Prefix Sum Application

1 Polynomial Evaluation Problem

Given $a_0, a_1, ..., a_{n-1}$ and x_0, we want to compute $P(x_0)$, where:

$$P(x) = \sum_{i=0}^{n-1} a_i x^i \tag{1}$$

1.1 Serial

Because we need to traverse every a and add them together, so the runtime will be:

$$T(n, 1) = \Theta(n) \tag{2}$$

1.2 Parallel

1.2.1 Algorithm

Assume we have p processors, and assume coefficients are block distributed. Then the processor P_i contains:

$$a_{i\frac{n}{p}}, a_{i\frac{n}{p}+1}, ..., a_{(i+1)\frac{n}{p}-1} \tag{3}$$

The algorithm is explained below:

1. P_0 reads x_0 and broadcast it.
2. Perform n-element parallel prefix with 1 as the first element, x_0 as all other elements and multiplication as the operation. Inside the operation, usually it does not have the operation like x^3, so we need to multiply x to achieve this.
3. For example, the first processor may have:

$$1, x, x^2, x^3 \tag{4}$$

4. Each processor computes n/p terms locally and adds them
5. Sum the partial results, one on each processor

1.2.2 Runtime Analysis

Recall the previous chapter, within one processor we need to do n/p multiplication, and after that we need to add p values together. Based on the divide-and-conquer method, the runtime will be $\log p$. So the total **computation time** will be:

$$\Theta(\frac{n}{p} + \log p) \tag{5}$$

Assume every communication will have the **latency** as τ, and the time taken to send each unit of data as μ, then the total **communication time** will be:

$$\Theta((\tau + \mu) \log p) \tag{6}$$

To be efficient, we want to balance the computational work and overhead due to communication between processors in a parallel computing environment. This depends on the number of processors:

1. If p is too small, then $\frac{n}{p}$ becomes large, and the computation time will be dominated by the work that each processor must perform, which means the potential benefits of parallelism are not fully realized.
2. If p is too large, then the term $\log p$ becomes significant, indicating that the overhead from communication or coordination between an excessive number of processors can become a bottleneck, negating the benefits of additional parallelism.

Therefore, we approximate the number of processors by:

$$\Theta(\frac{n}{p} + \log p) \approx \Theta((\tau + \mu) \log p) \tag{7}$$

In other words, we have:

$$\frac{n}{p} = O(\log p) \tag{8}$$

After numerical method, we can get approximately:

$$p = O(\frac{n}{\log n}) \tag{9}$$

2 Fibonacci Sequences Problem

For the Fibonacci sequence, we assume:

$$f_0 = f_1 = 1 \tag{10}$$

Then the rule is:

$$f_i = f_{i-1} + f_{i-2} \tag{11}$$

In a matrix form:

$$\left[f_i, f_{i-1}\right] = \left[f_{i-1}, f_{i-2}\right] \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \tag{12}$$

Then if we define:

$$V_i = \left[f_i, f_{i-1}\right] \tag{13}$$

And:

$$M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \tag{14}$$

Then the sequence could be generalized as:

$$V_i = V_1 \times M^{i-1} \tag{15}$$

Then the problem could be transferred as the prefix problem.

3 Tree Accumulation

Tree accumulation in computer science is a method used to aggregate data from a collection of elements in a tree-like structure. In a tree data structure, each node typically contains some data and references to its child nodes. The **accumulation** involves applying an operation to combine the data from child nodes and propagate the result up the tree. Tree accumulation can be performed in parallel because operations on different branches of the tree can be done independently before being combined at higher levels.

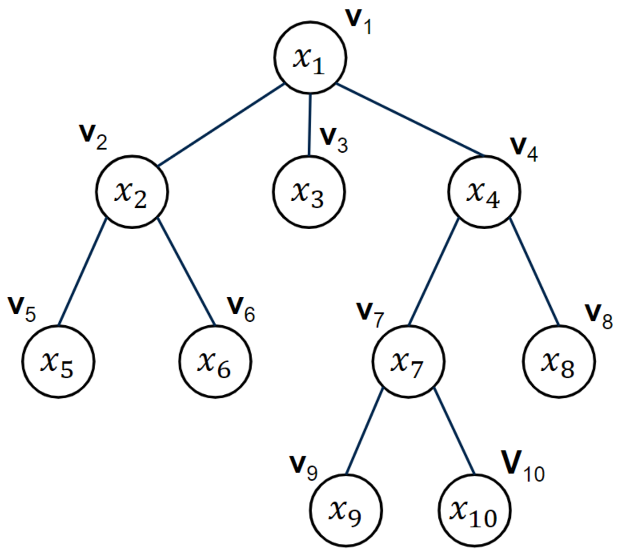

Figure 1: Tree Accumulation

3.1 Euler Tour in Tree

An Euler tour in a tree involves traversing every edge **exactly twice**, once in each direction. This is because, to return to the starting point in a tree and visit all vertices, each edge must be traversed once to reach a new vertex and then again to return. This type of traversal ensures that the tour can visit all vertices of the tree and return to the starting point without leaving any edge untraversed.

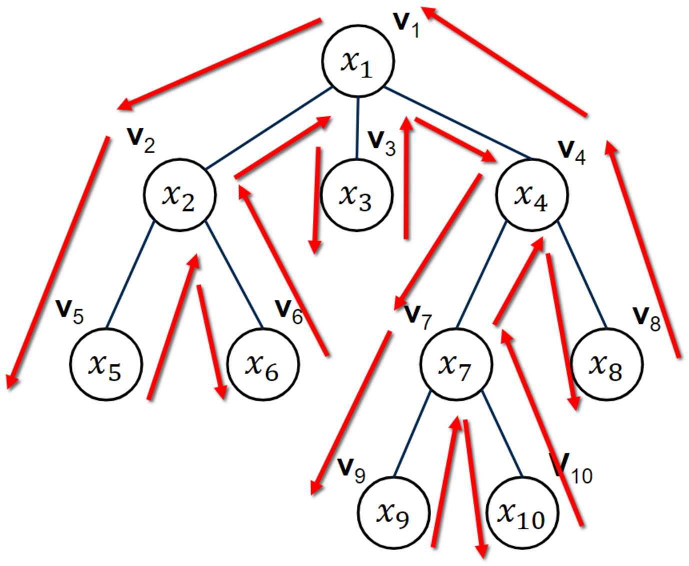

Figure 2: Euler Tour

Basically, each time we traverse to the very end leaf of one branch, then go back to the previous level, visit other leaves. It can be expressed as:

$$ET = v_1, v_2, v_5, v_2, v_6, v_2, v_1, v_3, v_1, v_4, v_7, v_9, v_7, v_{10}, v_7, v_4, v_8, v_4, v_1 \tag{16}$$

In a tree with n nodes, there are $n-1$ edges because a tree is an acyclic connected graph, which means there is exactly one path between any two nodes. For the Euler tour, each edge is visited exactly twice, so the total number of edge traversals is $2(n-1)$. **However the Euler tour is expressed using nodes.** Therefore we need to account for the starting point of the tour once again. So the total length is:

$$|ET| = 1 + 2(n-1) \tag{17}$$

3.2 Upward Accumulation (UA)

For a node v, sum all numbers in the subtree. For example in the previous graph:

$$UA(v_4) = x_4 + x_7 + x_8 + x_9 + x_{10} \tag{18}$$

Notice that **always count the node itself**. In parallel computing, we can use the combination of Euler tour and prefix sum to calculate the upward accumulation. Suppose we have a simplified example:

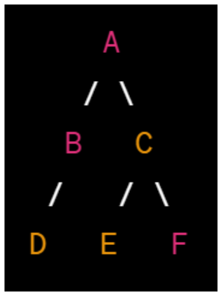

Figure 3: Simple Tree

Assume the values at each leaf are as follows: $D = 4$, $E = 1$, $F = 3$. Now the Euler tour is:

$$\text{ET} = ABDBACECFCA \tag{19}$$

When we visit a node **the first time, we assign its original value**, otherwise we assign 0. Therefore we get the value and prefix sum array:

Node	A	B	D	B	A	C	E	C	F	C	A
Value	0	0	4	0	0	0	1	0	3	0	0
Prefix	0	0	4	4	4	4	5	5	8	8	8

Table 1: Prefix sums array for upward accumulation.

Intuitively, we have the following observations (choose node C for observation):

1. The prefix sum after the **last occurrence** of node C includes **the sum of all the values in the subtree of C**
2. The prefix sum just before the **first occurrence** of C (1 index before C) includes **the sum of all the values from the start up to node C (not including node C).**

Assume:

1. PS: resulting prefix sums array
2. i_f: index of first occurrence of v
3. i_l: index of last occurrence of v

Then the upward accumulation could be calculated as:

$$UA(v) = PS[i_l] - PS[i_f - 1] \tag{20}$$

3.3 Downward Accumulation (DA)

For a node v, sum all numbers in ancestors of v. For example in the Euler tour graph:

$$DA(v_7) = x_7 + x_4 + x_1 \tag{21}$$

We also use the simplified tree as an example, so the Euler tour will be the same. But this time we assign $B = 2, C = 3$. Now at the first occurrence of v_i we assign x_i and the last occurrence of v_i we assign $-x_i$, otherwise we assign 0. **If the node is leaf node, we assign 0.** Therefore the prefix sum array will be:

Node	A	B	D	B	A	C	E	C	F	C	A
Value	0	2	0	-2	0	3	0	0	0	-3	0
Prefix	0	2	2	0	0	3	3	3	3	0	0

Table 2: Prefix sums array for downward accumulation.

As discussed before, the prefix sum just before the first occurrence of C (1 index before C) includes the sum of all the values from the start up to node C (not including node C). Now if we just take the index of node C, we will get the DA of C. The rules are summarized below. If v_i is an internal node:

$$DA(v_i) = PS[i_f] \tag{22}$$

If v_i is a leaf node:

$$DA(v_i) = PS[i_f] + x_i \tag{23}$$

4 Sequence Alignment

4.1 Introduction

Sequence alignment in parallel computing is a method used in bioinformatics to align sequences of DNA, RNA, or proteins to identify regions of similarity. (**Alignment**, in a broad sense, refers to the arrangement of objects or elements in a straight line or in correct or appropriate relative positions.) The goal is to find the **optimal alignment** between two or more sequences, which can provide insights into their evolutionary relationships, functional and structural characteristics, and help identify conserved or significant regions. We define the optimal alignment as **minimum number of substitutions, insertions, and deletions needed to turn one sequence into another.**

4.2 Alignment Score

The alignment score has the following rules:

1. The score shows the alignment quality.
2. Gaps are inserted to denote insertions/deletions.

3. Every column of an alignment is a match, mismatch or a gap.

4. Matches are preferred so have a positive score (1), mismatch (0), gap (-1)

For example:

A	T	G	A	-	C	C
A	-	G	A	A	T	C
1	-1	1	1	-1	0	1

Table 3: Alignment Example

Then the score of this alignment is 2.

4.3 Algorithm

Assume the inputs include two sequences:

$$A = a_1, a_2, ..., a_m \tag{24}$$

$$B = b_1, b_2, ..., b_n \tag{25}$$

and the scores for **only** match/mismatch ($f(a, b)$) and the number of gap g. Then the algorithm includes:

1. Create a table T with size $(m + 1) \times (n + 1)$. The reason for this dimension is to **include the case where one of the sequences is aligned against a zero-length counterpart, allowing for initial gaps.**

2. Now define $T[i, j]$ as the best score between $a_1, a_2, ..., a_i$ and $b_1, b_2, ..., b_j$

3. The value of $T[i, j]$ is determined by taking the maximum of the following three possibilities:

   (a) The score of aligning a_i and b_j ($f(a_i, b_j)$) plus the best score for aligning the subsequences $a_1, a_2, ..., a_{i-1}$ with $b_1, b_2, ..., b_{j-1}$, which could be expressed as: $T[i-1, j-1] + f(a_i, b_j)$.

   (b) The best score of aligning $a_1, a_2, ..., a_{i-1}$ with $b_1, b_2, ..., b_j$ minus the gap penalty, which is $T[i-1, j] - g$

   (c) The best score of aligning $a_1, a_2, ..., a_i$ with $b_1, b_2, ..., b_{j-1}$ minus the gap penalty, which is $T[i, j-1] - g$

   In summary, we have:

$$T[i, j] = \max \begin{cases} T[i-1, j-1] + f(a_i, b_j) \\ T[i-1, j] - g \\ T[i, j-1] - g \end{cases} \tag{26}$$

4. The sequential time is approximated as $O(mn)$ due to the dimension.

4.4 Example

Suppose we want to align the sequences $ATGCT$ and $ACCGCT$, assume that we assign 2 for match, 0 for mismatch, and the result is shown below:

	Gap	A	T	G	C	T
Gap	0	-1	-2	-3	-4	-5
A	-1	2	1	0	0	0
C	-2	1	2	1	2	1
C	-3	0	1	2	3	2
G	-4	0	0	3	2	3
C	-5	0	0	2	5	4
T	-6	0	2	1	4	7

Here we define $T[Gap, Gap]$ as $T[1, 1]$, then start from this point. Notice that $T[i-1, j-1]$ is the previous diagonal point, $T[i-1, j]$ is the cell above, $T[i, j-1]$ is the cell on the left. Therefore:

1. $T[1, 1]$: No diagonal, up or left cell, so 0.
2. $T[1, 2]$: *Gap* in the row meets A in the column, so $g = 1$. Only left cell, so $T[1, 2] = T[1, 1] - g = -1$. Similar with all the first column and first row values. Notice here if we move from $T[1, 1]$ to $T[1, 2]$, we use the *Gap* **in the row** to align with the letters **in the column**, so the *Gap* penalty keeps increasing **in horizontal direction**. Similarly, if we move from $T[2, 1]$ to $T[1, 2]$, we use the *Gap* **in the column** to align with the letters **in the row**, so the *Gap* penalty keeps increasing **in vertical direction**.
3. $T[2, 2]$: This time we need to consider three conditions.
   (a) Diagonal: $T[1, 1] = 0$. Because at $[2, 2]$, A matches A, f(A,A) = 2, therefore:
   $$T[2, 2] = T[1, 1] + f(A, A) = 2 \tag{27}$$
   (b) Cell above: from $T[1, 2]$ to $T[2, 2]$, we use A in column to align with the letters in row. Now the alignment is between $-A$ and $--$, so:
   $$T[2, 2] = T[1, 2] - 1 = -2 \tag{28}$$
   Rule of thumb: when we move from the cell above, it means we add a gap in the row sequence, so the gap penalty will add 1.
   (c) Cell on the left: from $T[2, 1]$ to $T[2, 2]$, similarly it means we add a gap in the column sequence, so the gap penalty will add 1. So:
   $$T[2, 2] = T[2, 1] - 1 = -2 \tag{29}$$

Similarly, we can get all the assigned values in the graph. Notice that for internal nodes, negative values will become 0.

4.5 Parallel Prefix

Now we want to calculate i row from $i-1$ row using parallel prefix method.

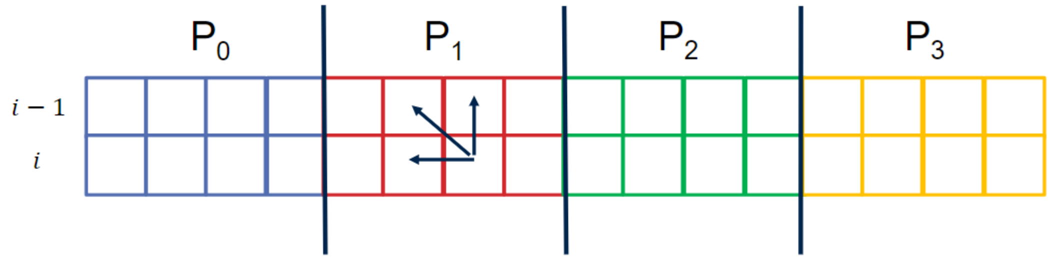

Figure 4: Parallel Prefix

We can start by split up the comparison process by two parts:

$$w[j] = \max \begin{cases} T[i-1, j-1] + f(a_i, b_j) \\ T[i-1, j] - g \end{cases} \tag{30}$$

$$T[i, j] = \max \begin{cases} w[j] \\ T[i, j-1] - g \end{cases} \tag{31}$$

However, both cases include $-g$, hard for calculation. We want to get rid of this process by building up the following expression:

$$x[j] = T[i, j] + jg \tag{32}$$

After rearrangement:

$$x[j] = \max \begin{cases} w[j] + jg \\ T[i, j-1] - g + jg \end{cases} \tag{33}$$

$$x[j] = \max \begin{cases} w[j] + jg \\ T[i, j-1] + (j-1)g \end{cases} \tag{34}$$

$$x[j] = \max \begin{cases} w[j] + jg \\ x[j-1] \end{cases} \tag{35}$$

If we already store $x[j-1]$ value, then we can assign $w[j]+jg$ to $x[j]$ and only need to do *max* operation once.

The complete algorithm is summarized below:

1. Use **right shift** to send last element of $i-1$ row on each processor (because from the previous convention, we don't need the $[i, j-1]$ point any more.)

2. Create vector:

$$A[j] = w[j] + jg = jg + \max \begin{cases} T[i-1, j-1] + f(a_i, b_j) \\ T[i-1, j] - g \end{cases} \tag{36}$$

3. Compute parallel prefix on A using *max* as operator and store results in x, including the comparison with $x[j-1]$

4. Now we have $x[j]$, then we compute:

$$T[i, j] = x[j] - jg \tag{37}$$

4.6 Runtime Analysis

Now first we consider the runtime for calculating only one row:

1. For step 2 in parallel algorithm, we need to traverse all the elements in each processor in $i-1$ row, so the runtime will be $O(\frac{n}{p})$

2. Then the communication among processors is expressed as $O((\tau + \mu) \log p)$

Therefore, the computation time for calculating all the rows will be $O(\frac{mn}{p} + m \log p)$ and the communication time will be $O((\tau + \mu) m \log p)$. Now if we want to balance these two runtime and become more efficient, we need:

$$p = O(\frac{n}{\log n}) \tag{38}$$