Solución para Lisp++ (Distributed Google Code Jam)

Iván Alejandro Soto Velázquez

A00225851

Tecnológico de Monterrey, Campus Querétaro

A00225851@itesm.mx

Abstract

El presente artículo presenta mi solución para un problema que es encontrado en la ronda 2 del concurso de programación Distributed Google Code Jam del 2016. Dicho problema, Lisp++, pregunta cuál es la longitud del prefijo más largo de una cadena de paréntesis abiertos y cerrados tal que, agregando una secuencia posiblemente vacía de paréntesis al final de ese prefijo, se obtenga una secuencia balanceada. Debido a las restricciones del problema, los concursantes se ven obligados a formular una solución que corra en paralelo.

La solución es, a diferencia de aquella de los concursantes, presentada en tres librerías diferentes. Para cada una se presentará la complejidad en tiempo de la solución y se obtiene el tiempo de una instancia de su ejecución para la entrada que demanda más trabajo. Asimismo, se hará una comparación entre el tiempo de ejecución entre las librerías utilizando métricas estándar.

Keywords: distributed, google, code, jam, lisp++, java threads, openmp, tbb

1 Planteamiento del problema

Un nuevo lenguaje de programación será creado, inspirado en Lisp. Dicho lenguaje será llamado Lisp++. Un programa válido de Lisp cumple las siguientes reglas: (P se refiere a algún programa ya válido de Lisp++)

- () Solamente un paréntesis abierto y un paréntesis cerrado
- (P) Un programa encerrado por un par de paréntesis.
- PP Dos programas (no necesariamente iguales) consecutivos

El compilador de Lisp++ en que se está trabajando debe evaluar una cadena consistiendo de caracteres (y/o) para determinar si es un programa válido de Lisp++, y proveer tal resultado al usuario. Si el programa es válido, el compilador deberá imprimir -1. De lo contrario, deberá imprimir la longitud del prefijo más largo que puede ser extendido a un programa válido si son añadidos cero o más caracteres adicionales al final. Si ese prefijo es el prefijo vacío, el compilador deberá imprimir 0. Particularmente, si la cadena de entrada no es un programa válido, pero puede ser extendida a un programa válido, el compilador deberá imprimir la longitud de la cadena de entrada. Restricciones: $|S| \leq 10^9$

Por ejemplo:

- ()(()()) es un programa válido, por lo que el compilador deberá imprimir -1.
- ((())) no es un programa válido. El prefijo (()) es el prefijo más largo que es o puede ser extendido a un programa válido (en este caso, ya es uno), por lo que el compilador deberá imprimir 4. El único prefijo más largo es ((())) (en efecto, la cadena completa), pero no hay manera de añadir (ningún número de) caracteres al final para convertirlo en un programa válido.
-) no es un programa válido. El prefijo) no puede ser extendido en un programa válido. El prefijo vacío no es un programa válido, pero fácilmente puede ser extendido en uno (añadiendo (), por ejemplo). Así que el compilador deberá imprimir 0.

Dada una cadena, ¿qué deberá imprimir el compilador de Lisp++?

El enunciado del problema se encuentra en la siguiente dirección:
https://code.google.com/codejam/contest/7244486/dashboards=p2

2 Solución

Primero, pensemos en cómo resolver este problema a través de una solución secuencial. Este problema es un tanto clásico y se puede resolver usando como idea la forma en que una pila opera (LIFO).

Cuando encontramos un paréntesis abierto, añadimos ese paréntesis al tope de la pila. En algún momento puede que encontremos un paréntesis cerrado, y hay que eliminar el paréntesis abierto que está en el tope de la pila. Podemos observar que la pila, si no está vacía, siempre consistirá de paréntesis abiertos. Remover un paréntesis abierto de la pila corresponde a formar una pareja de paréntesis, y por la definición de una cadena de paréntesis válida presentada en el planteamiento del problema, remover tal pareja no altera la validez de la solución. Otra observación es que el número de paréntesis abiertos siempre debe ser mayor o igual al número de paréntesis cerrados para todo prefijo de la cadena. En caso contrario, habrá al menos un paréntesis cerrado que ningún paréntesis abierto podrá cancelar. Por último, la cadena es válida si, tras procesar todos los paréntesis de la cadena, la pila está vacía. Esto es porque una pila vacía implica que todo paréntesis abierto fue apareado con un paréntesis cerrado.

Para resolver el problema usando un contador, podemos observar que, porque la pila solo contiene paréntesis abiertos, no hay necesidad de identificar cada caracter de la pila, por ser el mismo. Podemos simular la pila sumando uno al contador cuando nos encontramos con un paréntesis abierto, y restar uno cuando encontramos un paréntesis cerrado. Si el contador se vuelve negativo, nunca será posible lograr que la secuencia sea balanceada, y al final el contador deberá ser cero.

Ahora, teniendo esta información, ¿cómo sabemos que imprimirá el compilador de Lisp++? Regresando a la solución del contador, si en algún punto dicho contador se

vuelve negativo, imprimimos la longitud del prefijo antes de procesar el paréntesis cerrado que ocasionó la negatividad del contador. Si el contador jamás se volvió negativo, imprimimos -1 si el contador es cero al final, o la longitud de la cadena en caso contrario. Esta solución tiene complejidad $O(|S|)$, donde S es la cadena que fue procesada.

Teniendo en mente la solución secuencial, podemos empezar a pensar en cómo paralelizar. Podemos ver que la validez de una cadena de paréntesis hasta una posición i depende de la validez de la cadena hasta una posición $i-1$, así recursivamente hasta llegar a la cadena vacía. Pensando en términos de subcadenas de longitud 1 no ayuda a formular una solución que realmente sea paralela. Aquí entra la observación clave:

¿Qué tal si en vez de dividir la cadena en subcadenas de longitud 1, y suponiendo que estamos en la subcadena i, esperando a que obtengamos el balance para todo prefijo que termina en j para $j < i$ de manera ordenada (por lo que hay que esperar a un $j = i-1$, y así para cada prefijo que termina en $k \leq j$), dividimos la cadena en subcadenas de una longitud K, calculamos el balance en cada subcadena i, esperando a una subcadena $i-1$, que dicha también espera a una subcadena $i-2$, así hasta la subcadena vacía?

Por ejemplo, dado $S = (())()$ y $K = 2$, introducimos cada subcadena de longitud K en una función *calculate* que regresa el balance y se obtiene lo siguiente:

- $calculate("((") = 2$
- $calculate("))") = -2$
- $calculate("()") = 0$

Suponiendo que hacemos estas invocaciones de *calculate* en orden, esperando a que la anterior acabe, ¿cómo sabemos si aún la cadena puede potencialmente estar balanceada?

Si de la secuencia de subcadenas $S_{s_1}, S_{s_2}, ..., S_{s_{\frac{|S|}{K}}}$ estamos en $i = 2$, donde obtuvimos $calculate(S_{s_2}) = -2$, y porque $calculate(S_{s_1}) = 2$, podemos observar que el balance negativo de la segunda subcadena se puede equilibrar con el balance positivo de la primera subcadena, siendo así que $calculate(S_{s_1}) + calculate(S_{s_2}) = 0$. Si después consideramos $calculate(S_{s_3}) = 0$, vemos que el balance se mantiene en cero y por lo tanto la cadena de paréntesis es válida.

Dadas estas observaciones, llegamos a lo siguiente: Dado $S_{s_1}, S_{s_2}, ..., S_{s_{\frac{|S|}{K}}}$, la secuencia de subcadenas puede ser convertida en una balanceada agregando paréntesis al final si para todo $1 \leq i \leq K$, $\sum_{j=1}^{i} S_{s_j} \geq 0$. Si $\sum_{j=1}^{|S|} S_{s_j} = 0$, es una cadena balanceada sin necesidad de agregar paréntesis al final.

¿Qué pasaría si existe un i tal que $\sum_{j=1}^{i} S_{s_j} < 0$? Para dicho primero i, eso significa que en esa subcadena es que se identifica que lograr que la cadena sea balanceada es imposible. Porque en la solución secuencial establecimos que la longitud del prefijo justo antes de que el contador se volvería negativo corresponde al prefijo más largo que podría balancearse, tal prefijo termina también en esta subcadena. Podemos iterar sobre esta

subcadena, iniciando con un balance $\sum_{j=1}^{i-1} S_{s_j}$ y deteniéndonos en el primer momento en que el balance se vuelve negativo.

Por último, una observación crucial: no es necesario esperar a los balances de las subcadenas j si estamos procesando la subcadena i para $j < i$. Podemos asignar a cada thread una subcadena de manera independiente, y que cada uno notifique al thread master de sus resultados. Ya que acabaron todos los threads, podemos ordenarnos con respecto a la posición de inicio de su subcadena (o bien pudimos inicialmente colocarlo en su lugar correcto en un arreglo de longitud $|S|/K$), y después llevar a cabo la suma en cada prefijo como fue descrita anteriormente para resolver el problema. La complejidad de esta solución es $O(\frac{|S|}{T} + T * C)$, donde S es la cadena procesada, T es el número de threads involucrados y C es una constante que representa el overhead del mecanismo de paralelismo de cada librería.

3 Implementaciones

Utilizando tres tecnologías diferentes, Lisp++ fue solucionado. En el Distributed Google Code Jam, el número de nodos con los que los concursantes se podían comunicar eran hasta $N \leq 100$. Para asemejar lo más posible la solución esperada, las implementaciones en las 4 tecnologías divide el trabajo en a lo mucho $T = 100$ threads. Porque $|S| \leq 10^9$, $\frac{|S|}{T} \geq 10^7$, si la desigualdad se vuelve igualdad y porque 10^7 operaciones se realizan fácilmente en menos de un segundo, obtener que cada solución corre en menos de ese tiempo se vuelve factible.

Las funciones más comunes que vemos en las implementaciones son $solve$, $findErrorIndex$, que se encargan de calcular balances para cada subcadena y encontrar la longitud del prefijo de un sufijo en donde ocurre el primer balance ≤ 0.

3.1 Java Threads

Por la forma en que se planteó la solución, se ajusta perfectamente al paradigma que ofrecen los threads en Java. Cada thread se encarga de una subcadena, y al terminar, se calcula la suma de prefijos.

3.1.1 Lisp.java

```
package lisp;

/**
 * Created by ivan on 26/11/17.
 */
public class Lisp extends Thread {
  private String s;
  private int begin, end;
  private int balance;

  public Lisp(String s, int begin, int end) {
    this.s = s;
```

```
    this.begin = begin;
    this.end = end;
    this.balance = 0;
  }

  public int getBalance() {
    return balance;
  }

  public int getBegin() {
    return begin;
  }

  public int getEnd() {
    return end;
  }

  public void run() {
    for (int i = begin; i < end; i++) {
      if (s.charAt(i) == '(') {
        balance++;
      } else if (s.charAt(i) == ')') {
        balance--;
      } else assert(false);
    }
  }
}
```

3.1.2 Main.java

```
package lisp;

import java.util.Scanner;

public class Main {
  private static int T = 100;

  public static int findErrorIndex(String s, int begin, int end, int balance) {
    for (int i = begin; i < end; i++) {
      if (s.charAt(i) == '(') {
        balance++;
      } else if (s.charAt(i) == ')') {
        balance--;
        if (balance < 0) return i - begin;
      }
    }
    assert(false);
    return -1;
```

```
  }

  public static int solve(String s, Lisp[] threads) {
    int[] balances = new int[threads.length];
    int prefixSum = 0;
    int length = 0;
    for (int i = 0; i < threads.length; i++) {
      balances[i] = threads[i].getBalance();
      if (prefixSum + threads[i].getBalance() < 0) {
        length += findErrorIndex(s, threads[i].getBegin(), threads[i].getEnd(), prefi
        return length;
      }
      length += threads[i].getEnd() - threads[i].getBegin();
      prefixSum += threads[i].getBalance();
    }
    if (prefixSum == 0) return -1;
    assert(length == s.length());
    return length;
  }

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    String s = sc.nextLine();
    long startTime = System.currentTimeMillis();
    Lisp threads[] = new Lisp[Math.min(s.length(), T)];
    int range = (s.length() + threads.length - 1) / threads.length;
    for (int i = 0; i < threads.length; i++) {
      threads[i] = new Lisp(s, i * range, Math.min(s.length(), (i + 1) * range));
    }
    for (int i = 0; i < threads.length; i++) {
      threads[i].start();
    }
    for (int i = 0; i < threads.length; i++) {
      try {
        threads[i].join();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    int ans = solve(s, threads);
    long stopTime = System.currentTimeMillis();
    System.out.println(ans);
    System.out.println("Program's execution time: " + (stopTime - startTime) + "ms");
  }
}
```

3.2 OpenMP

Para la solución en OpenMP, podemos crear tasks que serán ejecutados de manera paralela. La asignación ocurre bajo una directiva *pragma*, en la que se define el inicio y fin de una subcadena y se manda a llamar *solve* para calcular el balance.

3.2.1 openmpsol.cpp

```
/*------------------------------------------------------------
 *
 * Actividad de programación: Solución para Lisp++
 * Fecha: 26-Nov-2017
 * Autor: A00225851 Iván Alejandro Soto Velázquez
 *
 *------------------------------------------------------------*/


#include <bits/stdc++.h>
#include "utils/cppheader.h"
#include <omp.h>

using namespace std;

const int N = 1000010;

int T;
int n;
char s[N];
struct thread {
  int balance;
  int b, e;
} threads[111];

void solve(char *s, int tid, int b, int e) {
  int balance = 0;
  threads[tid].b = b;
  threads[tid].e = e;
  for (int i = b; i < e; i++) {
    if (s[i] == '(') {
      balance++;
    } else if (s[i] == ')') {
      balance--;
    } else assert(0);
  }
  threads[tid].balance = balance;
}

int find_error_index(char *s, int b, int e, int balance) {
  for (int i = b; i < e; i++) {
```

```
    if (s[i]== '(') {
      balance++;
    } else if (s[i] == ')') {
      balance--;
      if (balance < 0) return i - b;
    }
  }
  assert(false);
  return -1;
}

int work(char *s) {
  int tid, ntids;
  int range = (n + T - 1) / T;
  int start, fin;
  #pragma omp parallel private(tid, start, fin) num_threads(T)
  {
    tid = omp_get_thread_num();
    start = tid * range;
    fin = min(n, (tid + 1) * range);
    solve(s, tid, start, fin);
  }
  int prefix_sum = 0;
  int len = 0;
  for (int i = 0; i < T; i++) {
    if (prefix_sum + threads[i].balance < 0) {
      len += find_error_index(s, threads[i].b, threads[i].e, prefix_sum);
      return len;
    }
    len += threads[i].e - threads[i].b;
    prefix_sum += threads[i].balance;
  }
  if (prefix_sum == 0) return -1;
  assert(len == n);
  return len;
}

int main() {
  scanf("%s", s);
  double ms = 0;
  Timer t;
  t.start();
  n = strlen(s);
  T = min(n, 100);
  int ans = work(s);
  ms += t.stop();
  printf("%d\n", ans);
```

```
  fprintf(stderr, "Program's execution time: %.6lf ms\n", ms);
  return 0;
}
```

3.3 Intel TBB

Intel TBB ofrece *task_group*, entonces es posible escribir una solución similar a la de Java Threads. Para cada miembro del grupo de task, se le asigna un rango de la cadena, y se espera a todos los threads para calcular el resultado final.

3.3.1 tbbsol.cpp

```
/*----------------------------------------------------------------
 *
 * Actividad de programación: Solución para Lisp++
 * Fecha: 26-Nov-2017
 * Autor: A00225851 Iván Alejandro Soto Velázquez
 *
 *--------------------------------------------------------------*/


#include <bits/stdc++.h>
#include "utils/cppheader.h"
#include <omp.h>
#include <tbb/task_scheduler_init.h>
#include <tbb/task_group.h>

using namespace std;
using namespace tbb;

const int N = 1000010;

int T;
int n;
char s[N];
struct thread {
  int balance;
  int b, e;
} threads[111];

int find_error_index(char *s, int b, int e, int balance) {
  for (int i = b; i < e; i++) {
    if (s[i]== '(') {
      balance++;
    } else if (s[i] == ')') {
      balance--;
      if (balance < 0) return i - b;
    }
  }
```

```
  assert(false);
  return -1;
}

int solve(char *s) {
  int prefix_sum = 0;
  int len = 0;
  for (int i = 0; i < T; i++) {
    if (prefix_sum + threads[i].balance < 0) {
      len += find_error_index(s, threads[i].b, threads[i].e, prefix_sum);
      return len;
    }
    len += threads[i].e - threads[i].b;
    prefix_sum += threads[i].balance;
  }
  if (prefix_sum == 0) return -1;
  assert(len == n);
  return len;
}

class SolveTask {
  public:
  int tid, b, e;
  char *s;

  SolveTask(char *s, int tid, int b, int e) : s(s), tid(tid), b(b), e(e) {}

  void operator() () const {
    int balance = 0;
    threads[tid].b = b;
    threads[tid].e = e;
    for (int i = b; i < e; i++) {
      if (s[i] == '(') {
        balance++;
      } else if (s[i] == ')') {
        balance--;
      } else assert(0);
    }
    threads[tid].balance = balance;
  }
};

int main() {
  scanf("%s", s);
  double ms = 0;
  Timer t;
  t.start();
```

```
  n = strlen(s);
  T = min(n, 100);
  task_group tg;
  int range = (n + T - 1) / T;
  for (int i = 0; i < T; i++) {
    tg.run(SolveTask(s, i, i * range, min(n, (i + 1) * range)));
  }
  tg.wait();
  int ans = solve(s);
  ms += t.stop();
  printf("%d\n", ans);
  fprintf(stderr, "Program's execution time: %.6lf ms\n", ms);
  return 0;
}
```

3.4 Análisis de tiempo de ejecución

La siguiente tabla muestra los tiempos de ejecución de cada programa bajo una cadena de 10^7 caracteres y los speedups comparando las versiones secuenciales contra las versiones paralelas. El programa secuencial en C++ corrió en **36 ms** y la versión de Java en **42 ms**.

	Java Threads	OpenMP	Intel TBB
C++ secuencial	1.16	1.63	1.89
Java secuencial	1.35	1.90	2.21