Fast Factorials & Binomial Computation for Arbitrary Precision

By Henrik Vestermark (hve@hvks.com)

Abstract:

This paper is the subsequent chapter in a series that systematically addresses the real-world challenges of constructing an arbitrary-precision arithmetic toolkit. This installment specifically targets the algorithms and methodologies for calculating factorials, as well as falling and rising factorials, and binomial coefficients. Often categorized as basic operations in the literature, these functions are typically explained using straightforward recursion formulas. However, the paper argues that the simplistic approaches prove inadequate when you scale these calculations to arbitrary-precision arithmetic. The computational intricacies and performance considerations come to the forefront, necessitating more sophisticated algorithms and optimization techniques. The paper aims to unpack these complexities, offering a more comprehensive understanding and effective methods for arbitrary-precision computation of these mathematical functions.

Introduction:

We start with simple functions like the factorial, falling, rising factorials, and binomial coefficients, discuss the performance of the various methods, and recommend which method to use for arbitrary precision computations.
All of these simple functions can be handled using arbitrary integer precision arithmetic. As customary, the actual C++ source code for the calculations will be provided, utilizing the author's arbitrary precision Math library [1].

This paper is part of a series of arbitrary precision papers describing methods, implementation details, and optimization techniques. These papers can be found on my website at www.hvks.com/Numerical/papers.html and are listed below:

1. Fast Computation of Math Constants in arbitrary precision. HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.
2. Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision. HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.
3. Fast Square Root and inverse calculation for arbitrary precision math. HVE Fast Square Root & inverse calculation for arbitrary precision
4. Fast Exponential calculation for arbitrary precision math. HVE Fast Exp() calculation for arbitrary precision
5. Fast logarithm calculation for arbitrary precision math. HVE Fast Log() calculation for arbitrary precision
6. Practical implementation of Spigot Algorithms for Transcendental Constants. Practical implementation of Spigot Algorithms for transcendental constants
7. Practical implementation of π algorithms. HVE Practical implementation of PI Algorithms

8. Fast Trigonometric function for arbitrary precision. HVE Fast Trigonometric calculation for arbitrary precision
9. Fast Hyperbolic functions for arbitrary precision. HVE Fast Hyperbolic calculation for arbitrary precision
10. Fast conversion from arbitrary precision number to a string. HVE Fast conversion from arbitrary precision to string
11. Fast conversion from a decimal string to an arbitrary precision number. HVE Fast conversion from string to arbitrary precision
12. Fast Computation of Stirling's numbers in arbitrary precision. HVE Fast Computation of Stirling numbers in arbitrary precision
13. Fast Prime computation in arbitrary precision. HVE Fast Prime Computation in arbitrary precision
14. Fast PRNG in arbitrary precision. HVE Fast PRNG in arbitrary precision
15. Fast Fibonacci sequence in arbitrary precision. HVE Fast Fibonacci in arbitrary precision

Contents

The Arbitrary precision library

If you already are familiar with the arbitrary precision library, you can skip this section. There are two classes. One for *int_precision* that handles arbitrary precision integers and one for *float_precision* that handles all *floating-point* arbitrary precision. Since Factorial and binomial coefficients are integers, we only need to highlight the *int_precision* class.

Int_precision class

To understand the C++ code and text we have to highlight a few features of the arbitrary precision library where the class name is *int_precision*. Instead of declaring, a variable with any of the build-in integer type char, short, int, long, long long, unsigned char, unsigned short, unsigned int, unsigned long, and unsigned long long you just replace the type name with *int_precision*. E.g.

```
int_precision ip;  // Declare an arbitrary precision integer
```

You can do any integer operations with *int_precision* that you can do for any type of integer in C++. Furthermore, there are a few methods you will need to know.
One of them is .iszero() which simply returns true or false if the *int_precision* variable is zero or not zero. Another is .even() and .odd() which return the Boolean value of the number even and odd status. There are other methods but I will refer you to the user manual for the arbitrary precision package [1].

Internal format for int_precision variables

For the internal layout of the arbitrary precision number, we are using the STL vector library declared as:

```
vector<uintmax_t> mNumber;
```

uintmax_t is mostly a 64-bit quantity on most systems, so we use a vector of 64-bit unsigned integer to store our integer precision number.
The method .size() returns the number of internal vector entries needed to hold the number.
The number is stored such that the vector mNumber[0] holds the least significant 64-bit binary data. The mNumber[size()-1] holds the most significant 64-bit binary data. The sign is kept separately in a class field variable mSign, which means that the mNumber holds the unsigned binary vector data.

For more details see [1].

The History of Factorials

The factorial, denoted by an exclamation point (!), is a fundamental concept in mathematics, especially in combinatorics, algebra, and mathematical analysis. The history of the factorial is intertwined with the study of permutations and combinations. Here's a brief history:

1. *Ancient Times*: The idea of factorial can be traced back to ancient civilizations. The Indians and Greeks studied permutations and combinations, which inherently involve factorial calculations, even if they didn't use modern notation or names.

2. *Middle Ages*: In the 1200s, Jewish scholars gave descriptions related to the concept of factorial when discussing the number of possible permutations of the Hebrew Bible's verses.

3. *Early Modern Period*: Swiss mathematician Leonhard Euler is often credited with introducing and popularizing the modern notation "n!" for factorial in the 1700s. By this time, the product definition of factorials was well-understood, i.e., n! = n (n-1) ... (2) (1)

4. *18th and 19th Centuries*: Factorial became increasingly significant in the realms of analysis, combinatorics, and probability theory. Notably, factorials play a crucial role in the development of the binomial theorem and the definition of the exponential function.

5. *20th Century to Present*: Factorials continued to be of paramount importance, especially in the study of series and sequences, generating functions, and the analysis of algorithms in computer science. Moreover, the concept of factorial was extended to non-integer values using the gamma function.

Factorials are a fundamental concept taught in schools and are widely used in various areas of mathematics, physics, and engineering.

Applications for factorials

Factorials are deeply embedded in many areas of mathematics and its applications. Here's a more detailed look at some of their primary uses:

1. *Combinatorics and* Permutations: Permutations are the number of ways to arrange n distinct items in a specific order is n!. Combinations are when choosing r items out of n without replacement and when the order doesn't matter, the formula is $\frac{n!}{r!(n-r)!}$

2. *Probability*: Factorials play a pivotal role in determining the number of possible outcomes in various probabilistic scenarios. For example, the probability of a certain sequence happening in a set number of events can often be calculated using factorials.

3. *Binomial Theorem*: When expanding $(x + y)^n$, the coefficients of the terms are combinations (which involve factorials). This is related to the famous Pascal's Triangle.

4. *Calculus*: Taylor and Maclaurin Series. Factorials are used in the expansion of functions as an infinite series and differential Equations. Some solutions, especially for certain recursive relationships, involve factorials.

5. *Quantum Mechanics*: In quantum statistics, factorials can appear in the calculations of the number of ways particles can be arranged.

6. *Computer Science:* Particular algorithm Analysis where factorials can be used to describe the worst-case scenarios of certain algorithms, especially when looking at all possible configurations or orderings or in data Structures, where certain tree structures or recursive data structures might have relationships best described using factorials.

7. *Biology*: Population models and genetics sometimes use factorials when determining possible gene combinations or looking at potential evolutionary pathways.

8. *Number Theory*: Factorials are involved in various number theoretical problems, including properties of prime numbers and combinatorial number theory.

9. *Gamma Function*: This is a generalization of the factorial, allowing us to compute "factorials" of non-integer values (excluding negative integers).

These are just a few of the areas where factorials come into play. They are everywhere in mathematical applications showcasing just how fundamental and versatile they are.

Factorials

Factorials are fundamental mathematical operations frequently employed in various applications. They are commonly represented as either the iterative formula or the recursive formula.

The iterative formula is as follows:

$$n! = n \cdot (n-1) \cdot (n-2) \dots 2 \cdot 1 = \prod_{k=1}^{n} k \tag{1}$$

Or the recursive formula:

$$n! = n \cdot (n-1)! \tag{2}$$

Where 0!=1 and 1!=1.

However, when implemented using basic data types, such as integers, factorials can quickly lead to overflow issues.
Even when using the above code on a 64-bit system you get an overflow after the factorial of 20! and above. This limitation can be addressed using arbitrary precision arithmetic libraries to handle larger values.
This is one of the many situations where an integer arbitrary precision math library is coming in handy to save the day. [1]

Normally you can compute factorials using one of the basic methods below with or without some variations and optimization techniques.

1) *Factorial using recursion*. The straightforward recursive approach is to calculate factorials as shown in the provided code snippet. This method, while simple, may not be efficient due to repetitive calculations.
2) *Factorial using looping*. Loop-based implementations are often preferred over recursion due to better efficiency. The given loop-based code computes factorials iteratively, reducing the risk of stack overflow and improving performance.
3) *Factorial using binary splitting*. The binary splitting method offers an efficient alternative for calculating large factorials. It recursively splits the computation into two halves and then multiplies the results. This approach is particularly advantageous when using arbitrary precision arithmetic. The provided code outlines the binary splitting algorithm, along with its performance benefits.

There are others but usually not completely relevant for an exact computation of factorial. Here we can mention Stirling's approximate formula that for large values of n, gives an estimate for n! that becomes more accurate as n grows. The formula is:

$$n! \approx \sqrt{2\pi n}\,(\frac{n}{e})^n \tag{3}$$

While this doesn't give the exact value, it provides a close approximation for large n, which can be sufficient in many applications.

For non-integer values, the factorial can be generalized using the gamma function, denoted $\Gamma(n)$ where:

$$\Gamma(n) = (n-1)! \tag{4}$$

There are various methods to compute values of the gamma function, such as using continued fractions or series expansions. But these are more complicated than the simple definition of factorial.
Finally, we can mention prime factorization using Legendre's formula, for example, which allows for the calculation of the power of a prime p in the factorial n! This approach is more theoretical but has applications in a certain number of theoretical problems.

Depending on the scenario (e.g., the size of n, required precision, computational resources), one may choose the most appropriate method to compute the factorial.

Factorial using recursion

Most often you see it implemented as below straightforward code using the recursion.

Source factorial via recursion

```
uintmax_t factorial(uintmax_t k)
{
	if (k <= 1)
		return 1;
	return k * factorial(k - 1);
}
```

Factorial using the loop-based algorithm

As we were told back in the computer science class recursion is good but unrolling it to a loop-based algorithm is better. The above code rewritten into a more efficient loop-based implementation is.

Source factorial looping

```
uintmax_t factorialloop(uintmax_t k)
{
	uintmax_t res = 1;
	for (; k > 1; --k)
		res *= k;
	return res;
}
```

Switching to arbitrary precision for large factorial

To overcome the limitation of a maximum of 20! (unsigned data type in a 64-bit system), You need to switch to integer arbitrary precision arithmetic.
The loop-based source above looks like this in the author's own arbitrary precision math package, where the type for an integer in arbitrary precision is called *int_precision*. [1]

```
int_precision factorialloop(const int_precision& kip)
{
	uintmax_t k = (uintmax_t)kip;
	int_precision res = int_precision(1);
	for (; k > 1; --k)
		res *= int_precision(k);
	return res;
}
```

Switching to arbitrary precision allows you to go to "unlimited" factorials, however, the above algorithm is not particularly fast since arbitrary precision slows down the computation. One improvement could be to do some of the intermediate computations using 64-bit arithmetic. E.g., we could group two consecutive numbers together and then add this number to an arbitrary precision result, and to limit any overflow we could rearrange the factorial and start by multiplying the top n with 1 thereafter (n-1)2, etc.

$$n! = n \cdot 1 \cdot (n-1) \cdot 2 \ldots \left(\frac{n}{2}\right)\left(\frac{n}{2}-1\right) \tag{5}$$

This has the benefit, that the biggest number will be less than $\left(\frac{n}{2}\right)^2$ limiting the possibility of overflow. The arbitrary precision implementation will be as below:

```
int_precision factorialloopbalance(const int_precision& kip)
{
	uintmax_t l = 1, k=(uintmax_t)kip;
	int_precision res = int_precision(1);
	for (; k > l; --k,++l)
		res *= int_precision(k*l);
	if(k==l)
		res *= int_precision(k);
	return res;
}
```

This gives a nice speedup since the pairwise multiplication of k and l is done using 64-bit arithmetic. Continue along with that idea you can generalize it and take the pairwise multiplication many times depending on the range of numbers you are multiplying.

Source factorial loop-based max

```
int_precision factorial(const int_precision& kip)
{
	const uintmax_t UINTMAX_T_MAX= ~((uintmax_t)0);
	uintmax_t m=1, kk, prod, k=(uintmax_t)kip;
	int_precision res = int_precision(1);

	if (k <= 1)
		return res;
	for (kk = 1; k>m; --k, ++m)
	{
		prod = k * m;		// Never overflow
		if (kk < UINTMAX_T_MAX / prod)
			kk *= prod;
		else
		{
			res *= int_precision(kk);
			kk = prod;
		}
	}
	res *= int_precision(kk);
	if (k == m)
		res *= int_precision(k);
	return res;
}
```

The performance from the simple loop-based to the more advanced version above represents a speedup of a factor of 6-7 over the straightforward loop-based version. With arbitrary precision arithmetic, you always want to take speedup over simplicity.

Factorial using binary splitting

There is one method that needs to be explained and that is the computation using the binary splitting method. Normally with the balance loop-based approach, you are fine but if you are dealing with a very large factorial the binary splitting method will perform better than any of the other methods. The binary splitting methods split the computation recursively into two halves and then call itself for each half and finally multiply the two numbers together. In essence, it does the same number of multiplications as the other methods but does it more efficiently particularly when using arbitrary precision integer arithmetic. The binary splitting method calculates the division of two factorials $\frac{n!}{m!}$. To just find n! You set m=0. The algorithm is as follows and called with two integer parameters n and m.

```
Factorial_binary_splitting(n,m)
    hid=(n+m)/2
    high = Factorial_binary_splitting(n,mid)
    low = Factorial_binary_splitting(mid,m)
    Return high * low
```

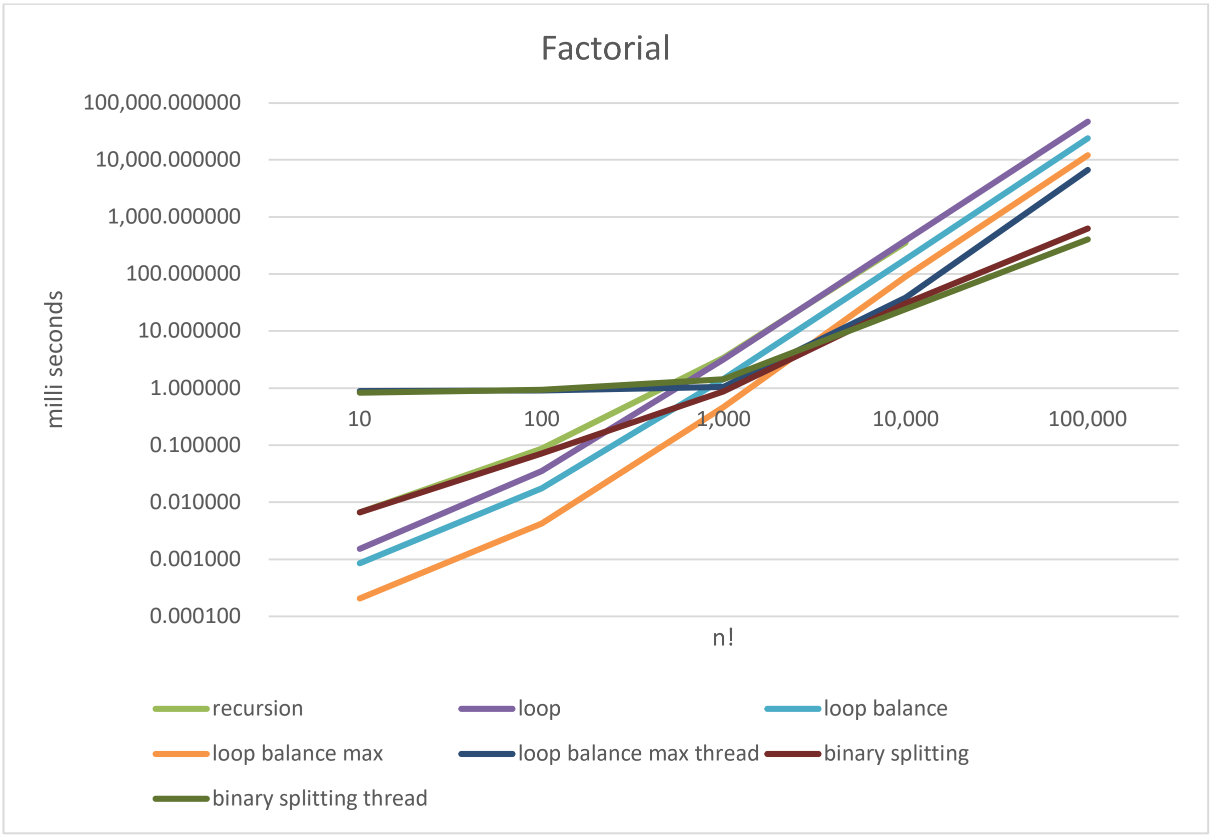
n! is computed with the call FactorialBinarySplitting(n,0);

Source factorial binary splitting

```cpp
int_precision factorial_binary_splitting(const int_precision& a, const
int_precision& b)
{
	const int_precision c1(1), c2(2);
	const int_precision diff = a - b;

	switch (static_cast<uintmax_t>(diff))
	{// Base cases
	case 0: return c1; break;
	case 1: return a; break;
	case 2: return a * (a - c1); break;
	case 3: return a * (a - c1) * (a - c2); break;
	default:// Fall through
		break;
	}
	int_precision m = (a + b) / c2;
	return factorial_binary_splitting(a, m) * factorial_binary_splitting(m, b);
}
```

Performance of Factorials

Looking at the performance chart below we see that the factorial using standard recursion and naive looping is not performing very well. Loop balancing is better but the loop balancing max (optimizing the use of 64-bit arithmetic) is even better. Implementing loop balancing as a two-thread can increase the performance even further however, initially the binary splitting method performed worse than the loop-based version. However, around the 5000! mark, we see that the binary splitting method overtakes the loop-based method and begins to perform significantly better than the loop-based methods. Again, making the binary splitting method multithreading further increases the performance.



Factorial using a hybrid approach

The hybrid approach combines the benefits of loop-based and binary splitting methods. It utilizes looping for smaller values of n and switches to binary splitting for larger values, offering an optimal balance of performance and accuracy. For optimal results, multithreading can be employed to further enhance the performance of the binary splitting method.
We can now present the hybrid factorial solutions that take advantage of both the loop-based approach and the binary splitting method and include threading for optimal performance.

Source Hybrid factorial with threading

```
int_precision factorial(const int_precision& n)
{
	const int_precision c0(0), c1(1), c2(2);

	auto factorial_loop = [&](const int_precision& n)
	{
		const uintmax_t UINTMAX_T_MAX = ~(0ull);
		uintmax_t m = 1, kk, prod, k = static_cast<uintmax_t>(n);
		int_precision res = c1;
		if (k <= 1)
			return res;
		for (kk = 1; k > m; --k, ++m)
		{
			prod = k * m;		// Never overflow
			if (kk < UINTMAX_T_MAX / prod)
				kk *= prod;
			else
			{
				res *= int_precision(kk);
				kk = prod;
			}
		}
		res *= int_precision(kk);
		if (k == m)
			res *= int_precision(k);
		return res;
	};

	// We need to use the std::function to be able to call the lambda
	// function recursively.
	std::function<int_precision(const int_precision&, const int_precision&)>
factorial_binary_splitting = [&](const int_precision& a, const int_precision& b)
	{
		const int_precision diff = a - b;

		switch (static_cast<uintmax_t>(diff))
		{// Base cases
		case 0: return c1; break;
		case 1: return a; break;
		case 2: return a * (a - c1); break;
		case 3: return a * (a - c1) * (a - c2); break;
		default:// Fall through
			break;
		}
		int_precision m = (a + b) / c2;
		return factorial_binary_splitting(a, m) *
factorial_binary_splitting(m, b);
	};

	// The balance loop base is fastest below 5000!
	if (n <= int_precision(5000))
		return factorial_loop(n);

	// Do the binary splitting method above 5000!
	// with or without threading. 2 ways Threading improves the speed with
	// 30-60% over the non-threaded version as shown below
	int_precision m = (n) / c2, high, low;
	std::thread first([&]()
		{
			high = factorial_binary_splitting(n, m);
		});		// interval [n...m]
	std::thread second([&]()
		{
			low = factorial_binary_splitting(m, c0);
		});		// interval [m...0]
	first.join();
	second.join();
	return high * low;
}
```

Recommendation for Factorials

You need a hybrid approach where below the 5000! you use the loop-based method (and maybe even the threaded version of that) and above the 5000! mark you switch to the binary splitting method.

Falling Factorials

Falling factorial describe factorial between two number m and n in failing (or descending notation):

$$n^{\underline{m}} = n \cdot (n-1) \cdot (n-2) \dots (n-m+1) = \prod_{k=n-m+1}^{n} k \qquad (6)$$

Falling factorials are useful when calculating the binomial coefficient:

$$\binom{n}{m} = \frac{n!}{m!(n-m)!} \qquad (7)$$

Since the falling factorial is:

$$\frac{n!}{(n-m)!} = n^{\underline{m}} \qquad (8)$$

And can be handled by one call to the falling factorial function instead of two calls to the factorial function.

See the next chapter for the computation of the binomial coefficient.

The falling factorial has two parameters n and m compared to the regular factorial and again we optimized the computation by doing interim computation using 64-bit arithmetic. In the particular example below, we specify the parameter as the definition of n and m and make the shortcut that n and m have a maximum of 2^{64}-1 or fit into an unsigned 64-bit integer. This limitation is a reasonable choice in nearly all cases. Since we try to do as much calculation using 64-bit integers we have to build in safeguard against unintended overflow. We know from the factorial chapter that function recursion is not as fast as loop-based computation so we go straight to the loop-based solution with arbitrary precision.

Source falling factorial with arbitrary precision

```
// Handles falling factorial
// Same as n!/(n-m)!== (n-m+1)*(n-m+2)*....*(n-1)*n
// Limitation for n is factorial (2^64-1) not entirely arbitrary precision
// but a reasonable max limit
// m is always less than n but no domain error is thrown if it is not
int_precision fallingfactorial(const int_precision& n, const int_precision& m )
{
	const uintmax_t UINTMAX_T_MAX = ~((uintmax_t)0);
	uintmax_t mm = static_cast<uintmax_t>(n - m) + 1;// mm is now the lower 
range
	uintmax_t nn = static_cast<uintmax_t>(n);
	uintmax_t kk, prod;
	int_precision res = int_precision(1);

	if (n.size() > 1|| m>n ) // more >= 2^64! or m>n
		throw int_precision::domain_error();

	if (nn <= 1)
		return res;
	for (kk = 1; nn > mm; --nn, ++mm)
	{
		prod = nn * mm; // Could potentially overflow
		// if no overflow on both conditions, then safely multiply it to kk
		if (prod < nn || kk < UINTMAX_T_MAX / prod)
			kk *= prod;
		else
		{  // multiply the current kk to the result
			res *= int_precision(kk);
			if(prod<nn)
			{	// overflow in pairwise multiplication
				res *= int_precision(nn);
				prod = mm;
			}
			kk = prod;
		}
	}
	res *= int_precision(kk);
	if (nn == mm)
		res *= int_precision(nn);
	return res;
}
```

Now that we have the falling factorial, we can improve on the factorial by splitting a factorial call for n! Into two parts.

$$\frac{n!}{(n-m)!}\ and\ (n-m)! \qquad (9)$$

Where we can run each calculation in parallel, see the source below where we use the C++ threading.

There is some overhead in setting up a thread so usually is first worthwhile when dealing with threading for falling factorial exceeding 2000!
A more interesting method is the binary splitting method for the falling factorial.

Falling Factorial using binary splitting

Like the factorial, we can gain some speed by reusing the binary splitting method for the falling factorial by noticing that $n^{\underline{m}}$ is indeed the two parameters to the binary splitting method *n, m*. By measuring and not surprisingly we see the same performance gain as for the factorial and see that the crossover is around 2,000 where the binary splitting method takes over.

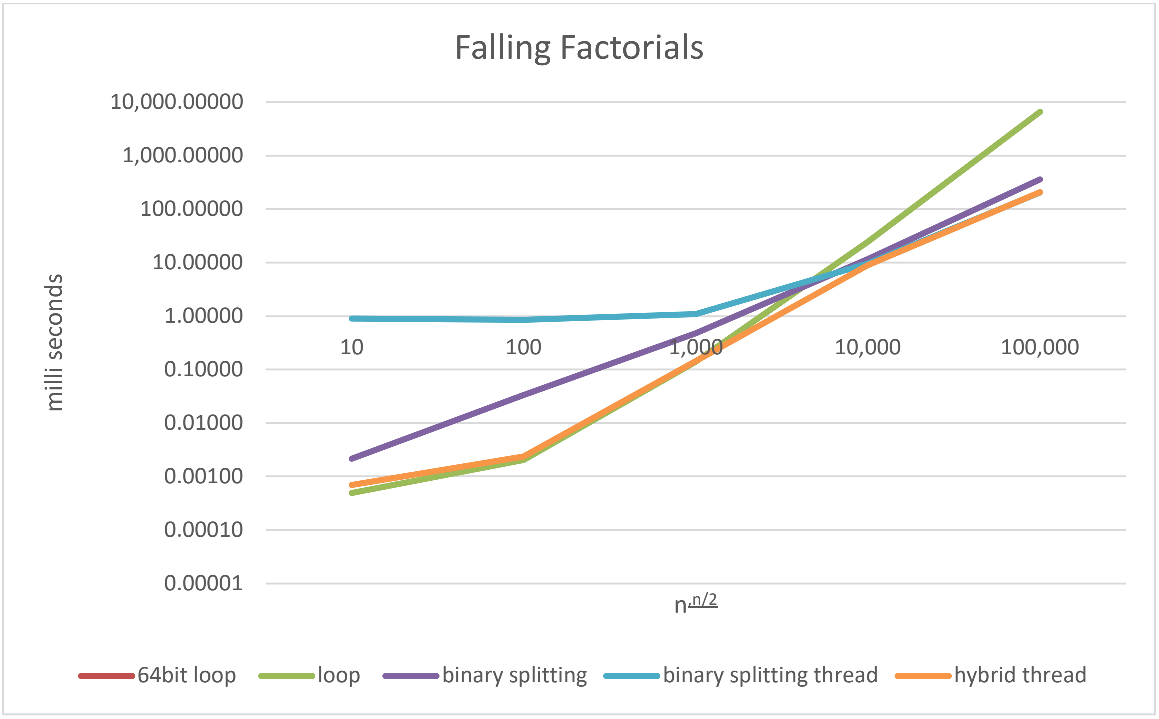
A threaded version of the binary_ splitting method first outperformed the regular binary splitting method by around 5,000. We present the source for our final hybrid version based on the above notes. n and m are the $n^{\underline{m}}$ from the definition of the failing factorials.

Source falling factorial hybrid implementation.

```
// Falling factorial as a hybrid thread version
// notice that maximum n & m is limited to 2^64-1. we should be enough for most
need
int_precision fallingfactorial_hybrid_thread(const int_precision& n, const
int_precision&m )
{
	const int_precision c0(0), c1(1), c2(2);

	auto fallingfactorial_loop = [&](const int_precision& n, const
int_precision& m)
	{
		const uintmax_t UINTMAX_T_MAX = ~(0ull);
		uintmax_t mm = static_cast<uintmax_t>(n - m)+1;// mm is now the
lower range
		uintmax_t nn = static_cast<uintmax_t>(n), kk;
		int_precision res = c1;

		if (nn <= 1)
			return res;
		for (kk = 1; nn > mm; --nn, ++mm)
		{
			uintmax_t prod;
			prod = nn * mm;      // Never overflow since n <= 2'000
			if (kk < UINTMAX_T_MAX / prod)
				kk *= prod;
			else
			{
				res *= int_precision(kk);
				kk = prod;
			}
		}
		res *= int_precision(kk);
		if (nn == mm)
			res *= int_precision(nn);
		return res;
	};

	// We need to use the std::function to be able to call the
	// lambda function recursively.
	std::function<int_precision(const int_precision&, const int_precision&)>
fallingfactorial_binary_splitting = [&](const int_precision& a, const
int_precision& b)
	{
		const int_precision diff = a - b;

		switch (static_cast<uintmax_t>(diff))
		{// Base cases
		case 0: return c1; break;
		case 1: return a; break;
		case 2:		return a * (a - c1); break;
		case 3: return a * (a - c1) * (a - c2); break;
		default:// Fall through
			break;
		}
		int_precision m = (a + b) / c2;
```

```
              return fallingfactorial_binary_splitting(a, m) *
fallingfactorial_binary_splitting(m, b);
       };

       // The balance loop base is fastest below 2000!
       if (n <= int_precision(2000))
              return fallingfactorial_loop(n,m);

       // Do the binary splitting method above 5000!
       // with or without threading. 2 ways Threading improves the speed with
       // 30-60% over the non-threaded version
       int_precision mm = n - m, mid = (n + mm) / c2, high, low;
       if (n <= int_precision(5000))
       {
              high = fallingfactorial_binary_splitting(n, mid);
              low = fallingfactorial_binary_splitting(mid, mm);
       }
       else
       {
              std::thread first([&]()
                     {
                            high = fallingfactorial_binary_splitting(n, mid);
                     });           // interval [n...mid]
              std::thread second([&]()
                     {
                            low = fallingfactorial_binary_splitting(mid, mm);
                     });           // interval [mid...mm]
              first.join();
              second.join();
       }
       return high * low;
}
```

Performance of falling factorial



We see nearly the same pattern as for the factorial the loop-based is faster up to around the 5,000-factorial mark. Then the binary splitting method takes over. By adding threading, you also get a boost in performance around the same mark. The hybrid thread is a combination of the loop-based falling factorial and the binary splitting method.

Rising Factorial

Rising factorial describe factorial between two number m and n in rising (or ascending notation):

$$n^{\overline{m}} = n \cdot (n+1) \cdot (n+2) \ldots (n+m-1) = \prod_{k=n}^{n+m-1} k \tag{10}$$

However, instead of creating a nearly similar function as for the failing factorial, you can map the rising factorial into the falling factorial and reuse some code.

$$n^{\overline{m}} = (n+m-1)^{\underline{m}} \tag{11}$$

Binomial coefficients

$$\binom{n}{m} = \frac{n!}{m!(n-m)!} \text{ where } 0 \le m \le n \tag{12}$$

As mentioned under partial factorials, the binomial coefficients can be implemented using the two different factorial functions.

Source for binomial

```
// binomial computation using arbitrary precision
int_precision binomial(const int_precision& n, const int_precision& m)
{
	int_precision nn(n), mm(m);
	if (mm > nn) return int_precision(0);  // Base case
	if (mm > nn - mm)  // Use the identity to reduce m
		mm = nn - mm;
	return fallingfactorial(nn, mm)/factorial(mm);
}
```

In the above function, we also use the identity that:

$$\binom{n}{m} = \binom{n}{n-m} \tag{13}$$

Sometimes there is a need for the calculation of a steady stream of an increasing number of binomial coefficients. E.g. for calculating Bernoulli’s number, there is a need for that. This can lead to optimization if you instead use one of the two binomial recurrences:

$$\binom{n+1}{m} = \frac{n+1}{n-m+1}\binom{n}{m} = \binom{n}{m} + \binom{n}{m-1} \tag{14}$$

Or:

$$\binom{n}{m+1} = \frac{n-m}{m+1}\binom{n}{m} \tag{15}$$

Reference

1) Arbitrary precision library package. Arbitrary Precision C++ Packages (hvks.com)
2) Numerical recipes in C++, 3rd edition, Cambridge University Press, New York, NY 2007
3) Henrik Vestermark, "HVE The math behind arbitrary". HVE The Math behind arbitrary precision.docx (hvks.com)