Prefix Sum Algorithms

CS F422: Parallel Computing
AY 2022-23, II Semester

Dhruv Rawat (2019B3A70537P)

April 30, 2023

1 Blelloch Algorithm

The Blelloch parallel prefix scan algorithm consists of two steps:

- `Reduction Phase/Up-sweep`: Up-sweep is the first phase where the objective is to build a partial prefix scan result for parts of the array in parallel. In this phase, we divide the array among all the participating threads, which then pair-wise computes the operation for the array elements belonging to their assigned portion.

Algorithm 1 Blelloch's Algorithm Upsweep Phase

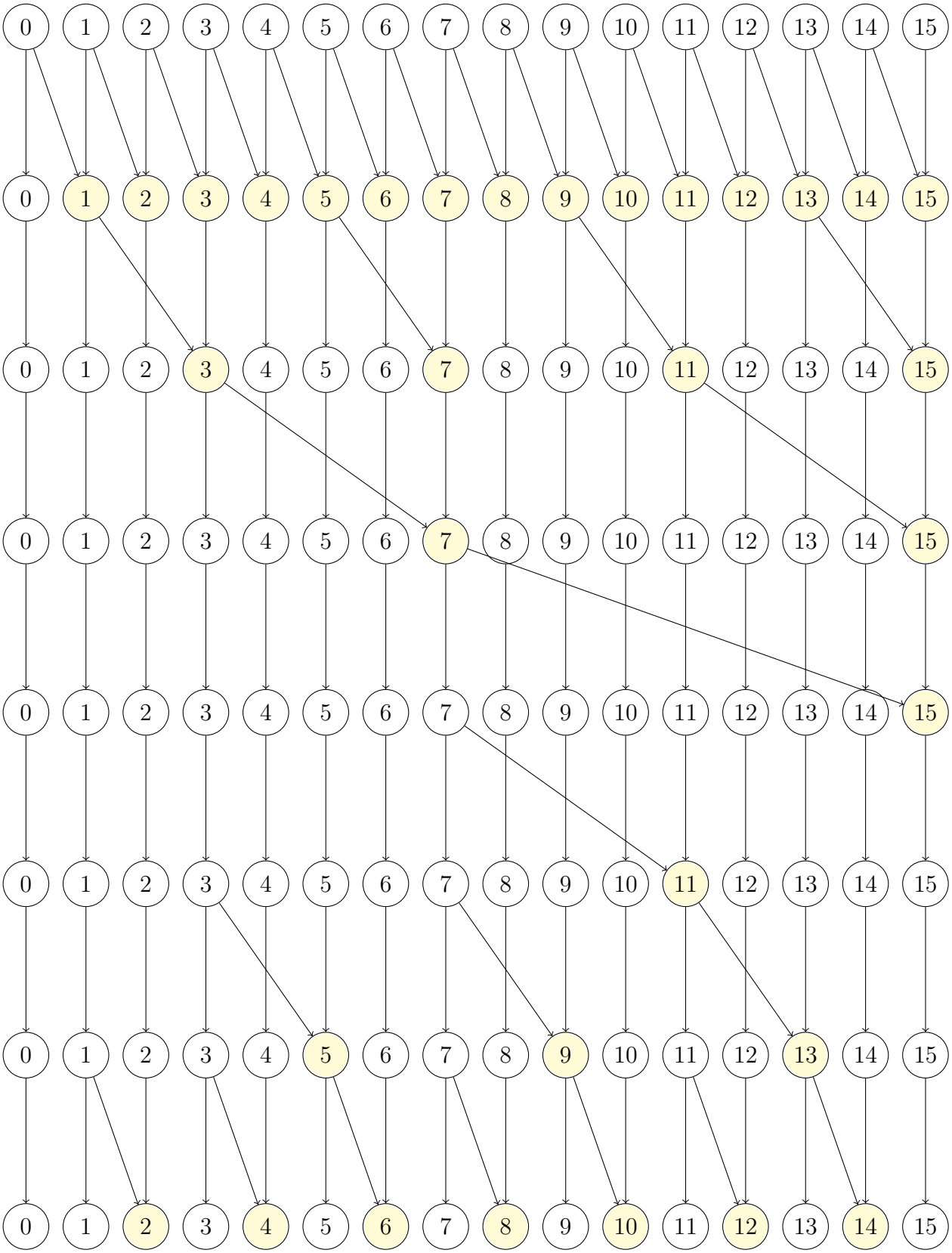
```
1: procedure UpSweep(A, n)
2:     for d = 0 to log_2 n − 1 do
3:         for i = 0 to n − 1 by 2^{d+1} in parallel do
4:             A[i + 2^{d+1} − 1] ← A[i + 2^{d+1} − 1] + A[i + 2^d − 1]
```

- `Down-sweep`: Down-sweep is the second phase of the algorithm, which executes after the up-sweep is completed. As a result of the up-sweep we get partial prefix scan output for subsets of the array, now to complete the scan, we first replace the root with an identity element (specific to the operator, 0 for addition) and then follow the below algorithm: On every iteration, each node passes its own value to its left child and passes to its right child the result of the operator applied to itself and its left child in the up-sweep tree.

Algorithm 2 Blelloch's Algorithm Downsweep Phase

```
1: procedure DownSweep(A, n)
2:     A[n − 1] ← 0
3:     for d = log_2 n − 1 down to 0 do
4:         for i = 0 to n − 1 by 2^{d+1} in parallel do
5:             t ← A[i + 2^d − 1]
6:             A[i + 2^d − 1] ← A[i + 2^{d+1} − 1]
7:             A[i + 2^{d+1} − 1] ← A[i + 2^{d+1} − 1] + t
```

1.1 Task Dependency Graph



1.2 Opportunities for Parallelism and Concurrency

1.2.1 Identifying Parallelism

- **Data parallelism**: A data dependency graph exhibits data parallelism when there are independent tasks applying the same operation to different elements of a dataset. The Blelloch algorithm is inherently parallelizable because it operates on an array of data. Data parallelism can be achieved by partitioning the input data into multiple chunks and processing each chunk concurrently on separate processors. Each processor can then perform the scan operation on its chunk independently.

- **Functional parallelism**: A data dependency graph exhibits functional parallelism when there are independent tasks applying different operations to different data elements. The Blelloch algorithm is composed of two main operations, the Up-Sweep and the Down-Sweep. These two operations can be executed in parallel on separate processors, enabling functional parallelism.

- **Pipelining**: The algorithm can be pipelined by overlapping the computation of successive stages. For example, while the Up-Sweep is being executed on one processor, the Down-Sweep can be executed on another processor. This can help to increase the throughput of the algorithm.

1.2.2 Degree of Concurrency

Mamximum degree of concurrency is the largest number of concurrent tasks at any point of execution. Assuming all nodes are equal-weighted, i.e. with weight 1.

As we can see in the task dependency graph above, for 16 elements, maximum degree of concurrency = 16. Hence we can say:

$$\textbf{Maximum Degree of Concurrency} = n = O(n)$$

To find critical path length, we see upsweep and downsweep phases independently. As we can see, the height of the tree for a 16 element dependency graph is 4 in each phase. Thus:

$$\textbf{Critical Path Length} = \log_2(n) + \log_2(n) = O(\log_2(n))$$

Thus, we can find total work for 16 elements by summing up no. of processes in each phase:

$$\textbf{Total Work} = (16 + 15 + 4 + 2 + 1) + (1 + 3 + 7) = 49$$

$$\textbf{Average Degree of Concurrency} = \frac{\text{Total Work}}{\text{Critical Path Length}} = \frac{49}{4+4} = 6.125$$

1.3 Design in MPI

Algorithm 3 Blelloch Algorithm Design in MPI

```
1: Initialize MPI
2: Get MPI_COMM_RANK and MPI_COMM_SIZE
3: MPI_Barrier()
4: Set numPerProc = n/commSize
5: Call start_find_sum()
6: Call start_find_psum()
7: MPI_Barrier()
8: Ends MPI
```

Algorithm 4 start_find_sum

```
1: procedure START_FIND_SUM(rank, size, *input, numPerProc, *overallSum)
2:     Get MPI_Status
3:     Find sum of array elements from 0 to numPerProc
4:     for level = 0 to log2(MPI_COMM_SIZE) do
5:         position ← rank/level^2
6:         if position is even then
7:             receiveFrom ← rank + level^2
8:             Receive receiveSum from rank receiveFrom process
9:             TotalSum += receiveSum
10:        else
11:            sendTo ← rank − level^2
12:            Send TotalSum to rank sendTo process
13:            Kill current process
14:        MPI_Barrier()
```

Algorithm 5 start_find_psum

```
1: procedure START_FIND_PSUM(rank, size, *input, numPerProc, *overallSum)
2:     Get MPI_Status
3:     if rank == 0 then psum ← TotalSum
4:     for level = log2(MPI_COMM_SIZE) − 1 down to 0 do
5:         if process is on current level then
6:             position ← rank/level^2
7:             if position is even then
8:                 receiveFrom ← rank + level^2
9:                 Send psum, receive receiveSum from rank receiveFrom process
10:                psum -= receiveSum
11:            else
12:                sendTo ← rank − level^2
13:                Receive psum, Send TotalSum to rank sendTo process
14:            MPI_Barrier()
15:    Put prefixSums in input
```

1.4 Performance Metrics

1.4.1 Speedup

We know that $T_{serial}(n,1) = O(n)$ and $T_{parallel}(n,p) = 2(\lceil n/p \rceil + \lceil \lg n \rceil) = O(n/p)$ [Since $n/p \geq \lg n$]. Therefore

$$\boxed{\textbf{Speedup} = \frac{T_{serial}(n,1)}{T_{parallel}(n,p)} = \frac{n}{2(\lceil n/p \rceil + \lceil \lg n \rceil)}} \tag{1}$$

1.4.2 Efficiency

$$\boxed{\textbf{Efficiency} = \frac{Speedup}{p} = \frac{n}{p * 2(\lceil n/p \rceil + \lceil \lg n \rceil)}} \tag{2}$$

1.4.3 Cost

We know, cost of a parallel algorithm is the product of its time complexity and the no. of processors, p.

$$\boxed{\textbf{Cost} = p * O(n/p) \simeq O(n)} \tag{3}$$

1.4.4 Isoefficiency Metrics

T_1 = Time taken by algorithm to execute on single processor = $n = W$
T_p = Time taken by algorithm to execute on p processor = $2(\lceil n/p \rceil + \lceil \lg n \rceil)$
T_0 = Total time spent by all processors doing work that is not done by sequential execution
We know

$$p * T_p = T_1 + T_0$$

i.e.

$$\begin{aligned} T_0 &= p * T_p - T_1 \\ &= 2p(\lceil n/p \rceil + \lceil \lg n \rceil) - n \\ &= 2n + 2p\lceil \lg n \rceil - n \\ &= n + 2p\lceil \lg n \rceil \\ &= W + 2p\lceil \lg W \rceil \end{aligned}$$

Hence, we have $T_0(W,p) = W + 2p\lceil \lg W \rceil$

$$\boxed{\textbf{Isoefficiency Metric} = \frac{1}{1 + \frac{T_0(W,p)}{W}} = \frac{1}{1 + \frac{W + 2p\lceil \lg W \rceil}{W}} = \frac{1}{2 + \frac{2p\lceil \lg W \rceil}{W}}} \tag{4}$$

1.5 Cost Optimality

A **cost optimal** parallel algorithm is an algorithm for which the cost is in the same complexity class as an optimal sequential algorithm.

Cost of a sequential Prefix-Sum algorithm = $O(n)$

Cost of Blelloch's Prefix-Sum algorithm = $O(n)$

Hence, **Blelloch's prefix-sum algorithm is cost-optimal**

1.6 Experimental Results

All experiments were performed on an ®Intel ™Core i7-8550U CPU with 4 cores.

Table 1: Execution Time (in microseconds) for different values of p

p	n=1000	n=10000	n=100000	n=500000	n=1000000	n=2000000
1	0.023821	0.252587	2.552497	9.93422	18.725536	36.792498
2	0.511832	0.725823	1.725969	5.506916	10.680913	16.296096
4	1.00193	1.074103	1.248382	4.287543	6.087878	12.591893
8	206.935595	21.785144	11.580885	3.28435	4.789575	10.081781
16	434.941109	479.227497	413.942548	347.966557	380.88346	347.229041
32	959.201463	1054.780617	936.807629	880.874029	913.219297	900.770569

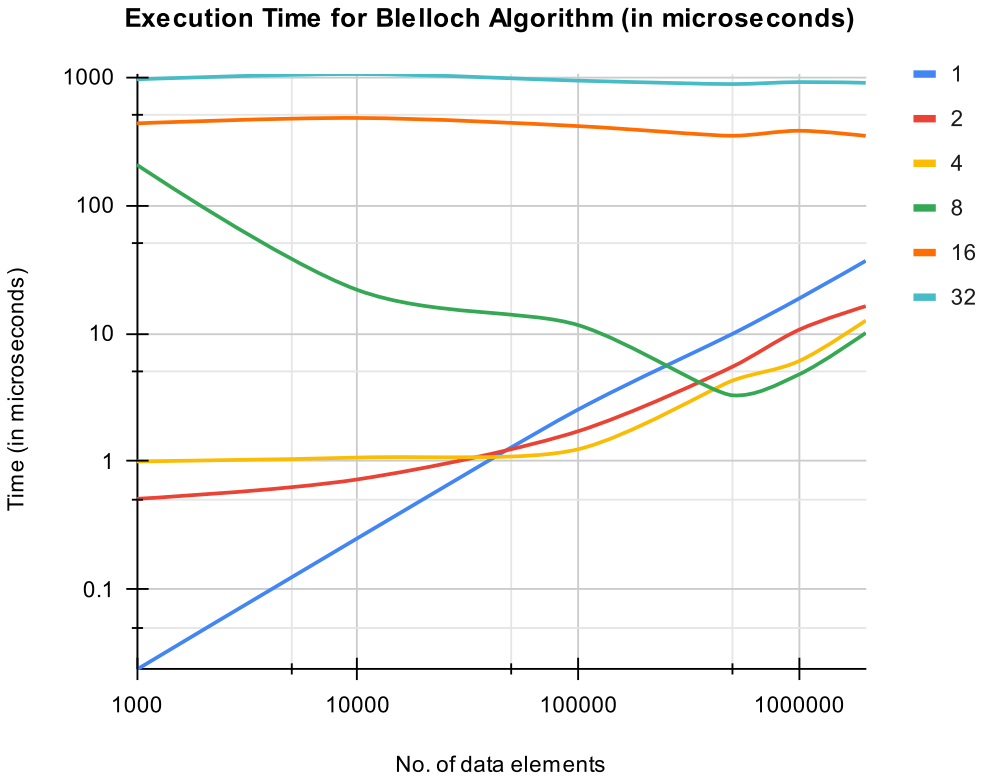


Table 2: Speedup for different values of p

p	n=1000	n=10000	n=100000	n=500000	n=1000000	n=2000000
1	1.0000000	1.0000000	1.0000000	1.0000000	1.0000000	1.0000000
2	0.0465407	0.3480008	1.4788777	1.8039534	1.7531775	2.2577492
4	0.0237751	0.2351609	2.0446442	2.3169960	3.0758724	2.9219195
8	0.0001151	0.0115945	0.2204060	3.0247142	3.9096446	3.6494046
16	0.0000548	0.0005271	0.0061663	0.0285494	0.0491634	0.1059603
32	0.0000248	0.0002395	0.0027247	0.0112777	0.0205050	0.0408456

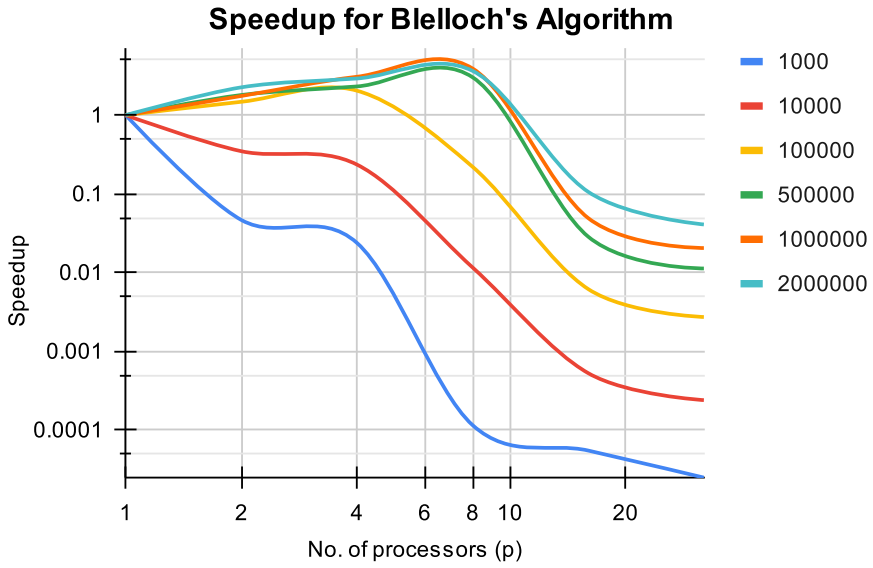
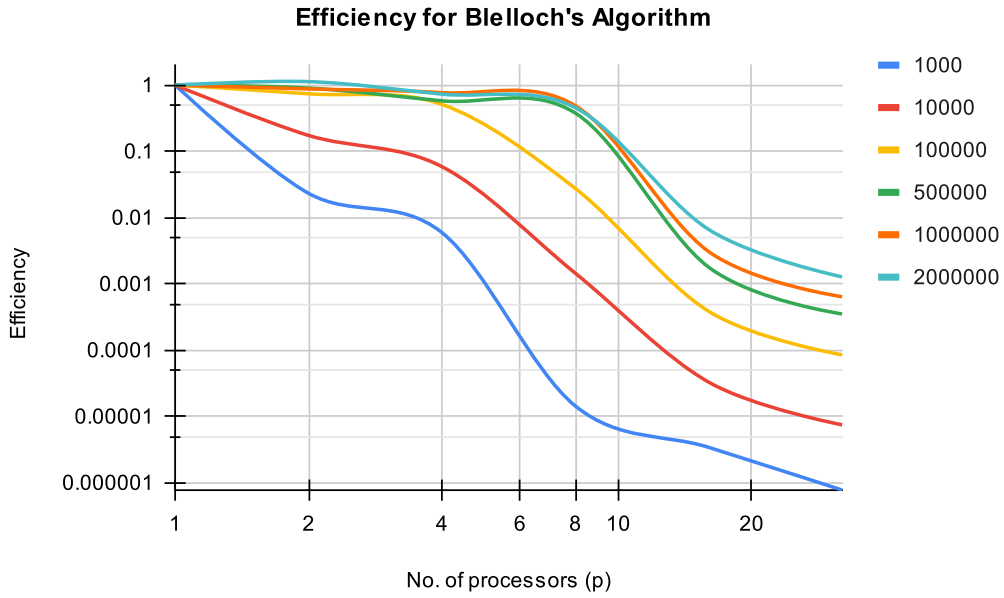


Table 3: Efficiency for different values of p

p	n=1000	n=10000	n=100000	n=500000	n=1000000	n=2000000
1	1.0000000	1.0000000	1.0000000	1.0000000	1.0000000	1.0000000
2	0.0232703	0.1740004	0.7394388	0.9019767	0.8765887	1.1288746
4	0.0059438	0.0587902	0.5111610	0.5792490	0.7689681	0.7304799
8	0.0000144	0.0014493	0.0275508	0.3780893	0.4887056	0.4561756
16	0.0000034	0.0000329	0.0003854	0.0017843	0.0030727	0.0066225
32	0.0000008	0.0000075	0.0000851	0.0003524	0.0006408	0.0012764



2 Hillis-Steele Algorithm

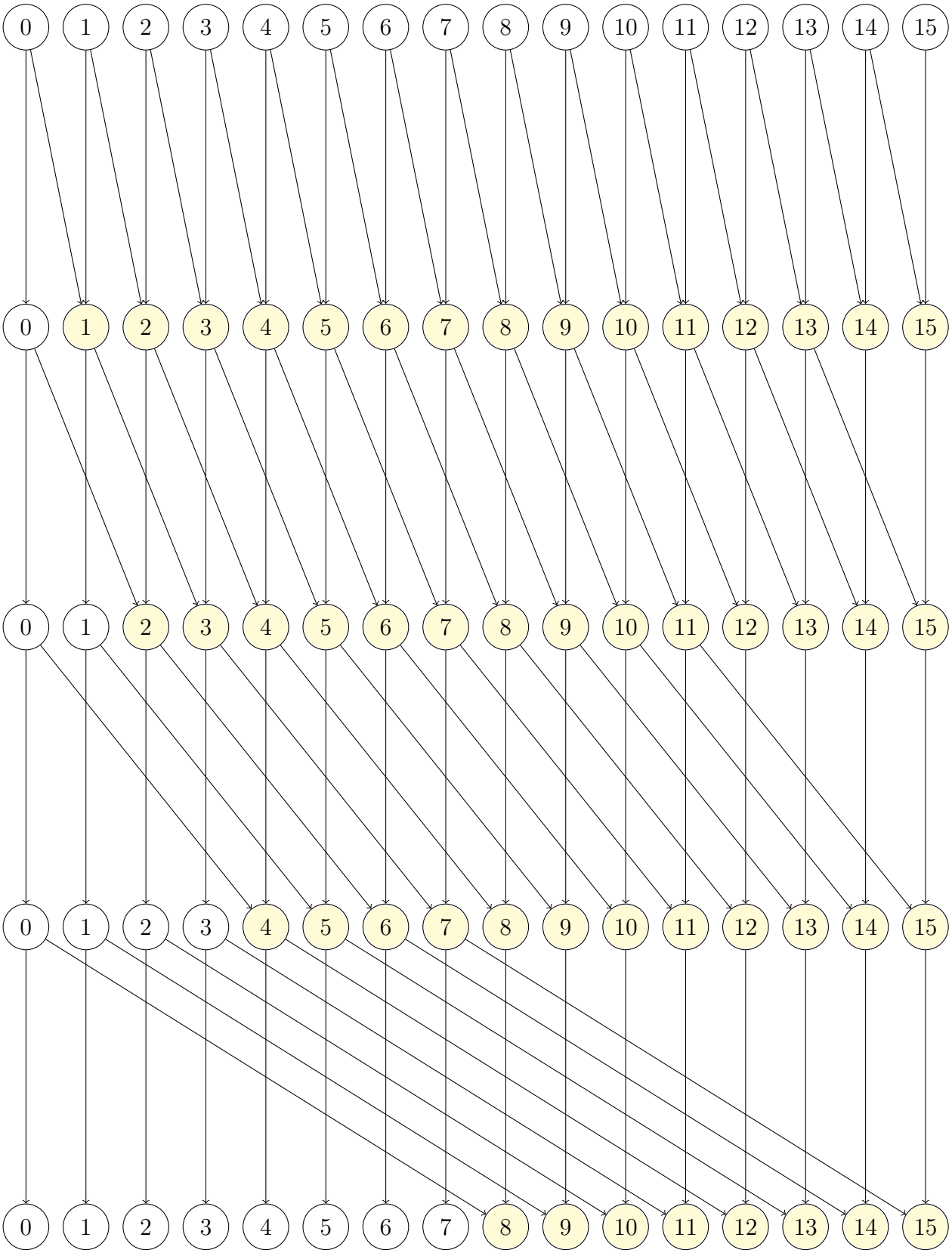
The Hillis-Steele prefix sum algorithm is a parallel algorithm for computing the prefix sum of an input array. The algorithm works by repeatedly combining pairs of elements in the input array in a binary tree fashion, using a series of parallel scans.

In the first iteration of the algorithm, each element of the input array is paired with the element that is 2^0 indices to its left, and the sum of each pair is stored in the right-hand element of the pair. In the second iteration, each element of the array that is 2^1 indices to the right of a power of 2 is paired with the element that is 2^1 indices to its left, and the sum of each pair is stored in the right-hand element of the pair. This process is repeated for $\log_2(n)$ iterations, where n is the length of the input array.

Algorithm 6 Hillis-Steele Algorithm

```
1: procedure HillisSteeleScan(x, n)
2:     for j = 1 to log_2 n do
3:         for all k in parallel do
4:             if k ≥ 2^j then
5:                 x[k] ← x[k − 2^(j−1)] + x[k]
```

2.1 Task Dependency Graph



2.2 Opportunities for Parallelism and Concurrency

2.2.1 Identifying Parallelism

- **Data parallelism**: The Hillis and Steele algorithm can be easily parallelized using data parallelism. The prefix sum calculation for each element in the input array is independent of the other elements. Therefore, each element can be processed in parallel. This approach can be implemented using SIMD (Single Instruction Multiple Data) instructions on modern CPUs and GPUs.
- **Functional parallelism**: The Hillis and Steele algorithm can also be parallelized using task parallelism. In this approach, multiple threads can be used to process different sections of the input array in parallel. This approach can be useful when the input array is very large, and the computation time for each element is relatively long.
- **Pipelining**: The Hillis and Steele algorithm can be pipelined to improve performance. In this approach, the input array is divided into smaller sub-arrays, and each sub-array is processed in a pipeline of stages. Each stage computes the prefix sum of the input sub-array and passes the result to the next stage. This approach can reduce the overall computation time by overlapping the computation of different sub-arrays.

2.2.2 Degree of Concurrency

Mamximum degree of concurrency is the largest number of concurrent tasks at any point of execution. Assuming all nodes are equal-weighted, i.e. with weight 1.

As we can see in the task dependency graph above, for 16 elements, maximum degree of concurrency = 16. Hence we can say:

$$\textbf{Maximum Degree of Concurrency} = n = O(n)$$

To find critical path length, we see upsweep and downsweep phases independently. As we can see, the height of the tree for a 16 element dependency graph is 4 in each phase. Thus:

$$\textbf{Critical Path Length} = \log_2(n) = O(\log_2(n))$$

Thus, we can find total work for 16 elements by summing up no. of processes in each phase:

$$\textbf{Total Work} = (16 + 15 + 14 + 12 + 8) = 65$$

$$\textbf{Average Degree of Concurrency} = \frac{\text{Total Work}}{\text{Critical Path Length}} = \frac{65}{4} = 16.25$$

2.3 Design in MPI

Algorithm 7 Hillis-Steele Algorithm Design in MPI

```
1: Initialize MPI
2: Get MPI_COMM_RANK and MPI_COMM_SIZE
3: MPI_Barrier()
4: Set numPerProc = n/commSize
5: Send numPerProc sized chunk of elements to each process using MPI_Scatter
6: for step = 0 to log2(element_count) do
7:     power ← 2^step
8:     for k = 0 to numPerProc do
9:         if k ≥ power then
10:            localSum[k] ← localData[k-power] + localData[k]
11:        else
12:            localSum[k] ← localData[k]
13:    Swap localSum and localData
14: if rank = 0 then
15:    Send last value in localData to process with rank = 1
16: else
17:    if rank > 0 & rank ≠ size-1 then
18:        Receive value from rank-1 process and add this to each element in localData
19:        Send last value in localData to process rank + 1
20:    else
21:        Receive value from rank-1 process and add this to each element in localData
22: Collect all prefix sums from each process using MPI_Gather
23: MPI_Barrier()
24: Ends MPI
```

2.4 Performance Metrics

2.4.1 Speedup

We know that $T_{serial}(n,1) = O(n \lg n)$ and $T_{parallel}(n,p) = O(\lceil n/p \rceil \lg n + \lceil n/p \rceil)$. Therefore

$$\boxed{\textbf{Speedup} = \frac{T_{serial}(n,1)}{T_{parallel}(n,p)} = \frac{n \lg n}{\lceil n/p \rceil \lg n + \lceil n/p \rceil}} \tag{5}$$

2.4.2 Efficiency

$$\boxed{\textbf{Efficiency} = \frac{Speedup}{p} = \frac{n \lg n}{p * (\lceil n/p \rceil \lg n + \lceil n/p \rceil)}} \tag{6}$$

2.4.3 Cost

We know, cost of a parallel algorithm is the product of its time complexity and the no. of processors, p.

$$\boxed{\textbf{Cost} = p * O(\lceil n/p \rceil \lg n) \simeq O(n \lg n)} \tag{7}$$

2.4.4 Isoefficiency Metrics

T_1 = Time taken by algorithm to execute on single processor = $n = W$
T_p = Time taken by algorithm to execute on p processor = $\lceil n/p \rceil \lg n + \lceil n/p \rceil$
T_0 = Total time spent by all processors doing work that is not done by sequential execution
We know

$$p * T_p = T_1 + T_0$$

i.e.

$$\begin{aligned} T_0 &= p * T_p - T_1 \\ &= p(\lceil n/p \rceil \lg n + \lceil n/p \rceil) - n \\ &= n \lg n + n - n \\ &= n \lg n \\ &= W \lg W \end{aligned}$$

Hence, we have $T_0(W, p) = W + 2p\lceil \lg W \rceil$

$$\boxed{\textbf{Isoefficiency Metric} = \frac{1}{1 + \frac{T_0(W,p)}{W}} = \frac{1}{1 + \frac{W + 2p\lceil \lg W \rceil}{W}} = \frac{1}{2 + \frac{2p\lceil \lg W \rceil}{W}}} \tag{8}$$

2.5 Cost Optimality

A **cost optimal** parallel algorithm is an algorithm for which the cost is in the same complexity class as an optimal sequential algorithm.

Cost of a sequential Prefix-Sum algorithm = $O(n)$

Cost of Hillis-Steele's Prefix-Sum algorithm = $O(n \lg n)$

Hence, **Hillis-Steele's prefix-sum algorithm is <u>not cost-optimal</u>**

2.6 Experimental Results

Table 4: Execution Time (in sec) for different values of p

p	n=1000	n=10000	n=100000	n=500000	n=1000000	n=2000000
1	0.207342	2.135189	12.994063	32.473464	68.847152	130.603583
2	0.048763	0.255135	2.70596	13.250024	26.938487	59.795909
4	0.034336	0.200666	1.167292	6.306477	13.792232	32.47846
8	0.028474	0.106216	1.170489	7.091356	17.838404	38.434396
16	0.020581	0.079604	0.787075	4.418177	9.889613	20.315884
32	0.021974	0.053179	0.397156	2.235072	5.033473	29.133808

3 Conclusions

Here you briefly summarize your findings.

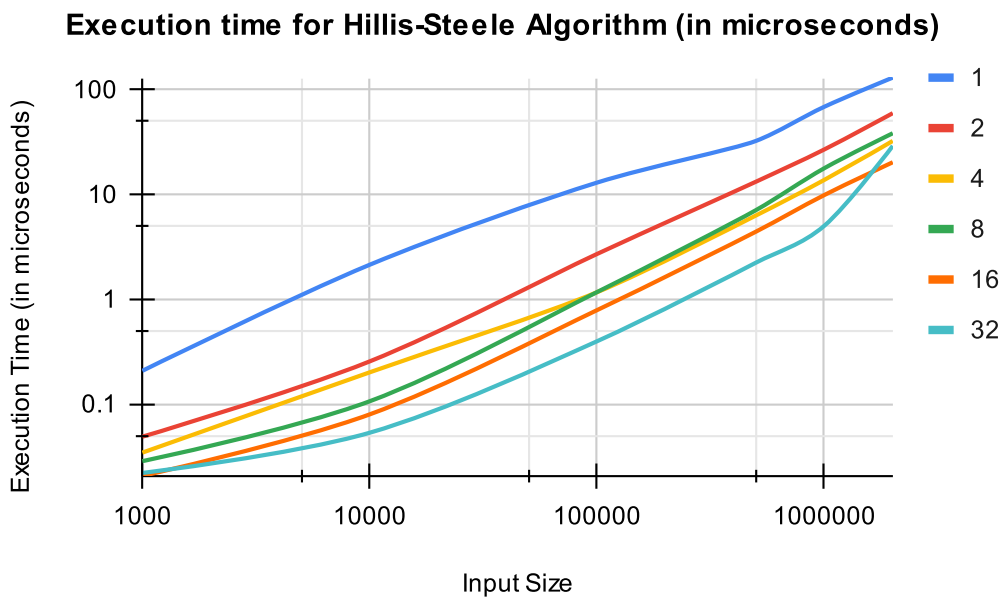


Table 5: Speedup for different values of p

p	n=1000	n=10000	n=100000	n=500000	n=1000000	n=2000000
1	1.000000000	1.000000000	1.000000000	1.000000000	1.000000000	1.000000000
2	4.252035355	8.368859623	4.80201592	2.450823032	2.555717105	2.184155826
4	6.03861836	10.64051209	11.13180164	5.149224202	4.991733898	4.021236937
8	7.281800941	20.10232922	11.10139694	4.579302463	3.859490569	3.398091205
16	10.07443759	26.82263454	16.50930725	7.349969003	6.961561792	6.428643863
32	9.435787749	40.15098065	32.71778092	14.52904604	13.67786258	4.482887476

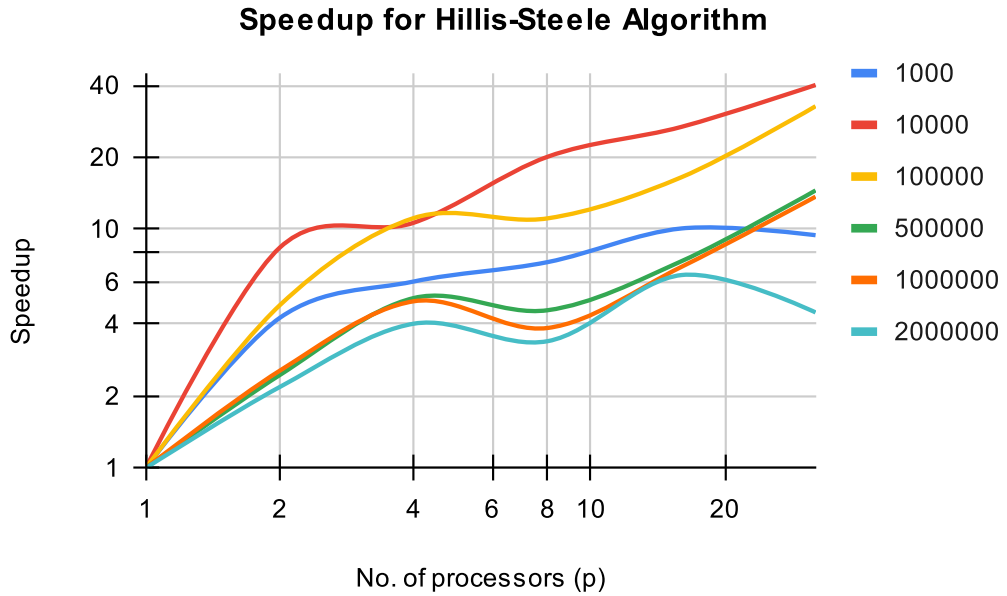
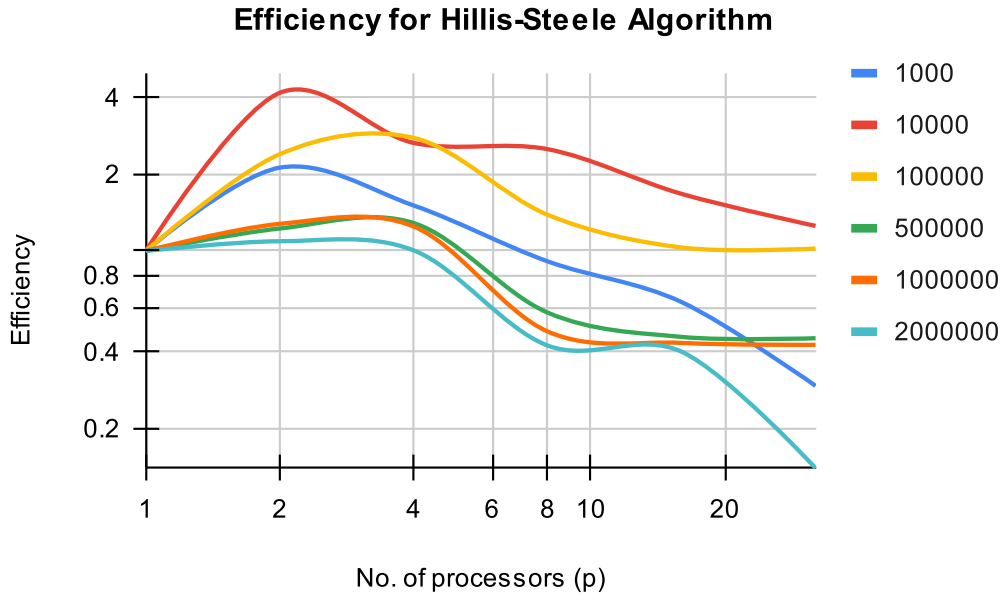


Table 6: Efficiency for different values of p

p	n=1000	n=10000	n=100000	n=500000	n=1000000	n=2000000
1	1.0000000	1.0000000	1.0000000	1.0000000	1.0000000	1.0000000
2	2.1260177	4.1844298	2.4010080	1.2254115	1.2778586	1.0920779
4	1.5096546	2.6601280	2.7829504	1.2873061	1.2479335	1.0053092
8	0.9102251	2.5127912	1.3876746	0.5724128	0.4824363	0.4247614
16	0.6296523	1.6764147	1.0318317	0.4593731	0.4350976	0.4017902
32	0.2948684	1.2547181	1.0224307	0.4540327	0.4274332	0.1400902



References

[1] *Prefix Sum*, available at `https://en.wikipedia.org/wiki/Prefix_sum`.

[2] Hillis, W. Daniel; Steele, Jr., Guy L. (December 1986). "Data parallel algorithms". Communications of the ACM. 29 (12): 1170–1183.

[3] *Prefix Sums and Their Applications*, Guy E. Blelloch, School of Computer Science, Carnegie Mellon University