Санкт-Петербургский государственный политехнический университет Институт компьютерных наук и технологий
«Высшая школа программной инженерии»

КУРСОВАЯ РАБОТА

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

Выполнил Алейников П. И.
3540904/00202

Преподаватель Левченко А.В

Санкт-Петербург

2020

Цель работы

Цель работы – программное обеспечение, разработанное по технологиям многоязыкового и параллельного программирования и эффективно задействующее современную целевую архитектуру. В ходе работы будут проделаны следующие этапы:

1. Выбор задания, на котором будет проверяться производительность с различными оптимизациями.
2. Разработка однопоточного невекторизованного кода.
3. Разработка однопоточного векторизованного кода.
4. Разработка многопоточного невекторизованного кода.
5. Разработка многопоточного векторизованного кода.
6. Разработка приложения на другом языке с динамическим вызовом ранее созданного кода.
7. Сравнение и выводы.

Платформа

Разработка производилась в OS Linux Manjaro 20.2, на машине с Intel Core i5 8265U.

Библиотека писалась на Rust с использованием компилятора rustc 1.45.0.

Вызывающая программа была написана на Java 11.

Задание

Библиотека содержит три метода:

- Метод однопоточного сложения чисел. Метод принимает массив целых чисел и возвращает результат сложения
- Метод многопоточного сложения, с числом потоков равным числу потоков процессора (на платформе разработки их 8). Метод принимает массив целых чисел и возвращает результат сложения
- Метод многопоточного сложения с варьируемым числом потоков. Метод принимает массив целых чисел и число потоков, а возвращает результат сложения

JNI

Для взаимодействия Java с Rust использовалась технология JNI [1]. Соответственно был написан Java Class описывающий декларацию методов, которые будут в Rust библиотеке.

```
package com.company;

class RustJni {

   private static native int sumPerfectSquare(int [] arr);

   private static native int sumPerfectSquarePar(int [] arr);

   private static native int sumPerfectSquarePar(int [] arr, int
thread_count);

   static {
       System.loadLibrary("libmpp");
   }

   public static void main(String[] args) {}
}
```

И далее был сгенерирован заголовочный фа йл для Си библиотеки при помощи команды `javac -h . RustJni`, который пришлось переписать на Rust.

Разработка однопоточного кода

Метод `Java_com_company_RustJni_sumPerfectSquare` - это сгенерированный и адаптированный под Rust шаблон метода `sumPerfectSquare`. В нём помимо чтения и преобразования входных параметров происходит замер времени сложения при помощи библиотеки `std::time::SystemTime` и само сложение при помощи вызова vec.iter().sum(), то есть метода сложения у итератора вектора с числами из входного массива.

```
#[no_mangle]
pub extern "system" fn
Java_com_company_RustJni_sumPerfectSquare(env: JNIEnv,
                                          _class: JClass,
                                          input: jintArray)
                                          -> jint {
```

```
    let input_size: jsize =
        env.get_array_length(input).expect("Couldn't get java array
size!").into();
    let mut vec :Vec<i32> = vec![0; input_size as usize];
        env.get_int_array_region(input, 0,  &mut
vec).expect("Couldn't get java array!");
    let now = SystemTime::now();
    let sum = vec.iter()
        .sum();
    match now.elapsed() {
        Ok(elapsed) => {
            println!("sumPerfectSquare time: {} ns",
elapsed.as_nanos());
        }
        Err(e) => {
            println!("Error: {:?}", e);
        }
    }
    sum
}
```

Разработка многопоточного кода

Для выполнения сложения в многопоточном режиме использовалась библиотека Rayon [2], которая позволяет не явно распараллелить работу с коллекциями предоставляю итератор `par_iter()`, который имеет API аналогичное стандартному итератору коллекций `iter()`, но выполняет методы параллельно.

Соответственно был реализован метод `Java_com_company_RustJni_sumPerfectSquarePar___3I`, который соответсвует методу `sumPerfectSquare(int [] arr)`.

```
#[no_mangle]
pub extern "system" fn
Java_com_company_RustJni_sumPerfectSquarePar___3I(env: JNIEnv,

_class: JClass,

input: jintArray)

                                                    ->

jint {
```

```
    let input_size: jsize =
        env.get_array_length(input).expect("Couldn't get java array
size!").into();
    let mut vec :Vec<i32> = vec![0; input_size as usize];
    env.get_int_array_region(input, 0,  &mut vec).expect("Couldn't
get java array!");
    let now = SystemTime::now();
    let sum = vec.par_iter()
        .sum();
    match now.elapsed() {
        Ok(elapsed) => {
            println!("sumPerfectSquarePar time: {} ns",
elapsed.as_nanos());
        }
        Err(e) => {
            println!("Error: {:?}", e);
        }
    }
    sum
}
```

И метод `Java_com_company_RustJni_sumPerfectSquarePar___3II`, который соответсвует методу `sumPerfectSquare(int [] arr, thread_count)`. Данный метод использует число потоков, переданное параметром в метод. Это достигается путём создания `rayon::ThreadPool` и исполнения `par_iter()` уже внутри этого пула.

```
#[no_mangle]
pub extern "system" fn
Java_com_company_RustJni_sumPerfectSquarePar___3II(env: JNIEnv,

_class: JClass,

input: jintArray, thread_count: jint)
                                                                ->
jint {
    let input_size: jsize =
        env.get_array_length(input).expect("Couldn't get java array
size!").into();
    let mut vec :Vec<i32> = vec![0; input_size as usize];
    env.get_int_array_region(input, 0,  &mut vec).expect("Couldn't
get java array!");
```

```
    let pool =
rayon::ThreadPoolBuilder::new().num_threads(thread_count as
usize).build().unwrap();
    let now = SystemTime::now();
    let sum = pool.install(|| {
        vec.par_iter()
            .sum()
    });
    match now.elapsed() {
        Ok(elapsed) => {
            println!("sumPerfectSquarePar with pool size: {} time:
{} ns", thread_count, elapsed.as_nanos());
        }
        Err(e) => {
            println!("Error: {:?}", e);
        }
    }
    sum
}
```

Компиляция библиотеки с векторизацией

Для компиляции библиотеки был описан Cargo.toml, который является файлом конфигурации для Cargo - сборщика Rust проектов. Он содержит стандартные секции с описанием проекта и зависимостей, а также секцию [lib], в которой мы сообщим Cargo, что надо собрать C dynamic library.

```
[lib]

crate_type = ["cdylib"]
```

Сборка производится командой: `$ cargo build --release`

Однако следующей командой можно попросить компилятор сохранить assembler файлы библиотеки: `$ RUSTFLAGS="--emit asm" cargo build --release`

В assembler файле библиотеки мы можем увидеть вызов инструкции `paddd`, которая соответствует MMX инструкции сложения

```
.LBB76_16:
```

```
  movdqu (%rbx,%rsi,4), %xmm2
  paddd  %xmm0, %xmm2
  movdqu 16(%rbx,%rsi,4), %xmm0
  paddd  %xmm1, %xmm0
  movdqu 32(%rbx,%rsi,4), %xmm1
  movdqu 48(%rbx,%rsi,4), %xmm3
  movdqu 64(%rbx,%rsi,4), %xmm4
  paddd  %xmm1, %xmm4
  paddd  %xmm2, %xmm4
  movdqu 80(%rbx,%rsi,4), %xmm2
  paddd  %xmm3, %xmm2
  paddd  %xmm0, %xmm2
  movdqu 96(%rbx,%rsi,4), %xmm0
  paddd  %xmm4, %xmm0
  movdqu 112(%rbx,%rsi,4), %xmm1
  paddd  %xmm2, %xmm1
  addq   $32, %rsi
  addq   $4, %rdi
  jne    .LBB76_16
```

Если в библиотеки суммируемый массив i32 заменить на массив f32, то инструкция `paddd` из assembler файла пропадёт, а вместо них появится инструкция `addss`, которая соответствует сложение чисел с плавающей запятой в SSE.

```
.LBB76_18:
  addss  (%rax), %xmm0
  addss  4(%rax), %xmm0
  addss  8(%rax), %xmm0
  addss  12(%rax), %xmm0
  addss  16(%rax), %xmm0
  addss  20(%rax), %xmm0
  addss  24(%rax), %xmm0
  addss  28(%rax), %xmm0
  addq   $32, %rax
  cmpq   %rcx, %rax
  jne    .LBB76_18
```

Из выше описанного эксперимента с заменой типа массива, можно сделать вывод, что MMX инструкции в assembler библиотеки есть ничто иное как автоматическая векторизация сложения элементов массивов. Так же если убрать метод не параллельного сложения из библиотеки, то MMX инструкции останутся значит векторизация и параллельный код.

Компиляция библиотеки без векторизацией

Чтобы отключить автоматическую векторизацию библиотеку надо собрать её командой : `$ RUSTFLAGS="--emit asm -C no-vectorize-loops -C no-vectorize-slp" cargo build --release`

Если оставить только флаг `no-vectorize-loops`, то остануться вызовы инструкции `paddd`, которые соответствуют методам с параллельным сложением.

Внешняя программа

Как говорилось ранее внешняя программа написана на Java. В ней реализован следующий main метод:

```
int N = 1_000_000;
int [] arr = new int [N];
for (int i = 0; i < N; i++) {
   arr[i] = 1;
}
System.out.println(RustJni.sumPerfectSquare(arr));
System.out.println(RustJni.sumPerfectSquarePar(arr));
System.out.println(RustJni.sumPerfectSquarePar(arr, 2));
```

В main вызывается все три метода библиотеки с массивом единиц длинной N. Метод `sumPerfectSquarePar` вызывается для двух потоков.

Сборка

Для финальной сборки в проекте написано два bash скрипта, которые собирают rust библиотеку и Java программу и запускают её. Отличия в скриптах лишь в том, что первый скрипт компилирует библиотеку с векторизацией, второй без векторизации. Далее приведён пример скрипта сборки с векторизацией.

```
#!/bin/bash
cd rust_lib
cargo clean
RUSTFLAGS="--emit asm" cargo build --release
cp target/release/liblibmpp.so ../java/
cd ../java/src
javac com/company/RustJni.java
```

```
java -Djava.library.path=../ com.company.RustJni
```

Оценка времени выполнения

Для оценки времени выполнения был написан Python скрипт который запускает 1000 раз Java программу и парсит вывод программы, чтобы забрать время выполнения вычислений выводимое Rust библиотекой. Далее скрипт считает среднее время выполнения и выводит его.

```
import os


N = 1000
sum1 = 0
sum2 = 0
sum3 = 0
for i in range(0, N):
    stream = os.popen('bash run.sh')
    output = stream.read()
    sum1 += int(output.split('\n')[0][22:-2])
    sum2 += int(output.split('\n')[2][25:-2])
    sum3 += int(output.split('\n')[4][43:-2])

print('no parallel', sum1/N)
print('parallel 8 thread', sum2/N)
print('parallel 2 thread', sum3/N)
```

Скрипт соответственно вызывался для библиотекой с векторизацией и без.

Векторизация	Последовательно	2 потока	8 потоков
Да	236829.997	222933.197	737150.101
Нет	445468.794	341135.86	759691.663

Таблица 1: Сравнение времени выполнения (нс) программы в зависимости от использованных методов оптимизации

Выводы

В ходе данной курсовой работы было разработано многоязыковое программное обеспечение, эффективно использующее возможности современных процессоров. При этом были рассмотрены методы оптимизации программ для Rust, который оказались схожи с теми, что даёт g++ и OpenMP для C++, так как также имеется автоматическая

векторизация и средства неявной многопоточности. Примечательно, что ни одного цикла в Rust в итоге не было написано, а использовались итераторы, которые компилятор Rust смог векторизовать.

Также была рассмотрена схема вызова Rust кода из Java посредство JNI. Это стало возможно благодаря тому что Rust может подстраиваться под C ABI.

Из оптимизаций лучше всего себя показала параллельность в 2 потока + векторизация, но тут надо понимать что это только для массивов большого размера (>= 1 000 000). А на втором месте оказалась обычая векторизация. Так же остаётся вопрос почему программа в 8 потоков показала себя хуже, чем программа в 2 потока. Возможно это связано с реализацией Rayon или с особенностями реализации нашей библиотеки.

Список литературы

1. Rust + JNI https://docs.rs/jni/0.18.0/jni/
2. Rust Rayon https://docs.rs/rayon/1.5.0/rayon/