Mini Project

2013105088 장효원

# 개요

첫 프로그래밍 언어가 만들어지고 난 이후, 지금까지 수 많은 프로그래밍 언어가 생겨왔고 각자의 목적 또한 모두 다르다. 지금까지의 프로그래밍 언어 패러다임을 살펴보면 크게 세가지로 나눌 수 있는데, C와 같은 절차적 언어, Java 같은 객체지향 언어, Scala와 같은 함수형 언어가 있으며, 각 패러다임의 목적도 다르고 언어가 다 다르기 때문에 언어의 퍼포먼스를 비교한다는 것은 불가능과 같은 이야기지만, 여기에서는 똑같은 알고리즘을 구현하여 각자의 속도를 비교하고자 한다.

# 테스트 환경

모델명: MacBook Pro (13-inch, 2017, Two Thunderbolt 3 ports)

프로세서 이름: Intel Core i5

프로세서 속도: 2.3 GHz

프로세서 개수: 1

총 코어 개수: 2

L2 캐시(코어당): 256 KB

L3 캐시: 4 MB

메모리: 8 GB

# 테스트 할 알고리즘

Quicksort

**algorithm** quicksort(A, lo, hi) **is**

**if** lo < hi **then**

p := partition(A, lo, hi)

quicksort(A, lo, p - 1 )

quicksort(A, p + 1, hi)

**algorithm** partition(A, lo, hi) **is**

pivot := A[hi]

i := lo - 1

**for** j := lo **to** hi - 1 **do**

**if** A[j] < pivot **then**

**if** i != j **then**

i := i + 1

swap A[i] with A[j]

i := i + 1

swap A[i] with A[hi]

**return** i

# 코드

C언어에서 구현한 코드

int partition(int[] array, int left, int right) {  
 int wall = left;  
 int pivot = right;  
  
 for (int i = left; i < right; i++) {  
 if (array[i] <= array[pivot]) {  
 int temp = array[wall];  
 array[wall] = array[i];  
 array[i] = temp;  
 wall++;  
 }  
 }  
 int temp = array[wall];  
 array[wall] = array[pivot];  
 array[pivot] = temp;  
  
 return wall;  
}  
  
void quicksort(int[] array, left, right) {  
 if (left >= right) {  
 return;  
 }  
  
 int pivot = partition(array, left, right);  
 quicksort(array, left, pivot - 1);  
 quicksort(array, pivot + 1, right);  
}

java에서 구현한 코드

private static void quicksort(int[] array, int left, int right) {  
 if (left >= right) {  
 return;  
 }  
  
 int pivot = *partition*(array, left, right);  
 *quicksort*(array, left, pivot - 1);  
 *quicksort*(array, pivot + 1, right);  
}  
  
private static int partition(int[] array, int left, int right) {  
 int wall = left;  
 int pivot = right;  
  
 for (int i = left; i < right; i++) {  
 if (array[i] <= array[pivot]) {  
 int temp = array[wall];  
 array[wall] = array[i];  
 array[i] = temp;  
 wall++;  
 }  
 }  
 int temp = array[wall];  
 array[wall] = array[pivot];  
 array[pivot] = temp;  
  
 return wall;  
}

Scala에서 구현한 코드

**def** swap(a: Array[Int], pos1: Int, pos2: Int): Unit = {  
 **val** stash = a(pos1)  
 a(pos1) = a(pos2)  
 a(pos2) = stash  
}  
  
**def** quicksort(a: Array[Int], low: Int, hi: Int): Unit = {  
 **if** (low < hi) {  
 **val** p = *partition*(a, low, hi)  
 *quicksort*(a, low, p-1)  
 *quicksort*(a, p+1, hi)  
 }  
}  
  
**def** partition(subArray: Array[Int], low: Int, hi: Int): Int = {  
 **val** pivot = hi;  
 **var** i = low;  
 **for** (  
 j <- low to hi  
 **if** subArray(j) < subArray(pivot)  
 ) {*swap*(subArray, i, j); i+=1}  
  
 *swap*(subArray, i, pivot);  
 **return** i  
}

# 실행시간

평균 시간이 걸리는 quicksort라 그런지 웬만한 사이즈로는 0밀리세컨드가 나와서 입력크기를 10만개짜리로 주었다. 다음은 각자 100번 실행한 후 평균 실행 시간이다.

C: 212ms

Java: 184ms

Scala: 160ms

# 결과분석

일반적인 통념인 Java가 JVM 상에서 코드가 돌아가기 때문에 구조적으로 네이티브 코드를 돌리는 C에 비해 빠를 수가 없다라는 것과 전혀 다른 결과가 나왔다. 이러한 결과를 알기 위해서는 JVM의 구동 방식에 대해 알 필요가 있으며 다음은 자바프로그램 실행과정이다.

1. 프로그램이 실행되면 JVM은 OS로부터 이 프로그램이 필요로 하는 메모리를 할당받는다.
2. 자바 컴파일러(javac)가 자바 소스코드(.java)를 읽어들여 자바 바이트코드(.class)로 변화시킨다.
3. Class Loader를 통해 class파일들을 JVM으로 로딩한다.
4. 로딩된 class 파일들은 Execution engine을 통해 해석된다.
5. 해석된 바이트코드들은 Runtime Data Areas에 배치되어 실질적인 수행이 이루어지게 된다.

스크린샷이(가) 표시된 사진



자동 생성된 설명

항상 이러한 과정을 거치기 때문에 전체적인 실행속도가 C보다 떨어지는 것이며, IO와 같은 커널의 시스템콜을 이용해야하는 코드가 아니면 자바의 실행속도가 C보다 떨어지지 않는다. 또한 JVM에서 처음부터 메모리를 할당받기 때문에 C에 비해서 전체적으로 메모리가 더 필요한 경우가 많다.

성능적인 측면에서는 생각보다 자바가 부족하지 않은 모습을 보여줬지만, 생산성의 측면에서보면 자바가 매우 장황하게 코드가 짜여진다. 파일 입출력을 하기 위해서도 여러 객체들을 만들어주어야 하며, 이에 따라 C에 비해 코드가 불필요하게 길어지는 경향이 있다. 이는 자바의 태생적 한계로서, 개발 목적이 OS에 관계없이 서버를 제어하는 코드를 만들려하였기 때문에 많은 소프트웨어 공학들을 접목시키려 하여 모듈화를 많이 하기 때문이다. 프로젝트의 규모가 커진다면 자바는 이러한 점 때문에 좀 더 간편하게 모듈화를 하고 추상화를 시켜 관리할 수 있지만 C는 코드가 매우 복잡해진다. 하지만 프로그램의 규모가 작다면 이러한 구조를 여전히 유지하기 때문에 장황할 수 밖에 없는 코드가 나오게 되는 것이다.