1 Required Variables

In this section we refer all the variables with its type definition and data it stores that are used in the algorithm.

1.1 reducedStr

Type: Character Array

Description

This stores the reduced format of the input string which is to be compiled. It only contains the following characters ('<', '>', '='). In the algorithm we refer '<' as yield and '>' as a take.

1.2 nxtIdx

Type: Integer Array

Description

This is used to get the next index during an iteration or any other purpose. Generally '$i+1$' is used as the next index for 'i', but in the algorithm nxtIdx[i] is used as the next index. The array is updated frequently during the run time of the algorithm.

1.3 globalRefernceTable

Type: *Partition Array (User Defined Class)

Description

Each element of the array is an reference to a Partition object and each object is responsible for a range of data of reducedStr, and no two elements in the globalRefernceTable are responsible for the same index in reducedStr. This observation helps us to process the array in parallel since no two elements have a dependency.

We use smart pointers to handle the references, so that even in case of a mistake in handling the dangling pointer the memory is properly cleaned. The class structure and kind of data it stores is defined below.

1.4 lockTable

Type: Boolean Array

Description

Each element of the array represents the respective element in the globalRefernceTable. If the element in lockTable is $True$, it implies that the respective array element of globalRefernceTable is still under process and the process wants to use it must either wait until it gets finished or can move on with any other element.

2 Functions Used

The purpose of this section is to define all the functions that are used in the algorithm.

2.1 Solver

Arguments: Integer idx, Functor process — Return Type: void

Description

This function takes the index (idx) of the globalReferenceTable and applies the function which is passed as an argument (process) to it. This locks the respective index as long as the process is taking place. The function 'process' contains the logic on how the reduction happens. All the solvers for each index is called in parallel.

2.2 hasYields

Arguments: Integer idx1, Integer Idx2 — Return Type: boolean

Description

This function returns boolean value depending on whether there exists an 'yield' in the given range (idx1, idx2).

2.3 hasTakes

Arguments: Integer idx1, Integer Idx2 — Return Type: boolean

Description

This function returns boolean value depending on whether there exists a 'take' in the given range (idx1, idx2).

2.4 InitData

Arguments: String reducedStr — Return Type: void

Description

This function creates Partition objects with proper construction and assign them to respective globalReferenceTable indexes. Respective solver is applied on each of those indexes.

2.5 Handler

Arguments: Integer size — Return Type: void

Description

This takes the size of the globalReferenceTable as the argument and executes multiple iterations as long as the size reduces to 1. In each iteration, some of the indices of globalReferenceTable will be merged which leads to reduction in size.

2.6 Merger

Arguments: Integer idx1, Integer idx2 — Return Type: void

Description

This takes two consecutive indices of the globalReferenceTable and merge the data in them optimally. This is how reductions happen in the program. This function is also responsible for the updates in the nxtIdx array.

3 Pseudo Code

In this section we define the pseudo codes of all the functions that are mentioned in the previous section.

3.1 Solver

Required variables:

Algorithm 1: solver

3.2 InitData

Required variables:

```
input  : Reduced string of the given input text
output : Void, globalReferenceTable is instantiated and solver is applied
         for each element in globalReferenceTable

n ← length of reducedStr;
randomK ← rand(n);
// rand gives a random value, which is used as the length to make
   an arbitrary cut
for i ← 1 to n do
    idx1 = i;
    idx2 = max(i + k, n);
    yield ← hasYield(idx1, idx2);
    take ← hasTake(idx2, idx2);
    insert into globalReferenceTable ← new Partition(idx1, idx2, yield, take);
    parallel solver(sizeof globalReferenceTable);
end
```

Algorithm 2: initData