Accelerating sequence alignment using parallel prefix

DS 202

Ghanshyam Chandra, 06/04/2022

Agenda

- Preliminaries
- Prefix Sums
- Parallel Prefix Sums
- Sequence alignement with affine gap costs
- Parallel sequence alignment with prefix sums
- Implementation (OpenMP)
- Experiments

Preliminaries

Collective Communication Primitives

- **Broadcast:** In a Broadcast operation, one processor has a message of size l to be sent to all other processors. This operation takes $O((\tau + \mu)log(p))$ time.

- **Reduce:** Consider n data items $x_0, x_1, \ldots, x_{n-1}$ and a binary associative operator $\oplus$ that operates on these data items and produces a result of the same type. We want to compute $s = x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1}$. This operation takes $O(\frac{n}{p} + (\tau + \mu)log(p))$.

Reference: S. Aluru et al. / J. Parallel Distrib. Comput. 63 (2003) 264–272 265

Preliminaries

Speedup

- Let T_1 = running time using single processing elements
- Let T_p = running time using p identical processing elements
- **Speedup** $S_p = \frac{T_1}{T_p}$, Theoritically, $S_p \leq p$
- Here $S_p = p$ (Perfect or linear or ideal speedup)
- **Efficiency(η)** = $\frac{T_1}{pT_p}$

Reference: https://www3.cs.stonybrook.edu/~rezaul/Fall-2017/CSE548/CSE548-lecture-12.pdf

Preliminaries

Parallelism and span law

- We define T_p = runtime on p identical processing elements
- Then span, T_∞ = runtime on an infinite number of identical processing elements
- **Parallelism:** $P = \dfrac{T_1}{T_\infty}$
- Parallelism is an upper bound on speedup, i.e., $S_p \leq P$
  - **Span Law:** $T_p \geq T_\infty$

Reference: https://www3.cs.stonybrook.edu/~rezaul/Fall-2017/CSE548/CSE548-lecture-12.pdf

Preliminaries

Work Law & Work Optimality

- **Cost** of solving a problem:
  - T_1 (Sequentially)
  - pT_p (Parallelly)
- Then, **Work Law:** $T_p \geq \dfrac{T_1}{p}$
- Let T_s = runtime of the optimal or fastest known serial algorithm.
- A Parallel algorithm is cost optimal or work optimal provided
  - $pT_p = \Theta(T_s)$

Reference: https://www3.cs.stonybrook.edu/~rezaul/Fall-2017/CSE548/CSE548-lecture-12.pdf

Prefix Sums

- **Prefix Sums:** Prefix sum takes associated binary operator $\oplus$ and an ordered set $[a_1, \ldots, a_n]$ of n elements and returns ordered set.
  - $[a_1, (a_1 \oplus a_2), \ldots, (a_1 \oplus a_2 \oplus \ldots \oplus a_n)]$
- Computing the scan of an n-element requires $n - 1$ serial operations.
- T**ime Complexity:** $O(n)$, **Space Complexity:** $O(n)$

Reference: http://courses.csail.mit.edu/18.337/2004/book/Lecture_03-Parallel_Prefix.pdf

Parallel prefix sums (Overview)

Now lets assume we have n processors each having one element of the array.

To get a total sum of all the elements b_n can be computed efficiently in parallel.

- Recursively break the array in two halves, and add the sums of the two halves.
- Associated with computation is complete binary search tree with each internal node representing sum of the its descendent node.
- With p processors , this algorithm takes $O(log(p))$ steps. If we have have only $p < n$ processors then the total time will be $O(n/p + log(p))$ and communication will start from second step.
- With an architecture like **Hypercube** or **fat-tree** we can embed complete binary tree so that the communication is performed directly by communication links.

Reference: http://courses.csail.mit.edu/18.337/2004/book/Lecture_03-Parallel_Prefix.pdf

Parallel Prefix Algorithm (Inclusive)

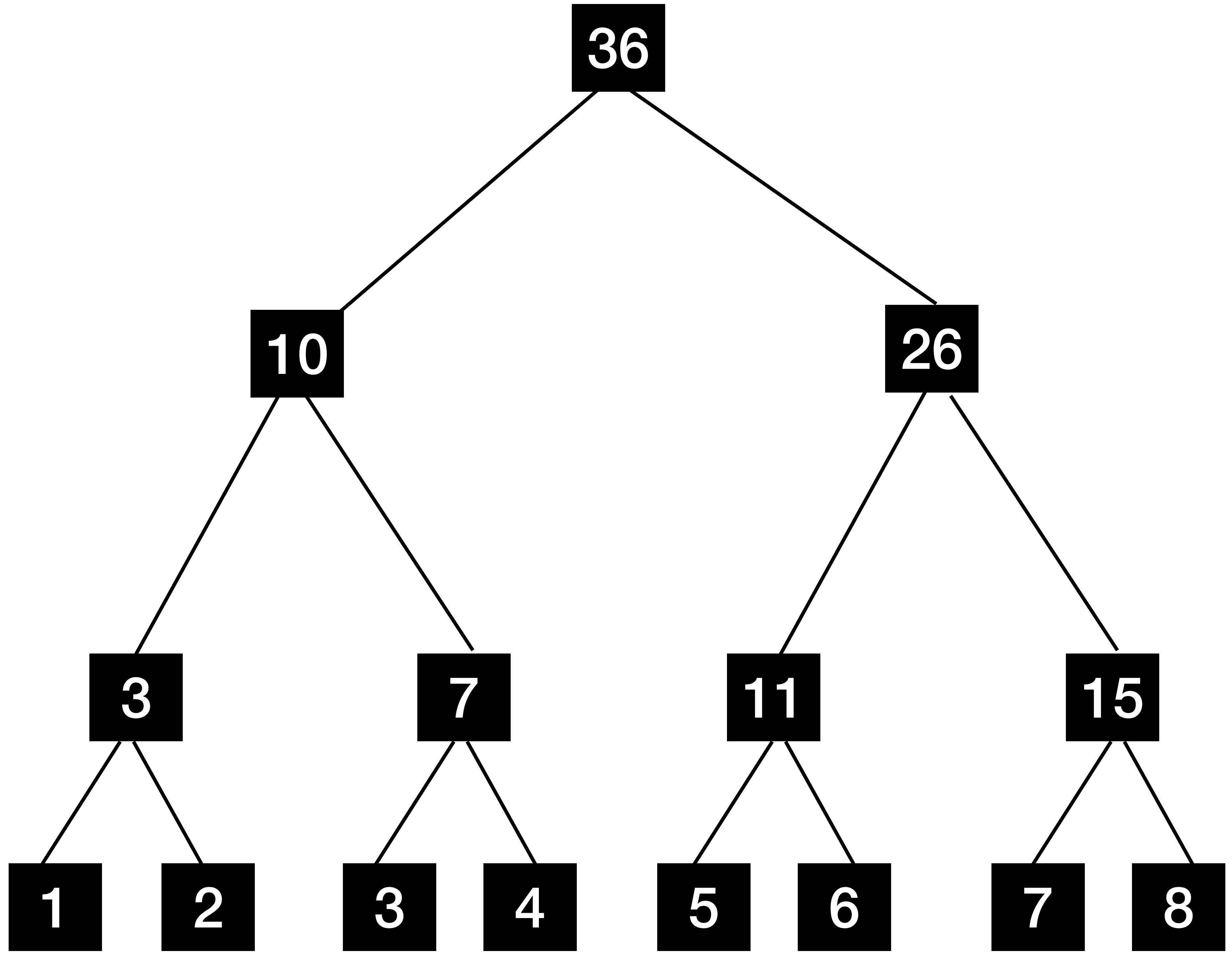
scan($[a_i]$):

- 1) Compute Pairwise sums, communicating with adjacent processor
    - $c_i := a_{i-1} \oplus a_i$ (if i is even)
- 2) Compute the even entries of the output by recursing on the size $\frac{n}{2}$ array of pairwise sums
    - $b_i := scan([c_i])$ (if i is even)
- 3) Fill in the odd entries of the output with a pairwise sum
    - $b_i := b_{i-1} \oplus a_i$ (if i is odd)
- return$[b_i]$

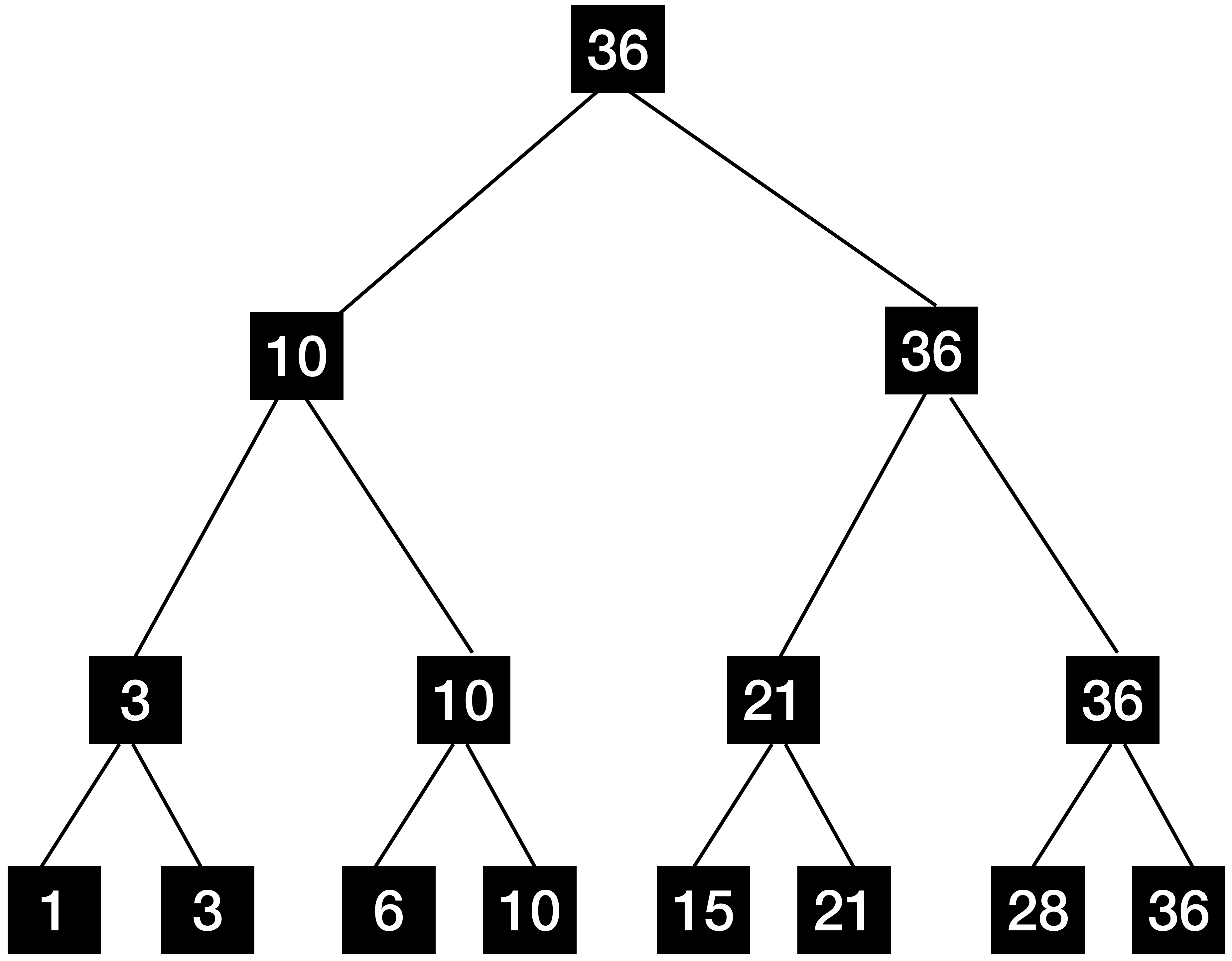
Reference: http://courses.csail.mit.edu/18.337/2004/book/Lecture_03-Parallel_Prefix.pdf

Action of the Parallel Prefix algorithm

Up the tree



Down the tree (Step (2) & (3))



Reference: http://courses.csail.mit.edu/18.337/2004/book/Lecture_03-Parallel_Prefix.pdf

Parallel Prefix Algorithm (Exclusive)

exclude_scan([a_i]):

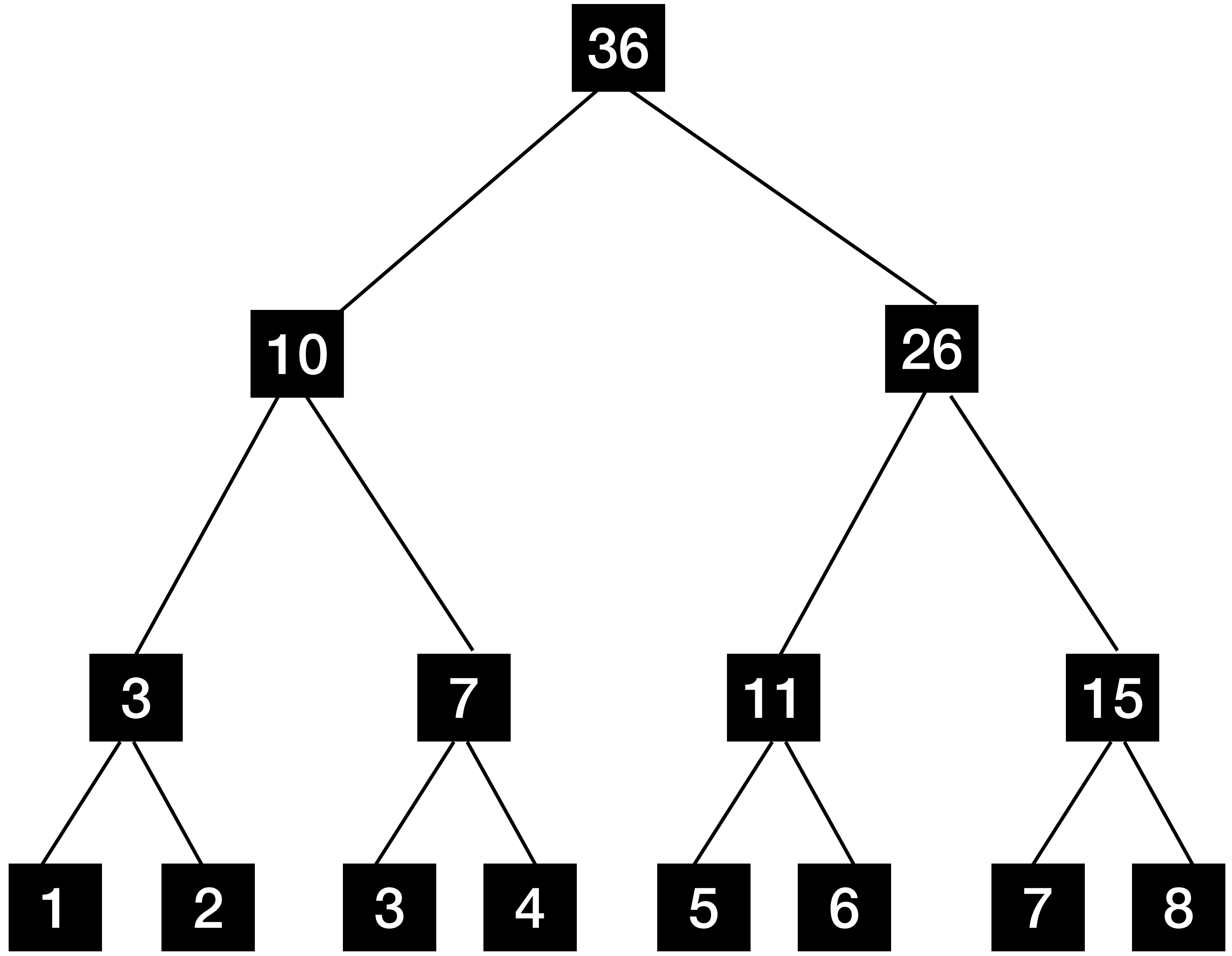
$[0, a_1, (a_1 \oplus a_2), \ldots, (a_1 \oplus a_2 \oplus \ldots \oplus a_{n-1})]$

- $b_i := excl\_scan([c_i])$ (if i is odd)
- $b_i := b_{i-1} \oplus a_{i-1}$ (if i is even)

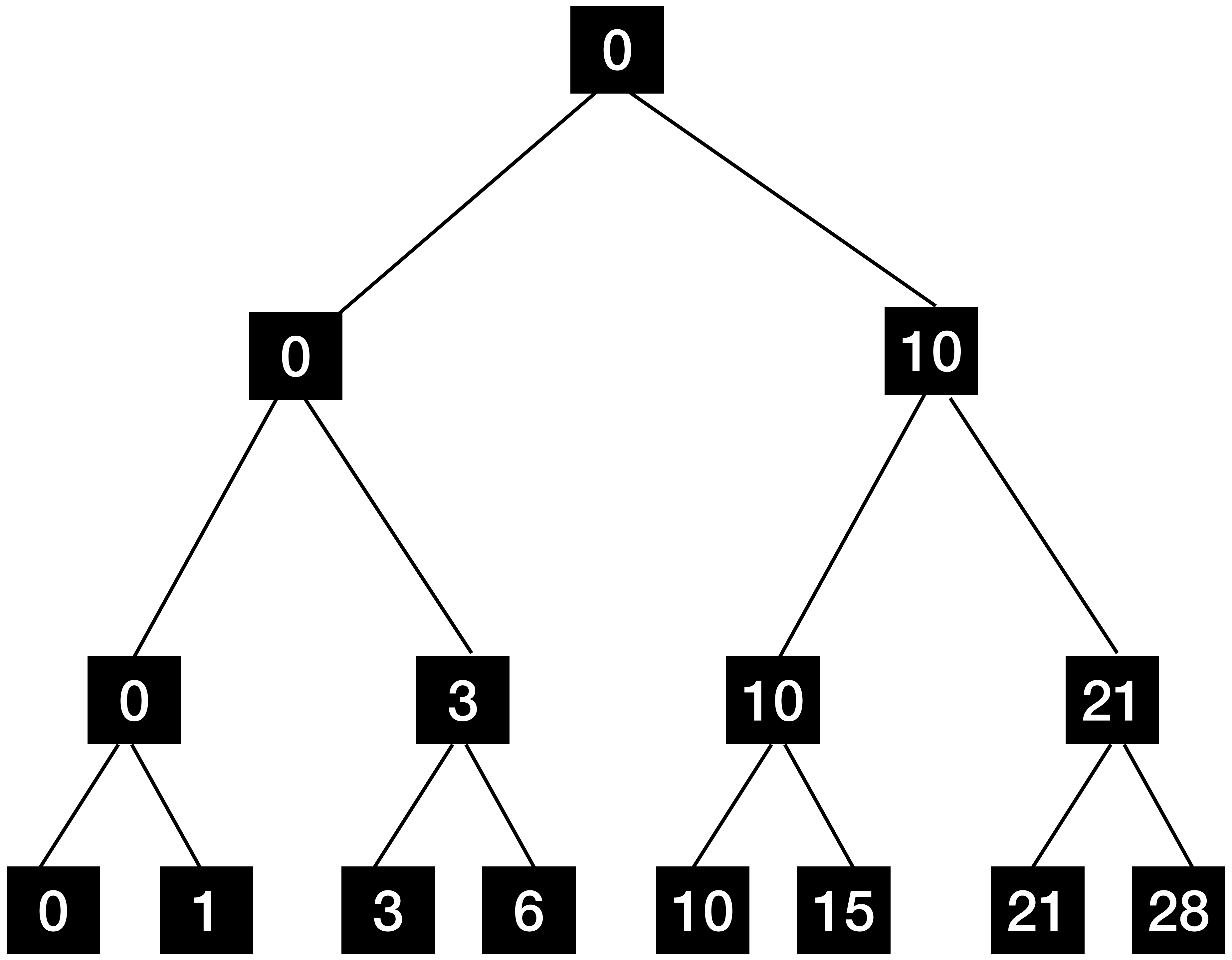
Reference: http://courses.csail.mit.edu/18.337/2004/book/Lecture_03-Parallel_Prefix.pdf

Action of the Parallel Prefix (Exclude) algorithm

Up the tree



Down the tree



Reference: http://courses.csail.mit.edu/18.337/2004/book/Lecture_03-Parallel_Prefix.pdf

Complexity Analysis:

Assumption: $n = 2^k$ **for some** $k \geq 0$

- **Work:** $T_1(n) = \begin{cases} \Theta(1), & \text{if } n = 1, \\ T_1(\frac{n}{2}) + \Theta(n), & \text{otherwise}. \end{cases}$ $= \Theta(n)$

- **Span :** $T_\infty(n) = \begin{cases} \Theta(1), & \textbf{if } n = 1, \\ T_\infty(\frac{n}{2}) + \Theta(1), & \textbf{otherwise}. \end{cases}$ **=** $\Theta(log(n))$

- **Parallelism (P) :** $\dfrac{T_1(n)}{T_\infty(n)} = \Theta(\dfrac{n}{log(n)})$

https://www3.cs.stonybrook.edu/~rezaul/Fall-2017/CSE548/CSE548-lecture-12.pdf

Complexity Analysis (realistic case):

Assuming each processors has $\frac{n}{p}$ elements (n is multiple of p & $n >> p$)

- 1) Each processor computes the prefix sums of the $\left(\frac{n}{p}\right)$ elements it has locally.
- 2) Using the last prefix sum on each processor, run a p-element parallel prefix algorithm.
- 3) On each processor, combine the result obtained by the parallel prefix algorithm with each local prefix sum computed previously.
- Steps (1) and (3) involve local computation only and each has $\left(\frac{n}{p}\right)$ run-time. Step (3) is the same as the parallel prefix algorithm where the number of elements equals the number of processors. Therefore, the run-time of the algorithm is:
    - **Computation Time:** $O(\frac{n}{p} + log(p))$
    - **Communication Time:** $O((\tau + \mu)log(p))$

Complexity Analysis (realistic case):

Assuming each processors has $\frac{n}{p}$ elements (n is multiple of p & $n >> p$)

- How to compute optimal number of processors?
- We say, parallel algorithm is optimal iff the cost of the algorithm is same as sequential runtime
  - $T_p = \Theta(\frac{n}{p} + log(p))$
- Cost of parallel algorithm is $pT_p = \Theta(n + plog(p))$. As long as $n = \Omega(plog(p))$, the cost is $\Theta(n)$, which is the same as sequential runtime.

Grama et al.

Complexity Analysis (realistic case):

Assuming each processors has $\frac{n}{p}$ elements (n is multiple of p & $n >> p$)

- How to compute optimal number of processors?
- We say **Efficiency(η) = 1**
  - $$\Theta(1) = \frac{\Theta(n)}{p\Theta(\frac{n}{p} + log(p))}$$
  - $\implies plog(p) = O(n)$
- Which gives optimal number of processor while still being efficient.

Complexity Analysis :

Computing upper bound on number of Processors.

- How to compute maximum number of processors, while still being efficient?
- **Work Law:** $T_p \leq \dfrac{T_1}{p}$, since $T_p = T_\infty$ for maximum Parallelism.
  - $p \leq O\left(\dfrac{n}{log(n)}\right)$
- $p = O\left(\dfrac{n}{log(n)}\right)$ is upper bound on number of processors that can be utilised efficiently.

Seqeunce alignment with affine gap costs $(g + hk)$

- We define simple scoring function: $f(c_1, c_2) = \begin{cases} 1, & c_1 = c_2, c_1, c_2 \in \Sigma \\ 0, & c_1 \neq c_2, c_1, c_2 \in \Sigma \end{cases}$
- Where Σ is the alphabet and $\Sigma = \{A, C, G, T\}$
- To find optimal alignment of sequences A and B using affine gap penalty functions, we will use Dynamic Programming and will maintain three tables T_1, T_2, T_3 each of size $(m + 1) \times (n + 1)$ (given |A| = n, |B| = m, where m < n). In T_1 we store score of match/mismatch of a_i with b_j, In T_2 we store score of , $'-'$ must be matched with b_j and in T_3 we store score of, a_i must be matched to $'-'$.

Seqeunce alignment with affine gap costs $(g + hk)$

- The tables can be filled with following equations.
- $$T_1[i, j] = f(a_i, b_j) + max \begin{cases} T_1[i-1, j-1], \\ T_2[i-1, j-1], \\ T_3[i-1, j-1], \end{cases}$$
- $$T_2[i, j] = max \begin{cases} T_1[i, j-1] - (g+h), \\ T_2[i, j-1] - g, \\ T_3[i, j-1] - (g+h), \end{cases}$$
- $$T_3[i, j] = max \begin{cases} T_1[i-1, j] - (g+h), \\ T_2[i-1, j] - (g+h), \\ T_3[i-1, j] - g, \end{cases}$$

Seqeunce alignment with affine gap costs $(g + hk)$

- **Initialisation:** The first row and column of each table are initialized to $-\infty$, except in the following cases ($1 \leq i \leq m$, $1 \leq j \leq n$) :
    - $T_1[0,0] = 0$
    - $T_2[0,j] = h + gj$
    - $T_3[i,0] = h + gi$

Parallel sequence alignment with prefix sums.

- Filling Three tables row by row parallely can be done with the help of prefix sums.
- Row i of T_1 and T_3 can be directly computed since it only depends upon the previous row information which has been precomputed.
- For T_2 we need Information from the same row hence we will use prefix sums to compute entries in table T_2.

Parallel sequence alignment with prefix sums.

- We define, $w[j] = max \begin{cases} T_1[i,j-1] - (g+h) \\ T_3[i,j-1] - (g+h) \end{cases}$
- Then, $T_2[i,j] = max \begin{cases} w[j] \\ T_2[i,j-1] - g \end{cases}$
- Let, $x[j] = T_2[i,j] + jg$
- We can rewrite, $x[j] = max \begin{cases} w[j] + jg \\ T_2[i,j-1] + (j-1)g \end{cases}$

Parallel sequence alignment with prefix sums.

- Now, $x[j] = max \begin{cases} w[j] + jg \\ x[j-1] \end{cases}$
- Since $w[j] + jg$ is known for all j, $x[j]$'s can be computed using parallel prefix with **max** as the binary associative operator.
- Then, $T_2[i, j], (1 \leq j \leq n)$ can be derived using
  - $T_2[i, j] = x[j] - jg$
- Thus each row can be computed using parallel prefix.

Parallel sequence alignment with prefix sums.

Distributed memory model

- For simplicity, assume m and n are multiples of p. Processor i is responsible for computing columns $i\left(\frac{n}{p}\right)+1$ to $(i+1)\left(\frac{n}{p}\right)$ of tables.
- Distribution of sequence B is trivial since b_j is needed only in computing column j. Therefore, processor i is given $b_{i(\frac{n}{p})+1}, \ldots, b_{(i+1)(\frac{n}{p})}$.
- Each a_i is needed by all the processors at the same time when row i is being computed. We distribute sequence A among all the processors to reduce storage. Processor i stores $a_{i(\frac{m}{p})+1}, \ldots, a_{(i+1)(\frac{m}{p})}$ and broadcasts it to all processors when row $i(\frac{m}{p})$ is about to be computed.
- If there is enough space, each processor can store a copy of A and broadcasting is eliminated.

Parallel sequence alignment with prefix sums.

Distributed memory model

- Computing $T_1[i,j]$ needs $T_1[i-1,j-1], T_2[i-1,j-1]$ and $T_3[i-1,j-1]$ also computing $w[j]$ requires $T_1[i,j-1], T_3[i,j-1]$, which may not be available locally (for extreme left columns on each processor). Each processor k can communicate and get these five entries from its preceding processor.

- The size of message is constant and independent of table sizes.

- Computing each row takes, $O(\frac{n}{p} + (\tau + \mu)log(p))$ time.

- Each of the p broadcasts for broadcasting portions of sequence A takes, $O(\frac{m}{p} + (\tau + \mu(\frac{m}{p})log(p)))$ time.

Parallel sequence alignment with prefix sums.

Distributed memory model

- **Computation time:** $O(\frac{mn}{p})$
- **Communication time:** $O((\tau + \mu)mlog(p))$

Experiments

Execution Time (s)

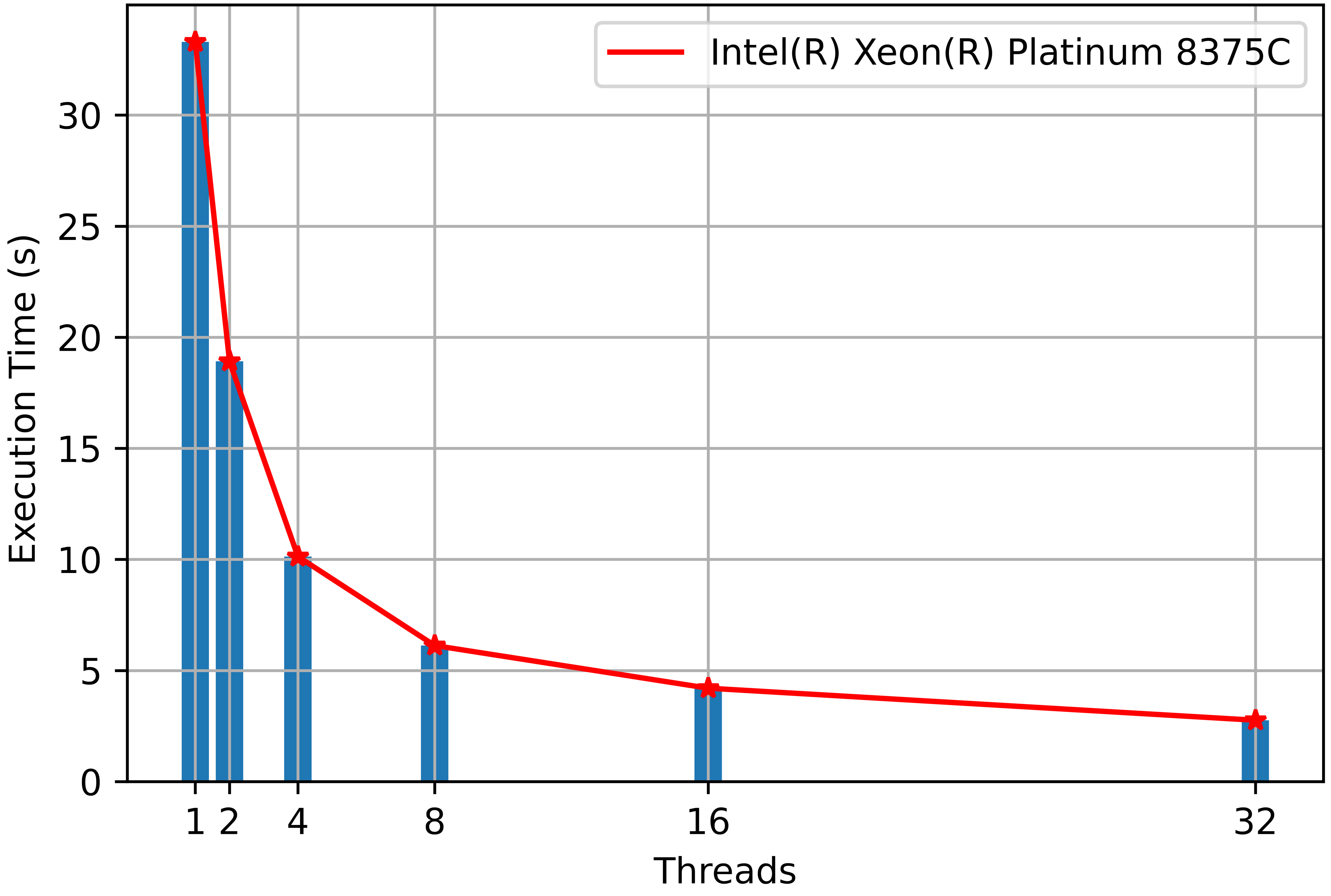


Figure (1): Execution Time (s) v/s Threads

Experiments

Speedup

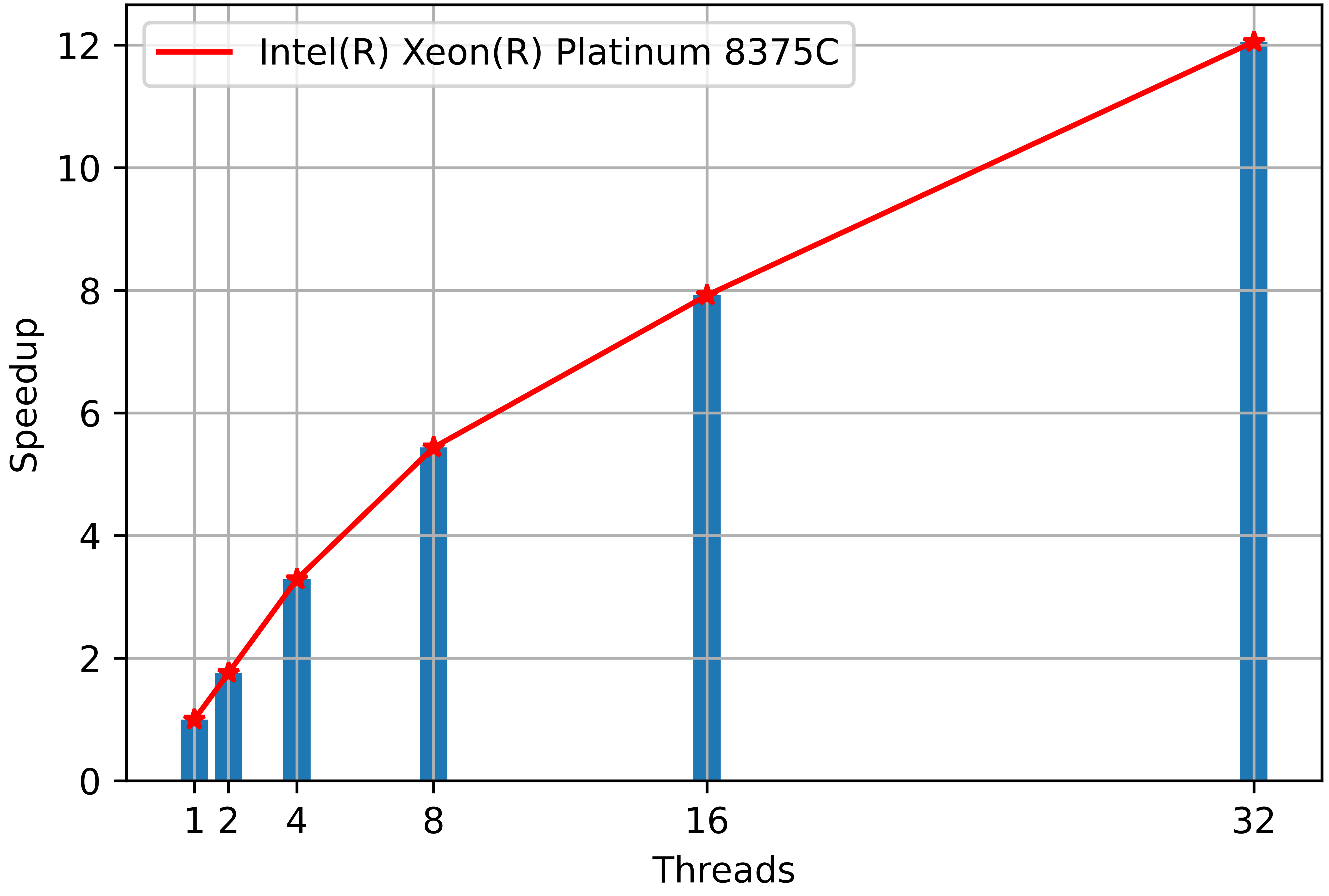


Figure (2): Speedup v/s Threads

Thanks!

ghanshyamc@iisc.ac.in

ParSeqAI: https://github.com/gsc74/ParSeqAI