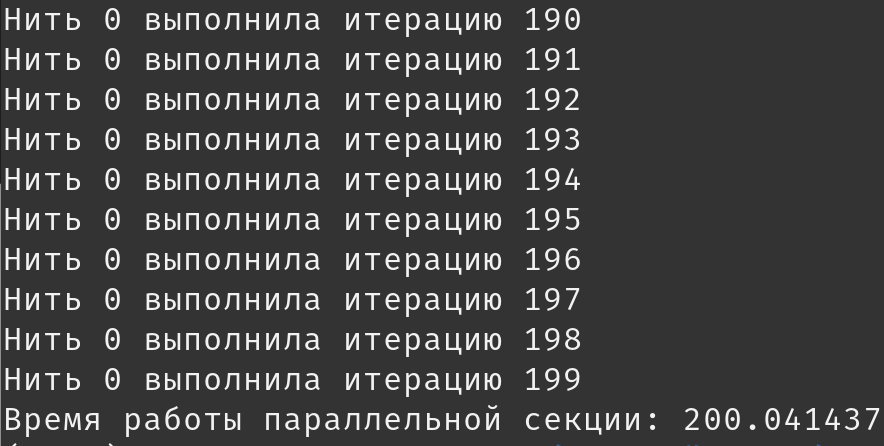
(\*\*\*)



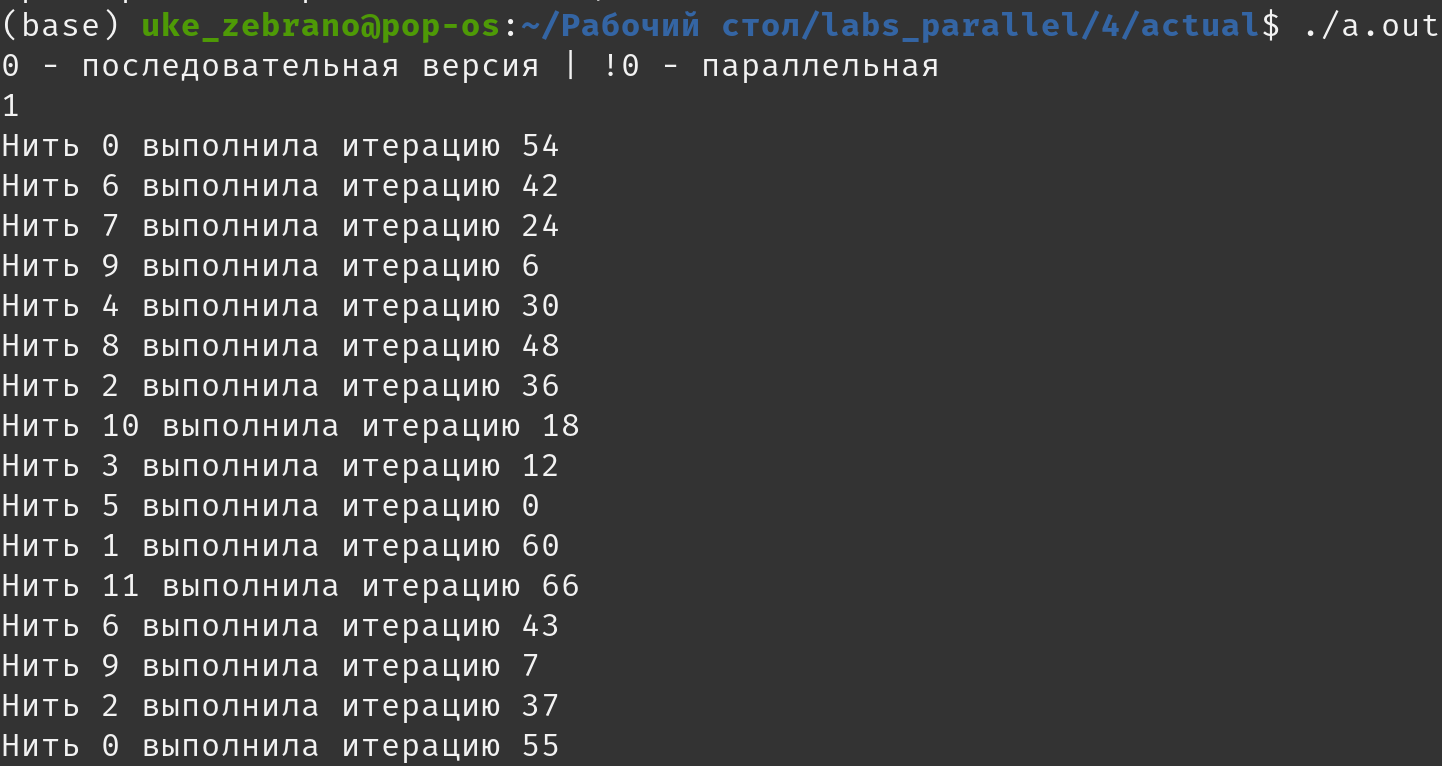
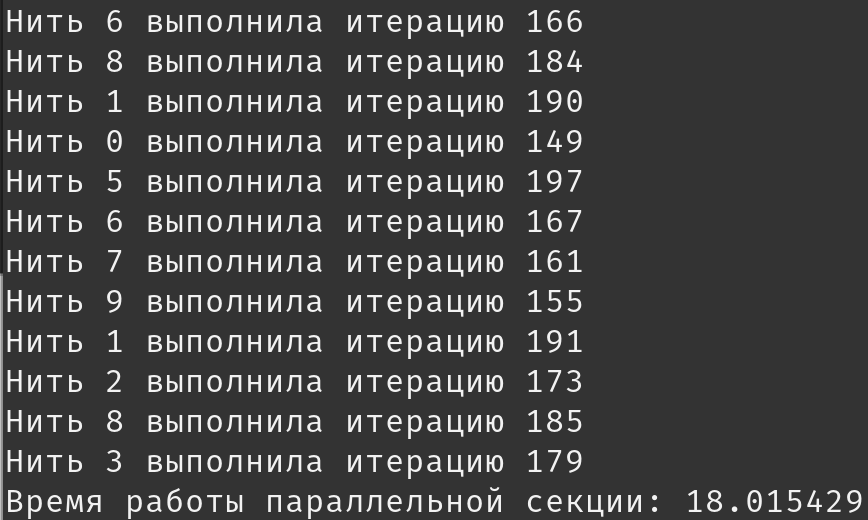
#pragma omp for schedule (dynamic, 6)

Последовательная версия

(\*\*\*)

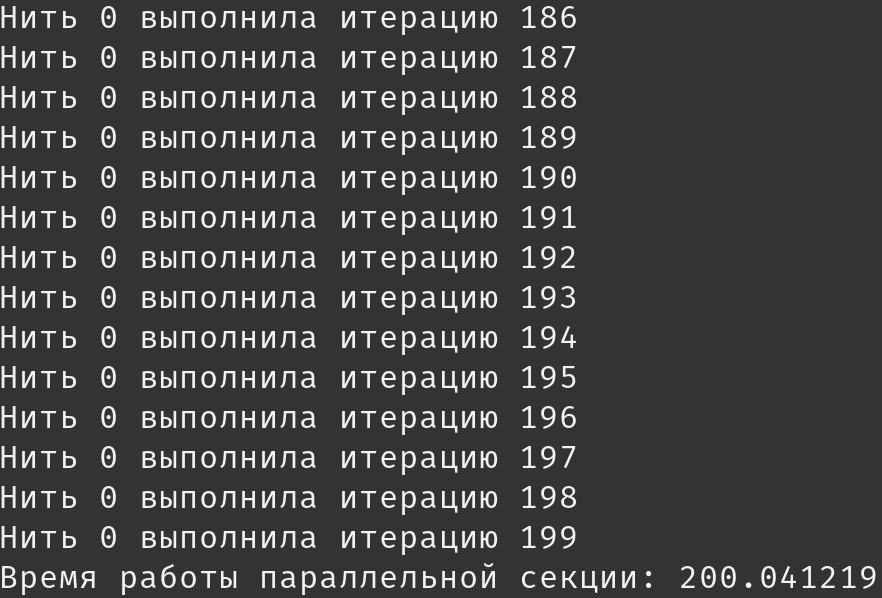
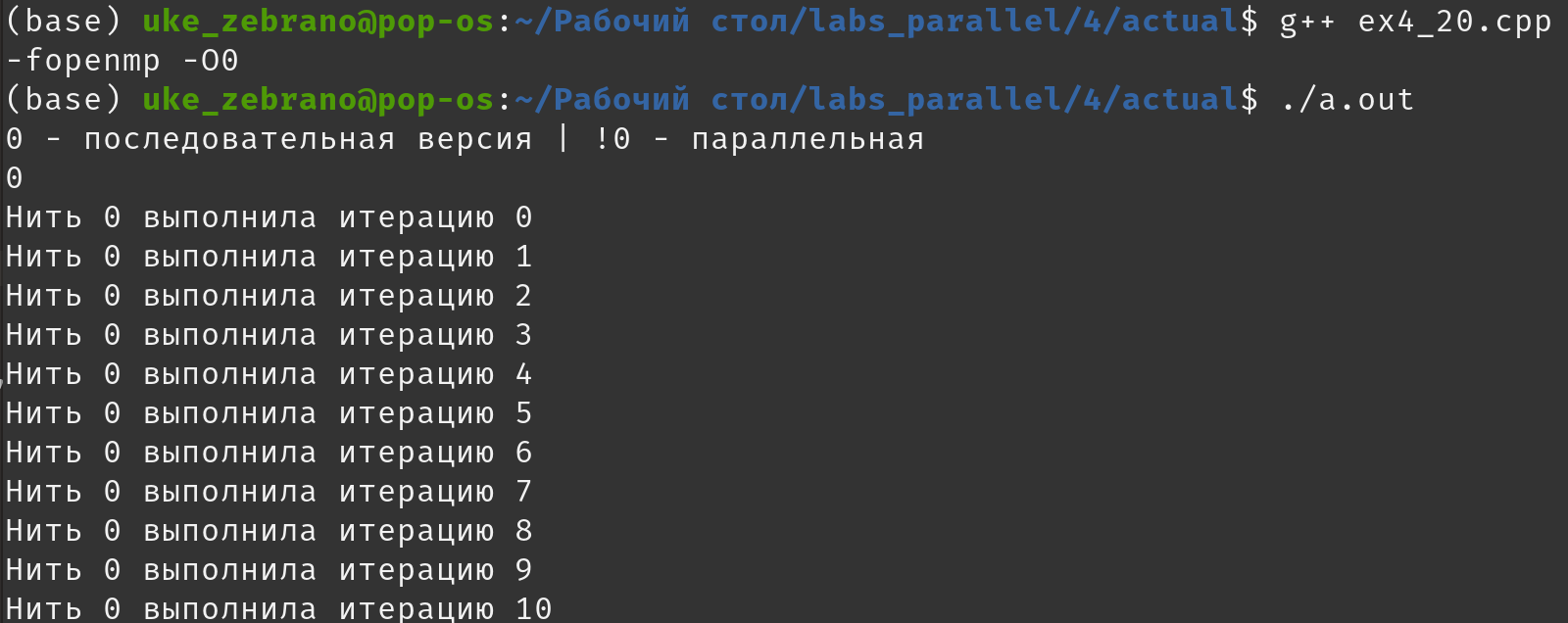


Параллельная версия

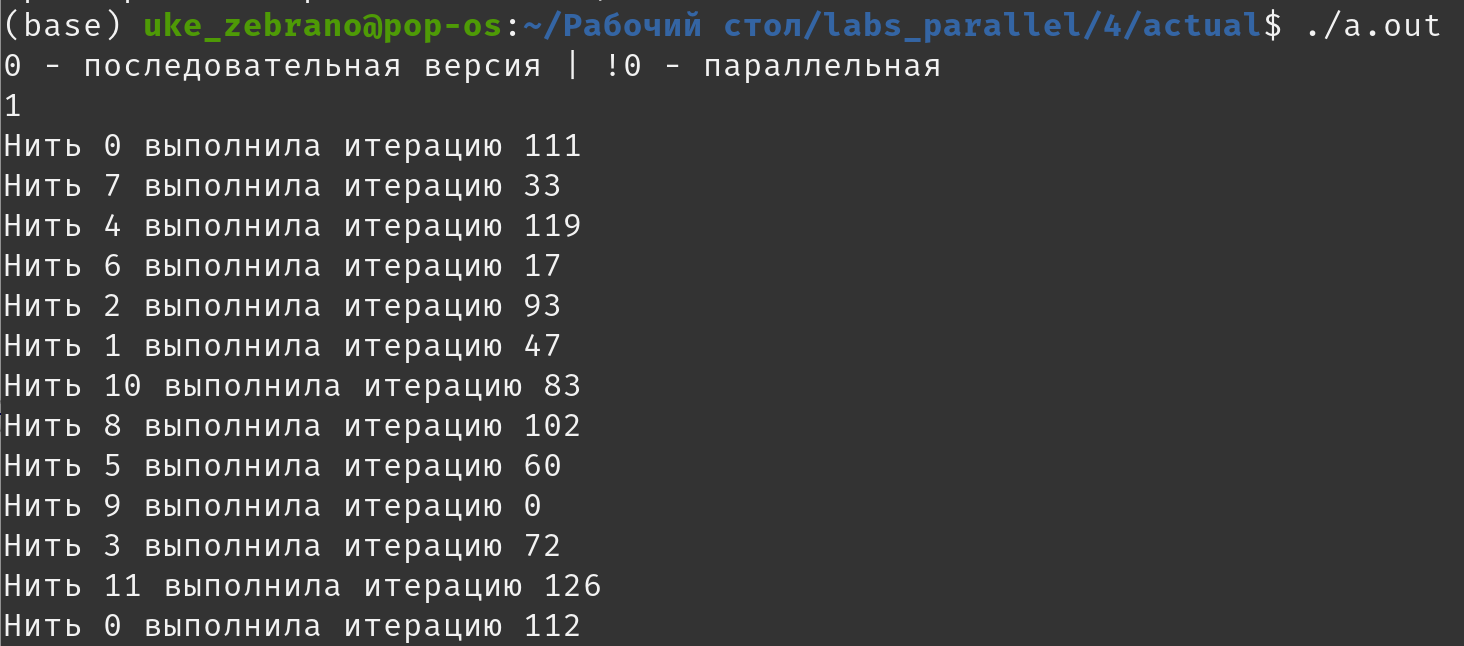
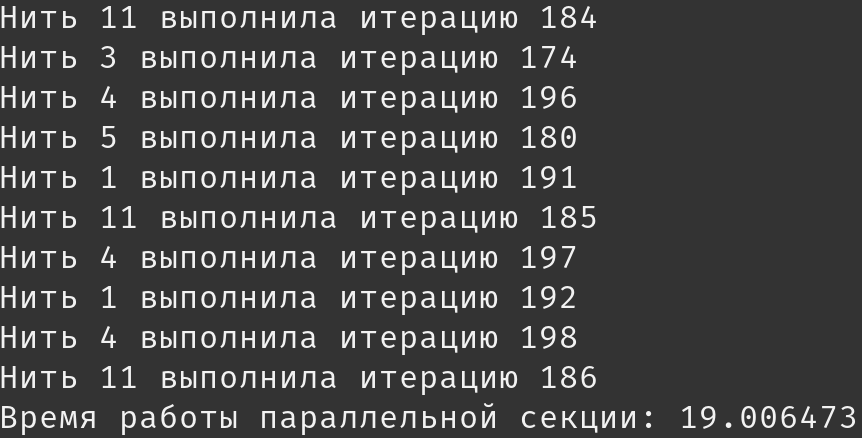
(\*\*\*)  


#pragma omp for schedule (guided, 6)

Последовательная версия

(\*\*\*)  
  
  
  
  


Параллельная версия

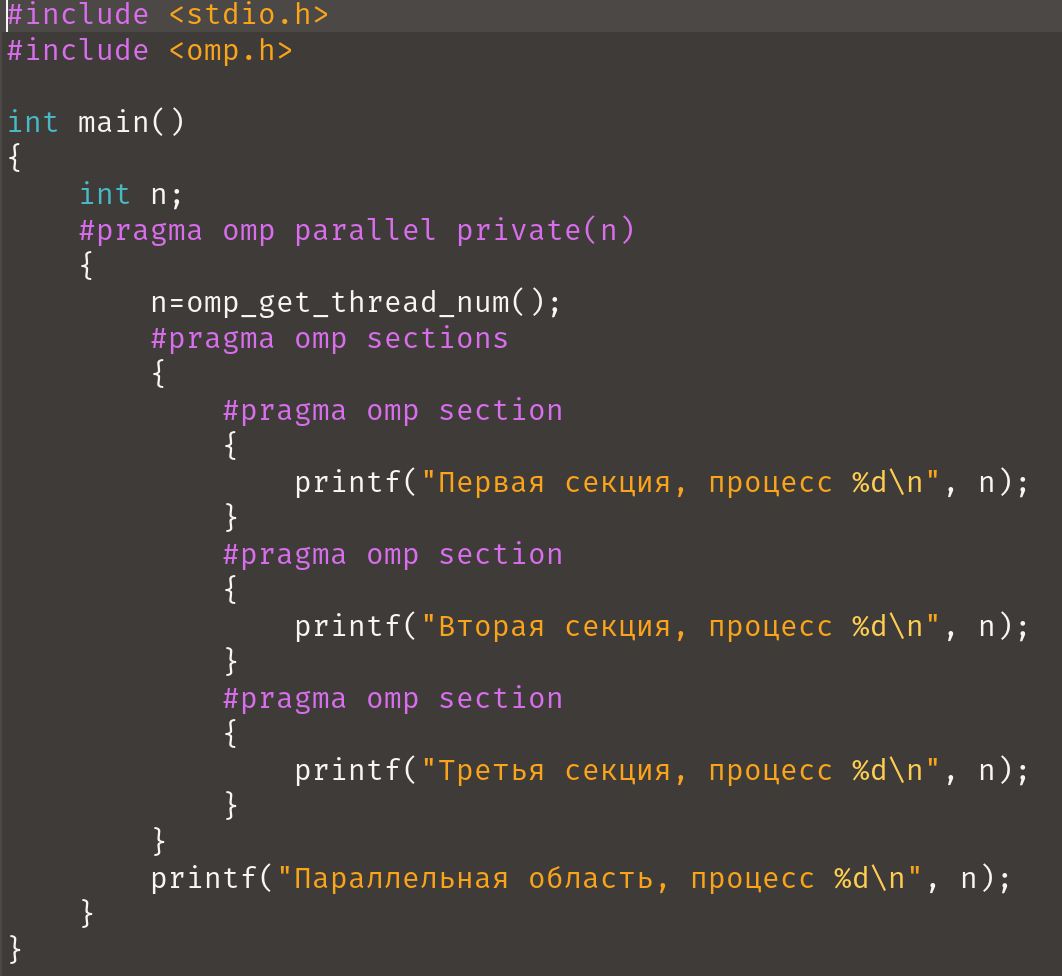
(\*\*\*)  
  
  
  
  
  
  
Здесь аналогично прошлому примеру, только одной нитью выполняется по 6 итераций, что отражается на скорости работы. Последовательные версии рассматривать нет смысла, они все одинаково а районе двухсот секунд.

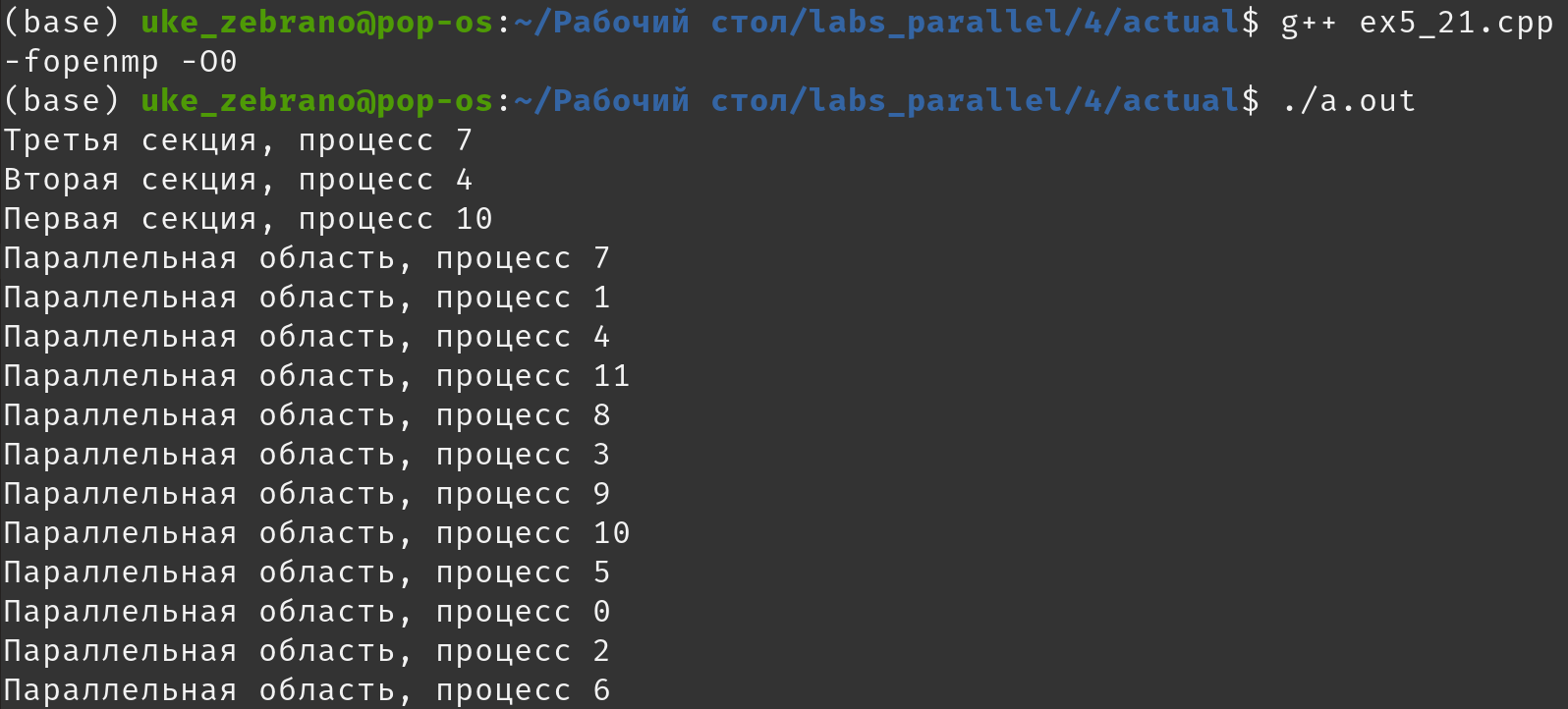
Время параллельных версий:

Static — ~18 секунд

Dynamic — ~18 секунд

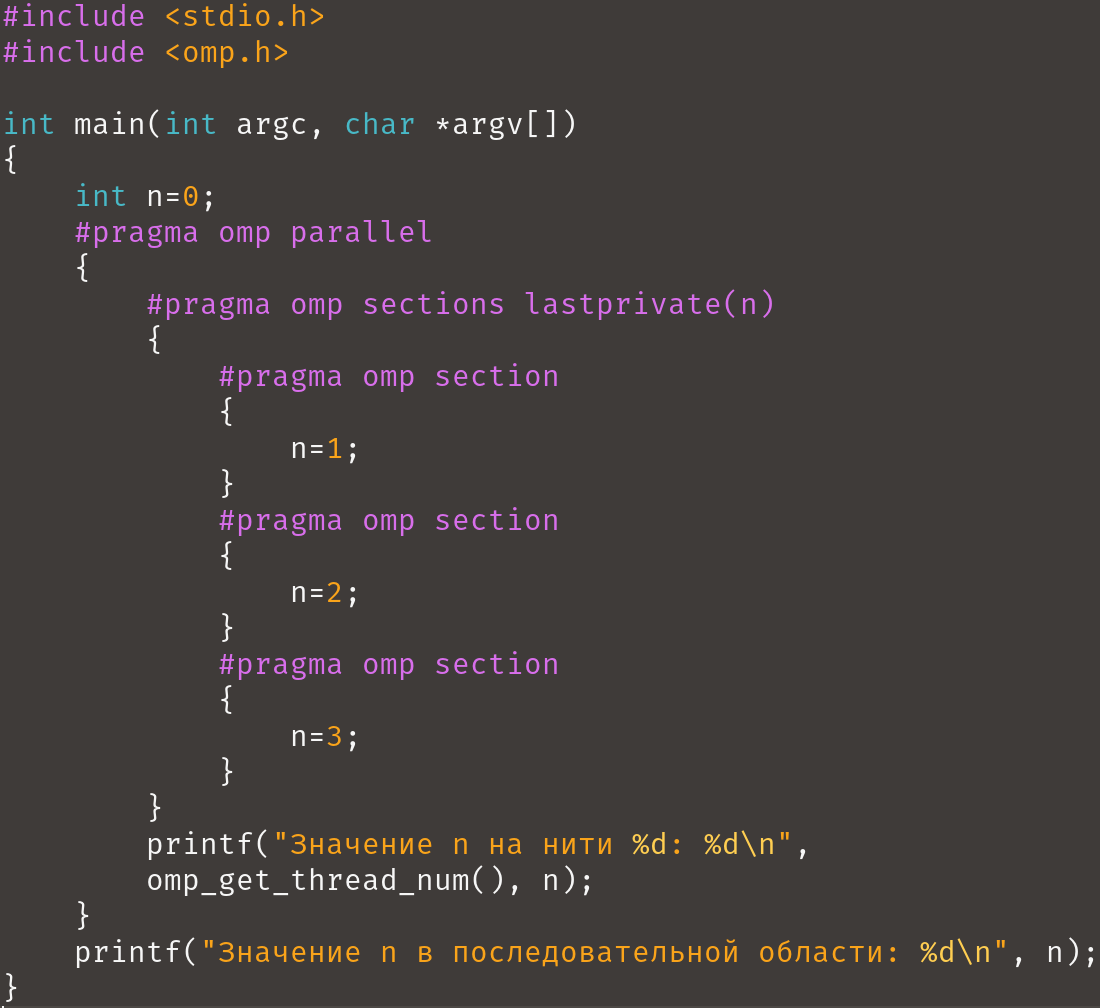
Guided — ~19 секунд, оказался дольше остальных

Пример 5 (в методичке обозначается как пример 21) Директива sections  
Цель данного примера — продемонстрировать работу sections  
  
Предположительно, программа распределит секции на 3 нити, они выведут сообщение со своим номером, далее все нити выведут одинаковое сообщение со своим номером.  
Результат:



В результате секции взяли на выполнение 3 нити, 7-4-10, далее все нити вывели свой номер (в том числе и 7-4-10). Ожидания оправдались.

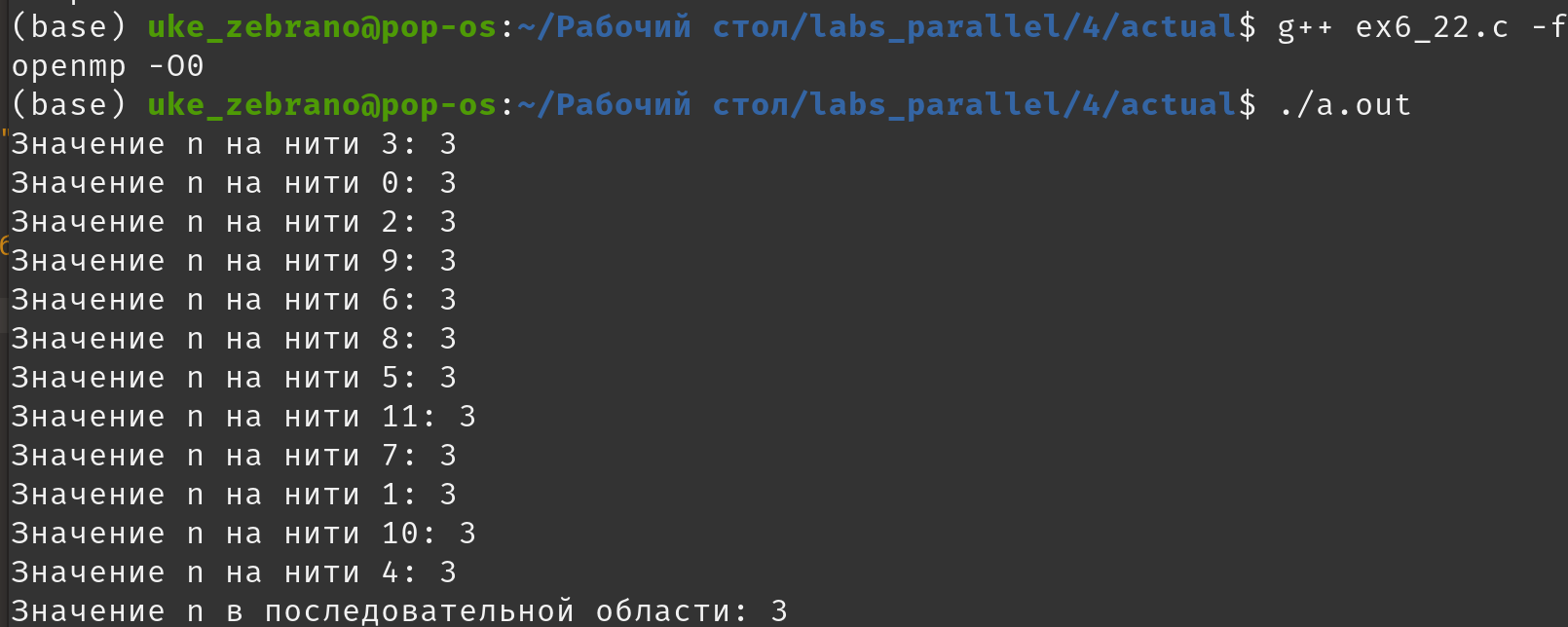
Пример 6 (в методичке обозначается как пример 22) Опция lastprivate.

  
Продемонстрируем использование опции lastprivate. Опция lastprivate используется вместе с директивой sections. Переменная n объявлена как lastprivate переменная. Три нити, выполняющие секции присваивают своей локальной копии n разные значения.

На выходе из области sections значение n из последней секции присваивается

локальным копиям во всех нитях, поэтому все нити напечатают число 3. Это

же значение должно сохраниться для переменной n и в последовательной области.



Вывод: Изучено распределение работы между имеющимися нитями.