Санкт-Петербургский государственный политехнический университет
Институт компьютерных наук и технологий

**«Высшая школа программной инженерии»**

Санкт-Петербург

2020

# КУРСОВАЯ РАБОТА

По дисциплине «Технология использования гибридных суперкомпьютеров для обработки больших данных»

Выполнил  Каравашкин Л.А.

3540904/00202

Преподаватель Левченко А.В

# Цель работы

Цель работы - программное обеспечение, разработанное по технологиям многоязыкового и параллельного программирования и задействующее современную целевую архитектуру.

# Задачи

1. Разработать техническое задание на программное обеспечение.
2. Разработать динамически подключаемый код по обработке данных.
    a. Разработать однопоточный невекторизованный код.
    b. Разработать однопоточный векторизованный код.
    c. Разработать многопоточный невекторизованный код.
    d. Разработать многопоточный векторизованный код.
3. Разработать головную программу.
4. Провести сравнительный анализ.

# Ход работы

## Выбор технического задания

В качестве функции, к которой будет применяться оптимизация, была выбрана функция для поэлементного сложения значений трех массивов.
Языком для внешней библиотеки был выбран c++, а для головной программы был выбран Python.

## Разработка библиотеки на c++

Для всех функций был добавлен ключ -O0 для того, чтобы отключить стандартные оптимизации языка.

### 1. Однопоточный невекторизованный код:

```
#pragma GCC optimize("-O0")
long long CalcSimple(float * a, float * b, float * c, float * resArr, int n) {
  auto start = std::chrono::high_resolution_clock::now();
  for (int i = 0; i < n; i++) {
    resArr[i] = a[i] + b[i] + c[i];
  }
  auto finish = std::chrono::high_resolution_clock::now();
  return std::chrono::duration_cast < std::chrono::nanoseconds > (finish - start).count();
}
```

Простой цикл без оптимизации.
Функция принимает 4 массива, 3 из которых для сложения и 1 для сохранения результата.
В выходном значении передается время выполнения функции.

### 2. Однопоточный векторизованный код

```
#pragma GCC optimize ("-O0")
long long CalcVect(float * a,float * b, float * c, float* resArr , int n) {
  auto start = std::chrono::high_resolution_clock::now();
  #pragma omp simd
  for(int i=0;i<n;i++) {
     resArr[i] = a[i]+b[i]+c[i];
  }
  auto finish = std::chrono::high_resolution_clock::now();
  return std::chrono::duration_cast<std::chrono::nanoseconds>(finish-start).count();
}
```

pragma omp simd указывает компилятору применить к циклу векторизацию.

### 3. Многопоточный невекторизованный код

```
#pragma GCC optimize ("-O0")
long long CalcParallel(float * a,float * b, float * c, float* resArr , int n) {
  auto start = std::chrono::high_resolution_clock::now();
  #pragma omp parallel for
```

```
 for(int i=0;i<n;i++) {
    resArr[i] = a[i]+b[i]+c[i];
 }
 auto finish = std::chrono::high_resolution_clock::now();
 return std::chrono::duration_cast<std::chrono::nanoseconds>(finish-start).count();
}
```

pragma omp parallel for указывает компилятору распараллелить цикл по нескольким потокам. Так как отсутствует зависимость по данным между итерациями, то это должно увеличить скорость выполнения.

## 4. Многопоточный векторизованный код

```
#pragma GCC optimize ("-O0")
long long CalcVectParallel(float * a,float * b, float * c, float* resArr , int n) {
 auto start = std::chrono::high_resolution_clock::now();
 #pragma omp parallel for simd
 for(int i=0;i<n;i++) {
    resArr[i] = a[i]+b[i]+c[i];
 }
 auto finish = std::chrono::high_resolution_clock::now();
 return std::chrono::duration_cast<std::chrono::nanoseconds>(finish-start).count();
}
```

pragma omp parallel for simd - одновременной использование векторизации и распараллеливания цикла по потокам.

## 5. Сборка библиотеки

g++ -fopenmp -fPIC -c -o clib clib.cpp
g++ -shared -lgomp -fopenmp -o clib.so clib

# Разработка головной программы

## 1. Организация работы с библиотекой

Подключение библиотеки:
lib = ctypes.CDLL('./clib.so')

Адаптер функций для библиотеки:

```
def funcConfig(func, arraySize):
    func.restype = ctypes.c_longlong
    func.argtypes = [ctypes.POINTER(ctypes.c_float * arraySize), ctypes.POINTER(ctypes.c_float * arraySize),
                     ctypes.POINTER(ctypes.c_float * arraySize), ctypes.POINTER(ctypes.c_float * arraySize),
                     ctypes.c_int]
    return func
```

Адаптер принимает функцию вида lib.function_name и размер массива с которым функции нужно работать и возвращает функцию python.

## 2. Описание программы

Тело программы представляет из себя применение библиотечных функций, сбор статистики зависимости времени выполнения функций от количества элементов в массивах и валидацию результатов за счет нескольких циклов и подсчета средних значений времени.

При запуске программа предлагает пользователю выбрать максимальное количество элементов в массивах и количество циклов валидации. После выполнения пользователю выводится время выполнения функций для выбранного количества элементов, а также графики зависимости времени от количества элементов и времени на один элемент от количества элементов.

## 3. Запуск программы

python3 main.py

# Результат

После выполнения программы для максимального количества элементов в массиве равным 5 000 000 и 5 циклов валидации получены следующий графики:

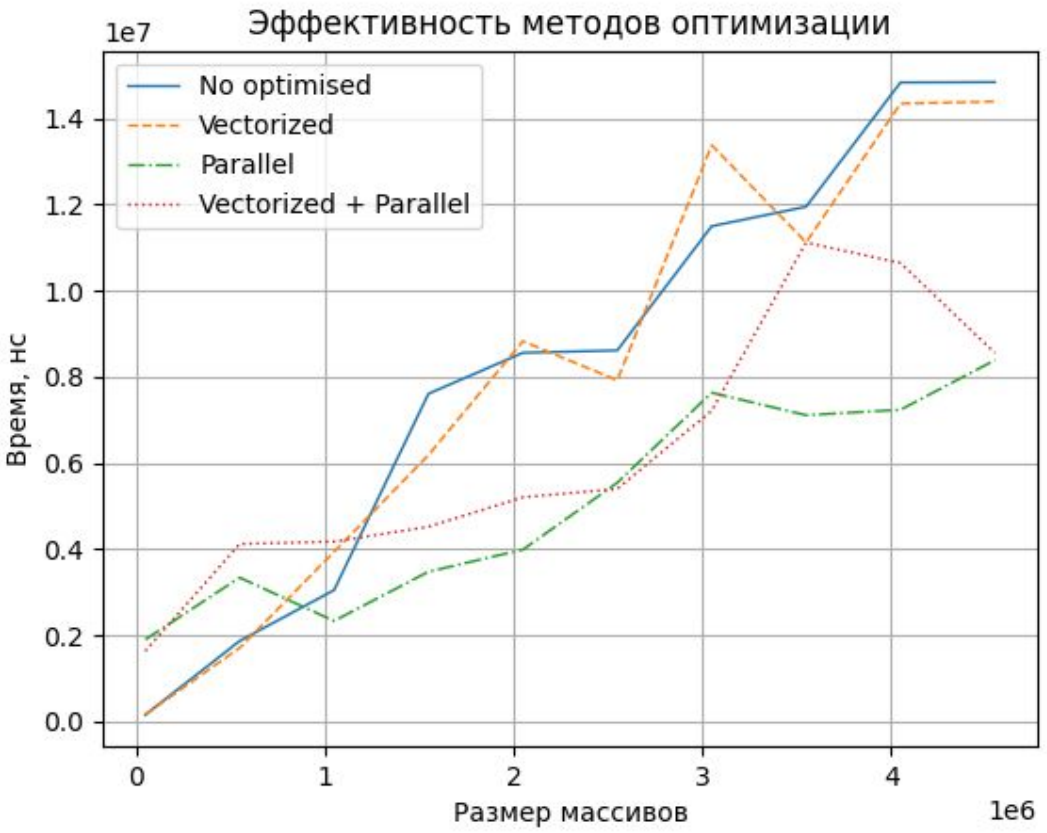


Рис 1.Зависимость времени выполнения функций от размера массива

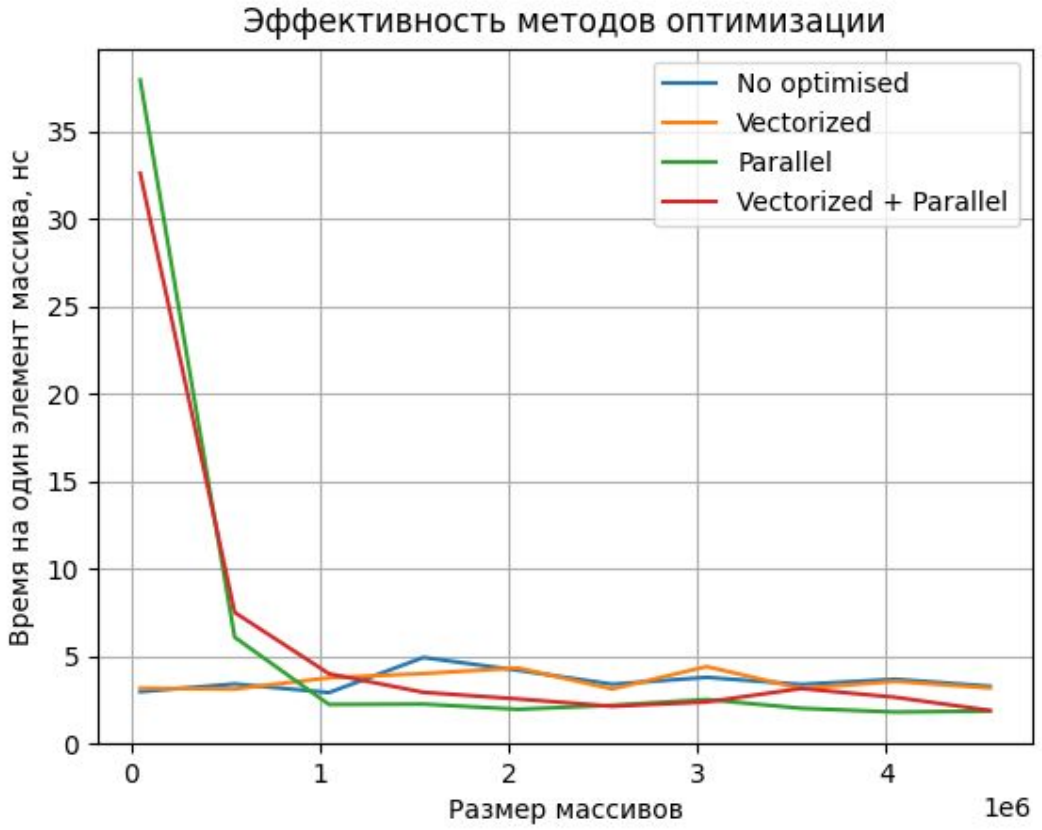


Рис 2. Зависимость времени обработки одного элемента массива от размера массива.

По графикам видно, что до одного миллиона значений в массиве методы использующие параллельность только замедляют выполнение функции, однако при достаточном размере массива эти методы доказывают свою эффективность.
Векторизация в сравнении с функцией без нее показывает переменные результаты в зависимости от размера массива.

# Вывод

В ходе работы было написано многоязыковое программное обеспечение, в котором за тяжелые вычисления отвечала написанная нами библиотека на C++,  а за организацию работы и вывод статистики отвечало приложение на Python.

Было продемонстрировано использование методов оптимизации вычислений с использованием openmp в функции для поэлементного сложения массивов.

Также было проведено сравнение эффективности этих методов оптимизации: при достаточно большом значении элементов в массиве наиболее эффективны методы с использование параллельного выполнения в нескольких потоках, однако в ином случае скорость работы снижается. Векторизация в полученных результатах имела переменную эффективность и абсолютные значения времени выполнения были соразмерны функции без векторизации.