**ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ**

**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ**

**«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»**

Факультет компьютерных наук

Департамент программной инженерии

**Домашнее задание №3   
по дисциплине «Архитектура вычислительных систем»**

**Практические приемы построения многопоточных приложений**

Исполнитель

студент группы БПИ196-1

Махнач Ф. О.

17.11.2020 г.

**Оглавление**

[1. Текст задания 2](#_Toc56538954)

[2. Однопоточный аналог приложения на языке Python 2](#_Toc56538955)

[3. Реализация алгоритма 3](#_Toc56538956)

[3.1. Первая реализация алгоритма на языке C++ 3](#_Toc56538957)

[3.2. Распараллеливание 4](#_Toc56538958)

[3.3. Оптимизация проверки на равенство цифр десятичной записи 5](#_Toc56538959)

[4. Прочие части программы 6](#_Toc56538960)

[5. Тестирование программы 7](#_Toc56538961)

# Текст задания

Вывести список всех целых чисел, содержащих от 4 до 9 значащих цифр, которые после умножения на будут содержать все те же самые цифры в произвольной последовательности и в произвольном количестве.   
 Входные данные: целое положительное число . Количество потоков является входным параметром

# Однопоточный аналог приложения на языке Python

Для общего понимания сути задачи была написана примитивная версия программы на языке Python:



Реализация алгоритма следующая: для каждого значения из указанного диапазона  
 1) Создаём множество (контейнер из уникальных элементов) из всех цифр числа ;  
 2) Создаём множество из всех цифр числа ;  
 3) Если множества равны – добавляем ив результирующее множество;  
Здесь отсутствует:  
 1) Использование потоков;  
 2) Проверка на входной параметр ;  
 3) Вывод в файл;

Выводы, сделанные в результате реализации примитивной версии программы:  
 1) Для определения равенства множества цифр для чисел можно использовать set – контейнер, содержащий упорядоченное множество уникальных элементов. Такой способ неэффективен, однако он наиболее прост для восприятия и понимания (и первым пришёл на ум).  
 2) Из формулировки условия задачи считаю, что если число является искомым, то число также является искомым (подходит под критерии).  
 3) Для диапазона [1e3; 1e9) программа работает слишком долго. Вменяемый результат удалось получить лишь на [1e3; 1e6).  
 4) Для вывода результата в отсортированной форме также придётся использовать контейнер set, однако такой подход, скорее всего, потребует большого количества памяти в время исполнения, а также потребует синхронизации доступа к этому контейнеру разных потоков. Для 1e9 понадобится 4\*1e9 байт475Мб. Конечно, на практике значений будет меньше, но не на порядок.  
 5) Проверка для каждого значения из диапазона независима. Из предложенных моделей построения многопоточных приложений наиболее подходящей мне кажется **итеративный параллелизм**.

# Реализация алгоритма

## Первая реализация алгоритма на языке C++

Первоначально идея использования контейнера set была перенесена реализацию на C++. Таким образом, основная часть алгоритма выглядела следующим образом:  


Здесь последовательно вызывается функция CheckNum для каждого числа из диапазона [1e3; 1e9) (числа из 4-9 цифр). В CheckNum создаются множества из цифр чисел и сравниваются на равенство. Если число проходит такую проверку, оно добавляется в result\_set – контейнер, содержащий результат.  
Часть кода, отвечающая за ввод-вывод не приведена для краткости (см. п. 4).  
Конечно, это всё еще линейная программа. Необходимо добавить потоки выполнения и распределить между ними задачи.

## Распараллеливание

Так как каждая проверка является независимой задачей, можем использовать итеративный параллелизм: имея m потоков (std::thread) в массиве, i-ый поток будет обрабатывать каждый m-ое значения начиная с 1000+i (1000 – нижняя граница, т.е. самое малое четырёхзначное число. Аналогичный способ распределения потоков был показан на семинарах (например, [тут](http://softcraft.ru/edu/comparch/practice/thread/01-simple/thread01cpp/sqsum.cpp)).   
После добавления распараллеливания изменилась функция ProcessNums и добавилась функция CalculateMultiThread, запускающая потоки:

Помимо этого, для безопасной работы с result\_set был добавлен mutex:

void CheckNum(int num) {

std::set<int> num\_digit\_set = MakeSetOfDigits(num);

std::set<int> prod\_digit\_set = MakeSetOfDigits(static\_cast<int64\_t>(num) \* ::n);

if (num\_digit\_set == prod\_digit\_set) {

glob::result\_set\_mutex.lock();

glob::result\_set.insert(-num);

glob::result\_set.insert(num);

glob::result\_set\_mutex.unlock();

}

}

Однако запуск программы показал, что она всё ещё недостаточно эффективна: диапазон [1e3; 1e6] обрабатывался >40 секунд.

## Оптимизация проверки на равенство цифр десятичной записи

Очевидно, использование контейнера set в функции CheckNum несёт в себе издержки. На практике нам необходима лишь информация о 10 цифрах, т. е. для каждого числа k нам нужно знать, какие из 10 цифр используются в его десятичной записи, а какие нет. Эту информацию можно закодировать в 10 битах: если i-ый бит единица, то цифра i входит в десятичную запись числа. Тогда числа состоят из одних и тех же цифр тогда и только тогда, когда их 10-битные коды будут совпадать.

Это несложно реализовать, используя 16-битный целочисленный тип (за неимением лучшего). Функция нахождения 10-битного кода числа:

uint16\_t GetBitmapOfDigits(int64\_t num) {

uint16\_t bitmap = 0;

while (num != 0) {

// Отмечаем биты, соответствующие каждой цифре.

bitmap |= glob::digit\_masks[num % 10];

// Проходим по каждой цифре числа.

num /= 10;

}

return bitmap;

}

где glob::digit\_masks – массив подготовленных масок с единственным битом на i-ой позиции, где i – индекс в массиве:

void PrecomputeDigitMasks() {

for (size\_t i = 0; i < 10; ++i) {

glob::digit\_masks[i] = 1 << i;

}

}

Изменённая функция CheckNum:

void CheckNum(int num) {

// Вычисляем маску цифр для num и для num \* n.

uint16\_t num\_map = GetBitmapOfDigits(num);

uint16\_t prod\_map = GetBitmapOfDigits(static\_cast<int64\_t>(num) \* glob::n);

if (num\_map == prod\_map) {

glob::result\_set\_mutex.lock();

// Если num подходит, то -num также подходит.

// Заносим -num и num в множество.

glob::result\_set.insert(-num);

glob::result\_set.insert(num);

glob::result\_set\_mutex.unlock();

}

}

Текущая версия уже может быть использована по назначению – на диапазоне [1e3; 1e8) программа работает 25 секунд, на [1e3; 1e9) – 6 минут (на моём компьютере). Безусловно, это довольно долго, но в то же время значительно лучше предыдущей версии.

# Прочие части программы

Все глобальные переменные, использующиеся в программе, выделены в пространство имён glob:

namespace glob {

// Нижняя граница диапазона проверяемых значений.

const int LOWER\_BOUND = 1e3;

// Верхняя граница диапазона проверяемых значений.

const int UPPER\_BOUND = 1e9;

// Входной параметр n -- то, на что мы умножаем каждое число из диапазона.

int n;

// Нижнее ограничение входного параметра n.

const int n\_lower = 1;

// Верхнее ограничение входного параметра n.

const int n\_upper = 10;

// Входной параметр -- число потоков.

size\_t num\_of\_threads;

// Множество значений x таких, что {цифры x} = {цифры x \* n}.

std::set<int> result\_set;

// Мьютекс для контроля доступа к result\_set.

std::mutex result\_set\_mutex;

// Битовые маски, соответствующие каждой цифре (напр 1 -> 0000000010, 7 -> 0010000000).

uint16\_t digit\_masks[10];

}

Для ввода значений n и числа потоков используются соответствующие функции:

// Читает параметр N из консоли.

bool TryReadN() {

std::cout << "Please, enter N: ";

std::cin >> glob::n;

return glob::n\_lower < glob::n && glob::n < glob::n\_upper;

}

// Читает число потоков из консоли.

bool TryReadNumberOfThreads() {

std::cout << "Please, enter number of threads: ";

std::cin >> glob::num\_of\_threads;

return 0 < glob::num\_of\_threads &&   
 glob::num\_of\_threads <= std::thread::hardware\_concurrency();

}

Для вывода результата в файл используются следующая функция:

// Записывает множество чисел в файл.

void WriteSetToFile(const std::set<int>& set, const std::string& path) {

std::ofstream out;

out.open(path);

if (out.is\_open()) {

for (int i : set) {

out << i << std::endl;

}

}

out.close();

}

Сама функция main:

int main() {

// Считаем битовые маски, которые понадобятся в функции GetBitmapOfDigits.

PrecomputeDigitMasks();

bool reading\_successful = TryReadN();

if (reading\_successful == false) {

std::cerr << "Wrong input! n must be a integer in range [2; 9]." << std::endl;

return 1;

}

reading\_successful = TryReadNumberOfThreads();

if(reading\_successful == false){

std::cerr << "Wrong input! Num of threads cannot be less than 1 or more than " << std::thread::hardware\_concurrency() << "!" << std::endl;

return 1;

}

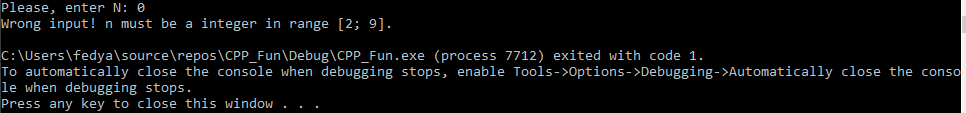
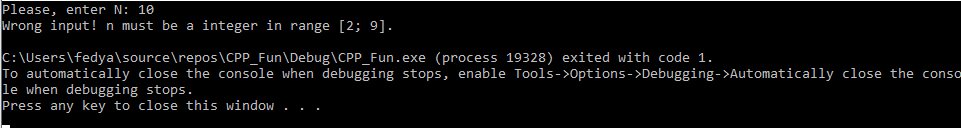
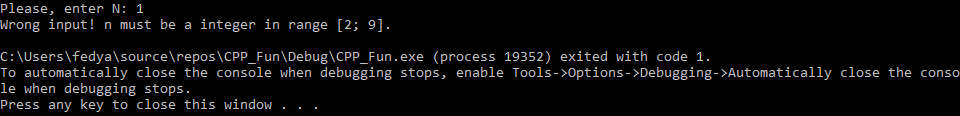
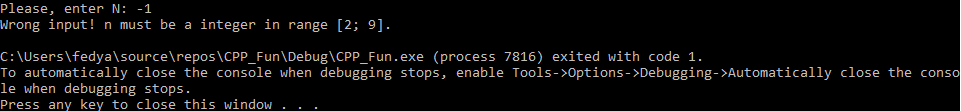
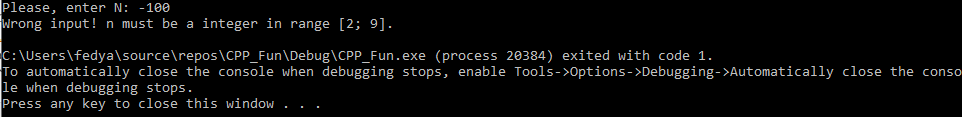
CalculateMultiThread();

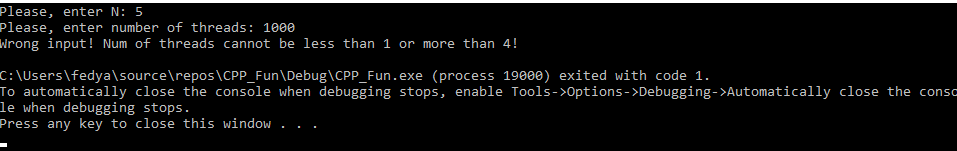
WriteSetToFile(glob::result\_set, "out.txt");

return 0;

}

# Тестирование программы

Результаты вывода программы при различных входных параметрах приведены в папке tests репозитория с работой. Название файла output\_N.txt обозначает, что это вывод программы при входных параметрах n = N и кол-во потоков = 4 (максимальное на моём компьютере), например, output\_3.txt.   
Также протестированы следующие случаи некорректного ввода:  
1) n = 0  
2) n = 10  
3) n = 1  
4) n = -1  
5) n = -100

6) n = 5 num\_of\_threads = 1000  


7) n = 7 num\_of\_threads = -15

