Ghanshyam Chandra (18380)

Biological sequence comparison with parallel prefix sums.

DS 202

2022-04-06
ghanshyamc@iisc.ac.in

1 Introduction.

Preliminaries:
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 $s = x_0 \oplus x_1 \oplus \cdots \oplus x_{n-1}$. This operation takes $O(\frac{n}{p} + (\tau + \mu)log(p))$.

Speedup $\&$ **Efficiency :**
Let T_1 = running time using single processing elements .
Let T_p = running time using p identical processing elements .

$$\text{Speedup : } S_p = \frac{T_1}{T_p}, \text{ Theoretically , } S_P \leq p$$

Here $S_p = p$ for perfect or linear speedup.

$$\text{Efficiency : } E = \frac{T_1}{pT_p}, \text{ Theoretically , } E \leq 1$$

Span : T_∞ = runtime on an infinite number of identical processing elements.

$$\text{Parallelism : } P = \frac{T_1}{T_\infty}$$

Parallelism is an upper bound on speedup, i.e. $S_p \leq P$

$$\text{Span Law : } T_p \geq T_\infty$$

Let T_1 = cost of solving problem sequentially and pT_p is cost of solving problem parallely, then

$$\text{Work Law : } T_p \geq \frac{T_1}{p}$$

Let T_s = runtime of the optimal or fastest known serial algorithm, then a parallel algorithm is said to be cost optimal provided

$$pT_p = \Theta(T_s)$$

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

$$[a_1, (a_1 \oplus a_2), \ldots, (a_1 \oplus a_2 \oplus \cdots \oplus a_n)]$$

Computing the scan of an n-element requires $n - 1$ serial operations.

Parallel prefix sums: 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 recursively.

- Associated with computation is complete binary search tree with each internal node representing sum of the its descendent node.
- With n processors , this algorithm takes $\mathcal{O}(log(n))$ steps. If we have have only $p < n$ processors then the total time will be $\mathcal{O}(n/p + log(n))$ and communication will start from second step.

2 Algorithm

a) $scan([a_i])$(Inclusive Scan)

- 1) Compute Pairwise sums, communicating with adjacent processor

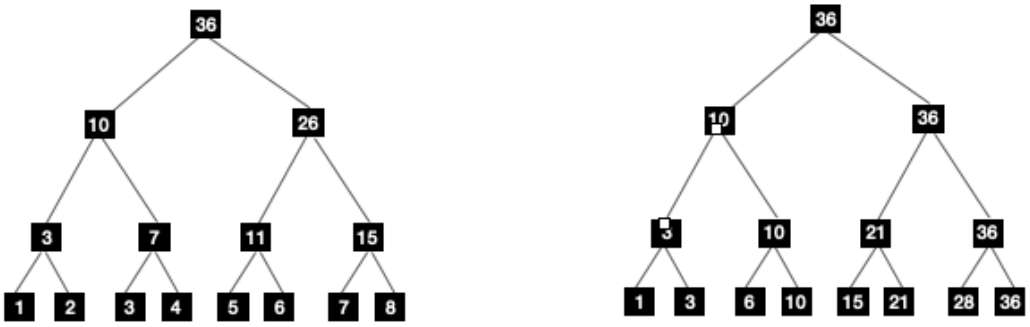
$$c_i := a_{i-1} \oplus a_i, \text{ if i is even}$$

- 2) Compute the even entries of the output by recursing on the size $\frac{n}{2}$ array of pairwise sums

$$b_i := scan([c_i]), \text{ if i is even}$$

- 3) Fill in the odd entries of the output with a pairwise sum

$$b_i := b_{i-1} \oplus a_i, \text{ if i is odd}$$



(a) Up the tree (step 1)

(b) Down the tree (step 2 & 3)

Figure 1: Action of Inclusive scan.

Explanation:

- We follow steps from Inclusive scan algorithm and we use step 1 to build the binary tree in Figure(1)(a) , we do pairwise sum and add the sum of leaf nodes to its parent node , any internal node of the tree will store inclusive scan upto its descendent nodes, also $scan([c_i])$ will return value stored in parent internal node.

- We follow step 2 and 3 to move down the tree in Figure(1)(b) where we take each level in Figure(1)(a) as input for the corresponding level in Figure(1)(b) , whenever we will compute odd leaf node we will use step 3 (sum the preceding leaf in the same tree and the current node in input tree(Up the tree)), whenever we will compute even leaf we will just use value from the parent internal node since each internal node holds inclusive scan upto its internal nodes.

b) $excl\_scan([a_i])$(Exclusive Scan)

- 1) Compute Pairwise sums, communicating with adjacent processor

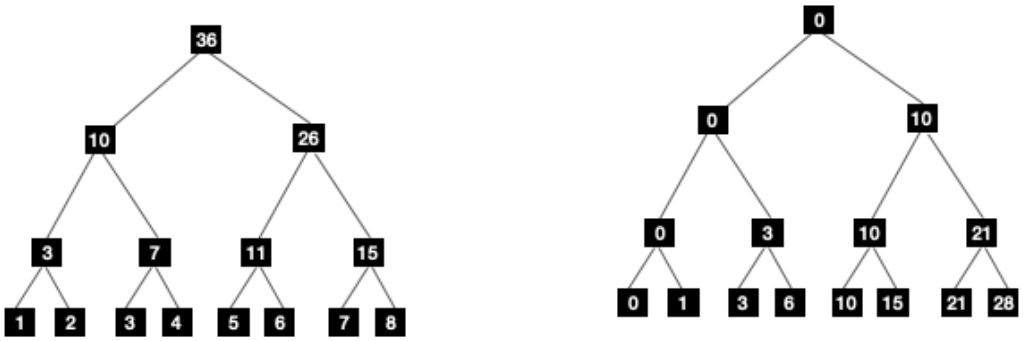
$$c_i := a_{i-1} \oplus a_i, \text{ if i is even}$$

- 2) Compute the even entries of the output by recursing on the size $\frac{n}{2}$ array of pairwise sums

$$b_i := excl\_scan([c_i]), \text{ if i is odd}$$

- 3) Fill in the odd entries of the output with a pairwise sum

$$b_i := b_{i-1} \oplus a_{i-1}, \text{ if i is even}$$



(a) Up the tree (step 1)

(b) Down the tree (step 2 & 3)

Figure 2: Action of Exclusive scan.

Explanation:

- We follow steps from Exclusive scan algorithm and we use step 1 to build the binary tree in Figure(2)(a) , we do pairwise and add the sum of leaf nodes to its parent node , any internal node of the tree will store exclusive scan upto its descendent nodes, also $excl\_scan([c_i])$ will return value stored in parent internal node.

- We follow step 2 and 3 to move down the tree in Figure(2)(b) where we take each level in Figure(2)(a) as input for the corresponding level in Figure(2)(b) , whenever we will compute even leaf node we will use step 3 (sum the preceding leaf in the same tree and the preceding node to the current in input tree(Up the tree)), whenever we will compute odd leaf we will just use value from the parent internal node since each internal node holds exclusive scan upto its internal nodes.

Complexity analysis:
Assumption: $n = 2^k$ for some $k \geq 0$.

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

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

$$\text{Parallelism (P): } \frac{T_1(n)}{T_\infty(n)} = \Theta(\frac{n}{log(n)})$$

Realistic case:

- 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:** $O((\tau + \mu)log(p))$.

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.
Alternate approach:
We say Efficiency(η) = 1

$$\Theta(1) = \frac{\Theta(n)}{p\Theta(\frac{n}{p} + log(p))}$$
$$\implies \quad plog(p) = O(n)$$

Which gives optimal number of processor while still being efficient.

How to compute maximum number of processors, while still being efficient?
Work Law: $T_p \leq \frac{T_1}{p}$, since $T_p = T_\infty$ for maximum Parallelism (span is upper bound on T_p).

$$p \leq O\Big(\frac{n}{log(n)}\Big)$$

$p = O\Big(\frac{n}{log(n)}\Big)$ is upper bound on number of processors that can be utilised efficiently.

c) Sequence alignment (with affine gap costs)

Sequence alignment with affine gap cost $(g + hk)$:
We define a simple scoring function

$$f(c_1, c_2) = \begin{cases} 1, & c_1 = c_2,\ c_1, c_2 \in \Sigma \\ 0, & c_1 \neq c_2,\ c_1, c_2 \in \Sigma \end{cases}$$

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 $(m+1) \times (n+1)$. In T_1 we store , a_i must be matched with b_j, In T_2, '-' must be matched with b_j and in T_3, a_i must be matched to '-'. The tables can be filled with following equations.

$$T_1[i,j] = f(a_i, b_j) + max \begin{cases} T_1[i-1, j-1], \\ T_2[i-1, j-1], \\ T_3[i-1, j-1], \end{cases}$$

$$T_2[i,j] = max \begin{cases} T_1[i, j-1] - (g+h), \\ T_2[i, j-1] - g, \\ T_3[i, j-1] - (g+h), \end{cases}$$

$$T_3[i,j] = max \begin{cases} T_1[i-1, j] - (g+h), \\ T_2[i-1, j] - (g+h), \\ T_3[i-1, j] - g, \end{cases}$$

Initialisation: first row and column of each tables are initialised to $-\infty$ except the cases ($1 \leq i \leq m, 1 \leq j \leq n$):
$T_1[0,0] = 0, \;\; T_2[0,j] = h + gj, \;\; T_3[i,0] = h + gi.$
Complexity: Space; $\mathcal{O}(mn)$, Time; $\mathcal{O}(mn)$.

d) 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 is precomputed, for T_2 we need Information from the same row hence we need to use prefix sums to compute entries in table T_2.

$$w[j] = max \begin{cases} T_1[i, j-1] - (g+h), \\ T_3[i, j-1] - (g+h), \end{cases}$$

Then,

$$T_2[i,j] = max \begin{cases} w[j], \\ T_2[i, j-1] - g, \end{cases}$$

Let, $x[j] = T_2[i,j] + jg$

$$x[j] = max \begin{cases} w[j] + jg, \\ T_2[i, j-1] + (j-1)g, \end{cases}$$

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

Then each row of T_2 can be computed using $T_2[i,j] = x[j] - jg$.

Explanation:

- 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.
- Computing $T_1[i,j]$ needs $T_1[i-1,j1], T_2[i-1,j-1]$ and $T_3[i-1,j-1]$ also computing $w[j]$ requires $T_1[i,j1], T_3[i,j-1]$, 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.

Complexity analysis:

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

3 Experiments

We implemented ParSeqAl for biological sequence alignment with affine gap cost using parallel prefix sums. Since in practice $\Theta(\frac{n}{p} + log(p))$ and $\Theta(\frac{n}{p} + p)$ are roughly same number since $\frac{n}{p} >> log(p) \approx p$ (for shared memory parallelism), hence we have implemented later due to ease of implementation with nearly same scalability $\Theta(\frac{n}{p} + log(p)) \approx \Theta(\frac{n}{p} + p) \approx \Theta(\frac{n}{p})$, for $n >> p$.

a) Getting ParSeqAl

Compilation:

```
git clone https://github.com/gsc74/ParSeqAl.git
cd ParSeqAl
make clean
make
```

Sample Run :

```
export OMP_NUM_THREADS=8
./bin/ParSeqAl data/first data/second > log/log.txt
```

Runnig with own examples :

```
export OMP_NUM_THREADS=8
./bin/ParSeqAl example_1 example_2
```

Format for examples :

```
>Reference
ATCGATCGGCTATCGGA
ATCGATCGGCTATCGGA
```

```
>Query
ATCGATCGGCTATCGGA
ATCGATCGGCTATCGGA
```

b) Experimental Setup :

AWS EC2 c6i.metal : Amazon EC2 C6i.metal instances are powered by 3rd generation Intel Xeon Scalable processors (Intel(R) Xeon(R) Platinum 8375C, IceLake), with 128 threads and 256 GB of RAM.
Since our test case is small hence we have only utilised 32 threads.

c) Results :

We extracted the reference and query sequence from random regions of E.coli reference genome, with Reference length = 26000bp and Query length = 16400bp .

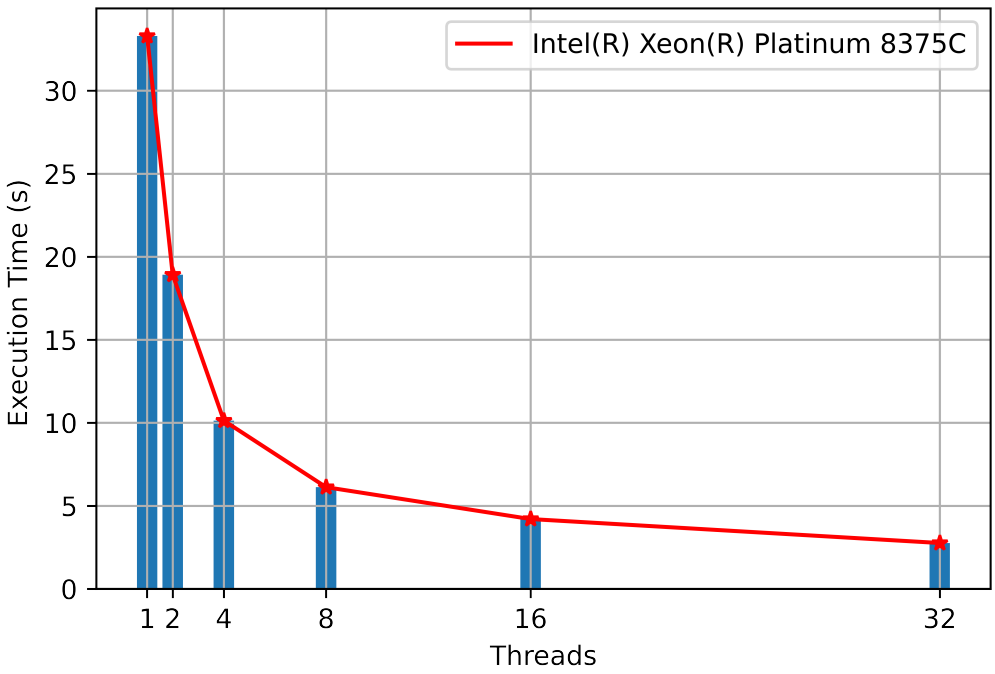


Figure 3: Execution Time(s) v/s Threads

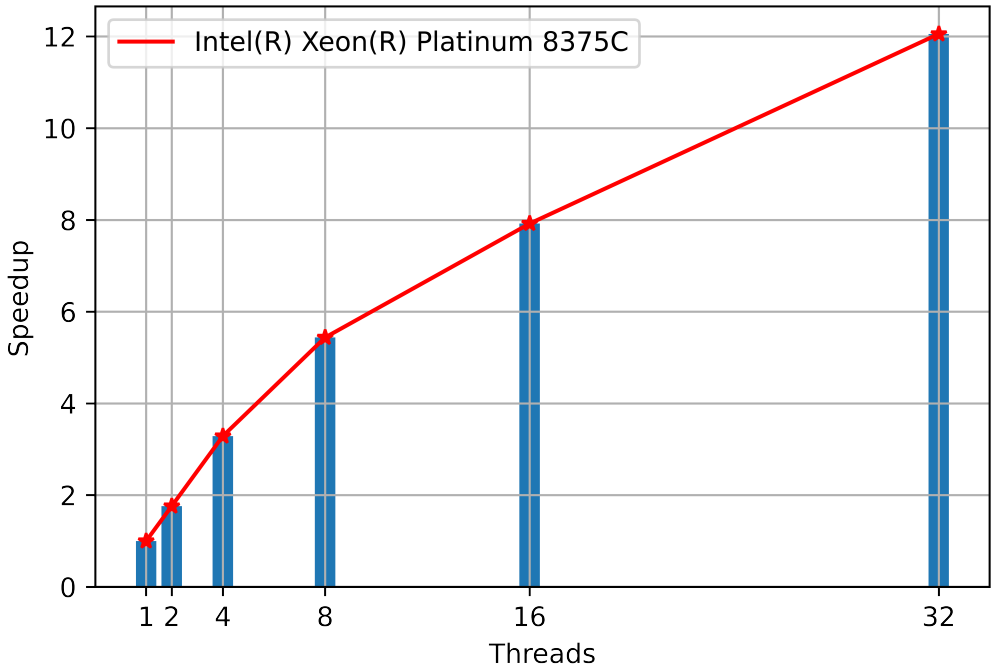


Figure 4: Speedup v/s Threads

Figure(3) clearly depicts with increase in number of threads execution time is decreasing, On the other hand Figure(4) depicts with fixed problem size, speedup with respect to sequential program increases with increase in number of threads but not linearly , hence we are observing sub-linear speedup.
Observations:

- Sub-linear speedup.
- Strong scalable upto 32 threads.

4 References :

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

2 CSE548-lecture-12

3 Lecture-3-Parallel-prefix-sums

4 Introduction to Parallel Computing, Second Edition. By Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar

5 Aluru 6220 class notes.