Binary-Grouped Factorial: A Parallel-Friendly Approach via Power-of-Two Extraction and Tree Reduction

Brian Klemm

Independent Researcher
`Brian@BrianKlemm.com`

8/3/25

Abstract

We introduce the *Binary-Grouped Factorial*, a self-contained algorithm that computes $n!$ in environments offering big-integer multiplication and bit-shifts but lacking a built-in factorial function. It extracts the power-of-two component via Legendre's formula [1], then performs a binary-tree reduction on grouped odd terms. The result is a parallel-friendly, quasi-linear method—within a small constant of GMP's prime-optimized routine [2] —that can be dropped into Java's `BigInteger`, C#'s `BigInteger`, JavaScript's `BigInt`, OpenSSL's BIGNUM, or similar APIs without external libraries.

1 Introduction

In this paper we present the *Binary-Grouped Factorial*, a self-contained two-phase algorithm designed to work in any environment with arbitrary-precision multiplication and bit-shifts but no native factorial function. This algorithm factors out all powers of two via Legendre's formula (Stage 1) [1] and then reduces the odd factors in a balanced binary tree (Stage 2) to compute $n!$. While the GNU Multiple Precision Arithmetic Library (GMP) [2] leverages prime sieving and Fast Fourier Transform (FFT)-based multiplication for peak throughput, our method delivers near-GMP performance without relying on external libraries—making it ideal for Java's `BigInteger`, C#'s `BigInteger`, JavaScript's `BigInt`, OpenSSL's BIGNUM, and similar APIs.

This work began as an exploration of factorial structure through the lens of binary decomposition. By separating even and odd contributions and grouping exponentiations of odd integers, we arrived at a closed-form structure well-suited for parallel computation. Although not quite matching GMP's raw speed, the Binary-Grouped Factorial is conceptually clean and deterministic, making it ideal for teaching and for environments that prize predictable parallel workloads.

The full C++ implementation, along with benchmarks and source code for this algorithm, is available at: `https://github.com/BrianKlemm-GIT/BinaryGroupedFactorial`

Structure Overview. The high-level structure of our method is illustrated below. The input $n!$ is decomposed into two phases: one computes the power-of-two component via Legendre's formula [1]; the other groups the odd terms into exponentiated factors. These are reduced in a binary tree and finally combined with the shift to yield $n!$.

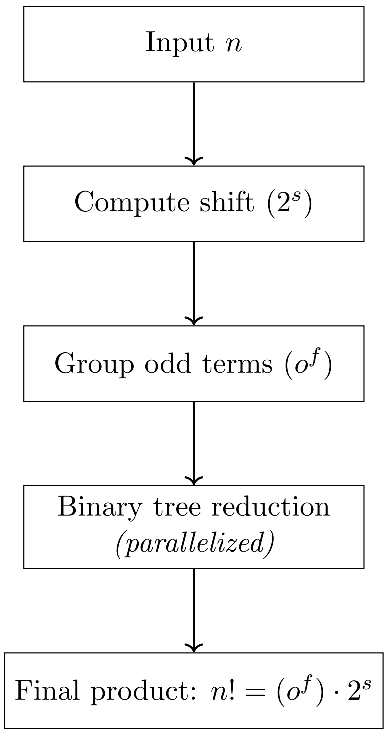

Figure 1: Overview of the computation pipeline.
The factorial is computed in four stages:
1) extract the power-of-two shift;
2) group odd terms;
3) reduce the odd terms with a parallel binary tree; (see Appendix A for more on 'reduce')
4) apply the shift to produce the final result.

Computational Rationale. This ordering minimizes computational cost by keeping intermediate values small for as long as possible. Extracting the power-of-two shift first avoids including a large number of trivial multiplications in the main reduction. Grouping and exponentiating odd terms separately allows for efficient parallel execution with smaller operands. The final shift is applied only once, after the primary product is complete, making the most of GMP's optimized left-shift operation. [2] This structure reduces memory pressure and arithmetic depth while preserving full numerical accuracy.

2 Methodology

In this section we detail the mathematical underpinnings of our approach: first extracting the power-of-two exponent via Legendre's formula, then grouping odd terms into exponentiated factors and outlining the binary-tree reduction strategy.

2.1 Computing the Power-of-Two Shift via Legendre's Formula

The exponent of 2 in $n!$ is calculated using Legendre's [1] formula:

$$\text{shift}(n) = \sum_{k=1}^{\infty} \left\lfloor \frac{n}{2^k} \right\rfloor, \quad \text{(terms vanish once } 2^k > n) \tag{1}$$

Legendre's formula (Eq. 1) computes the total exponent of 2 in $n!$, which lets us factor out all powers of 2 in a single step. Such a bit-level perspective—the "binary soul" of the factorial—shows that once you work in base 2, the structure of $n!$ falls into clear, two-power and odd components.

Example. For $n = 15$, the shift can be computed as:

$$\left\lfloor \frac{15}{2} \right\rfloor + \left\lfloor \frac{15}{4} \right\rfloor + \left\lfloor \frac{15}{8} \right\rfloor + \left\lfloor \frac{15}{16} \right\rfloor = 7 + 3 + 1 + 0 = 11$$

So we factor out 2^{11} from 15!, and reduce the remaining computation to the odd terms.

Computational Note. The binary shift operation is extremely fast in binary arithmetic. Each division by a power of 2 corresponds to a right-shift operation, and computing the total shift takes only $O(\log n)$ time due to the exponential decay of the terms. In practice, the shift count can be computed with a simple loop over $\log_2(n)$ terms. Furthermore, once the exponent is known, the multiplication by $2^{\text{shift}(n)}$ can be performed as a single left-shift — a constant-time operation in arbitrary-precision libraries like GMP. This early factoring removes a large number of trivial multiplications from the main reduction phase, simplifying both memory and arithmetic overhead.

Example Code (GMP). Here is a minimal implementation of the shift computation and its application using GMP's `mpz_class`:

```
#include <gmpxx.h>

// Computes the exponent of 2 in n! using Legendre's formula
unsigned long compute_shift(unsigned long n) {
    unsigned long shift = 0;
    while (n > 0) {
        n >>= 1;          // equivalent to floor(n / 2)
        shift += n;
    }
    return shift;
}

// Applies 2^shift to a result via a fast binary left-shift
mpz_class apply_shift(const mpz_class& x, unsigned long shift) {
    mpz_class result = x;
    mpz_mul_2exp(result.get_mpz_t(), result.get_mpz_t(), shift);
    return result;
}
```

This computes the Legendre shift in $O(\log n)$ time and applies it as a single binary left-shift, which is highly efficient. GMP's `mpz_mul_2exp` performs this operation in-place with no intermediate allocations or multiplications.

2.2 Grouped Exponentiation

$$f = \left\lfloor \log_2\left(\tfrac{n}{o}\right) \right\rfloor + 1 \tag{2}$$

We compute grouped terms of the form o^f where o is odd, and each exponentiated odd term o^f captures the contribution of odd factors to $n!$, i.e. every factor not accounted for by the power-of-two shift. This reflects how many multiples of o appear in the factorial product, stratified by binary expansion.

Example. For $n = 15$, the odd integers up to n are:

$$\{1, 3, 5, 7, 9, 11, 13, 15\}$$

For each o, we compute $f = \lfloor \log_2(15/o) \rfloor + 1$:

o	$15/o$	$f = \lfloor \log_2(15/o) \rfloor + 1$
1	15	4
3	5	3
5	3	2
7	2.14	2
9	1.66	1
11	1.36	1
13	1.15	1
15	1	1

Thus, the grouped terms are:

$$1^4, \quad 3^3, \quad 5^2, \quad 7^2, \quad 9^1, \quad 11^1, \quad 13^1, \quad 15^1$$

2.3 Parallel Reduction

The use of a binary tree structure to reduce exponentiated odd terms offers two key computational advantages. First, it minimizes the multiplicative depth—the longest chain of sequential multiplications—to $\log_2 k$ for k grouped terms. This is crucial for both parallelism and numerical stability, as it balances the computational load across processors and avoids large intermediate values early in the reduction. Second, the tree enables efficient parallel execution. At each level, multiple independent products can be computed simultaneously, making full use of available threads. This structure naturally fits a divide-and-conquer paradigm, where subproducts are combined recursively in a balanced and scalable way. This design caps thread fan-out, allowing predictable scheduling and cache-friendly reduction paths. At each level of the reduction tree, the number of concurrent threads is proportional to the number of parent nodes, halving with each level up to the recursion limit.

We use a thread pool to recursively divide and reduce the product in a binary tree fashion. A fixed recursion depth of 3 limits the thread count to 8, regardless of the system's maximum concurrency.

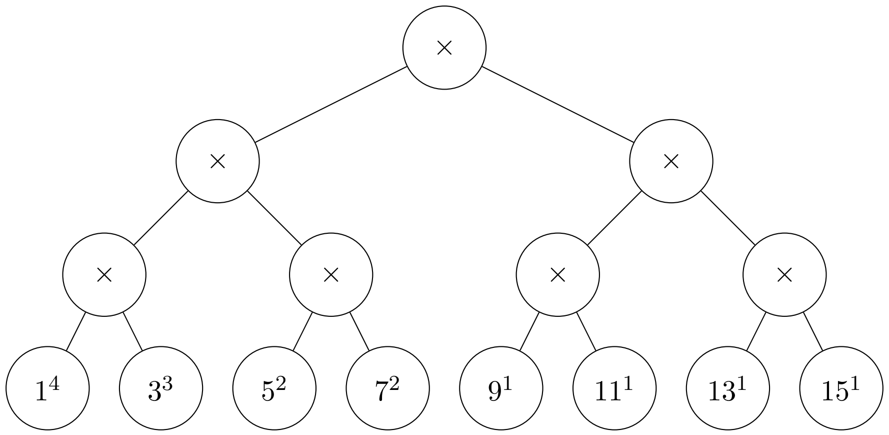

Figure 2: Binary tree reduction of exponentiated odd terms for $n = 15$. Each node represents a partial product computed in parallel.

2.4 Final Combination of Powers and Shift

Once the odd-term product P is obtained via binary-tree reduction, the final factorial result is computed by a single left-shift:

$$n! = P \times 2^{\text{shift}(n)}. \tag{3}$$

This is done with one in-place call to GMP's `mpz_mul_2exp` [2], which applies the shift in constant time with no further multiplications.

3 Implementation

Our implementation follows the four-stage pipeline described in Section 2, mapping each conceptual phase to a corresponding code module. While the methodology separates exponentiation and reduction (Stages 2 and 3), in practice these are implemented together in a single function for simplicity and efficiency. This combination avoids unnecessary intermediate structures and allows tighter control over memory and parallel task flow. Thus, our codebase is structured into three primary routines, corresponding to four logical stages:

1. **Shift Extraction (Module 1):** Computes the Legendre exponent (Eq. 1) [1].
2. **Grouped Exponentiation and Reduction (Modules 2 & 3):** Builds and reduces odd-factor products using a single unified routine.
3. **Final Combination (Module 4):** Applies the power-of-two multiplier to yield $n!$.

3.1 Module 1: Shift Extraction and Precision Setup

Arbitrary-Precision Arithmetic

We use GMP's `mpz_t` and `mpz_class` types [2] to support factorials of arbitrary size.

Shift Extraction

The first module implements Legendre's formula in $O(\log n)$ time via a simple right-shift loop:

```
unsigned long compute_shift(unsigned long n) {
    unsigned long shift = 0;
    while (n > 0) {
        n >>= 1;
        shift += n;
    }
    return shift;
}
```

Module 2 & 3: Grouped Exponentiation & Reduction

The second module builds the list of exponentiated odd factors and reduces them in parallel:

- `compute_powers_grouped` gathers all o^f for odd o (Eq. 2).
- `parallel_reduce_product` uses a fixed-depth thread pool to perform a binary-tree fold of the vector.

Thread Pool Design

We employ a simple fixed-size thread pool (see `thread_pool.h`) that:

- Spawns T worker threads on construction and maintains an internal task queue.
- Provides a `submit(f,args...)` method returning a `std::future<...>`, allowing callers to enqueue arbitrary jobs and later retrieve results.
- Gracefully shuts down by setting an atomic flag, notifying all workers, and joining threads in its destructor.

This pool underpins our recursive binary-tree reduction by letting us spawn up to $2^3 = 8$ simultaneous subtasks (depth limit 3) without oversubscribing the system.

Module 4: Final Combination

Finally, we combine the two components in-place:

```
mpz_mul_2exp(result.get_mpz_t(), prod.get_mpz_t(), shift);
```

This single call applies the 2^{shift} multiplier in constant time, completing the computation of $n!$.

4 Benchmarks

We evaluated the performance of our Binary-Grouped Factorial algorithm against GMP's built-in factorial implementation (`mpz_fac`) [2] across a range of input sizes, from $n = 10^4$ to $n = 10^8$. All tests were conducted on a 72-thread machine, using 8 threads for the Binary-Grouped implementation unless otherwise noted.

n	Threads	Binary Grouped (ms)	GMP (ms)	Validated
10,000	72 (8)	7.19	0.28	yes
50,000	72 (8)	12.06	8.95	yes
100,000	72 (8)	24.76	18.39	yes
1,000,000	72 (8)	213.77	117.56	yes
10,000,000	72 (8)	3,779.63	2,236.34	yes
100,000,000	72 (8)	54,313.7	36,118.3	yes

Table 1: Performance comparison with GMP on a 72-thread machine.

The "Binary Grouped" column reports runtimes (in milliseconds) for our algorithm, while the "GMP" column shows the corresponding runtimes using GMP's `mpz_fac`. The "Threads" column lists the hardware concurrency and the number of threads used in our implementation. All outputs were cross-validated for correctness against GMP's results.

At smaller values of n, GMP's highly optimized internals offer better raw performance. However, as n increases, the Binary-Grouped approach maintains a consistent scaling profile, especially when parallelized, and remains within a factor of 1.5–2$\times$ of GMP for large inputs. Given that GMP is a highly tuned, low-level library, this performance is encouraging—and demonstrates the viability of a structural, parallel-first approach to factorial computation.

5 Complexity Analysis

We now analyze the bit-operation cost of the Binary-Grouped Factorial algorithm by breaking it into its three phases.

1. Shift Extraction

Legendre's formula (Eq. 1) [1] is computed via a simple right-shift loop:

$$\text{shift}(n) = \sum_{k=1}^{\infty} \left\lfloor \frac{n}{2^k} \right\rfloor, \quad \text{(terms vanish once } 2^k > n\text{)}$$

which runs in

$$O(\log n)$$

bit-operations.

2. Grouping & Exponentiation

We form exponentiated odd terms o^f for each odd $o \le n$ and exponent

$$f = \lfloor \log_2(n/o) \rfloor + 1.$$

There are $O(n)$ such terms, each requiring a "pow-ui" on $b = O(\log n)$-bit numbers in $O\big(M(b)\,\log f\big)$, where $M(k)$ is the cost to multiply two k-bit integers [3]. Hence this phase costs

$$O\big(n\,M(\log n)\,\log\log n\big).$$

3. Parallel Binary-Tree Reduction

We multiply the $O(n)$ exponentiated terms in a balanced binary tree of depth $O(\log n)$. At level i, operands have up to $O(i \log n)$ bits, and each level performs $O(1)$ large-integer multiplications in parallel. Summing over levels gives

$$O\big(\log n \ \times \ M(n \log n)\big).$$

Overall Cost

Combining the three phases, the total bit-operation complexity is

$$O(\log n) \ + \ O\big(n\, M(\log n) \ \log\log n\big) \ + \ O\big(\log n\, M(n \log n)\big) \ = \ O\big(\log n\, M(n \log n)\big),$$

since $M(n \log n)$ dominates $n\, M(\log n)$ for large n.

Practical Multiply Models. - With schoolbook multiplication $M(k) = O(k^2)$, the complexity becomes

$$O\big(n^2 \log^3 n\big).$$

- With FFT-based multiplication $M(k) = O(k \log k)$, it is

$$O\big(n \log^2 n \ \log\log n\big),$$

i.e. essentially quasi-linear in the output size.

By contrast, GMP's `mpz_fac_ui` [2] achieves

$$O\big(M(n \log n)\big)$$

by sieving primes and binary-splitting only true prime factors. Our method incurs an extra $\log n$ factor asymptotically but remains within a small constant on multicore hardware thanks to its parallel-friendly structure.

6 Discussion

The Binary-Grouped Factorial fills a niche in modern development environments that offer arbitrary-precision integers but lack a built-in factorial function. Languages and libraries such as Java's `BigInteger`, C#'s `System.Numerics.BigInteger`, JavaScript's `BigInt`, and OpenSSL's BIGNUM API all provide efficient multi-word arithmetic—including multiplication, exponentiation, and bit-shifts—but none include a native `factorial(n)` call. In these settings, developers typically implement a naive loop or a binary-splitting product [4]; our method provides a drop-in, parallel-friendly alternative that:

- Leverages only shifts and multiplies—operations guaranteed in every arbitrary-precision API.
- Exposes fine-grained parallelism through a fixed-depth binary-tree reduction, making it well-suited for multi-core CPUs.
- Delivers quasi-linear scaling in the size of the result ($O(n \log^2 n \ \log\log n)$ with fast multiplies), within a small constant of GMP's prime-sieving approach [2].

“Free” FFT via GMP When linked against GMP [2], our implementation benefits from its highly optimized multiplication routines—often FFT-based for large operands—at no extra effort. In environments whose big-int library uses only schoolbook or Karatsuba multiplication without FFT, one may observe a relative performance drop; integrating an FFT-based multiply or a small-prime hybrid can help restore throughput.

Luschny’s Collection Other open-source efforts, such as Luschny’s collection of optimized factorial functions [5], explore prime-based and divide-and-conquer strategies tailored to specific architectures or languages. These emphasize raw throughput and tuning, whereas our approach aims for structural clarity, platform-independence, and parallel scalability.

Use Cases Beyond standard desktop and server applications (where GMP or equivalent is available), the Binary-Grouped Factorial is particularly attractive in:

- **Educational contexts**, for teaching parallel algorithms and number-theoretic insights without hiding everything in a black-box FFT multiplier.
- **Lightweight runtimes** or **embedded systems,** (e.g. microcontrollers, WebAssembly modules) that support big-integer arithmetic but cannot link against an arbitrarily large integer library without a built-in fast factorial function.
- **Deterministic or safety-critical systems**, requiring predictable task graphs and bounded parallelism.

Limitations and Future Work While the Binary-Grouped Factorial does not match the raw asymptotic performance of GMP’s prime-sieving implementation [2] ($O(M(n \log n))$), its structure is conceptually cleaner and more portable. The extra $O(\log n)$ factor can become significant for very large n, especially in environments without fast multiplication (e.g., FFT-based routines). However, the method remains practical for a wide range of real-world cases and is particularly well-suited to parallel execution.

Future work includes:

- **Dynamic depth control**, allowing the reduction tree to adapt its recursion depth based on operand sizes or available concurrency.
- **Workload-aware scheduling**, improving thread-level balance by assigning tasks based on operand bit-widths or expected multiplication cost.
- **Cross-language reference ports**, including idiomatic implementations in JavaScript, Python, Rust, and other modern languages with built-in big integers but no fast factorial. These ports could serve as drop-in tools for developers lacking native support.
- **Runtime benchmarking across ecosystems**, comparing the Binary-Grouped Factorial against naïve loops, recursive methods, and language-standard libraries to identify environments where its structure yields real performance gains.
- **Structural extensions,** to computations like multinomial coefficients, partition functions, or compositions—where similar odd/even decompositions and balanced reductions can offer algorithmic efficiency and clarity.

For small n, precomputed lookup tables (LUTs) remain the optimal solution, offering instant retrieval without any computation. Users targeting the 10^3–10^4 range or below should prefer LUTs when feasible.

7 Conclusions

The Binary-Grouped Factorial offers a fresh, structurally transparent approach to computing $n!$, separating powers-of-two from odd components in a way that naturally supports parallelism. Its tree-based design favors environments that prioritize clarity, determinism, and bounded concurrency—particularly where external libraries like GMP are unavailable or unwieldy.

Rather than compete with highly tuned, prime-sieving implementations, this algorithm targets practical and portable use. Its strengths lie in being easy to reason about, easy to parallelize, and easy to port—qualities that align with both teaching goals and systems constraints.

For small n, table-based lookup remains superior. But in general-purpose codebases with big-integer support, this method offers a viable middle ground: lightweight, efficient, and conceptually clean.

Future directions include:

- **Adaptive reduction strategies**, including dynamic depth control and workload-aware task scheduling.
- **Reference ports**, in languages like JavaScript, Python, Rust, and C#, which support big integers but lack a built-in factorial.
- **Benchmarking across runtimes**, to identify environments where this structure offers measurable advantages.
- **Structural generalizations**, to functions like multinomials or partition counts that benefit from similar decompositions.

The result is not just a different way to compute a factorial—but a reusable design principle for parallel numerical computation. Portable, predictable, and easy to adopt, the Binary-Grouped Factorial fills a quiet but important niche in the modern algorithmic toolkit.

References

[1] Legendre's formula. *Wikipedia, The Free Encyclopedia.* `https://en.wikipedia.org/wiki/Legendre%27s_formula`

[2] GNU MP Manual, version 6.3.0: Integer Functions. `https://gmplib.org/manual/Integer-Functions.html`

[3] Richard P. Brent and Paul Zimmermann. *Modern Computer Arithmetic.* Cambridge University Press, 2010. See Chapters 1 and 3 for multiplication complexity models.

[4] Donald E. Knuth. *The Art of Computer Programming*, Volume 2: Seminumerical Algorithms (3rd ed.). Addison-Wesley, 1997. See Section 4.3.3 for factorial algorithms and binary splitting.

[5] Peter Luschny. Fast Factorial Functions. `https://github.com/PeterLuschny/Fast-Factorial-Functions` Useful comparative implementations including prime-based and divide-and-conquer factorials.

A Binary-Tree Reduction Clarification

In this paper, “reduce” refers specifically to the recursive, pairwise (binary-tree) multiplication of the grouped odd-term factors—i.e. a parallel tree-based fold using the multiplication operator—rather than any abstract “mathematical reduction” operation. This essentially means when you multiply two terms together you 'reduce' the number of terms from two to one.