On Accelerated Methods to Evaluate Sums of Products of Rational Numbers*

Howard Cheng†
Symbolic Computation Group
University of Waterloo
Waterloo, Canada
hchcheng@scg.math.uwaterloo.ca

Eugene Zima
Symbolic Computation Group
University of Waterloo
Waterloo, Canada
ezima@daisy.uwaterloo.ca

ABSTRACT
In this paper we consider the problem of fast computation of sums of n-ary products of rational numbers, for large n. We present improvements to the standard binary splitting algorithm which are due to numerous factors, including changing the standard arbitrary precision integer representation to one that is more suitable for such computations, unrolling, and chains of recurrences techniques. For the computation of $\zeta(3)$ to 640000 decimal digits, we achieve a speedup factor of 2.65 over the standard binary splitting algorithm, which compares favorably to the ideal case in which the numerator and the denominator can be reduced by their greatest common divisor at no cost. If asymptotically fast multiplication is not available (as in the Java Development Kit), a speedup of an order of magnitude is easily obtained.

Categories and Subject Descriptors
I.1.1 [**Symbolic and Algebraic Manipulation**]: Expressions and Their Representation—*Representations (general and polynomial)*

General Terms
Representation of integers, binary splitting, chains of recurrences

1. INTRODUCTION
Advanced algorithms to perform basic operations on arbitrary precision integers are very well known. Most computer algebra systems (such as Maple) and specialized number theory packages (such as Piologie [8]) contain implementations of these algorithms. Fast arbitrary precision arithmetic is very important in applications such as high-precision floating point computations of values of elementary functions, π, Apéry's constant, and so on. Modern algorithms for such computation are performed over rational numbers until the very last step which involves the conversion from the exact rational value to the high-precision floating point value [3, 5]. In this paper we will consider $\zeta(3)$-like computational tasks. Given a natural number N, we wish to compute

$$S(N) = \sum_{n=0}^{N-1} \frac{a(n)}{b(n)} \prod_{i=0}^{n} \frac{p(i)}{q(i)}, \tag{1}$$

where a, b, p, q are polynomials with integer coefficients.

A widely used approach to the computation of (1) is binary splitting. All intermediate results are represented by the standard base-b representation $X = \sum_{k=0}^{L-1} x_k b^k$, which will be called the *standard representation* in this paper. The binary splitting algorithm is asymptotically faster than repeated multiplication only if the multiplication of L-digit integers is asymptotically faster than $O(L^2)$, because it does not reduce the overall operational complexity of computations. In this paper we will show how it is possible to reduce both the operational and bit complexity of (1), as well as the memory requirement for the calculation.

Creative telescoping is one of the great tools for finding new formulae for well-known constants [1, 2]. Acceleration potentials given by new formulae appear promising. However the use of these formulae for actual computations seems to be too straightforward, and it will be shown that they do not speed up computations as much as they first appear to. In fact, we will show that applying unrolling to a simpler formula can be superior when combined with the other improvements proposed.

For illustration, we consider formulae (2) and (3) for $\zeta(3)$ which were used to compute this constant up to 1000000 and 128000000 decimal digits, respectively [5, 9]:

$$\zeta(3) \approx \frac{1}{2} \sum_{n=0}^{N-1} \frac{(-1)^n \left(205\, n^2 + 250\, n + 77\right) \left((n+1)!\right)^5 (n!)^5}{\left((2\, n+2)!\right)^5} \tag{2}$$

and

$$\zeta(3) \approx \frac{1}{24} \sum_{n=0}^{N-1} \frac{(-1)^n a(n) ((2n+1)!(2n)!n!)^3}{(3n+2)!((4n+3)!)^3}, \tag{3}$$

*This work was supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada.
†Supported by the Natural Sciences and Engineering Research Council Postgraduate Scholarship and the Alberta Heritage Scholarship Fund.

ISSAC 2000, St. Andrews, Scotland
©2000 ACM 1-58113-218-2/00/0008 $5.00

Digits of $\zeta(3)$	Formula (2)		Formula (3)	
	Size of numerator	Useful digits (%)	Size of numerator	Useful digits (%)
10000	61278	20.6	57278	21.3
20000	132556	19.1	124115	19.7
40000	285111	17.7	267350	18.3
80000	610220	16.6	572941	17.1
160000	1300441	15.6	1222365	16.0
320000	2760884	14.7	2597699	15.1
640000	5841773	13.9	5501340	14.2

Table 1: Proportion of useful digits in the numerator before final floating point division.

where $a(n) = 126392n^5 + 412708n^4 + 531578n^3 + 336367n^2 + 104000n + 12463$. Formula (2) provides approximately 3.01 additional decimal digits of accuracy for each extra term, while approximately 5.04 digits are obtained for each extra term in formula (3).

Straightforward application of binary splitting to formula (2) (as it is described in [5]) for $N = 10000$ produces a rational number with the numerator and the denominator having 208367 decimal digits. Simple inspection shows that the greatest common divisor (*gcd*) of the numerator and the denominator is an integer having 170331 decimal digits. This suggests that only 38036 digits (less than 20% of the result) are useful. When N grows, this percentage decreases (Table 1 contains corresponding statistics collected for (2) and (3)). There is no doubt that carrying unnecessary digits slows down the computation. There is also no evidence that the binary splitting technique as described in [5] takes care of cancellation at intermediate steps of computations, although some part of the cancellation factors can be predicted. Unfortunately, divisions have to be performed to remove the predicted factors, which may become computationally prohibitive as N increases.

In this paper, we examine various techniques for accelerating the evaluation of sums of the form (1). These techniques include an alternate representation of integers called the *partially factored representation*, unrolling, and chains of recurrences. The partially factored representation decreases the cost of multiplications and divisions, and it allows common factors in the numerator and the denominator of a rational number to be easily removed. Unrolling is a simple way to obtain formulae with faster convergence. Finally, chains of recurrences [10] can be applied to the computation of $a(n)$ to speed up the computation of each term. Bit complexity is reduced by using the partially factored representation, and operational complexity is reduced by unrolling and chains of recurrences. Although the computation of $\zeta(3)$ is used for illustration, these techniques can be used on other similar sums. Some examples [4, 5] are the evaluation of elementary functions, hypergeometric functions, and the Gamma function at rational points, as well as

$$\frac{1}{\pi} = \sum_{n=0}^{\infty} \binom{2n}{n}^3 \frac{42n+5}{2^{12n+4}}. \tag{4}$$

The rest of the paper is organized as follows. Section 2 gives the necessary preliminaries. Section 3 describes the partially factored representation of integers. The application of this representation to (1) is discussed in Section 4. The combination of this representation with unrolling and chains of recurrences together with experimental results are presented in Section 5, while Section 6 gives concluding remarks and discusses how our method can be used for evaluating series in parallel.

All timings presented here were obtained from running our C++ implementation built on the top of the **Piologie 1.2.1** library [8]. The hardware used is a SUN A25-BA Enterprise server with 1 Gb of RAM and two 400 MHz CPUs[1]

2. PRELIMINARIES

In this section, we recall known techniques to expedite computations, explore possibilities of combining these techniques, and set the acceleration targets provided by the "ideal" case.

2.1 Binary splitting

The idea of binary splitting for the evaluation of (1) is the following [5]. Given bounds n_1, n_2 consider the partial sum

$$S = \sum_{n=n_1}^{n_2} \frac{a(n)}{b(n)} \frac{p(n_1)\cdots p(n)}{q(n_1)\cdots q(n)}. \tag{5}$$

Let $\alpha \geq 0$ be a *cut-off* value. Compute integers $P = p(n_1)\cdots p(n_2)$, $Q = q(n_1)\cdots q(n_2)$, $B = b(n_1)\cdots b(n_2)$ and $T = BQS$. If $n_2 - n_1 \leq \alpha$, these values are computed directly. If $n_2 - n_1 > \alpha$, they are computed using binary splitting:

- choose $n_m = \lfloor \frac{n_1+n_2}{2} \rfloor$,
- compute components P_l, Q_l, B_l, T_l corresponding to the left interval $n_1 \leq n \leq n_m$,
- compute components P_r, Q_r, B_r, T_r corresponding to the right interval $n_m + 1 \leq n \leq n_2$,
- compute $P = P_l P_r$, $Q = Q_l Q_r$, $B = B_l B_r$ and $T = B_r Q_r T_l + B_l P_l T_r$.

Application of this algorithm to (1) starts with $n_1 = 0$ and $n_2 = N - 1$. After this the final floating point division $S = \frac{T}{BQ}$ is performed.

For example, in formula (2), $a(n) = 205n^2 + 250n + 77$, $b(n) = 1$, $p(0) = 1$, $p(n) = -n^5$ for $n > 0$, and $q(n) = 32(2n+1)^5$. In formula (3), $a(n) = 126392n^5 + 412708n^4 + 531578n^3 + 336367n^2 + 104000n + 12463$, $b(n) = 1$, $p(0) = 1$, $p(n) = -n^5(2n-1)^3$ for $n > 0$, $q(0) = 10368$, and $q(n) = 24(3n+1)(3n+2)(4n+1)^3(4n+3)^3$ for $n > 0$. In both cases, $b(n) = 1$ and so we can eliminate B from the computation.

The trace of the binary splitting recursive calls can be represented by a binary tree. We will refer to this tree as the

[1]We used only one CPU in all experiments.

binary splitting tree. The leaf nodes correspond to direct evaluation of partial sums at the *cut-off level*. The success of the application of binary splitting to a particular evaluation of a linearly convergent series is due to the fact that at each node of the binary splitting tree integers of relatively close sizes are multiplied. This provides a balance of operand sizes to take advantage of asymptotically fast integer multiplications.

Before proceeding further we give two quotations characterizing binary splitting: "If the multiplication was implemented as an $M(N) = O(N^2)$ algorithm, the binary splitting algorithm would provide no speedup over step-by-step evaluation" [5]. "It is perhaps worth underlining the observation that these acceleration methods apply only to the bit complexity. The operational complexity is not reduced" [3].

2.2 Chains of recurrences and computation unrolling

The chains of recurrences (CR) technique to accelerate computations was described, for example, in [10]. The combination of this technique with loop unrolling applied to the computation of products was described in [6]. In this paper we will deal with chains of recurrences for polynomials, which are in fact tables of finite differences for polynomials taken at fixed points with a fixed step. Given a polynomial $p_k(i)$ of degree k, which we have to evaluate at $i = a, a+h, a+2h, \ldots$, we represent this with CR

$$\Phi(i = a, h) = \{\varphi_0, +, \varphi_1, +, \ldots, +, \varphi_{k-1}, +, \varphi_k\}(i = a, h),$$

where φ_j is the jth finite difference of $p_k(i)$ taken at the point a with the step h. For example,

$$\begin{aligned} i &= \{0, +, 1\}(i = 0, 1) = \{3, +, 2\}(i = 3, 2), \\ i^3 - 2i + 1 &= \{1, +, -1, +, 6, +, 6\}(i = 0, 1) \\ &= \{1, +, 4, +, 48, +, 48\}(i = 0, 2). \end{aligned}$$

The last example suggests that by using this representation it is possible to compute the values of $i^3 - 2i + 1$ for $i = 0, 2, 4, \ldots$ at the cost of 3 additions at every point i. The presence of the $+$ signs in this representation is explained by the fact that non-trivial chains of recurrences (for non-polynomial expressions) can also have other operation signs in place of $+$. For example, $i! = \{1, *, 1, +, 1\}(i = 0, 1) = \{1, *, 2, +, 10, +, 8\}(i = 0, 2)$. We do not give more definitions on the interpretation and construction of CRs here, because they can be found in the cited literature. What is important here is that we are able to construct CRs from the given input $p_k(i), i, a$, and h efficiently (often "almost" multiplication-free as in [6]).

Given natural numbers N, α such that $\alpha \mid N$ and a sum

$$S(N) = \sum_{n=0}^{N-1} f(n),$$

the unrolling of this sum by a factor of α transforms it into the form

$$S(N) = \sum_{k=0}^{N/\alpha-1} \left(\sum_{i=0}^{\alpha-1} f(\alpha \cdot k + i) \right).$$

The inner sum $\tilde{f}(k, \alpha) = \sum_{i=0}^{\alpha-1} f(\alpha \cdot k + i)$ is subject to additional transformations (such as normalizations, simplifications, etc.) in order to obtain it in the form

$$\frac{\tilde{a}(k)}{\tilde{b}(k)} \prod_{i=0}^{k} \frac{\tilde{p}(i)}{\tilde{q}(i)}.$$

It is straightforward to check that:
1) if $S(N)$ was in the form (1) then such transformations are always possible;
2) if $S(N)$ was in the form (1) with $b(n) = 1$ then it is possible to write $\tilde{f}(k, \alpha)$ such that $\tilde{b}(k) = 1$ also.

The outer sum $\sum_{k=0}^{N/\alpha-1} \tilde{f}(k, \alpha)$ is computed by binary splitting, in which the depth of recursion is $\lfloor \log_2 \alpha \rfloor$ less than it is for the original sum.

For example, the sum (2)

$$\frac{1}{2} \sum_{n=0}^{N-1} \frac{(-1)^n \left(205\, n^2 + 250\, n + 77\right) ((n+1)!)^5\, (n!)^5}{((2\, n + 2)!)^5}$$

after unrolling by a factor of 2 gives

$$2 \sum_{k=0}^{N/2-1} \frac{((2\, k + 2)!)^5\, ((2\, k)!)^5\, a(k)}{((4\, k + 4)!)^5},$$

where $a(k) = 6710880\, k^7 + 29259440\, k^6 + 53729328\, k^5 + 53699400\, k^4 + 31429470\, k^3 + 10726375\, k^2 + 1968260\, k + 149555$.

2.3 On straightforward combination of techniques

It seems worthwhile to try the following straightforward combination of binary splitting, chains of recurrences technique, and unrolling:

1. Assume $N = 2^l \cdot \alpha$ (this guarantees the equality of the number of terms at each branch of the binary splitting tree on the cut-off level).

2. Unroll the given sum by a factor of α:

$$\sum_{k=0}^{2^l-1} \tilde{f}(k, \alpha).$$

3. Apply binary splitting to the last sum, using chains of recurrences to compute the values $\tilde{f}(0, \alpha), \tilde{f}(1, \alpha), \ldots$.

Observe that these chains of recurrences have the form

$$\frac{\{\varphi, *, \psi_0, +, \psi_1, +, \ldots, +, \psi_\beta\}}{\{\tilde{\varphi}, *, \tilde{\psi}_0, +, \tilde{\psi}_1, +, \ldots, +, \tilde{\psi}_{\tilde{\beta}}\}}, \tag{6}$$

where β and $\tilde{\beta}$ are proportional to α.

Using (6) will reduce the number of multiplications to be performed at the cut-off level by a factor of α, replacing them by a similar amount of additions.

Simple analysis of the distribution of the workload among the levels of the binary splitting tree shows that the expected

	Formula (2)		Formula (3)		
Digits	Size of T	Time (s)	Size of T	Time (s)	Speedup
10000	61278	1.85	57278	1.70	1.09
20000	132556	5.82	124115	5.54	1.05
40000	285111	16.79	267350	15.48	1.08
80000	610220	46.92	572941	41.75	1.12
160000	1300441	126.07	1222365	113.38	1.11
320000	2760884	326.05	2597699	304.74	1.07
640000	5841773	805.76	5501340	776.81	1.04

Table 2: Computation of $\zeta(3)$ by binary splitting.

	Formula (2)		Formula (3)		
Digits	Size of T	Time (s)	Size of T	Time (s)	Speedup
10000	45813	1.89	38716	2.33	0.81
20000	102382	6.55	76231	6.61	0.99
40000	224625	18.18	193964	17.90	1.02
80000	489069	48.03	426327	45.14	1.06
160000	1058137	117.55	926933	117.43	1.00
320000	2276859	299.46	2006774	286.69	1.04
640000	4873542	781.44	4319660	684.88	1.14

Table 3: Computation of $\zeta(3)$ by binary splitting, with gcd removed whenever $8 \le n_2 - n_1 < 16$.

savings in running time given by (6) are relatively small. Indeed, let $M(v)$ denote the complexity of multiplication of two v-digit integers. Let L be the size of operands involved in the computation on the $(m+1)$th level of the binary splitting tree. Then, on level m, operands will have size approximately $2L$ digits. The cost of one multiplication on level $m+1$ is $M(L)$ and on level m it is $M(2L)$. Level $m+1$ has twice as many nodes as level m. Therefore, the relative workload on level m compared to level $m+1$ is characterized by the fraction $\frac{M(2L)}{2M(L)}$. If Karatsuba multiplication [7] is used (i.e. $M(2L) \approx 3M(L)$), then $\frac{3}{2}$ more work is done on level m compared to the work on level $m+1$. This means that to some extent it does not matter how much time is spent on the lower levels of recursion, since computational overheads on upper levels are dominating.

If the complexity of arbitrary precision multiplication is almost linear (ideal case), we have $\frac{M(2L)}{2M(L)} \approx 1$, which means that the workload is evenly distributed among the levels of the binary splitting tree. In this ideal case, in order to save 50% of the running time we have to choose the cut-off level at half of the tree height, i.e. choose $\alpha \approx \sqrt{N}$. With such a choice of α most of the saved time however will be spent on the construction of chains of recurrences, which has complexity $O(\max(\beta, \tilde{\beta})^2)$. This observation suggests that brute-force combination of the described techniques will not help much and additional preliminary analysis of the evaluation of (1) is required.

2.4 On "faster and faster convergent series for $\zeta(3)$"

Fast convergence does not guarantee an "impressive" speedup of binary splitting based on faster series. A faster convergent series requires fewer terms to reach the needed accuracy, but the sizes of the numerator and the denominator of each term are larger than they are in a slower convergent series. This means that
1) each term is more difficult to compute;
2) combining terms involves multiplications of larger integers.

Consider (2) and (3). The second one converges faster by a factor of about $\frac{5}{3}$. Our experiments show that the speedup of binary splitting for (3) compared to (2) does not even approach this value. Table 2 presents the data obtained for (2) and (3) with standard binary splitting. Table 3 gives similar data for binary splitting with cancellation of the numerator and the denominator by their gcd at the cut-off level. We remark that the size of Q can differ from the size of T by at most one digit.

2.5 How fast could it be in the "ideal" case?

If we restrict ourselves to computing $\zeta(3)$ by binary splitting using formulae (2) and (3), the ideal case occurs when there are no common factors in the intermediate results. That is, $\gcd(P, Q, T) = 1$ at each step. This is also the ideal case for any acceleration method based on the removal of predicted common factors. In this section, we consider this ideal case and examine the maximum possible improvement in computation time, by removing all common factors at every step in the binary splitting process.

Tables 4 and 5 present experimental results in the ideal case. Columns 4 and 5 show the time and speedup if we assume that the gcd is given to us by an oracle at no cost, but three divisions are still performed to remove it from P, Q, and T. Columns 6 and 7 show the time and speedup if we assume that the oracle provides the reduced P, Q, and T at no cost. In column 3 of the tables, we show the total computation time including the gcd computations and divisions, and we see that the total computation time is dominated by

	Binary Splitting	GCD removed each step	GCD ignored		GCD ignored Division ignored	
Digits	Time (s)	Time (s)	Time (s)	Speedup	Time (s)	Speedup
10000	1.85	41.43	1.36	1.39	0.76	2.43
20000	5.82	164.39	3.69	1.58	1.87	3.11
40000	16.79	656.48	11.47	1.46	5.12	3.28
80000	46.92	2643.92	35.93	1.31	12.85	3.65
160000	126.07	10586.09	108.63	1.16	33.94	3.71
320000	326.05	44621.85	285.57	1.14	87.16	3.74
640000	805.76	184625.19	762.23	1.06	208.02	3.87

Table 4: Ideal case for formula (2).

	Binary Splitting	GCD removed each step	GCD ignored		GCD ignored Division ignored	
Digits	Time (s)	Time (s)	Time (s)	Speedup	Time (s)	Speedup
10000	1.70	34.93	1.05	1.62	0.61	2.79
20000	5.54	137.50	2.88	1.92	1.52	3.64
40000	15.48	551.62	9.06	1.71	4.23	3.66
80000	41.75	2220.23	29.28	1.43	11.25	3.71
160000	113.38	8929.10	90.01	1.26	29.94	3.79
320000	304.74	36319.67	243.55	1.25	77.94	3.91
640000	776.81	149148.38	723.80	1.07	195.01	3.98

Table 5: Ideal case for formula (3).

gcd computations. Experiments for larger numbers of digits were not performed due to the exponential increase in the amount of time required.

We see two trends from these results. First, if we ignore only the cost of gcd computations, the speedup decreases as the number of digits increases. This is due to the additional three divisions performed at each step. We can conclude that it is not sufficient to obtain the gcd at no cost—we must also be able to remove the gcd efficiently. In other words, acceleration methods based on the removal of predicted common factors alone are not sufficient even if the prediction is perfect and efficient. Second, if we also ignore the cost of divisions, the speedup slowly increases because of the reduced sizes of the operands.

3. PARTIALLY FACTORED REPRESENTATION OF INTEGERS

We now consider an alternate representation of integers, called the *partially factored representation*. Let $p_1, \dots, p_m$ be the first m primes. An integer X is represented as

$$X = \left(\prod_{i=1}^{m} p_i^{\alpha_i} \right) x, \tag{7}$$

where $\alpha_i \geq 0$, and x, called the *standard component*, is in standard representation. We will further assume that α_i can be represented in the single-precision integer type provided by the machine, so that additions and subtractions of exponents take constant time. Since p_i can be precomputed and stored in a table, only the α_i's need to be stored in an exponent vector. We will simply write the representation as $(\vec{\alpha}, x)$. Ideally, we would like to have $\gcd(p_i, x) = 1$ for all i, but it is not required in our representation. In the degenerate case, $\vec{\alpha} = \vec{0}$ and we essentially have the standard representation.

The conversion of integers from standard representation to the partially factored representation is performed by trial divisions by each p_i, until $\gcd(p_i, x) = 1$ for all i. The conversion in the other direction is performed by computing $\prod p_i^{\alpha_i}$ and then multiplying the result by x.

Multiplication (division) of partially factored integers is performed simply by the addition (subtraction) of the exponent vectors, together with the multiplication (division) of the standard components. When the standard components are small, multiplications and divisions become significantly faster. Note that if $\gcd(p_i, x) \neq 1$ for some i, some α_i may become negative after a division even though the division is exact. Since binary splitting does not require divisions, this does not occur in our applications.

To compute $(\vec{\alpha}, x_1) \pm (\vec{\beta}, x_2)$, we first compute the gcd of the exponent part, $\vec{\gamma}$, where $\gamma_i = \min(\alpha_i, \beta_i)$. The result is then $(\vec{\gamma}, x)$, where $x = \prod_{i=1}^{m} p_i^{\alpha_i - \gamma_i} x_1 \pm \prod_{i=1}^{m} p_i^{\beta_i - \gamma_i} x_2$. Thus, the gcd of the exponent vector is removed and the remaining integers are converted to standard representation before addition or subtraction takes place. Divisions by 2 (involving only shifting) are performed on x to reduce the size of the standard component. However, divisions by other primes are not performed as trial divisions are too costly for our applications.

This representation is motivated by the need to remove common factors efficiently. First, multiplications and divisions preserve the partial factors, while additions and subtractions preserve as many partial factors as possible without significant computation. As a result, common factors among intermediate results can be removed simply by inspecting the exponent vectors. No standard gcd computation, multiplication, or division is required. In effect, partial factoring

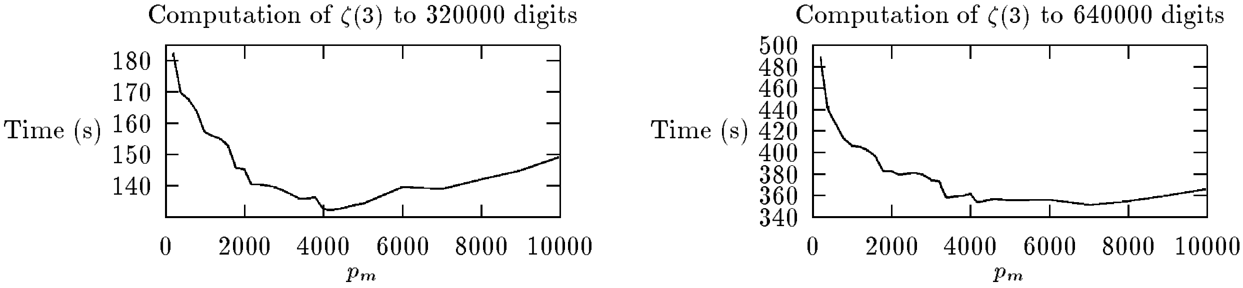

Figure 1: Computation time for formula (2) as p_m varies.

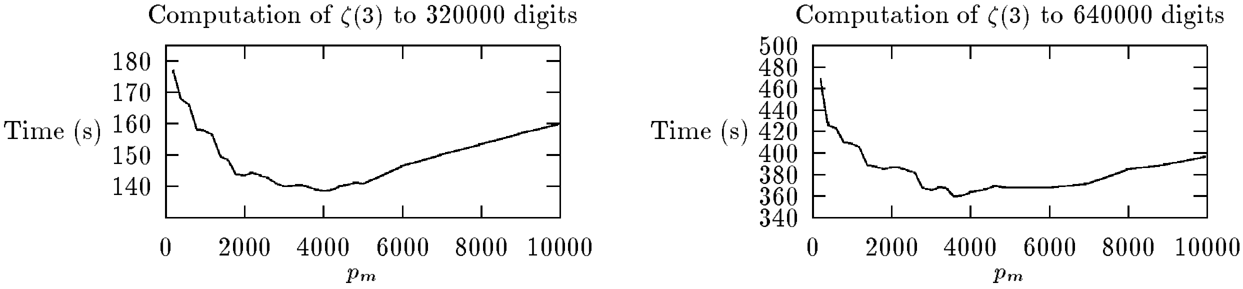

Figure 2: Computation time for formula (3) as p_m varies.

is performed on small initial operands, and the multiplications of the partial factors into the standard components are delayed until they are necessary. The delay allows common factors to be removed easily. The ideal case occurs when the integers have only small prime factors. We see from formulae (2) and (3) that both $p(n)$ and $q(n)$ have many small prime factors, and this representation can be helpful in reducing the bit complexity of the binary splitting process.

The cost of multiplications and divisions of integers in this representation is dominated by the cost of multiplying and dividing the standard components. By partially factoring the integers, we are able to reduce the size of the standard components, and hence the cost of multiplications and divisions of integers. The memory requirement for the computation of (1) is also reduced, allowing more terms to be computed when the amount of physical memory is fixed.

The drawback of this representation is that as m increases, the cost of conversions increases. As a result, the cost of additions and subtractions also increases as they involve conversions from partially factored representation to standard representation. Furthermore, there is a point of diminishing return. The probability of any given integer having a large prime factor is much less than the probability of it having a small prime factor. Therefore, making m too large is unlikely to be useful unless the given integers have some special properties.

We also remark that there is no need for $p_1, \dots, p_m$ to be the first m primes. Any choice of m primes can be used, and this may be useful for some applications.

4. APPLICATION OF PARTIALLY FACTORED REPRESENTATION TO THE COMPUTATION OF $\zeta(3)$

In this section, we show the result of applying the partially factored representation of integers to the computation of $\zeta(3)$. In each of our experiments, the computation is performed in the same way as in standard binary splitting, except that the partially factored representation is used to represent intermediate results. Also, the gcd of the exponent vectors among P, Q, and T is removed after each step. This is only necessary to prevent the overflow of the exponents, and is not necessary for reducing the sizes of the standard components as the common factors in the exponents are removed before additions and subtractions. At the final step, the gcd of the exponent vectors of T and Q is removed, and the two integers are converted to standard representation. This is followed by a division to obtain the floating point result.

In our experiments, we used $m = 574$ (all primes less than 4200) for formula (2) and $m = 503$ (all primes less than 3600) for formula (3) in the partially factored representation. This seems to give a good trade-off between reducing the size of the standard components and the conversion overhead, as shown in Figures 1 and 2. Furthermore, these choices of m work well with the different numbers of computed digits of $\zeta(3)$ used in the experiments. We see from the figures that the "optimal" value of m is not significantly affected as the number of digits doubles.

The difference in the choice of m for the two formulae is due to the fact that there are more small common prime factors in $p(n)$ and $q(n)$ in formula (2) than there are in (3). As m

	Formula (2)		Formula (3)	
Digits	Size of T (digits)	Time (s)	Size of T (digits)	Time (s)
10000	14398	1.34	14080	1.32
20000	29378	3.11	31601	3.21
40000	70146	8.04	75823	8.29
80000	171019	21.23	183326	21.77
160000	412254	53.86	436265	54.91
320000	975188	136.04	1018575	138.15
640000	2260445	348.57	2335702	355.24

Table 6: Timing results using partially factored representation of integers.

	Formula (2)		Formula (3)	
Digits	Size of T (digits)	Time (s)	Size of T (digits)	Time (s)
10000	14398	1.18	14080	1.29
20000	29378	2.84	31601	3.14
40000	70146	7.47	75823	8.14
80000	171019	20.05	183326	21.48
160000	412254	51.64	436265	54.41
320000	975188	130.68	1018575	136.43
640000	2260445	342.68	2335702	355.07

Table 7: Timing results using partially factored representation of integers, as well as chains of recurrences for $a(n)$.

increases, the cost of the conversion to standard representation increases quickly since the number of multiplications required increases. As a result, the cost of additions also increases. An analysis of the computation shows that the most time consuming part of the computation is the addition in $T = Q_rT_l + P_lT_r$. Approximately 60% of the time to compute $\zeta(3)$ to one million decimal digits by formula (2) is spent on conversions to standard representation during additions.

Table 6 shows the result of applying the partially factored representation to formulae (2) and (3). Comparing against the results in Table 2, we see that the partially factored representation achieves a speedup of 2.31 for formula (2) and a speedup of 2.19 for formula (3). We see that this is twice as fast as the case given in columns 4 and 5 of Tables 4 and 5, in which the cost of gcd computations is ignored but the cost of divisions is not. The speedup still compares favorably to the ideal case in which the cost of divisions is also ignored.

We also observe that using the partially factored representation with formula (2) results in a faster algorithm than using it with the "accelerated formula" (3). This is due to the fact that $a(n)$ is larger in formula (3), and so T is also larger as we do not perform trial divisions on the standard component after additions.

Table 7 shows the result of also applying chains of recurrences to the computation of $a(n)$ in each term. We see that chains of recurrences reduce computation time by a small amount for both formulae. In both cases, the time required to compute $a(n)$ is small compared to the total computation time, so only a small improvement can be expected. We also see that the improvement in formula (3) is much less, because the proportion of time used in computing $a(n)$ is smaller in this case. Nevertheless, the operational complexity is reduced.

5. UNROLLING AND UNROLLING TRADE-OFFS

In this section, we discuss the application of unrolling to formula (2). The unrolling of formula (2) by a factor of α gives

$$\zeta(3) \approx \sum_{k=0}^{N/\alpha-1} \frac{a(k)\,((\alpha k+\alpha)!)^5\,((\alpha k)!)^5}{((2\alpha k+2\alpha)!)^5}, \tag{8}$$

where $a(n)$ is a polynomial of degree $5\alpha - 3$. Since the construction of CR for $a(n)$ has complexity $O(deg(a(n))^2)$ [10], there is a limit at which the construction time for CR erases the speedup obtained by unrolling.

Table 8 shows the results of unrolling formula (2) for various unrolling factors α ($\alpha = 1$ means no unrolling), assuming that the CR for $a(n)$ has been precomputed[2]. Unrolling not only allows simplification of the unrolled terms, but also reduces the number of additions in the binary splitting process by a factor of α. As additions in the partially factored representation are costly, the reduction in computation time can be significant. For computing 640000 digits with an unrolling factor of $\alpha = 16$, a speedup of 2.65 is achieved over standard binary splitting.

However, unrolling by a large factor α leads to larger sizes of T and Q. This is due to the fact that the CR computes values of $a(n)$ in standard representation, and no trial divisions

[2] We remark that this precomputation does not take longer than 0.5 seconds with a C++ implementation [6].

Digits	$\alpha = 1$ (s)	$\alpha = 2$ (s)	$\alpha = 4$ (s)	$\alpha = 8$ (s)	$\alpha = 16$ (s)
10000	1.18	0.94	0.79	0.69	0.61
20000	2.84	2.34	2.02	1.82	1.70
40000	7.47	6.46	5.82	5.42	5.23
80000	20.05	18.02	16.68	16.04	15.55
160000	51.64	47.48	44.79	43.53	42.97
320000	130.68	122.35	116.94	114.31	113.75
640000	342.68	324.79	309.71	303.97	302.98
1280000	886.70	854.07	835.60	813.81	820.53

Table 8: Timing results for unrolling formula (2) by a factor of α, with chains of recurrences for $a(n)$.

are performed to reduce the size of the standard component. As a result, both the binary splitting process and the final floating point division may be slower. This explains the increase in computation time for 1280000 digits of $\zeta(3)$ when α is increased from 8 to 16.

Experiments have shown that unrolling formula (3) does not give significant improvements. Unrolling formula (3) leads to terms that are more complicated. By unrolling formula (2), we achieve an "accelerated formula" that converges faster than formula (3). It is also more suitable to our algorithm than formula (3) because the numerator and the denominator have more small prime factors. Thus, unrolling a simpler formula can lead to faster computation than using a formula with a faster convergence.

6. CONCLUDING REMARKS

We have shown that predicted cancellation techniques alone are not sufficient to accelerate the computation of sums of products of rational numbers. This motivates the partially factored representation which allows the computation and removal of common factors efficiently. The speedup provided by this representation is significant. When unrolling and chains of recurrences techniques are combined with the partially factored representation, even better improvements can be obtained. The three techniques presented reduce both the operational and bit complexity of (1), which is supported by our experimental results. For the range of decimal digits computed in our experiments, we can compute $2N$ digits of $\zeta(3)$ with our accelerated methods in the time required by standard binary splitting to compute N digits. In addition, the memory requirement for the computation is reduced. This technique can also be applied to the evaluation of other linearly convergent series. Preliminary experiments show that applying some of the techniques to the evaluation of $1/\pi$ to 50000 decimal digits using formula (4) gives a speedup factor of 6 over the standard binary splitting algorithm.

When asymptotically fast multiplication is not available (as in the Java Development Kit), the speedup is even more impressive. For example, we achieved a speedup of 11.4 compared to standard binary splitting for the computation of $\zeta(3)$ to 80000 digits, when the Fast Fourier Transform and Karatsuba multiplications were disabled.

It is easy to see that the binary splitting algorithm is parallelizable. Up to a certain level of the tree all computations associated with every subtree of this level can be done in parallel. This is used in [5, 9]. None of our improvements damages this property of the computations. Moreover we can find more opportunities for parallelism in the accelerated scheme. First, computations with chains of recurrences can be performed in parallel. Second, the partially factored representation of integers provides more possibilities for parallelization of basic arithmetic compared to the standard representation because all operations on exponent vectors can be performed componentwise.

7. ACKNOWLEDGEMENTS

The authors would like to thank Reinhold Burger (University of Waterloo) for his help in the preparation of this paper.

8. REFERENCES

[1] T. Amdeberhan. Faster and faster convergent series for $\zeta(3)$. *Electronic Journal of Combinatorics*, 3, 1996.

[2] T. Amdeberhan and D. Zeilberger. Hypergeometric series acceleration via the WZ method. *Electronic Journal of Combinatorics (Wilf Festschrift Volume)*, 4, 1997.

[3] J. Borwein and P. Borwein. *Pi and the AGM*. Wiley, 1987.

[4] S. Finch. Favorite mathematical constants. `http://www.mathsoft.com/asolve/constant/constant.html`.

[5] T. Haible and B. Papanikolaou. Fast multiprecision evaluation of series of rational numbers. Technical Report TI-97/7, University of Darmstadt, 1997.

[6] V. Kislenkov, V. Mitrofanov, and E. Zima. How fast can we compute products? In *Proc. of ISSAC'99, Vancouver, Canada*, pages 75–82. ACM Press, 1999.

[7] D. Knuth. *The art of computer programming*, volume 2. Addison-Wesley, third edition, 1997.

[8] S. Wedeniwski. `ftp://ftp.informatik.uni-tuebingen.de/pub/CA/software/Piologie/`.

[9] S. Wedeniwski. `http://www.math.temple.edu/~zeilberg/mamarim/mamarimhtml/Zeta3.txt`.

[10] E. Zima. Simplification and optimization transformations of chains of recurrences. In *Proc. of ISSAC'95, Montreal, Canada*, pages 42–50. ACM Press, 1995.