Homework 2 Report

KAI HOGAN

This report compares the performance of one sequential and two parallel prefix sum algorithms. The sequential algorithm computes prefix sums in a single pass through the array. The first parallel algorithm, Hillis-Steele scan (O(NlogN)), iteratively adds elements from increasing offsets at each step. The second parallel algorithm, two-pass approach (2(N-1)), first builds a binary tree of partial sums (upsweep phase), then uses these to compute the final prefix sums (downsweep phase). I tested each implementation on the Talapas system using 8, 16, 24, and 28 threads across workloads of 1,000, 10,000, 100,000, and 1,000,000 elements.

CCS Concepts: • **Computing methodologies** → **Parallel algorithms**; *Shared memory algorithms*; • **Theory of computation** → *Design and analysis of algorithms.*

ACM Reference Format:
Kai Hogan. 2025. Homework 2 Report. *ACM Trans. Graph.* 1, 1, Article 1 (October 2025), 2 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Findings

1.1 Sequential Algorithm vs Parallel Algorithm 1 (O(NlogN)):

The sequential algorithm consistently outperformed P1 across nearly all configurations. For 1,000 elements, the sequential algorithm was 1,000× faster with 8 threads, increasing to over 200,000× faster with 28 threads. At 10,000 elements, the gap narrowed to 20× with 8 threads but still grew to 6,700× with 28 threads.

For 100,000 elements, performance differences diminished significantly, only 2× faster with 8 threads, though the gap widened to approximately 450× at 28 threads. At 1,000,000 elements, the sequential algorithm maintained a modest 1.4-1.8× advantage for 8-24 threads but jumped to 181× faster at 28 threads.

Conclusion: The sequential algorithm proved more efficient in the vast majority of cases, with P1's parallel overhead severely impacting performance, especially at 28 threads.

Table 1: O(N-1) vs O(NlogN)

Threads	Elements	O(N-1) Time (s)	O(NlogN) Time (s)	Speedup (NlogN/N-1)
8	1,000	9.42×10^{-7}	5.66×10^{-4}	601.1× slower
8	10,000	2.82×10^{-5}	5.52×10^{-4}	19.6× slower
8	100,000	3.28×10^{-4}	7.57×10^{-4}	2.3× slower
8	1,000,000	1.75×10^{-3}	3.12×10^{-3}	1.8× slower
16	1,000	7.31×10^{-7}	8.63×10^{-4}	1180.4× slower
16	10,000	3.27×10^{-5}	9.54×10^{-4}	29.2× slower
16	100,000	3.35×10^{-4}	1.04×10^{-3}	3.1× slower
16	1,000,000	1.78×10^{-3}	2.45×10^{-3}	1.4× slower
24	1,000	9.82×10^{-7}	9.85×10^{-4}	1003.3× slower
24	10,000	3.43×10^{-5}	1.31×10^{-3}	38.1× slower
24	100,000	3.13×10^{-4}	1.34×10^{-3}	4.3× slower
24	1,000,000	1.86×10^{-3}	2.90×10^{-3}	1.6× slower
28	1,000	7.31×10^{-7}	1.50×10^{-1}	204,924× slower
28	10,000	2.89×10^{-5}	1.95×10^{-1}	6,741× slower
28	100,000	3.27×10^{-4}	1.50×10^{-1}	459× slower
28	1,000,000	1.51×10^{-3}	$\times10^{-1}$	181× slower

Fig. 1. Sequential algorithm vs P1 algorithm

1.2 Sequential vs Parallel Algorithm 2 (2(N-1)):

P2 was fairly efficient for large workloads with optimal thread counts but underperformed on smaller datasets. For 1,000 elements, P2 was 75-125× slower with 8-24 threads and far 362,000× slower at 28 threads. At 10,000 elements, P2 remained 4-6× slower for 8-24 threads and 12,900× slower at 28 threads.

Performance became competitive at 100,000 elements, where P2 was 1.3-1.5× faster with 8-16 threads and marginally slower with 24 threads. For 1,000,000 elements, P2's performance excelled with 8-24 threads (2.5-2.7× faster) but took a significant hit at 28 threads (351× slower).

Conclusion: P2 effectively parallelizes large workloads with moderate thread counts but suffers severe overhead penalties with excessive threading or small datasets.

Table 2: O(N-1) vs 2(N-1)

Threads	Elements	O(N-1) Time (s)	2(N-1) Time (s)	Speedup (2(N-1)/N-1)
8	1,000	9.42×10^{-7}	7.08×10^{-5}	75.1× slower
8	10,000	2.82×10^{-5}	1.39×10^{-4}	4.9× slower
8	100,000	3.28×10^{-4}	2.61×10^{-4}	1.3× faster
8	1,000,000	1.75×10^{-3}	6.93×10^{-4}	2.5× faster
16	1,000	7.31×10^{-7}	7.99×10^{-5}	109.3× slower
16	10,000	3.27×10^{-5}	1.78×10^{-4}	5.4× slower
16	100,000	3.35×10^{-4}	2.25×10^{-4}	1.5× faster
16	1,000,000	1.78×10^{-3}	6.58×10^{-4}	2.7× faster
24	1,000	9.82×10^{-7}	1.16×10^{-4}	118.1× slower
24	10,000	3.43×10^{-5}	1.61×10^{-4}	4.7× slower
24	100,000	3.13×10^{-4}	2.87×10^{-4}	1.1× slower
24	1,000,000	1.86×10^{-3}	7.16×10^{-4}	2.6× faster
28	1,000	7.31×10^{-7}	2.65×10^{-1}	362,482× slower
28	10,000	2.89×10^{-5}	3.75×10^{-1}	12,974× slower
28	100,000	3.27×10^{-4}	2.48×10^{-4}	1.3× faster
28	1,000,000	1.51×10^{-3}	5.30×10^{-1}	351× slower

Fig. 2. Sequential algorithm vs P2 algorithm

1.3 Parallel Algorithm 1 vs Parallel Algorithm 2:

P2 consistently outperformed P1 except under high thread contention. For 1,000 elements, P2 was 8-11× faster with 8-24 threads but 1.8× slower at 28 threads. This pattern repeated at 10,000 elements (P2: 4-8× faster for 8-24 threads, 1.9× slower at 28 threads).

At 100,000 elements, P2 maintained a 2.9-4.7× advantage for 8-24 threads and dramatically outperformed P1 by 605× at 28 threads—a

Author's Contact Information: Kai Hogan, khogan@uoregon.edu.

ACM 1557-7368/2025/10-ART1
https://doi.org/XXXXXXX.XXXXXXX

notable exception to the trend. For 1,000,000 elements, P2 was approximately 4× faster with 8-24 threads but 1.9× slower at 28 threads.

Conclusion: P2's two-pass approach generally outperforms P1's logarithmic design, though both algorithms suffer at 28 threads due to excessive parallelization overhead, with P2 showing anomalous resilience only at 100,000 elements.

1.4 Conclusion

P2's two-pass approach generally outperforms P1's logarithmic design across most configurations. However, both parallel algorithms suffer severe performance degradation at 28 threads due to excessive overhead. The sequential algorithm was generally more efficient for workloads under 100,000 elements and at 28 threads regardless of workload size. With optimal thread counts (8-16 threads) and large workloads (100,000+ elements), P2 surpassed the sequential algorithm while P1 approached comparable performance. These results demonstrate that parallel prefix sum algorithms only provide benefits when workload size justifies the overhead and thread count is optimal.

Table 3: O(NlogN) vs 2(N-1)

Threads	Elements	O(NlogN) Time (s)	2(N-1) Time (s)	Speedup (NlogN/2(N-1))
8	1,000	5.66×10^{-4}	7.08×10^{-5}	8.0× faster
8	10,000	5.52×10^{-4}	1.39×10^{-4}	4.0× faster
8	100,000	7.57×10^{-4}	2.61×10^{-4}	2.9× faster
8	1,000,000	3.12×10^{-3}	6.93×10^{-4}	4.5× faster
16	1,000	8.63×10^{-4}	7.99×10^{-5}	10.8× faster
16	10,000	9.54×10^{-4}	1.78×10^{-4}	5.4× faster
16	100,000	1.04×10^{-3}	2.25×10^{-4}	4.6× faster
16	1,000,000	2.45×10^{-3}	6.58×10^{-4}	3.7× faster
24	1,000	9.85×10^{-4}	1.16×10^{-4}	8.5× faster
24	10,000	1.31×10^{-3}	1.61×10^{-4}	8.1× faster
24	100,000	1.34×10^{-3}	2.87×10^{-4}	4.7× faster
24	1,000,000	2.90×10^{-3}	7.16×10^{-4}	4.0× faster
28	1,000	1.50×10^{-1}	2.65×10^{-1}	1.8× slower
28	10,000	1.95×10^{-1}	3.75×10^{-1}	1.9× slower
28	100,000	1.50×10^{-1}	2.48×10^{-4}	605× faster
28	1,000,000	2.73×10^{-1}	30×10^{-1}	1.9× slower

Fig. 3. P1 algorithm vs P2 algorithm