Design A Parallel Algorithm For The Parallel Prefixproblem That Runs In Time O(log N) With N/ Logn Processors

Design A Parallel Algorithm For The Parallel Prefixproblem That Runs In Time O(log N) With N/ Logn Processors

The parallel prefix problem, also known as the scan problem, is a fundamental computation pattern in parallel algorithms. It involves computing all the prefix sums (or more generally, prefix operations) of a sequence of data elements efficiently using multiple processors. Achieving a time complexity of O(log N) with a processor count of N/ log N is a significant stride in designing scalable parallel algorithms, especially for large datasets. This article delves into the design of such an algorithm, explaining the underlying concepts, detailed steps, and the theoretical foundations that enable optimal parallel performance.

Understanding the Parallel Prefix Problem

What Is the Parallel Prefix Problem?

The parallel prefix problem involves computing the prefix operation over a sequence of elements. Given an array A of N elements and an associative binary operation ⊕ (such as addition, multiplication, or maximum), the goal is to compute an array B such that:

$$ B[i] = A[1] \ ⊕ \ A[2] \ ⊕ \ ... \ ⊕ \ A[i] \quad \text{for} \ i=1,2,\ldots,N $$

This operation is fundamental in various applications, including prefix sums, polynomial evaluation, and parallel sorting algorithms.

Significance in Parallel Computing

Parallel prefix computations are critical because they often serve as building blocks for more complex algorithms. Efficiently solving the prefix problem in parallel reduces overall computation time and enhances the scalability of systems, especially in high-performance computing environments.

Challenges in Designing Parallel Prefix

Algorithms

While the prefix problem is straightforward sequentially, parallelizing it efficiently presents challenges:

- **Maintaining Associativity:** The operation must be associative to enable parallel computation without ambiguity.
- **Minimizing Time Complexity:** Achieving the theoretical lower bound of O(log N) time for the computation.
- **Optimal Processor Utilization:** Using as few processors as possible while maintaining efficiency.

The classic parallel prefix algorithm, known as the Blelloch scan, achieves O(log N) time with N/2 processors. Our goal extends this efficiency further by designing an algorithm that runs in O(log N) time with N/ log N processors.

Designing the Algorithm: Core Concepts

Achieving the desired performance requires a combination of parallel reduction and expansion phases, carefully orchestrated to utilize the processor resources efficiently.

Key Ideas

- **Divide and Conquer:** Break down the problem into smaller subproblems that can be processed in parallel.
- **Tree-Based Computation:** Use a balanced binary tree structure to perform prefix computations in a hierarchical manner.
- **Processor Allocation:** Allocate processors dynamically to balance load and minimize idle time.
- **Parallel Reduction and Distribution:** First perform a reduction phase to compute partial results, then a distribution phase to generate the final prefix outputs.

Step-by-Step Algorithm Design

The following sections outline the detailed steps involved in constructing the parallel prefix algorithm optimized for time and processor count.

1. Partitioning the Data

- Divide the array A of size N into blocks of size $\log N$.
- Number of blocks: $K = \frac{N}{\log N}$.

This partitioning allows the algorithm to process each block independently and in parallel.

2. Assigning Processors

- Allocate $\frac{N}{\log N}$ processors, assigning each block to a processor.
- Each processor handles its block of size $\log N$.

This assignment ensures that the total processors used are N/ log N, matching the target.

3. Local Prefix Computation within Blocks

- Each processor computes the prefix operation locally within its block sequentially or with a small parallel approach.
- For each block, compute local prefix sums:

$$
B_j[i] = A_{j} [1] \ ⊕ \ A_{j} [2] \ ⊕ \ ... \ ⊕ \ A_{j} [i]
$$

where A_j is the j-th block.

- Time complexity per block: O($\log N$)) (since block size is $\log N$)).

4. Building the Block Prefix Tree

- Create a tree structure over the block results to compute the prefix over block aggregates.
- Perform an up-sweep (reduce) phase:

- Pairwise combine the last element of each block's prefix with neighboring blocks.
- Propagate partial results up the tree in O($\log K$)) = O($\log (N/\log N)$)) steps, which simplifies to O($\log N$)).

- This phase aggregates the block results into a global prefix structure.

5. Propagating Partial Results Down the Tree

- Perform a down-sweep phase to distribute the prefix sums from the parent nodes to child nodes.
- Each processor updates its local prefix array accordingly, adding the prefix sum of all previous blocks.

6. Final Local Prefix Adjustment

- Each processor adds the prefix sum obtained from the block prefix tree to its local prefix array.
- This step ensures that each element’s prefix sum accounts for the contributions from all preceding blocks.

Analyzing Time and Processor Complexity

Time Complexity

- Local prefix computation within each block: O(\(\log N\))
- Up-sweep (reduce) over block aggregates: O(\(\log N\))
- Down-sweep for distribution: O(\(\log N\))
- Final local adjustments: O(\(\log N\))

Total time: O(\(\log N\))

Processor Utilization

- Total processors used: N/ log N
- Each processor handles a block of size \(\log N\).

This distribution ensures optimal utilization, balancing the workload and achieving the targeted processor count.

Algorithm Summary

Phase	Description	Time Complexity	Processors Used

Partitioning	Divide array into blocks	-	N / log N
Local Prefix	Compute within blocks	O(log N)	N / log N
Tree Reduction	Aggregate block results	O(log N)	N / log N
Distribution	Propagate prefix sums	O(log N)	N / log N
Final Adjustment	Add prefix sums to local arrays	O(log N)	N / log N

Total parallel runtime: O(log N)

Advantages of the Proposed Algorithm

- Optimal Time Complexity: Achieves the theoretical lower bound of O(log N).
- Efficient Processor Use: Utilizes N/ log N processors, balancing the workload.
- Scalability: Suitable for large datasets, leveraging parallelism effectively.
- Flexibility: Works with any associative binary operation, broadening applicability.

Practical Considerations and Implementation Tips

- Memory Management: Ensure that each processor has sufficient local memory to handle its block.
- Synchronization: Use barriers or synchronization primitives at key phases (up-sweep/down-sweep) to coordinate processors.
- Load Balancing: Verify that block sizes are uniform to avoid processor idle time.
- Handling Non-Divisible N: For datasets where N is not divisible by \(\log N\), pad the array with identity elements.

Conclusion

Designing a parallel prefix algorithm that runs in O(log N) time using N/ log N processors involves thoughtful partitioning, hierarchical computation, and efficient communication. By dividing the array into manageable blocks, performing local prefix computations, and then combining these results through tree-based reduction and distribution phases, the algorithm achieves optimal parallel performance. This approach not only demonstrates the theoretical power of parallel algorithms but also provides practical insights into scalable computation for large-scale data processing.

Implementing such an algorithm can significantly enhance the efficiency of high-performance computing systems, enabling faster data analysis, real-time processing, and more responsive parallel applications. As parallel computing continues to evolve, algorithms like this lay the groundwork for even more sophisticated and resource-efficient computational strategies.

Frequently Asked Questions

What is the primary goal when designing a parallel algorithm for the prefix problem that operates in O(log N) time?

The main goal is to compute the prefix sum (or similar prefix operation) efficiently by minimizing the total computation time to O(log N), leveraging a sufficient number of processors (N / log N) to achieve this time complexity.

How does the number of processors (N / log N) influence the design of the parallel prefix algorithm?

Using N / log N processors allows for dividing the input into smaller segments processed concurrently, enabling the algorithm to perform multiple computations simultaneously and reduce the overall runtime to O(log N). This processor count balances workload distribution and communication overhead.

What are the key steps involved in designing an O(log N) parallel prefix algorithm with N / log N processors?

The key steps include: partitioning the input array into segments, performing local prefix computations within each segment, performing a reduction phase to compute segment summaries, and then using these summaries to adjust local prefix results, all orchestrated to complete in O(log N) time.

Why is the parallel prefix problem important in parallel computing, and what applications benefit from such algorithms?

The parallel prefix problem is fundamental for enabling efficient parallel algorithms in various applications such as sorting, scanning, and graph algorithms. It allows these computations to be performed faster by exploiting concurrency, leading to significant performance improvements in high-performance computing tasks.

What are the challenges in achieving an O(log N) runtime for the prefix problem with N / log N processors, and how can they be addressed?

Challenges include managing communication overhead between processors, balancing the workload, and ensuring synchronization. These can be addressed by designing efficient communication protocols, using hierarchical algorithms, and carefully partitioning data to minimize inter-processor dependencies.

Additional Resources

Designing a Parallel Algorithm for the Prefix Problem with O(log N) Time Complexity Using N / log n Processors

---

Introduction

The prefix problem, also known as the prefix sum problem, is a fundamental computational challenge with widespread applications in parallel algorithms, including sorting, searching, and numerical computations. Efficiently solving this problem in parallel environments has been a central focus of theoretical computer science, especially within the PRAM (Parallel Random Access Machine) model.

In this review, we delve into the design of a parallel prefix algorithm that operates in O(log N) time utilizing N / log n processors. This approach aligns with the goal of optimizing both depth (parallel time) and processor utilization, achieving a balance that is critical for scalable performance. We will systematically explore the problem definition, existing solutions, the core design principles, the detailed algorithmic process, complexity analysis, and potential implementation considerations.

Understanding the Prefix Problem

Formal Definition

Given an array $A = [a_1, a_2, ..., a_N]$, the prefix sum problem involves computing an array $P = [p_1, p_2, ..., p_N]$ where each element:

$$
p_i = a_1 \oplus a_2 \oplus ... \oplus a_i
$$

Here, $\oplus$ denotes a binary associative operation (commonly addition, multiplication, or bitwise operations).

Key Properties

- Associativity: The operation must be associative to facilitate parallelization.
- Input Size (N): The total number of elements.
- Parallel Time (Depth): Our target is $O(\log N)$.
- Processor Count: $N / \log n$ processors, which implies a sublinear number relative to N, demanding efficient load balancing.

Historical Context and Existing Solutions

Sequential Prefix Sum Algorithms

- Naive approach: Sequentially sums elements, taking $O(N)$ time.
- Parallel solutions: Developed to exploit concurrency, reducing depth to $O(\log N)$.

Classical Parallel Prefix Algorithms

- Kogge-Stone Algorithm: Achieves $O(\log N)$ time with $O(N)$ processors.
- Sklansky and Ladner-Fischer Algorithms: Variations optimizing for processor count and memory.

Limitations of Previous Approaches

- High processor count (often $O(N)$) makes scalability challenging.
- Large constant factors due to multiple communication steps.
- Not tailored for scenarios where processor count is limited.

Our goal is to design an algorithm that maintains the logarithmic depth while reducing the number of processors to $N / \log n$.

Fundamental Design Principles

1. Work-Depth Tradeoff: Balance between total work and parallel depth.
2. Processor Utilization: Efficiently distribute the computation across fewer processors.
3. Recursive Divide-and-Conquer: Break down the problem into smaller subproblems.
4. Tree-Based Computation: Use balanced binary trees for combining partial results.
5. Parallel Prefix Computation: Leverage associative operations, enabling concurrent partial sums.

Core Algorithm Design

High-Level Strategy

- Partitioning the Input: Divide the array into blocks, each handled by a subset of processors.
- Hierarchical Computation:
- Local Prefix Computation: Compute prefix sums within each block.
- Global Aggregation: Combine block results to compute prefix sums across blocks.
- Reconstruction: Use the global results to adjust local prefix sums for the final output.

Step-by-Step Algorithm

Step 1: Divide Input into Blocks

- Partition the array A into $B = N / m$ blocks, where m is chosen based on the total number of processors $P = N / \log n$.
- For example, set $m = \log n$, leading to $B = N / \log n$ blocks.

Step 2: Assign Processors

- Allocate processors to each block.
- Each block is assigned approximately $m / B = \log n$ processors, enabling local parallel prefix computation.

Step 3: Local Prefix Computation

- Within each block, perform a parallel prefix sum using a standard tree-based method:
- Parallel Upward Pass: Perform pairwise combinations in $\log m$ steps to compute partial sums.
- Downward Pass: Propagate prefix sums to all elements in the block.
- Total time per block: $O(\log m) = O(\log \log n)$.

Step 4: Compute Block Totals

- Each block computes its total sum s_b.
- Collect these totals into an array $S = [s_1, s_2, ..., s_B]$.

Step 5: Global Prefix of Block Totals

- Perform a prefix sum on the array S using the same parallel prefix method:
- Since $B = N / \log n$, this step takes $O(\log B) = O(\log N - \log \log n)$.
- The result yields the accumulated sum up to each block, S_{acc}.

Step 6: Adjust Local Prefixes

- For each block b, add $S_{acc}[b-1]$ (the sum of all previous blocks) to each element's local prefix sum.
- This adjustment is done in parallel across blocks.

Step 7: Final Assembly

- Concatenate all adjusted local prefix sums to produce the global prefix array P.

Complexity Analysis

Parallel Time

- Local prefix within blocks: $O(\log \log n)$.
- Prefix sum of block totals: $O(\log N)$.
- Adjustment step: $O(1)$ per block, as it involves adding a scalar to each element in the block.
- Overall: Dominated by the global prefix sum of block totals, leading to $O(\log N)$.

Processor Utilization

- Total processors: $P = N / \log n$.
- Processors assigned:
- Each block handled by approximately $\log n$ processors.
- During local prefix computations, these processors operate in parallel.

- The global prefix of block sums is performed with $O(N / \log n)$ processors.

Work

- Total work across all steps: $O(N)$, matching the sequential complexity, which is optimal.

Practical Considerations and Optimizations

Memory Management

- Efficient data layout to minimize communication overhead.
- Use of shared memory or cache-aware data structures for faster access.

Communication Cost

- Minimizing synchronization points, especially during the global prefix sum.
- Using tree-based aggregation to reduce latency.

Processor Scheduling

- Load balancing to prevent bottlenecks.
- Adaptive division of input based on hardware capabilities.

Extending to Other Associative Operations

- The algorithm generalizes beyond addition to any associative operation such as multiplication, min, max, or bitwise operations.

Variations and Extensions

Hierarchical Prefix Computation

- Implement multi-level hierarchies for extremely large N.
- Use multiple layers of prefix computations to improve scalability.

Approximate Prefix Sums

- For applications tolerant to approximation, algorithms can be modified for faster execution.

Dynamic Input Handling

- Adapt the algorithm for streaming data or dynamically changing input arrays.

Summary and Conclusion

The described parallel prefix algorithm effectively leverages a divide-and-conquer approach combined with hierarchical prefix computations to achieve O(log N) runtime using N / log n processors. This method strikes an optimal balance between work efficiency and parallel depth, making it highly suitable for practical parallel processing environments where processor resources are limited.

Its core strengths include scalability, adaptability to various associative operations, and compatibility with modern parallel hardware architectures. By carefully orchestrating local computations, global aggregations, and adjustments, the algorithm embodies the principles of modern parallel algorithm design—maximizing concurrency while minimizing synchronization and communication overhead.

This approach not only advances theoretical understanding but also provides a foundation for practical implementations in high-performance computing systems, distributed data processing, and real-time analytics, where efficient prefix computations are often critical.

References

- Blelloch, G. E. (1990). Scans as primitive parallel operations. IEEE Transactions on Computers, 39(11), 1526-1538.
- Kogge, P. M., & Stone, H. S. (1973). A parallel algorithm for the prefix computation. IEEE Transactions on Computers, 22(8), 786-793.
- Ladner, R. E., & Fischer, M. J. (1980). Parallel prefix computation. Journal of the ACM, 27(4), 831-838.
- Valiant, L. G. (1981). A bridging model for parallel computation. Communications of the ACM, 27(8), 686-691.

In essence, the detailed design of a parallel prefix algorithm operating in $O(\log N)$ time with $N / \log n$ processors hinges on hierarchical decomposition, efficient intra-block prefix computations, and a global aggregation phase. This structured approach ensures optimal parallel performance, making it a cornerstone technique in the domain of parallel algorithms.

Design A Parallel Algorithm For The Parallel Prefixproblem That Runs In Time O Log N With N Logn Processors

Design A Parallel Algorithm For The Parallel Prefixproblem That Runs In Time O Log N With N Logn Processors

Related Articles

- How Many Computers? In A Simple Random Sample Of 195 Households, The Sample Mean Number Of Personal Computers
- A Conducting Rod Whose Length Is ? = 27.0 Cm Is Placed On Frictionless U-shaped Metal Rails That Is Connected
- Three Brothers Operating A Business Where The Action Of One Brother Is Binding On The Other Two Brotherswhat

Back to Home