Parallel Collision Search

Making money the old-fashioned way—
the NOW as a cash cow

David Wagner and Ian Goldberg
University of California, Berkeley
{daw,iang}@cs.berkeley.edu
CS 267 Final Project

May 23, 1997

Abstract

Parallel collision search is the hard problem underlying a number of interesting applications in electronic commerce, graph theory, and codebreaking. Several algorithms, including one by Rivest and Shamir and another by van Oorschot and Wiener, have been proposed to solve this problem. In this paper, we present enhancements to these algorithms to achieve practical efficiency and near-perfect scalability.

1 Introduction

An electronic payment system provides the ability for banks to produce "tokens" of data, each of which represents a small amount of money (typically on the order of cents or fractions of cents). These tokens can be sold to customers, who can spend them at shops over the Internet. The shops then redeem these tokens at the bank to receive their payment. One key factor in any micropayment system is how to prevent customers from "minting" their own tokens.

Whereas a number of electronic payment systems [1, 3, 9] solve this problem by using (somewhat heavyweight) digital signatures on tokens, the MicroMint electronic payment system proposed by Rivest and Shamir in [8] utilizes the *collision search* problem to allow the bank to mint tokens while making it infeasible for customers to do the same.

A high-performance solution to the collision search problem is extremely important because, instead of relying on a "trapdoor" function, in which some kind of data known only to the bank makes the problem easier to solve, this problem requires that the bank have more time and more computing power that it can devote to solving this problem than would-be forgers do. No single machine can provide the performance these applications demand. Because of the need for speed, parallel architectures are the only way we can hope to amass enough computing power to solve useful instances of the collision search problem. We therefore require a solution to this problem that performs well in parallel environments, as well as scaling well in both absolute and incremental fashions. But parallel computing brings with it new challenges, with innovations required in both algorithms and implementations. Therefore, this paper focuses on parallel collision search: challenges and solutions.

The goals of this project were twofold. First, we aimed to study previous and recent work on collision-finding algorithms. Specifically, we were interested in implementing and evaluating real-world performance on the NOW, attempting to gain some insight into likely bottlenecks with this practical experience.

Second, we hoped to extend that experience to handle a generalized ("k-way") collision search problem, which is used by MicroMint. Most of the previous work did not address this problem, but we hoped to be able to extend the algorithms to efficiently solve these problems too, with performance requirements motivated from MicroMint.

Collision search also has a number of other applications; for example, it is closely related to the graph-theoretic problem of finding cycles on a large sparse directed graph, as well as a number of other applications throughout codebreaking, a field with ever-increasing demands for high performance.

Background on the collision search problem is given in Section 2. Section 3 outlines previous work on this problem. Section 4 describes our enhancements to these previous algorithms. We describe our implementation of our algorithm and performance measurements in Sections 5 and 6, and Section 7 concludes.

2 Background

2.1 Problem statement and Mathematical Model

The collision search problem is easy to state: given a hash function h, we are to find a pair of inputs $x \neq y$ with $h(x) = h(y)$. In some applications, we may wish to find a great number of collisions efficiently. The hash function is viewed as a black box; we assume that h has no mathematical structure we can conveniently analyze.

Collision search is tied to the birthday paradox, which (in its best-known formulation) states that a party of 23 people will likely contain a pair of people who were born on the same day of the year. In general, on a planet with m days in a year, one needs about $\sqrt{m}$ people to find such a pair. Viewed in terms of collision search, the birthday paradox says that we will have a good chance of finding a collision in an n-bit hash function after hashing about $2^{n/2} = \sqrt{2^n}$ different values.

The collision search problem can also be generalized to the problem of finding k-way collisions: we are to find a set of k different inputs $x_1, \ldots, x_k$ with $h(x_1) = \ldots = h(x_k)$.

The mathematics can be tightened up a bit: we formally model $h : \{0,1\}^* \to \{0,1\}^n$ as a random function with n-bit outputs; therefore if we apply the hash function to any j different inputs, we will see j independent outputs, each uniformly distributed over $\{0,1\}^n$. The probability that no collision is observed is $(1-1/2^n)(1-2/2^n)\cdots(1-(j-1)/2^n) \approx e^{-j^2/2^{n+1}}$. We see that after hashing $1.177 \cdot 2^{n/2}$ values, the probability of observing a collision is about $1/2$. Furthermore, the expected number of values needed before observing a collision is $\sqrt{\frac{\pi}{2} \cdot 2^n} \approx 1.253 \cdot 2^{n/2}$ [10]. We see that the rough estimates given above are very close to the truth.

The analysis can be extended to estimate the case where we are searching for a number of collisions. After hashing j values for $j \gg 2^{n/2}$, the expected number of collisions found is approximately $j^2/2^{n+1}$. In other words, if one is searching for m collisions, a set of $\sqrt{m}\, 2^{n/2}$ hashed values will suffice. This means that later collisions start coming faster and faster than the first one; if many collisions are wanted, it would be a very bad idea to restart the search after each new collision. In other words, finding many collisions is less costly, per collision, than finding just a few. The problem exhibits a "soft" threshold at roughly $2^{n/2}$ hash evaluations: significantly before that point, collisions are quite unlikely, but a ways after the threshold collisions come increasingly quickly. In the more general case of k-way collisions, about $2^{(k-1)n/k}$ values are needed to find the first k-way collision, and $m^{1/k} 2^{(k-1)n/k}$ hashed values suffice to find about m k-way collisions. In particular, k-way collision search exhibits an even sharper threshold for $k > 2$: one has very little chance of observing even one collision much before the threshold, but after the threshold, collisions abound. This interesting behavior is critical to the design of the MicroMint electronic payment system.

2.2 MicroMint

The MicroMint electronic payment system relies heavily on properties of the collision search problem, and as such, motivates our exploration of parallel collision search. This is a three-party payment system: a broker (typically a bank) generates and redeems coins, customers purchase coins from the broker, and vendors offer electronic goods in exchange for such coins. A coin is nothing more than a bit string with a special structure: in particular, a MicroMint coin is just a k-way collision $(x_1, \ldots, x_k)$; the authors suggest $k = 4$ as a convenient choice.

This leads us to an interesting conundrum. Anyone who can generate k-way collisions efficiently can defraud the system and obtain goods for free; thus, all of the security from forgery lies in the difficulty of collision search. At the same time, the broker mints coins en masse by finding a great many collisions on high-performance platforms; therefore the business model of the broker rests on his ability to find collisions efficiently.

Now the alert reader might ask, "Aren't these conflicting requirements?" The answer is a resounding no, and the resolution of this apparent paradox provides interesting insights into MicroMint. Note that we may typically assume that the broker possesses more computing power than would-be forgers have access to; at the same time, to be profitable, the broker needs to mint many more coins than a typical forger would. But the collision search problem is perfectly suited to this scenario: it provides a fundamental economy of scale for the broker. Generating a great many collisions is cheaper (per collision) than finding just a few. After a large initial investment, the broker can mint coins efficiently, but without comparable capital, would-be counterfeiters cannot create forged coins as economically. (Here's another way to look at this situation: the collision search problem provides a natural threshold between "collisions extremely rare" and "collisions abound", and when the broker has more computing power than counterfeiters, the broker can tune the problem to place the threshold firmly inside that performance gap.)

Thus we see why a high performance solution to the collision search problem is critically important to the MicroMint broker.

2.3 Other applications

2.3.1 Graph theory

Graph theory is intimately involved in the collision search problem. As we will see later in Section 3.2, collision search can be related to the cycle-finding problem on a certain large sparse random directed graph determined in a natural way from the hash function. This insight is fundamental to a number of the most powerful algorithms for collision search; collision search algorithms benefit greatly from graph theory. The converse is also true; a number of the algorithms we describe can be used to identify cycles on large sparse random graphs.

2.3.2 Codebreaking

Collision search also finds a long list of applications in cryptanalysis, or codebreaking. First of all, ability to perform efficient collision search would give cryptanalysts the ability to break cryptographic-strength hash functions. Digital signatures, authentication, timestamping, and a number of other important cryptographic applications all rely on the infeasibility of finding collisions in cryptographic hash functions, so this problem is of great interest to cryptographers and system designers. Of course, blind search is not likely to find a collision in MD5 (which has 128-bit outputs) anytime soon, but cryptographers remain vitally interested in tracking the state of the art, so they can retire MD5 before Moore's law and collision search technology catches up with it.

Second, van Oorschot and Wiener have shown how to calculate discrete logarithms over an arbitrary general finite group by using a collision search algorithm. For a number of groups in widespread use (especially elliptic curve groups), the collision-based algorithm is the best known. A number of important public-key systems rely on the security of discrete logarithms over these groups, so users of such systems watch collision search technology closely to understand the security of their system.

Third, collision search can be used to optimize certain types of attacks known as "meet-in-the-middle" attacks; typically they would require enormous amounts of memory, but efficient collision search algorithms remove much of that need.

In the end, collision search finds widespread application to codebreaking because of a fundamental phenomenon: the powerful birthday paradox has been used to break many cryptographic systems, and collision search algorithms primarily focus on speeding up birthday paradox computations on high-performance parallel architectures.

3 Related work

3.1 MicroMint's naive algorithm

The simplest algorithm one could imagine for solving the collision search problem would have a single machine continually picking x's at random, calculating $h(x)$, and comparing this output to all previous outputs. Note that this requires storing each hash output generated; however, in practice the costs of I/O are much greater than those of computation, so the time spent accessing long-term storage will dominate. Of course, the computation costs are unavoidable (because of the birthday paradox), but the I/O costs form a considerable amount of waste which we would very much like to avoid.

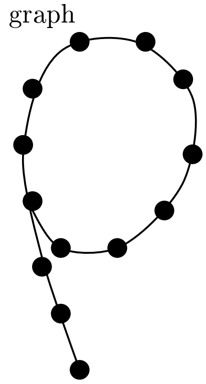
The original algorithm described in the MicroMint paper [8] attempted to solve the collision search using a naive parallelization of the above algorithm. p processors each continually pick x's at random, calculate $h(x)$, and stream them to a central server. This central server then stores and compares the expected $2^{n/2}$ hash values as before.

Needless to say, MicroMint's naive parallel algorithm is not the paragon of scalability. All of the I/O costs found in the serial algorithm remain. And there will be an extreme amount of communication (one message for every hash evaluation) sent to the central server. The server—and in particular the server's I/O subsystem, including both the network and disk interfaces—forms a central bottleneck, which inherently limits scalability. In short, communications costs will totally overwhelm the computational costs, thus preventing any hope of scalability.

3.2 Other serial algorithms

Pollard is credited with identifying the strong link between collision search and graph theory [4, 5]. Consider the (very large and sparse) graph with 2^n nodes (labelled 0 though $2^n - 1$), and a directed edge from x to y iff $h(x) = y$. Note that each node in this graph has outdegree exactly 1. Now pick a random starting point and perform a random walk on this graph; this amounts to choosing x_0 at random and iterating the recurrence $x_n = h(x_{n-1})$. Note that the resulting trail will eventually circle back and cross over itself; when it does, it will thereafter retrace the same steps (because each node has outdegree 1) and continue to cycle forever. The resulting path will look something like the Greek letter "rho" (ρ), as shown in Figure 1.

Figure 1: Pollard's Rho: viewing collision search as cycle-finding in a graph



The critical observation is that we can identify a hash collision from such a trail: if y is the node where the leader meets the cycle, x is its immediate predecessor on the leader, and x' is its immediate predecessor on the cycle, we see that $h(x) = y = h(x')$. Therefore, as Pollard noted, we can use any method for detecting cycles in a random walk to build a collision search algorithm. This key insight into the connection between collision search and graph theory is central to a number of efficient collision search algorithms. Algorithms based on this approach are known as "Pollard's rho" methods. Note that, by the birthday paradox, we expect to walk for about $2^{n/2}$ steps before cycling; therefore, Pollard's rho methods tend to detect a collision with about the same number of hash computations as more naive collision search algorithms.

One cycle-detection technique, called "distinguished points" [6, 7] (attributed to Rivest [10]), is worth mentioning in particular. The idea is to single out a small subset of the graph nodes as "distinguished" based on some simple recognizable property; a convenient choice is to identify a node x as distinguished if the first d bits of x are zero, for some d. The collision search algorithm would record only the distinguished points it encounters on its random walk; the algorithm can detect a cycle by recognizing when a distinguished point is encountered for the second time. Note that memory requirements are reduced by a factor of 2^d, while computation costs remain roughly the same.

Another clever technique, Floyd's cycle-detecting algorithm, avoids all memory costs. Think of it as two concurrent processes: one does an ordinary random walk, and another follows in its footsteps but at half the rate. At each step, one checks to see if the two processes are at the same node in the graph; if they are, a cycle has been detected. It is an amazing and beautiful fact that this simple algorithm will always correctly detect cycling [2]. (Floyd's algorithm potentially imposes up to a factor of three increase in computational cost, but that can be virtually eliminated with some additional algorithmic design [10].)

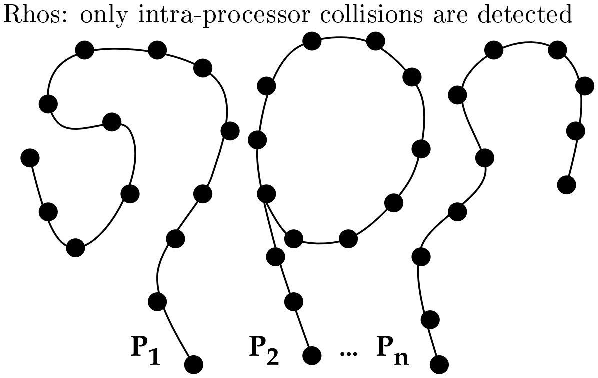
With these and other techniques, one can almost entirely eliminate the memory requirements and the expensive I/O operations that dominated performance of the naive algorithm. In particular, serial collision search algorithms run near the theoretical upper bounds: there simply is very little room for improvement. Serial collision search is a solved problem.

3.3 Parallel algorithms

Parallel algorithms for collision search have tended to mirror the serial ones; until recently, however, no good approach to parallel collision search was known.

One simple approach is to have each of p processors independently execute a random walk, and do Pollard's rho cycling checks independently on each; this approach is attributed to Brent [10].

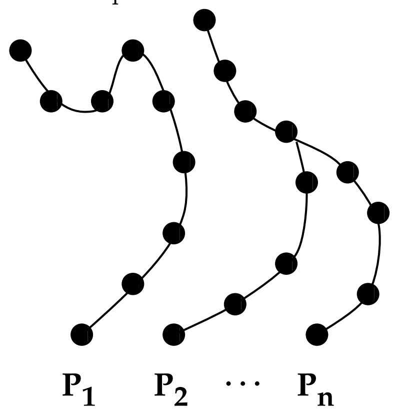
Figure 2: Running separate, independent Pollard's Rhos: only intra-processor collisions are detected



This avoids the need for any communication, but one pays for the lack of sophistication with a significant performance hit: the speedup is only $\sqrt{p}$ when using p processors. The reason for this is that after each processor has walked $2^{n/2}/\sqrt{p}$ steps, the expected number of collisions at each processor is about $1/p$; by linearity of expectations, we get that the expected total number of collisions across all processors is about 1 after a total of $2^{n/2} \cdot \sqrt{p}$ hash computations. See Figure 2.

There is a much better approach, as discovered and related by van Oorschot and Wiener in recent work [10]. Note that one of the major causes of inefficiency in the previous algorithm is the independence between processors: if two processors' trails converge (in a "lambda", or λ, shape, Figure 3) the algorithm will not notice, even though a useful collision can be obtained from such a useful convergence.

Figure 3: van Oorschot and Wiener's lambda: taking advantage of inter-processor collisions



In van Oorschot and Wiener's algorithm, processors periodically communicate the current state of their random walks to detect λ's, which ensures that we can take advantage of collisions between processors. In particular, the method of distinguished points is applied, and whenever a processor encounters a distinguished point, it communicates that to a central server; if that distinguished point has already been encountered once before (usually by some other processor—a λ event—although occasionally ρ's do happen too), the central server identifies that a collision has occurred and recovers it. With this innovation, processors spend most of the time in local computation, with occasional communication, thus improving efficiency.

Finally, we describe the particulars of how collisions are recovered from the incoming reports of distinguished points. A processor performing a random walk encounters points x_1, $x_2 = h(x_1)$, $x_3 = h(x_2)$, and so on. If 1 out of every 2^d points in the overall graph are distinguished, we also expect 1 out of every 2^d points in this sequence to be distinguished points. Each time the processor encounters a distinguished point (say x_j), it recalls the previous distinguished point it encountered (say x_i), and reports the triplet $(x_i, j - i, x_j)$ to the central server.

Once the central server receives two reports (x, c, y) and (x', c', y') that satisfy $y = y'$ but $x \neq x'$, it realizes a λ or ρ event has occurred, and begins a second phase of computation. The goal of this second phase is to recover the collision implied by the event; this is, to find $z \neq z'$ with $h(z) = h(z')$. We note that, by the way the triplets were reported, $h^c(x) = y = h^{c'}(x')$, where superscripts refer to iteration. Without loss of generality, $c' \geq c$, and we let $c'' = c' - c$ and $x'' = h^{c''}(x')$. Now we note that $h^c(x'') = h^{c'}(x') = h^c(x)$. It is also important to note that $x'' \neq x$, because x is a distinguished point, and so could not be on the path between x' and $y'(= y)$.

We now have $x \neq x''$ but $h^c(x) = h^c(x'')$, so we simply calculate $h^i(x)$ and $h^i(x'')$ for each i from 1 until the first such time as the two values are equal (which will happen at latest when $i = c$). The values from the previous iteration are then the desired collision.

For more details on this algorithm of van Oorschot and Wiener, we refer the interested reader to their paper [10]. We decided to investigate their algorithm in more detail, implement in on a real-world parallel processing platform, and examine how well it behaves in practice. We report on that investigation in further detail in the following sections.

It is worth pointing out that almost all of the previous work in this field concentrates on pairwise collisions, and ignores the problem of finding k-way collisions for $k > 2$. We remind the reader that MicroMint depends crucially on the k-way collision search problem. Therefore, we also set out to investigate the problem of k-way collision search. Can the previous work on 2-way collisions be generalized to apply to the k-way collision problem? Can one develop efficient, high-performance solutions for natural problem instances needed by applications such as MicroMint? We answer these questions in the affirmative in the following sections.

4 Enhancing the algorithm

We present two important improvements to the parallel collision search algorithm. The first gives better scalability by improving the data partitioning; the second extends the algorithm to handle k-way collisions for $k > 2$. These modifications make up a substantial part of this paper's contribution.

4.1 Data partitioning

Note that van Oorschot and Wiener's algorithm has a central bottleneck. It relies on a central server to maintain a central list of all distinguished points encountered to date; all processors periodically communicate with the central server. This has a number of disadvantages. First of all, the amount of communication required clearly grows as we increase the number of processors, and communication with the central server is likely to soon become a major bottleneck. Second, the amount of data processed by the central server also grows in the same fashion, so:

- the storage capacity required will increase, which means that soon main memory will be exhausted and we will be required to store the data on disk, which is substantially slower;
- the I/O bandwidth needed will increase, which means that it will soon become a bottleneck (and in fact that point will come even faster than one might otherwise expect because of the need to use slower storage such as hard disks); and
- the amount of computation needed to manage the data will increase, which means that it may eventually also become a bottleneck.

Third, the placement of all distinguished point data at the central server means that (unless special efforts are taken) all of the phase 2 calculations will occur at that one node, so the second phase will not be parallelized. If we are attempting to find a great many collisions, the poor performance of the second phase will become a serious problem. In short, the centralized data partitioning will severely limit the scalability of the algorithm.

In fact, we started to observe effects attributable to these problems in early experiments with relatively small numbers of nodes, which is what motivated us to examine the data partitioning issue. We suggest distributing the list of distinguished points among all of the processors to avoid these central bottlenecks. In particular, we suggest using a hash table (keyed on the reported distinguished point, but not on the previous distinguished point or the trail length between them), with buckets distributed among all the processors. When a processor has a distinguished point to report, instead of sending it to the central processor, it identifies its proper location in the distributed data structure, and sends it directly to the appropriate destination. The destination incorporates it directly into the hash table, checks to see whether this distinguished point has been seen before (note that this check is entirely local, due to the choice of partitioning), and if not, continues with its computation. (These communications are not synchronized, but occur at random times as distinguished points are encountered.) The random nature of h implies that it is appropriate to distribute the hash buckets among the processors in a uniform, static manner.

This improvement allows us to avoid all of the drawbacks of a central server and associated central bottleneck; in return, we expect to see significantly better scaling behavior. For instance, while the amount of data generated still grows linearly in the number of processors p, now the storage resources available also grow linearly in p. It turns out that we can keep the entire hash table in RAM, which speeds up accesses to the data structure significantly. Furthermore, I/O bandwidth and network bandwidth are now split among all of the processors, eliminating those bottlenecks. Finally, we note that a careful choice of partitioning ensures that data is distributed in such a way that each processor does its second-phase computation entirely on local data, and we get the desired parallelization and scalability there for free. This is an especially big advantage when generating collisions en masse, and even more so for k-way collisions ($k > 2$).

In short, we recommend our improved data partitioning scheme for anyone implementing van Oorschot and Wiener's parallel algorithm: there's no reason not to use it, and it greatly improves scalability.

4.2 k-way collisions

In this section, we generalize the parallel collision search algorithm to handle k-way collisions, for $k > 2$. We modify only the second phase; the first phase proceeds exactly as before (except that it needs to continue for far longer, to ensure that we have performed at least $2^{(k-1)n/k}$ hash calculations). The second phase examines each hash bucket, looking for clusters of entries which all share the same distinguished point value (but for which the previous distinguished point values are all distinct). We ignore clusters of size less than k, as they cannot possibly lead to k-way collision. Also note that, unlike 2-way collisions, having a cluster of size k is *not* sufficient to ensure a k-way hash collision for $k > 2$. For example, we could have k distinguished points $x_1, \ldots, x_k$ for which $h(h(x_i)) = y$ for all i (y is the common distinguished point). Then the k triplets $(x_i, 2, y)$ would be reported, but it may be that $h(x_i) = z$ for $1 \leq i \leq k-1$ and $h(x_k) = z'$ for two points $z \neq z'$ that also happen to satisfy $h(z) = y = h(z')$.

Suppose we are given a single cluster of size j. List the triplets in this cluster as $(x_1, c_1, y), (x_2, c_2, y), \ldots, (x_j, c_j, y)$, so that $c_1 \geq c_2 \geq \cdots \geq c_j$. We search for the possible k-way collision resulting from this cluster by repeatedly performing the following steps 1–3:

1. For each i such that $c_i = c_1$ (note that this will be all $i \leq i^*$ for some i^*, because of the ordering of the c_i), replace (x_i, c_i, y) with $(h(x_i), c_i - 1, y)$. Note that this replacement maintains the invariants that the c_i are in non-increasing order, and that $h^{c_i}(x_i) = y$ for each i.

2. After this replacement, you may find multiple entries in the list that have become the same. If there are k or more entries that are now the same, their previous values form (at least) a k-way collision, which you record.

3. Remove all duplicates from the list (thus shrinking j), and go back to step 1 until there are less than k triplets in the list.

This algorithm for the second phase is very efficient. Note that it performs (at most) the number of hash calculations that would be performed in the second phase for a $k = 2$ collision search (given the same set of triplets produced by the first phase). Furthermore, the clusters may be analyzed in parallel—and in fact, they will be, due to the data partitioning scheme described above—and such computations are purely local, requiring no communications. Therefore, we see that this is very scalable; in practice, the cost of the first phase will dominate the total cost of the k-way collision search effort.

This demonstrates that the successful parallel collision search algorithm of van Oorschot and Wiener extends very cleanly to the $k > 2$ case; moreover, much of our experience with the $k = 2$ algorithm applies directly to this extended algorithm as well.

5 Implementation

We implemented this algorithm on the NOW, a high-performance parallel platform, using GLUnix to select idle workstations. The NOW was well-suited to this project: it had some convenient (though far from perfect) software development tools, it supports incremental scalability as well as absolute scalability (which helps avoid any need for the dreaded "forklift upgrade"), and it was readily available for our use. We targetted the Sun UltraSPARC workstations, as they provide extremely high performance, a fast network interface, and there are a large number of them (80+) available through the NOW.

We coded our initial implementation with the MPI library. At the time this seemed reasonably well-suited to our application. However, at a late point in development (we had nearly completed implementation, or so we thought), we ran into one serious stumbling block: a bug in MPI. To achieve maximum performance, we desperately needed asynchronous, non-blocking primitives to both send and receive messages; however, after some investigation, we learned that the non-blocking network-polling primitive simply didn't work on the NOW due to some bug in the implementation of the MPI library. The local MPI expert was out of town for several weeks, and we had no hope of tracking down and fixing the bug ourselves, being relatively new to MPI programming. We were blocked on the MPI bug; we could have waited for the expert to return, but we decided the wisest thing to do was to re-implement on another language with better support for the primitives we needed. In hindsight, that was for the best—the fundamental problem was that we were using primitives that are part of the periphery of MPI, not the core functionality that everyone uses (and thus is presumably more stable and correct and higher-performance).

With that experience, we investigated the available tools a bit further, and decided that AM-2 (Active Messages 2) was a good choice, and would have made a much better fit for our application in the first place. One would be hard-pressed to imagine a pre-existing tool better suited to our needs:

- Our algorithm is totally message-driven; implementing with a shared-memory abstraction would have been very painful.
- AM-2 operates close to the metal, and it has a simple interface with low overhead for short messages, which fits the profile of our application's communication needs. (No fancy features, but we didn't really need bells and whistles; therefore, the simplicity and high performance of AM-2 were ideal.)
- The AM-2 library provides a convenient, easy way to get event-driven semantics without needing to write threaded code. The algorithm is very event-driven, and so this ease of development was well appreciated.

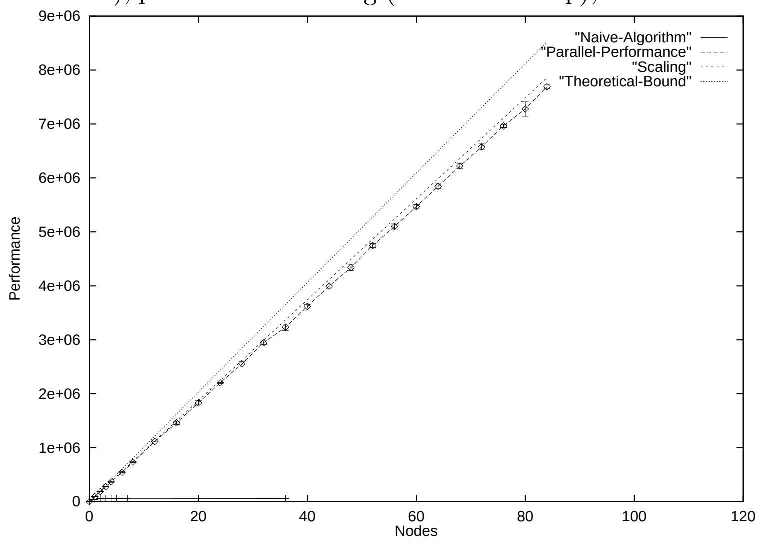
Furthermore, AM-2 was widely deployed on all the NOW UltraSPARC nodes; it was written here at Berkeley, so support seemed likely to be easier to find; good documentation and example code was readily available; several other parallel languages seemed to be implemented in AM-2, which suggested that AM-2 would be at least as stable as them; and AM-2 did not have any artificial restrictions on the number of nodes it could run on (such as requiring that they must be a power of two, for instance). Best of all, local AM-2 gurus were rumored to be highly available.

The MPI bug that prompted the move to AM-2 was eventually fixed. However, we do not regret the time it took to re-implement for AM-2. The MPI bug forced us to do a better job selecting our tools; as we found out, our initial choice had been sub-optimal. We learned a valuable lesson: a little time spent finding the best tool before coding pays off in the end.

We learned another lesson from the experience: some of the parallel development tools on the NOW are woefully inadequate. GLUnix was a mess, suffering from persistent stability problems (it would often get into a funny state, and need to be restarted) and from occasional configuration errors that violated the abstraction of a homogenous cluster. The NOW hardware also had occasional connectivity problems (the poor cross-cluster communications performance was notable), even including one notable instance (following some wiring upgrades—no surprise there) where the entire network became partitioned into two halves. Fortunately, Alan Mainwaring stepped in to fix the hardware problems each time those arose. In the end, the recurring GLUnix stability problems were the most noticeable.

However, we also observed a very promising phenomenon: these tools seemed to improve significantly during the last two months of the semester when the NOW saw a lot of usage from CS267 projects. Several students involved in the NOW project evidently put in a lot of hard work to make the NOW as usable as possible for CS267 students, and that paid off. We hope those trends will continue.

Figure 4: Performance of our enhanced collision search algorithm, as compared to the naive MicroMint algorithm (bottom; almost flat), perfect linear scaling (second from top), and the theoretical maximum (top)



6 Performance

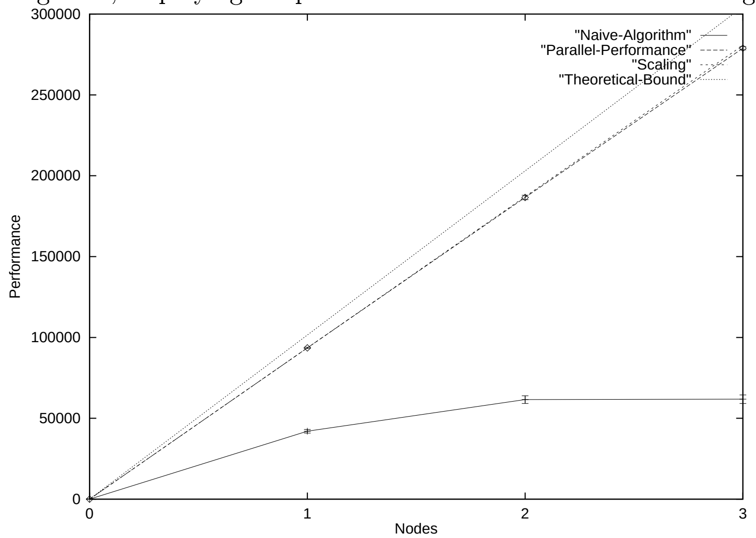
This study would not be complete without a careful look at the performance of our application. We conducted extensive measurements on the NOW UltraSPARCs, running on up to 84 nodes, and running a large number of measurements when we had reserved all the nodes to ensure other user's jobs did not interfere with our application. To ensure that a comparison between performance figures made sense, we scaled up the problem size proportionately as we increased the degree of parallelism.

The primary figure of interest is performance graphed against the number of processors. We have plotted our results in Figure 4. The bottommost (nearly flat) line in the graph shows the performance of our implementation of the naive MicroMint parallel collision search algorithm from [8]; Figure 5 shows a closeup of the plot near the origin, which is where all the interesting behavior happens for this algorithm. The next line above that gives the performance of our implementation of the parallel collision search algorithm (based on van Oorschot and Wiener's work, with our improvements as described in Section 4). The diagonal line just above that is the straight line passing though the origin and our data point for one processor: it depicts the performance that we would see if we had achieved perfect linear scaling; this lets us compare scaling behavior readily. Finally, the topmost line gives the theoretical upper bound on the maximum performance possible with any algorithm; this is the performance one would achieve if everything except Phase 1 calculations of h were free.

Figures 4 and 5 show us that the naive MicroMint parallel algorithm has very poor scaling behavior. After about 3 nodes, the performance curve is flat, and there is no benefit to adding additional processors; even before then, it is noticeably less than ideal. We attribute this to both the cost of I/O and communications. If the bottleneck was solely disk performance, the curve would flatten after 1 processor; if the bottleneck was solely network performance, we would expect the single-processor performance to be roughly comparable to the theoretical bound.

In contrast, we get excellent performance from the more sophisticated parallel collision search algorithm. We observe very close to perfect linear scaling all the way up to 84 nodes, which is a very positive res-

Figure 5: Closeup of Figure 4, displaying the performance of the naive MicroMint algorithm in more detail



ult, as both absolute and incremental scalability were important to us. Furthermore, we see that we are not far away from the theoretical maximum. That is well-suited to MicroMint (where real money is potentially at stake), because brokers want to be sure that would-be forgers can't get better performance than they with less powerful hardware just by upgrading their algorithms. Instead, we know that one simply cannot do much better (at least at these ranges of scale) than our implementation.

7 Conclusion

In this project, we studied the parallel collision search problem. We pointed out serious shortcomings of MicroMint's naive parallel algorithm, and were thus motivated to look for a better approach. We identified and eliminated a central bottleneck in van Oorschot and Wiener's parallel algorithm; with clever data partitioning, we were able to achieve excellent scaling behavior, including both absolute and incremental scalability. We also extended the algorithm to handle k-way collisions efficiently and cleanly. Finally, we implemented the algorithm on the NOW to gain practical experience with it on a real parallel architecture. Measurements on up to 84 processors indicate that our implementation performs extremely well, with efficiency near the theoretical maximum, and displays near-perfect scaling behavior.

References

[1] D. Chaum, "Blind Signatures for Untraceable Payments," *Proc. of CRYPTO'82*, Plenum, D. Chaum, R.L. Rivest, & A.T. Sherman (Eds.).

[2] D.E. Knuth, *The Art of Computer Programming*, vol.2, Addison-Wesley, 1981.

[3] T. Okamoto and K. Ohta, "Universal Electronic Cash," *Proc. of CRYPTO'91*, Springer.

[4] J.M. Pollard, "A Monte Carlo method for factorization," *BIT*, vol. 15 (1975), pp. 331–334.

[5] J.M. Pollard, "Monte Carlo Methods for Index Computation (mod p)," *Math. Comp.*, vol. 32, no. 143, July 1978, pp. 918–924.

[6] J.-J. Quisquater and J.-P. Delescaille, "How easy is collision search? Applications to DES," *Proc. of EUROCRYPT'89*, Springer-Verlag.

[7] J.-J. Quisquater and J.-P. Delescaille, "How easy is collision search? New results and applications to DES," *Proc. of CRYPTO'89*, Springer-Verlag.

[8] R.L. Rivest and A. Shamir, "PayWord and MicroMint: two simple micropayments schemes," presented at the 1996 Security Protocols Workshop, Cambridge, UK.

[9] M. Sirbu and J.D. Tygar, "NetBill: An Internet Commerce System Optimized for Network Delivered Services," *IEEE COMPCON'95*, 1995.

[10] P.C. van Oorschot and M.J. Wiener, "Parallel Collision Search with Cryptanalytic Applications," to appear, Sept 23 1996.