# 10 Trillion Digits of Pi: A Case Study of summing Hypergeometric Series to high precision on Multicore Systems

Alexander J. Yee
*University of Illinois Urbana-Champaign*[1]
*Urbana, IL*
*Email: a-yee@u.northwestern.edu*

Shigeru Kondo
*Asahimatsu Food Co. LTD*
*Iida, Japan*
*Email: ja0hxv@calico.jp*

***Abstract*— Hypergeometric series are powerful mathematical tools with many usages. Many mathematical functions, such as trigonometric functions, can be partly or entirely expressed in terms of them. In most cases this allows efficient evaluation of such functions, their derivatives and their integrals. They are also the most efficient way known to compute constants, such as $\pi$ and $e$, to high precision. Binary splitting is a low complexity algorithm for summing up hypergeometric series. It is a divide-and-conquer algorithm and can therefore be parallelized. However, it requires large number arithmetic, increases memory usage, and exhibits asymmetric workload, which makes it non-trivial to parallelize. We describe a high performing parallel implementation of the binary splitting algorithm for summing hypergeometric series on shared-memory multicores. To evaluate the implementation we have computed $\pi$ to 5 trillion digits in August 2010 and 10 trillion digits in October 2011 — both of which were new world records. Furthermore, the implementation techniques described in this paper are general, and can be used to implement applications in other domains that exhibit similar features.**

## I. Introduction

Hypergeometric series are a large class of infinite series that can represent many mathematical functions and constants. These include, but are not limited to, the majority of the trigonometric functions, special functions such as the the error and Bessel functions, as well as the derivatives and integrals of all other hypergeometric functions.

Evaluating such series to high precision is a very computationally intensive task. And as such, it is a prime target for parallelism in high-performance implementations. However, writing an efficient implementation is a non-trivial task that requires multi-precision arithmetic — itself a non-trivial task. Furthermore, it is complicated by the fact that series summation requires a large amount of memory and inherently exhibits asymmetrical workloads.

The first difficulty in summing up a series is the load-imbalance that is caused by the asymmetrical workload. Terms with higher indices require more work to evaluate than terms with lower indices. Classical parallel implementations of series summation are either unaware of or simply ignore this imbalance and evenly distribute the terms among the processor cores. We provide a technique that precomputes the partition sizes, such that each core gets the same amount of work.

The second difficulty is the amount of memory that is needed to sum a series to high precision. Series summation can consume a very large amount of memory and dynamic memory allocation of large amounts of memory can incur significant overheads. Furthermore, the memory usage can be very erratic; this makes it difficult to place upper-bounds on memory usage thereby increasing the risk of overrunning physical memory. In our implementation we use mathematical techniques to prove bounds on all memory usage. Then we combine such methods to implement our own custom memory allocator that provides low-overhead memory allocation. Our memory allocator is also deterministic with multi-threading which greatly reduces the amount of time spent on debugging.

For larger computations involving higher precisions, two more issues arise which apply both to hypergeometric series as well as multi-precision arithmetic in general. The first issue is insufficient memory. Current high-end desktop computers rarely have more than the 144GB of memory we used. This caps the precision of any high-precision computation to about 20 billion decimal digits. To reach higher precisions, secondary storage is needed — either in the form of disk or a network of multiple computers. The difficulty with secondary storage is speed. Hard drive performance is typically orders of magnitude slower than physical memory for both bandwidth and latency. Our implementation is based on combining many hard drives to increase performance as well as the total storage to allow for very large computations that require many times more memory than is physically available.

The other issue with large computations is reliability. Due to the long durations of computations involving extremely high precision, the probability of hardware failure increases. To address this issue, we implement an elaborate system of error-detection and correction. We also combine checkpointing with out-of-core computation to allow computations to be restarted after catastropic failures such as a power outage or a system crash.

[1]The work for this paper was done at Northwestern University.

| Constant | Previous Record | Previous Record Holder | New Record |
|---|---|---|---|
| $\pi$ | $2.7 \cdot 10^{12}$ | Bellard [2] | $10 \cdot 10^{12}$ |
| $e$ | $0.2 \cdot 10^{12}$ | Kondo & Pagliarulo | $1.0 \cdot 10^{12}$ |
| $\phi$ | $0.3 \cdot 10^{12}$ | Anastasov & Pagliarulo | $1.0 \cdot 10^{12}$ |
| $\sqrt{2}$ | $0.2 \cdot 10^{12}$ | Kondo & Pagliarulo | $1.0 \cdot 10^{12}$ |
| $\zeta(3)$ | $10 \cdot 10^{9}$ | Kondo & Pagliarulo | $100 \cdot 10^{9}$ |
| $\ln(2)$ | $10 \cdot 10^{9}$ | Kondo & Pagliarulo | $100 \cdot 10^{9}$ |
| $\ln(10)$ | $10 \cdot 10^{9}$ | Kondo & Pagliarulo | $31 \cdot 10^{9}$ |
| $G$ | $10 \cdot 10^{9}$ | Kondo & Pagliarulo | $31 \cdot 10^{9}$ |
| $\gamma$ | $10 \cdot 10^{9}$ | Kondo & Pagliarulo | $30 \cdot 10^{9}$ |

Table I
NEW RECORDS FOR DECIMAL DIGITS COMPUTED FOR CONSTANTS

We have implemented these techniques in a high performance framework for summing up hypergeometric sequences. To evaluate our framework, we have developed an application called *y-cruncher* [1]. and used it to compute several major constants to a new world record number of digits. The largest of these computations was for $\pi$, which we computed to 5 trillion digits in 90 days on a single high-end desktop computer. A follow up computation of 10 trillion digits of $\pi$ using the same program and the same computer took 371 days. Table I summarizes these new records.

## II. HYPERGEOMETRIC SERIES AND BINARY SPLITTING

The *hypergeometric series*

$$_pF_q(a_1, a_2, ..., a_p; b_1, b_2, ..., b_q; x)$$

is a series $\sum_{k=0}^{\infty} c_k x^k$ where the ratio between successive coefficient can be expressed as:

$$\frac{c_{k+1}}{c_k} = \frac{(k+a_1)(k+a_2)\cdots(k+a_p)}{(k+b_1)(k+b_2)\cdots(k+b_q)(k+1)}.$$

When the series converges, it defines a *hypergeometric function*.

Many important functions can be expressed partly or entirely as special cases of hypergeometric functions:

- The majority of the trigonometric functions.
- Many special functions such as: Error Function $erf(x)$, Exponential Integral $Ei(x)$, Bessel Functions $J_0(x)$, Airy Functions $Ai(x)$, etc.
- The derivatives and integrals of all the aforementioned, when convergent.

Binary Splitting [3] is an efficient technique for numerically summing up the first $n$ terms of a series. This includes all geometrically convergent hypergeometric functions at rational points as well as certain types of double-summations. In addition to sums, it can also be applied to product functions such as the factorial. The technique can also be used to sum up sequences such as the harmonic series and various divergent asymptotic series to a finite number of terms. Such cases arise in the computation of the Euler-Mascheroni Constant.

Some of the values that can be evaluated using Binary Splitting are:

- Most of the major constants: $e$, $\pi$, $\ln(2)$, $\zeta(3)$, etc.
- All the aforementioned functions at rational points.
- The Harmonic Series $H_x$ for integer x.
- The Gamma Function $\Gamma(x)$ at rational points.

Binary Splitting can be used to numerically sum up $n$ terms of a series in quasi-linear run-time — usually $O(n \log^3 n)$ or less[1]. By comparison, sequential evaluation is $O(n^2)$ and Horner's Method is $O(n^2 \log n)$. For geometrically convergent series (where the series converges to $N$ digits after $\Theta(n)$ terms), these complexities map directly to the run-times needed to evaluate a series to N digits. And given the large sizes of N that we are interested in, quasi-linear run-time is necessary to make the task at all feasible.

We will present the Binary Splitting algorithm using two examples. The first example is the factorial function. Although the factorial is a product rather than a summation, it is a simple example that covers the basics of Binary Splitting. To implement the factorial using Binary Splitting, we use the following recursion:

$$P(a, a+1) = a+1$$

$$P(a, b) = P(a, m)P(m, b)$$

for $a < m < b$

Then

$$P(a, b) = \prod_{k=a+1}^{b} k$$

and

$$n! = P(0, n).$$

The recursion can be used to split a product of $n$ consecutive integers $a(a+1)\cdots(a+n-1)$ into two subproducts of $\approx n/2$ consecutive integres. The method is applied recursively to the sub-products until one is left with a single integer.. This method allows $n!$ to be computed in $O(n \log^3 n)$ time, rather than the $O(n^2 \log^2 n)$ needed for a sequential evaluation, since most products involve smaller numbers.

The second example is of computing $e$ using its Taylor series expansion, which is a hypergeometric series:

$$e = \sum_{k=0}^{\infty} \frac{1}{k!}$$

To get $N$ digits of e, $n = \Theta(\frac{N}{\log(N)})$ terms are needed. If terms are summed one at a time, then it takes $O(\frac{N^2}{\log(N)})$ time to compute $N$ digits of $e$ as each term takes on average $O(N)$ time to add.

[1]We assume multiplication to be $\Theta(n \log n)$.

Below is a two-variable Binary Splitting recursion to compute $e$:

$$\sum_{k=0}^{n} \frac{1}{k!} = 1 + \frac{1}{n!} \sum_{i=1}^{n} \prod_{j=i+1}^{n} j = 1 + \frac{P(0,n-1)}{Q(0,n-1)},$$

where

$$P(a,b) = \sum_{k=a+1}^{b} \prod_{j=k+1}^{b} j,$$

and

$$Q(a,b) = \prod_{k=a+1}^{b} k.$$

$P()$ and $Q()$ are computed using the recursions

$$P(a,a+1) = 1,$$

$$Q(a,a+1) = a+1$$

$$P(a,b) = P(a,m)Q(m,b) + P(m,b)$$

and

$$Q(a,b) = Q(a.m)Q(m,b).$$

Using recursion, $P(0,n)$ and $Q(0,n)$ can be computed in time $O(n\log(n)^3)$. So the total complexity for obtaining $N$ digits of e is $O(N\log(N)^2)$.

This approach generalizes to more complicated series, but at the cost of more variables. Most series require three variables – including that of the popular formulas for $\pi$. The minimum number of Binary Splitting variables needed to compute various constants are shown in Table II.

For the computation of $\pi$, we used the following formula by the Chudnovsky brothers [4]:

$$\frac{1}{\pi} = \frac{\sqrt{10005}}{4270934400} \sum_{k=0}^{\infty} \frac{(6k)!(13591409 + 545140134k)}{(-1)^k (k!)^3 (3k)! 640320^{3k}}$$

This series converges very fast, with about 14 additional decimal digits per term. It was implemented using a 3-variable binary splitting recursion.

## III. Large Number Arithmetic Overview

The immediate consequence of using the Binary Splitting method is that arithmetic with large numbers is needed. The numbers returned by the recursive calls double in size at each level of recursion. Soon, they no longer fit into a single word.

We handle large number arithmetic much the same way others do. Our code expresses large numbers as arrays of 32-bit integers. Arithmetic such as addition, subtraction, and multiplication are done the same way as grade school methods but using base $2^{32}$ instead of base 10. We did not use 64-bit integers because most compilers lack support for the 128-bit integer type that would be needed to handle carry propagation. Other implementations do this by using inline assembly to access the x86-64 carry flags.

We use the long multiplication algorithm, which runs in $\Theta(N^2)$ time, for short products. For mid-range products, we use the Karatsuba algorithm [5] with $\Theta(N^{\log_2 3})$ run-time. And for the largest products, we use a variety of methods based on Fast Fourier Transforms which run in approximately $\Theta(N\log N)$ time [6].

Division and square roots are computed using Newton's Method. And finally, radix conversions are done using Scaled Remainder Trees [7].

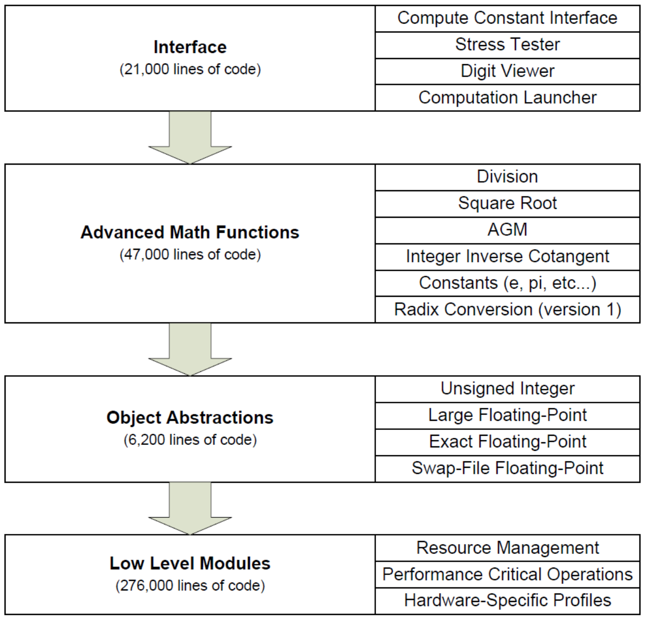


Figure 1. Software Design Layout for y-cruncher

## IV. Implementation

Figure 1 shows the design of y-cruncher. There are 4 primary layers of abstraction. Each layer is only dependent on the layers below it.

The top layer is the interface layer, which consists of all the user-visible features in y-cruncher. It also contains a dispatcher that invokes lower level functions to compute constants. The next layer is the math layer. This layer contains the implementations of all the constants as well as all advanced math functions such as division and square roots. The third is the object layer that implements the large number objects that are used by the math layer. Finally, the lowest layer is the modules layer, which implements resource managements, performance critical math functions, and hardware-specific optimizations.

Of the implementation aspects that we will discuss in depth in this work, the interfaces for multi-threading, memory management, and disk I/O are implemented in the modules layer, but they are used by all the layers. Binary

| Constant/Function | Formula/Algorithm | Minimum known Binary Splitting recursion variables |
|---|---|---|
| $n!$ | | 1 |
| $e$ | Taylor Series | 2 |
| $\pi$ | Chudnovsky/Ramanujan | 3 |
| Apery's Constant, $\zeta(3)$ | Amdeberhan-Zeilberger | 3 |
| Catalan's Constant, $G$ | Lupas Formula | 3 |
| Euler-Mascheroni Constant, $\gamma$ | Brent-McMillan Formula | 4 |

Table II
MINIMUM BINARY SPLITTING RECURSION VARIABLES NEEDED FOR VARIOUS CONSTANTS

Splitting is implemented entirely in the math layer and the code is usually specific to the value that it computes. Large multiplication and the FFTs they depend on are implemented in the modules layer and called internally and by the object layer. Most linear operations such as addition and subtraction are also implemented in the modules layer and called both internally and by the object layer.

## A. Multi-threading

In our implementation, the vast majority of the compute time is spent in the Binary Splitting algorithm for series summation and/or an FFT of some sort. Both of these are recursive divide-and-conquer algorithms. As a result, they are easily parallelized using a recursive thread fork/join programming model.

*1) Binary Splitting of Series:* The Binary Splitting algorithm offers several opportunities for parallelism. It is a divide-and-conquer algorithm that splits into completely independent tasks, and each task can therefore be run in parallel in separate threads. Each call to the binary splitting procedure is given a thread count. The two recursive calls are then run in parallel — each with half the thread count. The merge of the two halves are done using the full thread count. Since the merge consists mostly of large multiplications, they can utilize the multi-threaded FFT algorithms. Figure 2 shows the pseudo code for a Binary Splitting recursion with multi-threading.

New tasks are created recursively until the thread count reduces to 1 or when the size of the calls drops below a certain threshold. Computing sub-series in parallel increases memory usage, as each parallel task needs it own temporary memory. Therefore, the recursive thread-spawning is started several levels deep, where the tasks are smaller and less temporary memory is needed. The higher levels are all done sequentially, and the parallelism is in the multiplications rather than in the recursive calls.

The binary divide-and-conquer nature of binary splitting is the sole reason why y-cruncher only supports a power-of-two number of threads. For systems with a non-power-of-two hardware threads, the best results are usually achieved by rounding up to the next power-of-two and relying on the operating system to deal with the scheduling.

```
Vector<BigNum> bs_threaded(start, end, threads){
  // No more threading.
  if (threads == 1 || end - start < threshold)
    return bs_sequential(start, end);

  // Middle Index
  mid = (start + end) / 2;

  // Split the threads
  half_threads = threads / 2;

  // Run these two calls in parallel.
  bottom = bs_threaded(start, mid, half_threads);
  top    = bs_threaded(mid   , end, half_threads);

  // Merge the halves using all the threads.
  return merge(bottom, top, threads);
}
```

Figure 2. Pseudo code for the naïve divide-and-conquer multi-threading

*Load Balancing:* The size of the recursive tasks produced by the Binary Splitting algorithm for a series such as the one in the Chudnovsky Formula are inherently uneven. This is because the terms in the series increase in size as the indices increase; the tasks that are created to compute the later parts of the series are therefore larger than the tasks that compute the earlier parts. As a result, the straight forward implementation of Binary Splitting often leads to large load imbalance.

Figure 3 illustrates this imbalance when computing a factorial using binary splitting. To compute 100! one could choose 50 as the midpoint. First compute $(1 \cdot 2 \cdot 3 \cdot ... \cdot 50)$ and $(51 \cdot 52 \cdot 53 \cdot ... \cdot 100)$, then multiply the result together. But $(1 \cdot 2 \cdot 3 \cdot ... \cdot 50)$ has 215 digits while $(51 \cdot 52 \cdot 53 \cdot ... \cdot 100)$ has 311 digits; the first product can be computed much faster than the second.

Multiplication dominates the running time and has $\Theta(N \log N)$ complexity. Since multiplication is super-linear, performing a multiplication where the operands are roughly equal is faster than computing the same product by multiplying two unequal operands. A simple inductive argument shows that the total amount of work in multiplications is minimized by always splitting into equal size operands. Furthermore, this also reduces memory usage. This will be discussed in detail later.

There are no algorithmic load-balancing problems with the FFT algorithms since all computations are symmetrical.

In our factorial example we choose $58$ as the mid-point. So we would compute $(1 \cdot 2 \cdot 3 \cdot ... \cdot 58)$ and $(59 \cdot 52 \cdot 53 \cdot ... \cdot 100)$ — which are 260 and 265 bits long respectively.

To determine the correct split, we implement `size()` functions that accurately estimate the sizes of each variable after summing up a certain range of terms. Then we use binary search to find the correct split. Since this can be expensive, the method is only used above a threshold. Below this threshold, we revert to selecting the midpoint as the splitter. Implementing these `size()` functions is non-trivial, and to make it efficient it is usually done with asymptotic approximations:

- Exponentials are sized by multiplying the exponent by the logarithm of the base. This logarithm is expensive, but can be precomputed and cached.
- Factorials are sized using first order Stirling's Formula. The logarithm here is unavoidable.
- Products involving polynomials sometimes require multiple functions to approximate different regions of the domain. (This is not necessary for $\pi$, or any of the other constants that y-cruncher can compute. However, it may occur in the binary splitting implementations of other series.)

*Over-decomposition:* Using the load balancing methods described above, we usually achieve greater than $90\%$ load balance efficiency. The remaining imbalance is mostly caused by operating system overheads. To further increase the efficiency, we use over-decomposition to increase CPU utilization. In certain places, we use two or four times as many software threads as there are hardware threads. In many cases, this achieves more than $99\%$ CPU utilization with negligible overhead from resource contention.

*2) FFT algorithms:* Parallelization of the FFT algorithms follows a similar binary divide-and-conquer approach to binary splitting, but is generalized to handle more than two sub-recursions. The multi-threaded FFTs that we use are all implemented using a modified, fully recursive Bailey's 4-step approach [8].

Bailey's 4-step algorithm breaks a size $N = nm$ FFT into the following steps:

1) Perform $n$ size $m$ FFTs
2) Multiply by twiddle factors
3) Transpose the data
4) Perform $m$ size $n$ FFTs.

In our implementation, the twiddle factors are merged into step 1 and the transpose is completely skipped. So our code only has 2 steps (steps 1 and 4). Furthermore, we skip the bit-reversals since we are only interested in convolution, and therefore do not need the frequency domain variables to be in-order.

Both of these remaining steps consists entirely of independent smaller FFTs. These sub-transforms are parallelized simply by dividing them up among the available threads.

When the number of threads exceeds the number of sub-transforms, it is handled the same way as Binary Splitting. The sub-transforms are done in parallel and each sub-transform is done recursively using the same parallel FFT function. Like Binary Splitting, this recursion continues until the thread limit drops to 1. But unlike Binary Splitting, this recursion rarely gets deep since $n$ and $m$ are usually quite large and most machines do not have that many threads.

## B. Memory Management

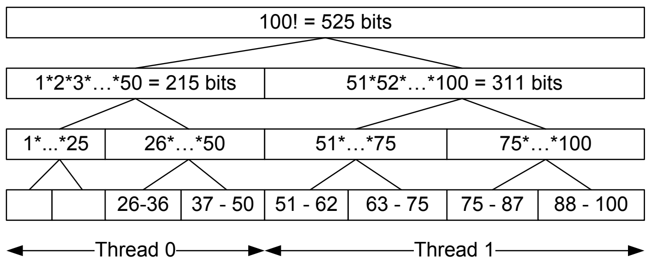
Memory allocation in y-cruncher is managed in a fairly elaborate and unconventional manner. We use a custom memory allocator with tightly managed memory.

Before we go into details, we must first explain the motivation. A standard memory allocators such as `malloc()` or `new` suffers from two defficiencies: Depending on the size of the allocation and the state of the memory heap, a call to `malloc()` or `new` can incur hundreds of cycles of overhead in a multithreaded environemnt. An even larger overhead is due to the memory zeroing that operating systems use to enforce memory security. In applications that are already bound by memory bandwidth, this zeroing operation can severely degrade performance, especially on systems with low memory bandwidth. On one of our machines, it takes about 20 seconds to allocate and commit 60GB of memory. By comparison, a 12 billion x 12 billion digit multiply takes 90 seconds using the same amount of memory. When that 60GB of memory needs to be freshly allocated, the 20 seconds is added to the time resulting in a $22\%$ performance degradation. Based on our experiments, the combined overhead of synchronization and zeroing results in a performance degradation of about 10–40$\%$ for computations of $\pi$ depending on the hardware.

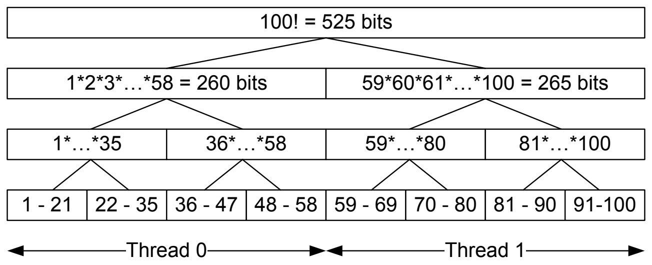
*1) Memory Preallocation:* To mitigate this, we preallocate and commit all the memory that we need at the start of the computation. Then we use our own custom memory allocator. This serves three purposes:

1) It eliminates all OS-related overhead, except for the initial allocation.
2) It allows all memory that is needed to be reserved at the start of the computation. This reduces the chance of a mid-computation failure due to a failed memory allocation.
3) Memory allocation becomes deterministic once the initial block has been allocated.

Although determinism is not required, it is strongly desired as it simplifies debugging. However, there are many drawbacks to this approach. First, building a custom memory allocator that is efficient and thread-safe is non-trivial, and making it deterministic is hard. We will show how we achieve this later. Secondly, we restrict ourselves to a single



(a) The Classic Approach: Selecting the midpoint by index results in an umbalanced recursion tree. Thread 0 has less work than thread 1.



(b) Our Balanced Approach: Selecting the midpoint by operand size produces a more balanced recursion tree. Both threads have the same amount of work.

Figure 3. Binary Splitting recursions. The diagrams are drawn to scale with respect to the size of the numbers.

large contiguous block of memory that is allocated at the start of the computation. This requires that the amount of memory is known ahead of time — after considerations such as fragmentation and multi-threading.

*2) Resource Maps:* The basic structure of the memory allocator that we used is a simplified resource map (see Figure 4). A memory allocator can be initialized from a block of contiguous memory. Incoming allocations are placed at the start of the heap and grow upwards. A table containing a list of active allocations is kept at the end of the heap and grows downwards. This design allows the heap to remain of fixed size while supporting a large number of active allocations.

The allocation policy is simple. Each allocation is placed into the first sufficiently large empty space. When no empty space is large enough for requested allocation, the allocation fails and halts the program. This policy was chosen for its simplicity and because it is a good match for the memory usage patterns of the y-cruncher:

1) Small allocations tend to be persistent. These typically map to large number objects which may have a very long lifetime.
2) Large allocations tend to be temporary. Such large allocations are created by the large multiplication functions. They are allocated at the start of a multiplication and freed at the end.
3) No small, persistent allocations are made before the large allocations are freed.
4) The large allocations may take up to $99\%$ of the combined free space in the heap. This is not by coincidence. The size of the heap is often chosen to be just large enough to fit them. We will discuss this in a later section.

The combination of the simple greedy heuristic allocation scheme and this usage pattern ensures that the *upper* unused portion of the heap is kept contiguous.

Our simple resource map allocator may take, in the worst case, time $\Omega(N^2)$ for making $N$ allocations because each allocation needs to traverse the entire table. This complexity is sub-optimal. However, the logarithmic depth of the re-

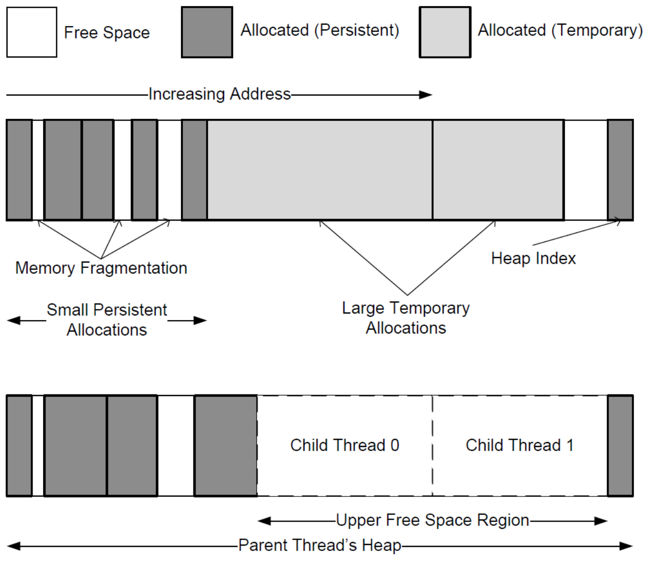


Figure 4. A Typical Memory Heap Layout: Child threads split the upper free space region.

cursions in y-cruncher ensures that $N$ never gets large, and thus keeps the overhead down. This combined with low-level optimizations to reduce memory allocation time is enough to make our memory allocator sufficiently efficient.

*3) Memory fork/join:* As mentioned in an earlier section, nearly all parallelism in y-cruncher fits into a fork/join programming paradigm. In places where the data is structured and pre-allocated, such as in the FFTs, no dynamic memory allocation is needed. However, in places where the data is unordered, such as with the binary splitting recursions, we need a classic memory allocator that works accross all threads. For these cases we introduce `fork()` and `join()` functions to our memory allocator.

Every time the computation is forked into two threads, we split the heap in half. This is done by taking the upper unused portion of the heap and creating two child heaps in its place. Due to our greedy allocation algorithm mentioned earlier, this upper unused portion is (provably) very large.

Each thread is then passed a child heap where all its allocations will go. When the threads join, the two child heaps are also joined. All active allocations in the child heaps are transferred to the parent. Due to the placement of the child heaps, allocations transferred from the upper child will fragment the free space in the parent heap. If needed, explicit memory defragmentation is therefore used. This is discussed in depth below.

Each thread is only allowed to access its own heap, and each heap has at most one thread accessing it. Thus, the threads need no synchronization. A parent heap with child heaps is never accessed until the two child threads join. Since this situation has no race conditions, this method is completely deterministic in correct operation and rarely non-deterministic in incorrect operation. Non-determinism can only occur when a programmer error causes one thread to access another thread's heap. However, this is extremely rare due to the large sizes of the heaps — a simple array-overrun will rarely cross a heap boundary. This method of forking heaps by splitting them in the middle requires the forked threads to have equal memory usage. The load-balancing methods described earlier ensures this. This method can be generalized to unequal memory usage provided that the ratios of the memory usage of the child threads can be predetermined.

*4) Explicit Defragmentation:* Even though the memory allocator is constructed to reduce memory fragmentation for the memory allocation pattern of this application, memory fragmentation still occurs. Therefore defragmentation is used to keep the fragmentation under control. This is done explicitly by the client code through APIs, instead of automatically by the memory allocator. Our memory allocator provides a `defrag()` function. This function takes a pointer, shifts its allocated block down as far as possible, and returns the new pointer. A `defrag()` function is implemented into every large number data structure that resides in memory. This is used in many places. For example, when joining threads, the `defrag()` function is called on all of objects inherited from the upper child.

*5) Precomputing Memory Requirements:* In order to make our memory allocator feasible at all, we need to be able to predetermine how much memory is needed at the start of the computation.

To do this, we implement *space* functions, `space()`, for virtually every function that needs extra memory. These are similar to the size functions used for load balancing. These space functions return an upper bound on the amount of memory that the corresponding function needs. For example, the space function for multiplication will return the amount of memory needed to perform any $N \cdot N$ digit multiplication or less. Similarly, there are space functions for Binary Splitting recursions and Newton iteration algorithms. These space functions typically call one another and extra care is needed to account for multi-threading and memory fragmentation.

To compute the global memory usage, our global space function will walk through the computation process and call the space functions of all sub-steps — thereby computing the maximum memory usage. This approach of propagating space function calls down works well at the global level. However, this approach relies on being able to compute memory bounds at the base cases.

The space function for multiplication is implemented by testing every possible set of parameters up to the requested size.

The space function for binary splitting is done by calling the size functions used for load balancing. Slight modifications are needed to ensure that the value is an upper-bound rather than an approximation.

## C. Out-of-Core

Due to the tremendous size of the computation, it is not possible to perform the entire computation in memory. For our computation of 5 trillion digits of $\pi$, we needed 22TB of storage while only having 144GB of memory[2]. That is 156x more memory than we had! Therefore, the only option was to use disk storage. For that we used an array of 16 x 2TB high-performance 7200RPM hard drives.

*1) Multiple Hard Drives:* Using disk in place of memory is not without its problems. Hard drives are extremely slow, not just for bandwidth, but also latency. Solid State Drives (SSDs) are better, but are also much more expensive and suffer from write wear.

To get around the bandwidth limitation, we use multiple hard drives in a raid 0 configuration. In our code, we abstract a *file pointer* which is like a normal *memory pointer* but to a file instead of memory. This abstraction is implemented using our own software raid 0. The advantage of using our own software raid instead of hardware raid is that we can combine an unlimited number of drives, while hardware raid is typically limited to 4 or 6 drives. Operating system supported raid can break these limits, but they tend to be suboptimal in performance. Manually managing all the drives allows us the freedom of specifying our own raid parameters and to change them on the fly to whatever is optimal for different stages of computation.

With the 16 hard drives that were used in our computation, we achieved about 2GB/s of sequential access bandwidth. This was enough to make large parts of overlapped computation/disk IO completely CPU bound.

*2) Out-of-Core Arithmetic:* At lower layers of abstraction, out-of-core support is implemented mostly by duplicating the in-memory code and modifying it to use file pointers instead of memory pointers. In this sense, disk is treated and used as main memory. RAM is treated as an explicit cache and all transfers between RAM and disk are done manually.

[2]Although we had 144GB of memory available to us, we chose to use only 96GB because it allowed us increase the operating frequency of the memory. 96GB of faster memory proved to be more desirable than 144GB of slower memory.

For simple single-pass operations like addition, our code streams through the operands by loading them from disk into memory in chunks that fit into memory. Once in memory, the in-memory versions of the same function are called to do the actual work. Then the result is flushed back to disk. No overlap is done since all of such single-pass operations are extremely fast.

For multiplication, the 3-pass and 5-pass convolution algorithms are used. The 3-pass algorithm is essentially an optimized implementation of Bailey's 4-step FFT algorithm and is as follows:

- Pass 1 – Non-sequential (strided) disk access
  1) Read from input
  2) Process the input operand
  3) Perform first steps of forward FFT.
- Pass 2 – Sequential disk access
  1) Perform final steps of forward FFT
  2) Perform pointwise multiplications
  3) Perform first steps of inverse FFT.
- Pass 3 – Non-sequential (strided) disk access
  1) Perform final steps of inverse FFT.
  2) Process output.
  3) Perform carry propagation.
  4) Store to destination.

Note the large number of steps that are done in each pass. Although this makes the code extremely complicated, it is necessary to avoid introducing extra passes over disk. Overlapping of computation and IO is done aggressively. In general, the 1st and 3rd passes are severely disk bound while the 2nd pass can be either disk or CPU bound depending on the ratio of disk bandwidth to CPU throughput.

As the product size increases, the disk access of the 1st and 3rd passes of the 3-step algorithm become increasingly less and less sequential – at some point, the run-time becomes dominated by disk seeks instead of disk reads/writes. This is where the 5-step algorithm is used. It is similar to the 3-step algorithm, except that the FFTs are split across 3 passes (again with the central passes merged). The 5-step algorithm has more sequential disk access at the cost of more disk I/Os (in bytes).

We note that our approach to the 3-step algorithm is not optimal when the 2nd pass is CPU bound. A better approach is to re-order the data such that most of the non-sequential disk access is in the 2nd pass. This allows the expensive disk seeks to be overlapped with the computation. However, this approach severely complicates the code as it requires implementing functions that can operate on partially transposed data. This approach is also is less effective for the 5-step algorithm and may even backfire.

*3) Out-of-Core Math and Functions:* At higher layers of abstraction, implement a high-precision floating-point object that resides on disk. It is nearly identical to the floating-point object that resides in memory, but has a file pointer instead of a memory pointer. For each function that operates on a swap object, a dispatcher chooses whether to pull all operands into memory and call the fast in-memory code (if there is enough memory to do so), or directly call the slower disk code.

By introducing a separate swap object, much of the code high level code in y-cruncher needs to be duplicated to operate on swap objects. When operations become small enough to fit into memory (such as lower levels of the Binary Splitting recursions), they automatically fall through to the in-memory implementations. Determining when something can be done in memory is achieved using the same `space()` functions discussed in the previous sections.

Although this code replication increases overall code size, it provides greater implementation flexibility and eases debugging by separating disk code from in-memory code.

*4) Other Notes:* The out-of-core code does not use the same fancy memory management that is used by the in-memory code. Instead, we revert to the classic memory allocation model – but with one difference: instead of a global `malloc()` and `free()`, it is creating and deleting files. However, most of the out-of-core functions will take an extra parameter specifying how much (RAM) memory it is allowed to use.

All out-of-core code is sequential since multi-threading is done at lower levels. Therefore, there are no concurrency issues that lead to messy solutions like memory fork()/join(). For file creation, the zeroing that plagues OS memory allocation is not present in Linux and can be overridden in Windows [9].

## D. Fault Tolerance

Due to the long duration of larger computations, fault-tolerance is needed to help ensure that the computation will finish correctly. This is achieved using a combination of application-specific error-detection, automatic repeat request, and checkpointing.

Faults can be caused by a large number of reasons, but they typically fall into one of the following 4 categories:

- Software Error: A typical programming bug. These are extremely rare at the later stages of code development.
- CPU Computation Error: A hardware error where an execution unit returns an incorrect result.
- CPU Logic Error: A hardware error in the general CPU logic. (such as flow control errors)
- Memory/Storage Error: This applies to memory/cache as well as disk. Similar to CPU errors, they can silently corrupt data. In the case of hard drive failures, it can lead to permanent data loss.

There are generally two types of hardware faults: *visible* and *slient*. *Visible* faults typically result in a segmentation fault or a system crash. (such as a Windows Blue Screen of Death (BSOD)) Nearly all CPU logic errors fall into this category. In most cases, *visible* faults are not a problem because they teriminate the program as soon as it occurs.

*Slient* faults, on the other hand, are not detected by the hardware and do not crash the system. These are very problematic because they do not terminate the program such errors will propagate and poison the computation.

By their very nature, *Visible* are not recoverable since the program is terminated. So we make no attempt to recover from such faults. In general, *visible* faults are rare because they tend to appear only on very unstable hardware. As a result, they are not a problem because unstable hardware will rarely make it far into a long computation. On the other hand, *slient* faults are predominant among stable hardware and are frequent enough to require handling.

Over a course of two years during the development of y-cruncher, we have encountered about 20 cases of hardware faults (excluding hard drive failures). Of these, only 4 were *visible* faults that lead to a crash or a BSOD. The rest were silent and were only detected via a failed redundancy check or an incorrect result. Of the 20 total faults, 7 of them occurred on an overclocked computer. Of those, only 5 could have been caused by overclocking of which 3 were actually confirmed to be due to overclocking.

Of peculiar interest is that our observed ratio of *visible* to *slient* faults is in direct contrast to the results of [10]. In our observations, roughly $80\%$ of faults are *slient* whereas the results of [10] show that the SDC (Silent Data Corruption) rate is less than $5\%$ for most applications, but up to $12\%$ for floating-point intensive applications. Although we are unsure the cause of this discrepancy, we suspect it is due to the differences in testing methodology. [10] uses manual injection to generate faults, whereas our results are from overclocking and long-term data collection of natural faults on commodity hardware.

Since the majority of the faults we encounter are *slient*, we implement our own software for error-detection. This is accomplished in a number of case-specific methods.

*1) Error Detection: Multiplication:* FFT-based algorithms for multiplication can be checked for errors by examining the sizes of the coefficients before performing carryout. If any coefficient is larger than the theoretical maximum, it implies an error. For floating-point FFTs, any coefficient that is not near an integer also implies an error. However, this latter test is fairly expensive in the absence of hardware support so we do not use it.

For all integer multiplies (where the entire product is kept), it possible to use modular hash checks to verify the product. A modular hash check is as follows.

Given an integer product:

$$C = A \times B$$

The following property also holds:

$$C \bmod p = ((A \bmod p) \times (B \bmod p)) \bmod p$$

where p is any natural number.

Should this relation fail, then it implies an error. $p$ is typically chosen to be a word-size prime number.

In y-cruncher we choose $p = 2^{61} - 1$. The choice of $2^{61} - 1$ (a Mersenne prime) is done solely for the reason of efficiency as computing a modulus over this number can be easily done without multiplications or divisions.

This modular hash check for multiplication is actually not used because it is expensive and redundant. FFT-based algorithms are already checked by analyzing the coefficients, and smaller products are covered by the Binary Splitting checks as discussed in the next section.

*2) Error Detection: Binary Splitting:* The lower levels of Binary Splitting recursions involve only integer addition, subtraction, and multiplication. These are all closed under the modular ring and thus can be verified by the modular hash check.

For efficiency, we do not verify the modular hash of every single operation. Instead we compute the hash once each term is generated. This hash (the modulus) is stored with the number and propagated through each operation. For example, when two values are multiplied together, their hashes are also multiplied together and reduced modulo p. This keeps the overhead at a negligible $O(1)$ for any one operation.

Only at higher levels of the recursion are the hashes actually verified against the numbers. This is usually done only once and just before the data is first swapped out to disk. When a hash fails to match the modulus of the number, it implies an error.

*3) Automatic Repeat Request:* When an error is detected, the relevant portion of the computation is repeated. It is repeated until the result is either correct or a certain threshold is reached – at which the error is deemed uncorrectable.

For multiplication, the product is done again. For Binary Splitting, the recursive call that produced the failed checksum is repeated. For a non-convergent Newton's Method iteration, the entire algorithm starts over. Figure 5 shows how a fault in the binary splitting recursion is handled.

Most errors are fixed on the first reattempt. Repeated failures usually indicate data corruption in some outer scope. Our code makes no attempt to recover such errors as the effects are no longer isolated to a small part of the program.

*4) Checkpointing:* In cases where the computation is halted (uncorrectable error, system crash, power outage, etc.), checkpointing is used to avoid needing to start the computation over from scratch.

At various places in a large computation, checkpoints are created. These checkpoints are created in a zero-overhead fashion that only involves renaming a few swap files. There is no data copying. An old checkpoint is not destroyed until the next checkpoint has been successfully made and flushed to disk.

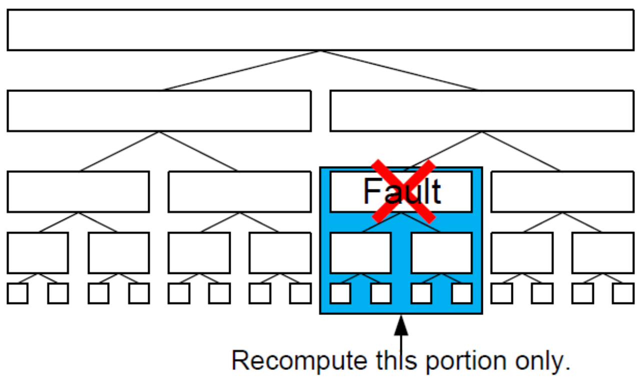


Figure 5. Fault-Tolerance in Binary Splitting: When a fault is detected, only the affected portion of the recursion is recomputed.

A computation can be resumed from a checkpoint provided that the hard drives are intact. To protect against hard drive failure, periodic backups are done manually.

Checkpointing proved crucial in both our computations of 5 trillion and 10 trillion digits of $\pi$. At just 8 days into our computation of 5 trillion digits, our error-detection code detected a silent hardware fault. The error could not be recovered, so we had to roll the computation back to the previous checkpoint.

In our computation of 10 trillion digits of $\pi$, we encountered a total of 9 hard drive failures. With each failure, the computation had to be rolled back to a checkpoint. Although the hard drive failures ultimately extended the total computation time from 190 days to 371 days, it would not have been possible for it to finish at all without checkpointing.

*5) Testing:* Since fault-tolerance code is activated by hardware errors, it cannot be tested the same way normal code is. Instead it was tested using a variety of indirect methods including (but not limited to) :

- Injecting errors via modifying the source code.
- Triggering actual hardware faults by:
  - Overclocking the hardware until instability.
  - Using failing hard drives that produce I/O errors.
  - Physically unplugging hard drives while in use.

## V. Evaluation

### A. Methodology

We evaluate our software based on three categories: compute time, memory usage, and reliability.

For compute time, we benchmarked our code against three other programs: GMP-Chudnovsky [11], Parallel GMP-Chudnovsky [12] and TachusPi [13]. These tests were done on two different machines (Table III). The first machine is a simple desktop with 4 physical cores and 2-way SMT for 8 threads. The second machine is a higher-end workstation with 8 physical cores and a large amount of memory. For

| | |
|---|---|
| Machine 1 | Intel Core i7 920 @ 3.5 GHz (overclock)<br>12 GB DDR3 @ 1333 MHz<br>Windows 7 Ultimate + Ubuntu Linux 10 |
| Machine 2 | 2 x Intel Xeon X5482 @ 3.2 GHz<br>64 GB DDR2 FB-DIMM @ 800 MHz<br>Windows 7 Ultimate + Ubuntu Linux 10 |

Table III
Benchmark Machines

consistancy, both machines dual-boot the same versions of Windows and Linux.

These benchmarks were carried by having the programs compute $\pi$ to a set number of digits. Then we recorded the total time needed to compute $\pi$ as well as the time needed for a full-sized multiplication. This was done for a whole range of sizes from 1 million digits to 10 billion digits and using 1 and 8 threads.

For sizes below 1 million digits, the benchmarks are too fast to obtain accurate timings. For y-cruncher and GMP, we can simply loop the benchmarks since we have access to the source code. But TachusPi is closed source, so that is not possible. For sizes above 10 billion digits, we simply don't have the memory to run them. Furthermore, our Core i7 machine can only go up to 2.5 billion digits. GMP needs significantly more memory than both our code and TachusPi, so it was not able to reach the maximum sizes that we were targeting.

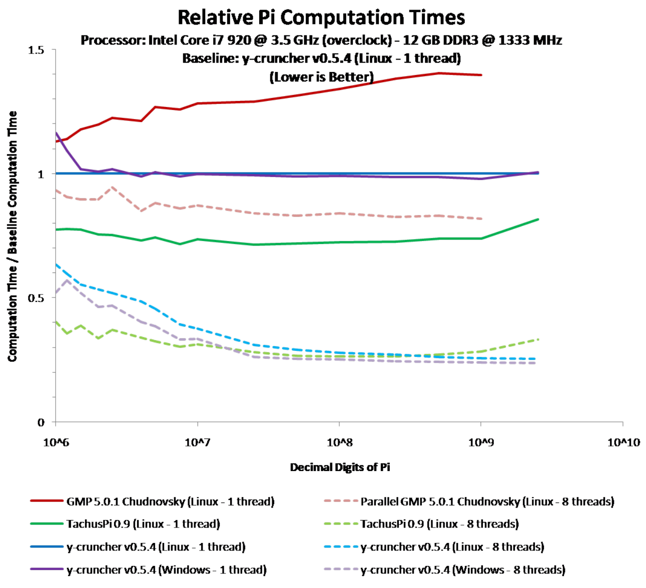
### B. Results

*1) Performance:* Figure 6 shows the relative run-times needed to compute $\pi$ to various numbers of digits. Solid lines are single-threaded. Dashed lines are multi-threaded. The baseline is our y-cruncher, running in Linux on 1 thread.

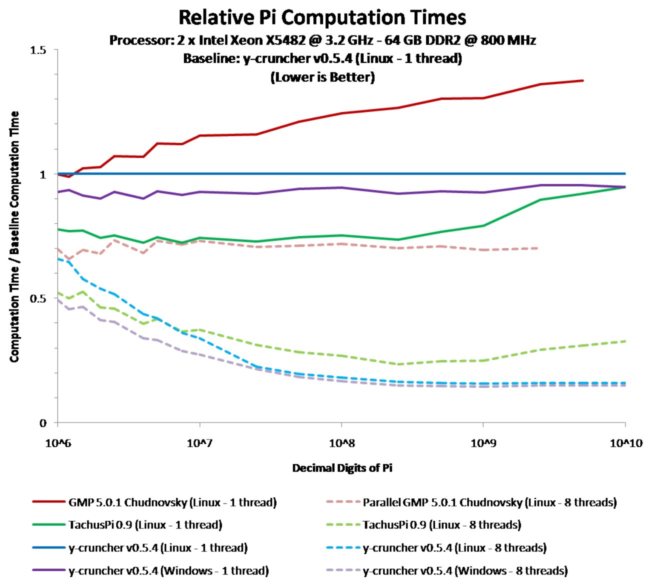
Our code achieves significant gains from multi-threading. For the Core i7, there are only 4 physical cores, but our code manages to achieve more than 4x speedup at the larger sizes due to hyperthreading. On the Xeon, which has 8 physical cores, our code gets about 6.5x speedup at the larger sizes. By comparison, neither GMP nor TachusPi scales as well. GMP suffers because it has no support for multi-threaded large multiplication.

The other observation is that TachusPi is faster than our code by about 20% for the single-threaded computations. This is due to the fact that our implementation omits some mathematical optimizations that would provide significant speedup. Specifically, our code does not do GCD Factorization [14], Middle Product [15], and FFT reuse. TachusPi is known to do at least the first two. (The author has not disclosed whether FFT reuse is also used.)

GCD factorization was omitted because we lacked an integer division function. Middle Product was not even known to us at the time we implemented y-cruncher. (TachusPi is newer than our y-cruncher.) And finally, FFT reuse was

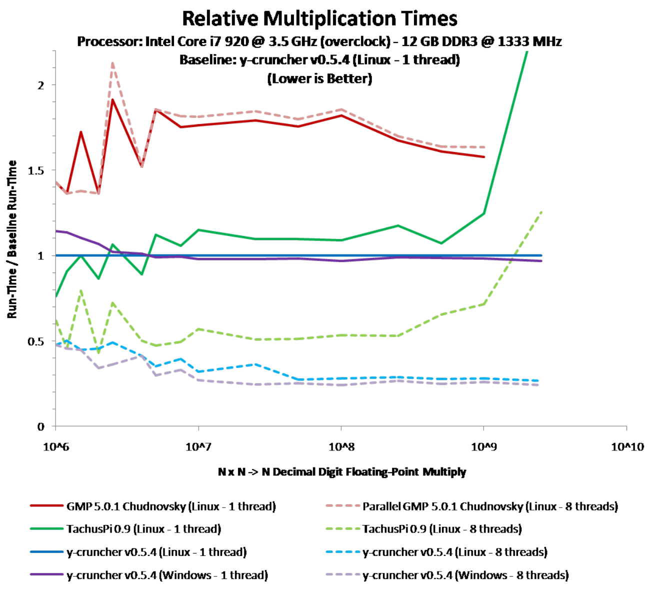


(a) Pi Computation Times: Intel Core i7 920 @ 3.5 GHz (overclock)

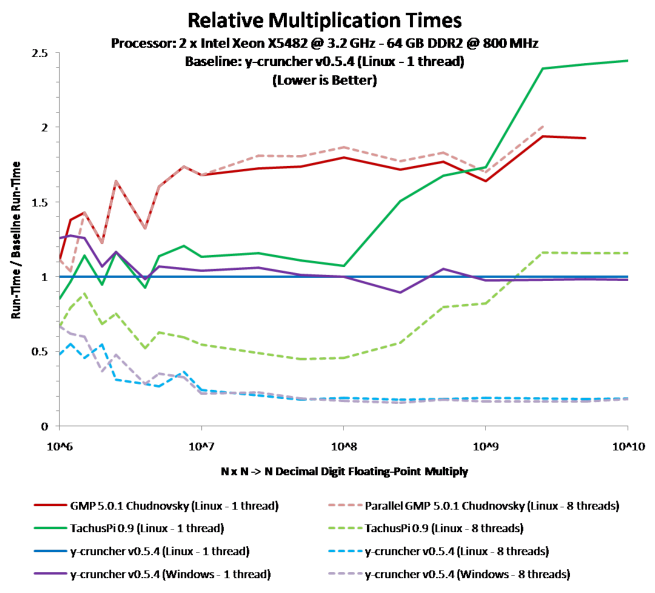


(b) Pi Computation Times: 2 x Intel Xeon X5482 @ 3.2 GHz

Figure 6. Pi computation times of various programs on different machines. Note that the GMP graphs stop short due to lack of memory.



(a) Multiplication Times: Intel Core i7 920 @ 3.5 GHz (overclock)



(b) Multiplication Times: 2 x Intel Xeon X5482 @ 3.2 GHz

Figure 7. Large multiplication times of various programs on different machines. Note that the GMP graphs stop short due to lack of memory.

omitted because it provides only marginal speedup at the cost of needing to break through several layers of object abstraction. Nevertheless, for sufficiently large computations, our code outperforms all other programs that we tested due mostly in part to the scalability that we achieve.

Figure 7 shows the relative run-times needed to perform a multiplication. This is important because multiplication is typically the dominant run-time factor in large number computations. Addition and subtraction is linear, and all common non-linear functions reduce to multiplication. The Binary Splitting algorithm is composed entirely of multiplications with an occasional addition or subtraction. As a result, multi-precision libraries are typically evaluated by the speed of their large multiplication.

The multiplication graphs show a slightly different situation from the $\pi$ graphs. In this case, our code holds up well against TachusPi for the single-threaded case while maintaining its strength in multi-core scalability.

Another observation is that TachusPi significantly slows down at about 1 billion digits. This slow-down appears to persist for larger sizes. This is possibly the result of an algorithm switch caused by either a poorly tuned threshold or a space-time tradeoff to reduce memory consumption. But we cannot be sure since we have little information on the internals of TachusPi.

Note we did not benchmark large computations that require disk due to the resource commitment that would be needed. However Shigeru Kondo [16] has noted that y-cruncher runs about 30% faster than TachusPi for a 100 billion digit computation of $\pi$ on a Core i7 with 12 GB of ram and 8 hard drives.

*2) Memory:* Aside from compute time, memory is actually a big issue for large computations. The problem with computing $\pi$ and other constants is that the total amount of memory that is needed is uncertain. It is rarely known ahead of time how much memory a computation needs. This makes it more difficult to construct hardware configurations for the purpose of running such large computations as underestimating memory usage will doom a computation from the start.

Prior to y-cruncher, PiFast [17] is the only known program that is able to predetermine memory and disk requirements. QuickPi [18] allows the user to specify a memory limit, but does not specify how much disk is needed. TachusPi allows the user to specify a memory limit and provides an upper bounds on disk usage. However, Shigeru Kondo [16] has noted that TachusPi has a tendency to significantly overrun the memory limit and thrash virtual memory.

Thanks to the plethora of `space()` functions used by various components in our code, y-cruncher is able to precisely determine (usually to within 1%) the memory and disk requirements. This makes it possible to configure "just enough" hardware for a target size without risk of underestimating the requirements.

There are programs (such as SuperPi [19]) that provide tight predetermined memory requirements. However, these programs only support a small number fixed sizes - their memory requirements are determined by actually running all the fixed sizes and hard-coding each of their memory requirements into the program.

In terms of the amount of memory/disk that is needed for a particular computation, our code is second behind TachusPi. Although we had some focus on reducing total memory usage, we did not do optimizations that interferred with performance or that increased the complexity of the code. We also did not do memory optimizations that required more than a trivial amount of effort.

At the time of our computation of 5 trillion digits of $\pi$, the price of hard drives was cheap compared to that of a high-end computer. Therefore our priorities were to maximize performance and code reliability rather than to reduce total memory usage as it was completely reasonable to acquire as many hard drives as we needed.

The main drawback of using many hard drives is a shortened MTTF of the array. Although none of the 16 hard drives failed during our computation 5 trillion digits of $\pi$, our follow-up computation of 10 trillion digits using 24 hard drives had a total of 9 hard drive failures.

A possible solution for future computations involving many hard drives is to use fault-tolerant raid.

*3) Reliability:* Fault-tolerance is a difficult feature to evaluate due to the difficulty of producing consistent faults in the absence of source code. And as such, we made no attempt to test the fault-tolerance of the programs that we benchmarked. So our analysis is mainly based on past performance and history.

PiFast and SuperPi are known to detect errors, but not correct them. On the other hand, PiFast offers checkpointing for swap computations. TachusPi also provides checkpointing and has also been observed to detect errors. It is not known if TachusPi is capable of correcting errors and it is not known to us if QuickPi offers any sort of fault-tolerance. GMP is open-sourced and does not appear to have any fault-tolerance capability.

We should note that there is a distinction between basic "sanity checks" versus "active" error-checking. In our definition, a simple test for convergence of a Newton's iteration would be considered a "sanity check" whereas a modular hash would be considered an "active" error-check.

We believe that most if not all of such programs implement sanity checks. But we expect active error-detection to be uncommon due to the performance costs that come with it. The documentation of TachusPi confirms that it does use active error-checking. But in the absence of source code, official documentation, or direct mention from the authors, we do not know if any of the other programs have active error-detection – let alone error-correction. It is possible that most of the error-detection capability that we found in the programs we tested are the result of basic sanity checks rather than active error-detection.

In our case, we have extensively tested the fault-tolerance of y-cruncher by artificially generating faults. However, it has also proven to be extremely robust and effective when faced with actual (legitimate) faults.

Generating faults by overclocking is arguably a valid way of producing legitimate faults. But there have been many instances of actual (unintended) hardware faults occurring during development of y-cruncher – not all of which were related to overclocking. Some involved overheating leading to CPU/memory errors and one involved a failing motherboard that failed to supply enough power to the processor. The Xeon machine that we used for our benchmarks has had its fair share of failures and it isn't even overclocked.

Aside from CPU/memory faults, there were also countless cases of hard drive failures, CRC errors, and bus transfer errors. In nearly all cases where the fault did not crash the program or the system, our code was able to correctly handle the fault as intended. In one case, y-cruncher was able to detect and recover from 3 independent hardware errors in a single computation lasting less than 10 seconds! The computation was also correct. (In this case, the hardware was an Intel Core i7 2600K overclocked to 5.2 GHz with insufficient voltage to maintain any degree of stability.)

To date there have only been three cases where a computation has finished with an incorrect value without triggering any warnings or errors. All three have all been traced to hard drive I/O bus errors caused by improperly over-clocked memory controllers. Since bus transfer errors are invisible to the system and our code implements no error-detection for I/O, such errors will pass through undetected if not covered by a higher level of error-detection. (Bus transfer errors do get logged in hard drive S.M.A.R.T. chips. This is how we were able to trace the cause of these three failed cases.)

### C. Discussion: Micro-Optimizations

We have discussed mostly high-level optimizations, but we should mention that our code benefits heavily from low-level micro-optimizations as well. The primary low-level optimizations that we use are loop-unrolling and function inlining. Most of the FFTs that we implement also make heavy use of SIMD instructions.

Recursive functions are optimized by aggressive manual inlining of the end-points. This allows us to maintain a recursive structure without incurring the overhead of recursion. Binary Splitting recursions are implemented on top of the object layer. However, we manage to optimize these by breaking below the object layer and calling the low-level math functions directly. In our implementation of $e$ using the recursion given earlier, we stop the recursion at $b-a<10$. The rest is implemented directly using one-to-one hardware instructions. For this particular example, we gain a 10% overall speedup over terminating the recursion at $b-a=1$.

Loop-unrolling is the other optimization that we use extensively. Typically, low-level loops are unrolled enough to fill up the registers without spills. Generally, this is very effective and can easily speed up loops by 1.5 - 2x. In nearly all cases, we can do better than the compilers as they tend to unroll the wrong loops and are too conservative with the right ones. The types of loops that gain the most from loop-unrolling are loops involving SIMD instructions. SIMD loops tend to be large, so compilers refuse to unroll them for fear of code-bloat. Even worse, SIMD instructions tend to have long latencies and such loops tend to be too large to fit into processor instruction re-order windows. So the only way is manual unrolling - which we find can easily give a 2 - 3x speedup. (not counting the speedup gained from SIMD)

For purposes of maintaining portability, we fall short of using hand-written assembly. However, in current development versions of y-cruncher we have started using snippets of inline assembly to access otherwise inaccessible instructions such as add-with-carry.

Overall, cache-level optimizations are done sparsely. Much of the code is recursive and cache-oblivious, so there is no urgent need for the code to be fully cache-aware. Cache-optimizations are generally limited to the radix selection in of the FFT algorithms.

In current development versions of y-cruncher, we take micro-optimization to a new extreme. Functions and loops are sometimes inlined or unrolled to the sizes of the instruction cache or the decoded uop cache. Much of the new code is now cache-aware and makes heavy use of explicit prefetching and data streaming. The results have been quite impressive as some of the new code can achieve floating point performance comparable to that of dense matrix computations.

The downside of extreme micro-optimization is the added stress that is brought on to the hardware. For system builders as well as the overclocking community, this can be very useful for stress and stability testing of new hardware. But for long running computations, increased stress can instigate hardware errors and put additional pressure on the fault-tolerance capability.

## VI. Conclusion

This paper has described a highly efficient implementation of the binary splitting algorithm for the evaluation of hypergeometric series, and described its use for the computations of 5 and 10 trillion digits of Pi. The paper identifies multiple issues that need to be handled in order to achieve the required level of performance: careful algorithm design, careful management of load balancing, custom memory management and micro-optimizations such as loop unrolling and inlining. The paper illustrates, once again, that the development of very efficient codes requires a concerned, coordinated effort at the algorithm level, as well as at the macro and micro implementation level; e.g., bypassing standard optimizations provided by a compiler and doing manual loop-unrolling and inlining; and bypassing standard system services and using a custom memory manager. Regular programming environments have to balance performance with ease of use and generality, and are not sufficient to create highly performing codes as described in this paper. It is interesting to speculate what would be a performance-oriented programming environment that would facilitate the crafting of code such as described here. Finally, the paper illustrates the use of algorithmic fault tolerance to overcome the high failure rate and, in particular, the high soft-error rate. Such techniques will be increasingly needed to support large computations as the reliability of components decreases with increased miniaturization, and the number of components increases.

## Acknowledgments

We would like to thank Shigeru Kondo for running the actual computation of Pi and for the numerous times he has helped in testing y-cruncher. Furthermore, we would like to thank Fredrik Kjolstad and Marc Snir for helping us write and review this paper.

## References

[1] A. Yee, "y-cruncher - multi-threaded pi program," http://www.numberworld.org/y-cruncher/, 2010.

[2] F. Bellard, "Computation of 2700 billion decimal digits of pi using a desktop computer," http://bellard.org/pi/pi2700e9/, 2010.

[3] B. Haible and T. Papanikolaou, "Fast multiprecision evaluation of series of rational numbers," TH Darmstadt, Tech. Rep. TI-97-7.binsplit, 1997.

[4] D. V. Chudnosky and G. V. Chudnovsky, "Approximations and complex multiplication according to ramanujan." *Boston, MA: Academic Press*, pp. 375–472, 1987.

[5] A. Karatsuba and Y. Ofman, "Multiplication of many-digital numbers by automatic computers," *Doklady Akad. Nauk*, pp. 293–294, 1963.

[6] A. Schnhage and V. Strassen, "Schennelle multiplikation grosser zahlen," *Computing vol. 7*, pp. 281–292, 1971.

[7] D. J. Bernstein, "Fast arithmetic," http://cr.yp.to/arith.html, 2004.

[8] D. H. Bailey, "FFTs in external or hierarchical memory," in *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 1990.

[9] Microsoft, "SetFileValidData function," http://msdn.microsoft.com/en-us/library/aa365544%28v=vs.85%29.aspx, 2011.

[10] M.-L. Li, P. Ramachandran, S. K. Shaoo., S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design," in *13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[11] H. Xue, "gmp-chudnovsky.c," http://gmplib.org/pi-with-gmp.html, 2011.

[12] H. Xue and D. Carver, "Parallel gmp-chudnovsky using openmp with factorization," http://gmplib.org/list-archives/gmp-discuss/2008-November/003444.html, 2011.

[13] F. Bellard, "Tachuspi," http://bellard.org/pi/pi2700e9/tpi.html, 2011.

[14] H. Xue, "(no subject)," http://groups.yahoo.com/group/pi-hacks/message/617, 2002.

[15] G. Hanrot, M. Quercia, and P. Zimmermann, "The middle product algorithm i. speeding up the division and square root of power series," 2002.

[16] S. Kondo, personal communication, 2010.

[17] X. Gourdon, "Pifast: the fastest windows program to compute pi," http://numbers.computation.free.fr/Constants/PiProgram/pifast.html, 2011.

[18] S. Lyster, "The fastest pi programs that will run on your pc," http://members.shaw.ca/francislyster/pi/chart.html, 2011.

[19] Y. Kanada, "Kanada laboratory home page," http://www.super-computing.org/, 2011.