Validate Parentheses

CSE 702 - Programming Massively Parallel Systems

Instructor - Dr. Russ Miller

Prepared by - Madhushri Patil

Agenda

- The Problem
- Applications
- Generating Parentheses dataset
- The Algorithm
- Results
- Observations
- Conclusions
- Challenges
- References

The Problem

Given a sequence of parentheses S, validate the sequence.

$S_i = <S_0, S_1, ... , S_{n-1}>$ where $0 <= i < n$

Sequence S_i is said to be valid if and only if -

- For every opening parenthesis ‘(’ there is a corresponding closing parenthesis ‘)’.
- The matched parentheses should be in the correct order, i.e., an opening parenthesis should appear before the closing parenthesis.

Example -

(( ) (( ))) Valid Parentheses

((( )) ((Invalid Parentheses

Applications

- Parentheses-heavy programming languages like Java, Javascript in order to ensure if the code has correct number of parentheses.

- Modern text editors / Integrated Development environments (IDE) that support highlighting the matching opening and closing parenthesis.

Generating Parentheses dataset

Implemented a Java program to generate parentheses for the given size n

```java
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;

public class GenerateParentheses {

    private static void generateParenthesis(String cur, int open, int close, int max, OutputStreamWriter osw) throws IOException {

        if(cur.length() == (max * 2)) {
            osw.write(cur);
            return;
        }

        if(open > 0)
            generateParenthesis(cur+"(", open-1, close, max, osw);
        if(Math.abs(open - close) > 0)
            generateParenthesis(cur+")", open, close-1, max, osw);
    }

    public static void main(String args[]) {
        int n = Integer.parseInt(args[0]);
        String fileName = args[1];
        try {
            File fout = new File(fileName);
            FileOutputStream fos = new FileOutputStream(fout);
            OutputStreamWriter osw = new OutputStreamWriter(fos);
            generateParenthesis("", n, n, n, osw);
            osw.close();
        } catch (Exception e) {
            System.out.println("File write error!");
        }
    }
}
```

The Algorithm

1. For each process, compute local parallel prefix sum as follows -
    a. Assign 1 for each left parenthesis and -1 for each right parenthesis
    b. Compute parallel prefix sum for the data each process has

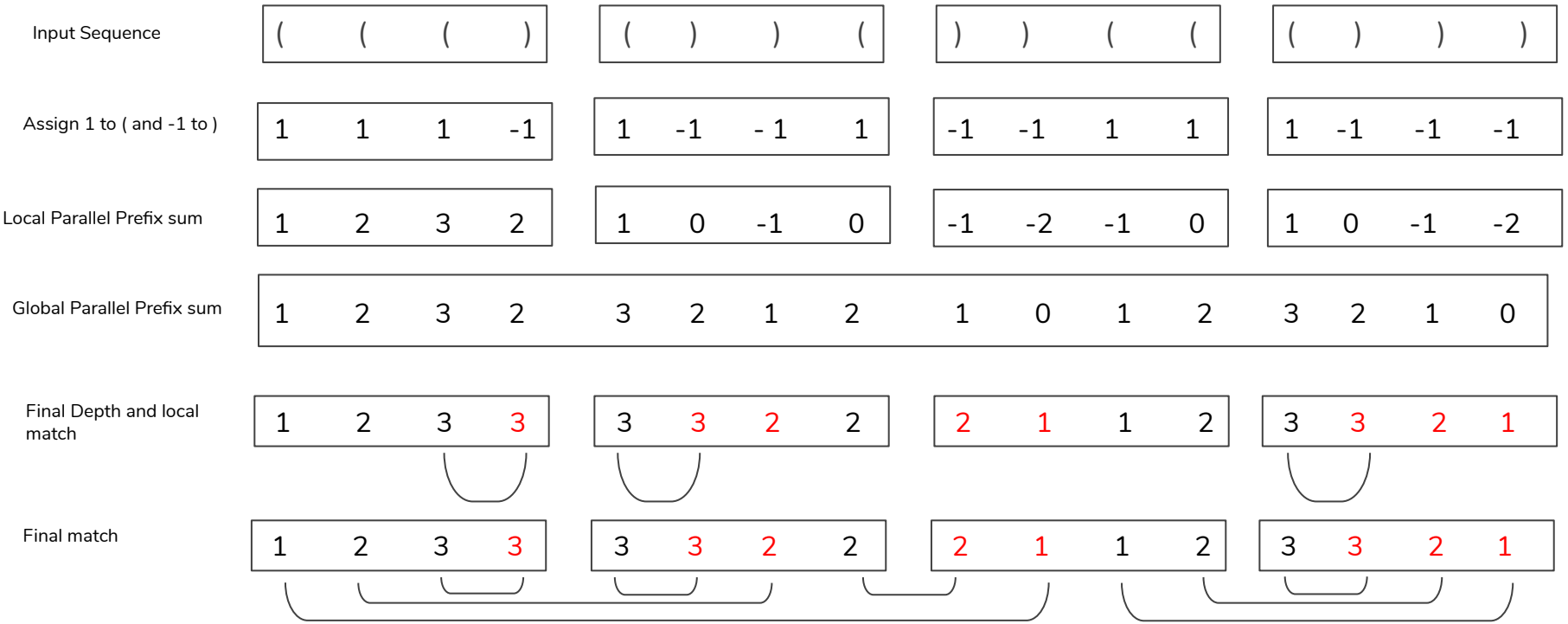
2. Compute global parallel prefix sum for the entire data sequence across all processes

3. For every process, compute the depth/nesting for parentheses as follows -
    a. Increment the value of prefix_sum[i] for every closing parenthesis ‘)’ : 0 <= i <= n

4. Once every process has final depth values, the algorithm does the following -
    a. Every process computes the match for every opening parenthesis in its local subsequence as -
        i. For each opening parenthesis, find its closest closing parenthesis with the same depth
        ii. Mark True in match[i] against each parenthesis for which a match is found
    b. Perform parallel merging and matching parentheses for the rest of the subsequences until Process 0 has the entire sequence.

5. Process 0 returns if the given sequence is Valid / Invalid iff every match[i] = True.

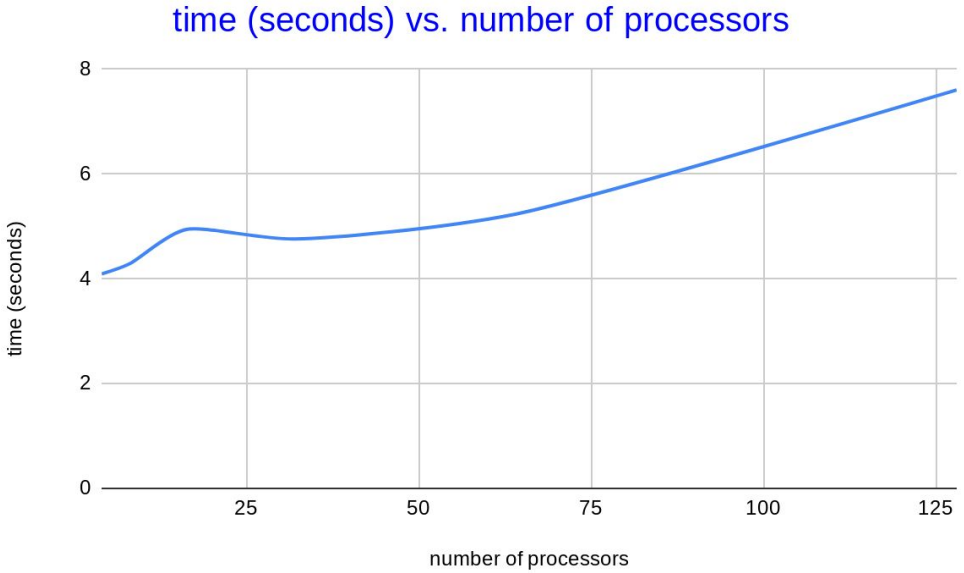
Example


Input Sequence
((( ) ( )) ( )) ((( )))
Assign 1 to (and -1 to)
1 1 1 -1 1 -1 -1 1 -1 -1 1 1 1 -1 -1 -1
Local Parallel Prefix sum
1 2 3 2 1 0 -1 0 -1 -2 -1 0 1 0 -1 -2
Global Parallel Prefix sum
1 2 3 2 3 2 1 2 1 0 1 2 3 2 1 0
Final Depth and local match
1 2 3 3 3 3 2 2 2 1 1 2 3 3 2 1
Final match
1 2 3 3 3 3 2 2 2 1 1 2 3 3 2 1

Results

Constant data per Processor
(~90K characters per Processor)

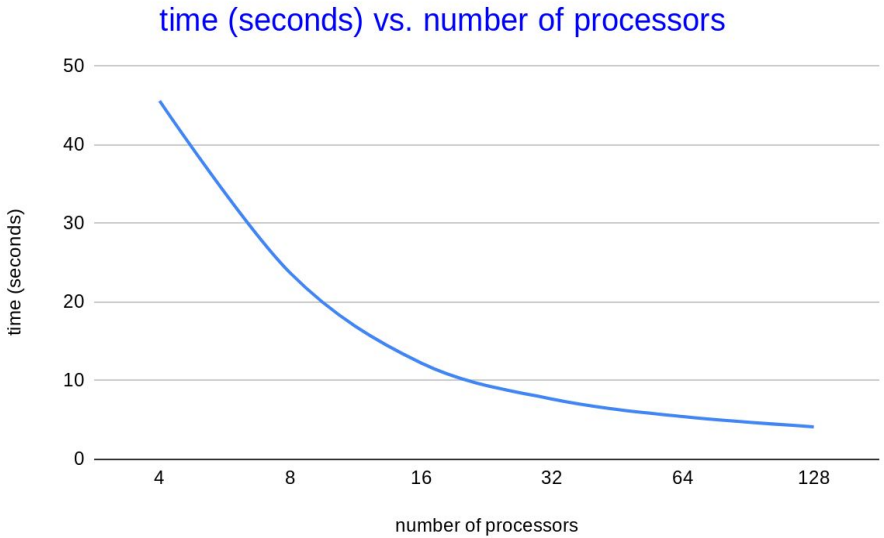
Number of Processors	Average time taken by each Processor (seconds)
4	4.09418
8	4.28352
16	4.93446
32	4.76223
64	5.23540
128	7.60596



Results

Constant total data (~1 Million characters)
Variable data per Processor

Number of Processors	Average time taken by each Processor (seconds)
4	45.60996
8	23.68024
16	12.24782
32	7.66502
64	5.43394
128	4.14124



Observations

Constant data per processor	Variable data per processor
● Response times are seen increased in cases where the matches found locally are comparatively less than those found in the other cases ● Cost of communication is reflected by the increase in time with increasing number of processors	● Significant decrease in the time required is observed upto 32 processors ● Cost of communication increases after 32 processors for the data considered in this experiment and hence the response time

Conclusions

- Increasing number of processors does not always result in better response times, as the cost of communication increases

- Constant data per processor does not always guarantee same response time at each processor, due to the communication involved between processors

Challenges

- Ran into insufficient memory errors when the input files were too big
- Running the script on 256 was time consuming and ran into a issues like getting incorrect results and memory errors.

References

- Algorithms Sequential & Parallel: A Unified Approach (Dr. Russ Miller, Dr. Laurence Boxer)
- Christos Levcopoulos, Ola Petersson, “Matching parentheses in parallel”, Discrete Applied Mathematics 40 (1992) 423-431

Thank You!