Looking for Class Records in the 3x+1 Problem by means of the COMETA Grid Infrastructure

Giuseppe Scollo

***Abstract*—The design of a parallel algorithm for the subject search on the Cometa Grid is presented. Known optimization techniques of the basic algorithm are supplemented with novel ones. These, and especially the higher performance enabled by the 64-bit architecture, outperform current implementations on 32-bit machines by a rough 5 speed-up factor. This work introduces the basic algorithm, its optimization techniques and their mathematical underpinnings. The most novel aspect of this contribution relates to the optimal integration of a new optimization technique, called Acceleration, which shortens the delay computation time, with known ones, such as Head cut-off, that stop delay computation at an early stage, under appropriate conditions. Seemingly, these two kinds of techniques work on orthogonal aspects of the computation, since Acceleration decreases the delay computation time for any given trajectory, whereas Head cut-off decreases the number of trajectories whose delay is eventually computed. So, one might expect that optimal integration of these techniques ought to result from their combination with parameter values that are respectively optimal for each technique considered in isolation. At a closer look, however, this expectation proves ill-founded, because of actual interference between the techniques in question. A key idea to obtain near-optimal integration of Acceleration with Head cut-off is that of appropriately smoothing the former in order to prevent the loss of information that would harm the effectiveness of the latter. This is exposed in detail in this work, together with time performance statistics out of a 5-month search that has more than doubled the size of the search space explored so far. The software is open source, written in standard C++, but for a few scripts, mostly meant for job monitoring and control on the Cometa Grid, and is going to be made freely available on the PI2S2 Web in the GRID CT Wiki.**

***Index Terms*—Collatz problem, class record, algorithm optimization, parallel search, optimization interference.**

I. Introduction

THE $3x+1$ problem, also known as the Collatz problem, and under several other names, concerns the behavior of the iterates of the function which takes odd integers x to $3x+1$ and even integers x to $\frac{x}{2}$. The $3x+1$ Conjecture asserts that, starting from any positive integer x, repeated iteration of this function eventually produces the value 1; or, in other words, that it has the $1 \to 4 \to 2 \to 1$ cycle as (unique) attractor.

The $3x+1$ problem is comprised of two distinct, separate questions:

1) existence of eventually periodic reductions with a non-trivial period (the aforementioned cycle being the trivial one),
2) existence of reductions which would not be eventually periodic.

G. Scollo is with the Department of Mathematics and Computer Science, University of Catania, Catania, Italy, and with Consorzio COMETA, Catania, Italy, e-mail: scollo@dmi.unict.it

The Conjecture says that both questions have a negative answer. The proof of both conjectures has turned out to be awfully hard to find, in that it is affected by all difficulties which one may expect from a purported chaotic behaviour of $3x+1$ reductions. Paul Erdös held the view that "Mathematics is not yet ready for such problems". We refer the interested reader to [1] for a detailed, historical account of the $3x+1$ problem, together with a comprehensive annotated bibliography on this problem and its several extensions and generalizations, that is maintained by the same author on *arXiv* [2], [3].

The present work does not aim at solving the $3x+1$ problem—a positive answer is rather assumed. Subject of interest here is the dynamical behaviour of the $3x+1$ iteration, and especially the computational challenge which arises when one classifies the eventually periodic $3x+1$ trajectories by their finite transient length, also referred to as the *delay* of a trajectory, and aims at finding minimal elements of delay classes, referred to as *class records*. More precise definitions of these and related concepts are as follows. Let f denote the subject function, and f^n its n-fold iterate, then the following definitions refer to the (discrete) dynamics of iterated f:

- *trajectory* at (origin) x: the infinite sequence x=f^0x, f^1x, f^2x, ...;
- *delay* of (trajectory at) x: the smallest n such that f^nx=1;
- *delay class* d: the set of those x which have delay d;
- *class record* (CR): the smallest member of a delay class;
- *delay record* (DR): an x such that every $y < x$ has a lower delay.

A couple of remarks are in place:

- every delay class is populated (2^d is the largest member of delay class d), so a CR exists in every delay class;
- every DR is a CR (by the definitions of these concepts), but the converse does not hold (for example, 5 is the CR of delay class 5, but it is not a DR, since 3 has delay 7; the latter happens to be a DR).

The computational challenge posed by the CR search is not just the trivial consequence of the fact that it will never end, since not only the search space is infinite but also the search target, that is the set of CR's, is infinite. It rather consists of a twofold aspect of the CR distribution:

1) exponential decay of their average density;
2) logarithmic growth of the average delay of trajectories.

The design of efficient, parallel algorithms for CR search is addressed in [4], in connection with algorithms and program optimization techniques for computing peak statistics of $3x+1$ trajectories. A case of special interest is their concept of "composite polynomial" or *Vermeulen polynomial*, whereby

several steps in a $3x+1$ trajectory are computed by a single expression, paying attention to guarantee that intermediate computation values never exceed the final value. This is indeed one of the ideas implemented in the parallel search program running on the Cometa Grid, but our novel contribution consists of an inductive definition of those polynomials, missing in the cited work, that enables their generation at preprocessing time.

The rest of this paper is organized as follows. The motivations for running the parallel CR search on the Cometa Grid are exposed in Section II, while the basic parallel structure of the search algorithm is presented in Section III. The main technical content of the paper is put in Section IV, which deals with the optimization techniques adopted to get execution speed-up. Performance figures relating to the explored search space are reported in Section V, together with a brief discussion of performance testing experiments aimed at empirical identification of optimal parameter values for the aforementioned optimization techniques. The current state of the search and the most significant results so far obtained are summarized in Section VI. Finally, Section VII gives an outline of future research perspectives and draws brief conclusions from this experience.

II. Distributed Search of Class Records

A distributed CR search effort has been set up since quite a few years by [5], who maintains a website with results and status of the ongoing progress of that endeavour. This has produced 2016 Class Records so far, exploring the search space up to $60200 \cdot 10^{12}$.

The present contribution arises as a follow-up of that effort, and is primarily aimed at expanding the current rate of search space exploration thanks to the computational power made available by the Cometa Grid infrastructure, supplemented by novel optimization techniques which are adjoined to those already implemented in the software run by the aforementioned distributed search. Additional motivation for running the parallel CR search on the Cometa Grid comes out of the following facts.

- As of today, in order to join the 3x+1 distributed search one is required to install a piece of software on a 32-bit machine equipped with a proprietary operating system. The author mostly works with a free one, and making the other one available for such a job actually entails the impossibility to use the machine for the daily work duties in the meantime (because of a dual boot installation). And, it's no short time (of the order of 19 days, according to the impact estimation on the 3x+1 search page [6]).
- The piece of software in question implements an exciting series of beautiful tricks and optimizations to speed-up the search, the most effective of which actually come from clever exploitation of mathematical properties of the subject function, see the enlightening technical details in [7], for instance. However, for efficiency reasons the software is written in C, with parts in 32-bit vendor-specific assembler, which is of little use in a Grid of 64-bit machines.
- The efficiency gain afforded by trading off high-level readability, in favour of low-level execution speed-up, amounts to a factor of 5, according to the aforementioned technical account. However, high-level algorithms are more easily amenable to analysis and further improvement, and the speed-up that may be obtained that way is... unpredictable! So, maybe lower, maybe not. This may offer some motivation to push the exercise beyond its currently known limits.
- Programming methodology is of professional interest to the author, paying attention to both efficiency and clarity of design. In a parallel execution context, the aim at finding optimal blends of both qualities becomes just a bit more challenging.

III. Basic algorithm and DAG parallelization

The obvious approach to parallel search in the problem at hand enjoys the simple structure of a Directed Acyclic Graph (DAG) of parallel processes illustrated in Fig. 1, where no interaction between concurrent processes is required—just a little care in the merge of their outcomes does the job.

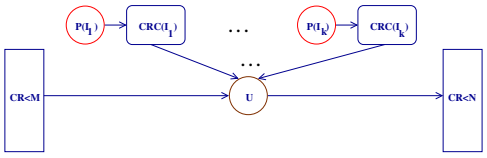

Fig. 1: DAG structure of CR parallel search

Each parallel process explores a given interval I_k of the search space, referred to as its search *slice*, and yields a set of *CR candidates* (CRC) in that slice, that consists of the lowest elements in delay classes that happen to be populated in the explored slice. Thus, if all CR's below a given integer M are known, the new CR's above M and below $N > M$ can be determined by partitioning this search interval into any convenient number of slices, then running the parallel search processes on them, and finally selecting the best (*i.e.* lowest) CRC found for each delay class that is not populated by any number below M.

It is apparent that the algorithm depends neither on the size of the search interval nor on the slice size, which are just parameters of its execution. This fact may be conveniently exploited to effectively manage the fairly frequent case that some of the running processes fail to complete their search, *e.g.* because of hardware or system failures at the Grid computing elements (CE) hosting them. Most often the Grid management system is able to re-allocate such processes (jobs, in Grid terminology) to another available CE, but this entails a restart of the computation (since no use is made of the Grid checkpointing services). When a process failure is detected at a time when most of the other processes have completed their search, the Grid re-allocation mechanism would entail doubling the total waiting time for the parallel search completion. This can be avoided by cancelling the restarted job and then conveniently *subslicing* its missing slice and launching a parallel search on it, thus by straightforward exploitation of a space-time tradeoff.

IV. Optimization Techniques

The computational engine of the CR search algorithm is the *delay computation* function, which, for a given positive integer x, yields its delay. The execution speed of this function is performance critical, hence it is reasonable to take it as main focus of optimization efforts. These give rise to two major classes of optimization techniques:

- algorithm optimization
- program optimization

Algorithm optimization techniques do not depend on the particular programming language chosen to implement the algorithm, and generally deliver the most significant contribution to execution speed-up. Effective programming techniques may add on this, albeit usually by a lower factor; several such "tricks" are implemented in the current version of the CR search program running on the Cometa Grid infrastructure, but they are not presented here, as the focus of the present paper is on *algorithm optimization*. The main techniques of this class adopted in the subject case study are presented below.

A. Sieving

The fastest computation is that which ... need not run (it only takes the time needed to decide that it need not run). In the problem at hand, such a case occurs more often than not, because of *coalescence* of trajectories, that is, their joining at some common point. Coalescence analysis lowers the number of trajectories that need actual delay computation by a near 6 factor, because of two main facts it brings to light:

- even CR's and CR's that are $\equiv 2 \pmod 3$, respectively are (1-step) predecessors and (2-step) successors of CR's, so they are easily derived from other known CR's, hence their delay need not be computed by the parallel search processes;
- CR's always fall outside certain congruence classes (see below), hence positive integers in these classes, too, need not be considered by the parallel search.

Sieving is precisely the technique whereby positive integers on which the delay computation function is to be invoked are first checked against filtering criteria, such as those which follow from the aforementioned facts. The first of them is a straightforward consequence of the CR definition, whereas the second one needs a little more elaborated coalescence analysis. This is best seen in a simple case first, and then characterized generally.

As a basic example, it is easy to show that no CR falls in the congruence class 5 (mod 8), except for the number 5 (which *is* a CR), because, for all $n > 0$, the trajectories of $8n$+5 and that of $8n$+4 coalesce at $6n$+4 after the same number of steps (3, in this case), hence they belong to the same delay class, and therefore $8n$+5 may never be a CR.

The coalescence situation just seen generalizes to congruence classes $(2^{k-2} + (k \bmod 2)(2^{k-1} - 1) \pmod{2^k})$, for $k > 2$. A check against this formula, together with the check for even or $\equiv 2 \pmod 3$ numbers, would yield an exact factor 6 reduction of the number of positive integers passing the check, but would be computationally expensive, and actually inefficient because the fraction of sieved out integers rapidly decreases with increasing k. It is thus far more efficient to be content with a finite, bounded approximation of the sieving criterion, say up to $k = 18$, that can be efficiently implemented by a pre-computed binary sieve.

B. Tail cut-off

An easy way of shortening the delay computation time is that of factoring out the values yielded by the delay computation function for positive integers below a given threshold. All trajectories leading to 1 (veritably all, thus) sooner or later fall below 2^t, for any given, fixed t. Delay computation may thus get quicker by storing the delay values $D(n)$ for all $n < 2^t$, and then adding $D(n)$ to the partially computed delay of x as soon as the trajectory starting at x reaches such an n.

The higher the value of the Tail cut-off parameter t, the greater the computation time saving, provided t is not too high. For, the storing of pre-computed delay values costs some memory, of course; when the size of this grows beyond a limit that depends on the available cache memory size, then fetching the value of $D(n)$ becomes slower, for it may entail a so-called minor page fault, that is, a cache update. In the present case, the choice of an optimal value for t also depends on other optimization parameters, hence the discussion of this subject is postponed until after the introduction of Acceleration and smoothing.

The amount of memory needed for Tail cut-off may be halved by only storing the values of $D(n)$ for $2^{t-1} \le n < 2^t$, as it is done in the distributed search algorithm [7], or, as it is done in the Cometa search algorithm, by only storing the values of $D(n)$ for all *odd* $n < 2^t$. The latter proves more convenient in the present case, for it is compatible with the adoption of the Acceleration technique for delay computation, that is introduced later below.

C. Head Cut-Off

Among the delay computations which do start, because the trajectory starting point passes the Sieving check, the fastest ones are those which ... need not finish. This is determined by using available data about Delay Records (DR), according to the following reasoning.

Let us say that two DR's are *consecutive* if there is no DR in between them. Let now $x_1 < x_2$ be a pair of consecutive DR's; then $x < x_2 \implies D(x) \le D(x_1)$, by DR definition. One may use this fact to stop delay computation of "hopeless" trajectories ahead of time, where a trajectory is hopeless if it becomes evident that it cannot be that of a CR. This is determined as follows.

Assume all CR's below M are known, and let u be the lowest delay class whose CR is not below M. Then $x \ge M$ may only be a CR if its delay is at least u. Now, if the trajectory of x falls below x_2 after d steps, then $D(x) \le d + D(x_1)$, hence if $d + D(x_1) < u$, then x cannot be a CR—its trajectory is hopeless. Equivalently, this is detected whenever the trajectory of x falls below x_2 after $d < u - D(x_1)$ steps. The higher DR in the pair is thus a threshold against which the value reached by the trajectory is compared, whereas the delay of the lower

DR in the pair, together with u determine until when, in terms of steps from the starting point, the comparison is applicable.

The combined action of Head cut-off and Tail cut-off is illustrated by the staircase-shaped sequence of thresholds in Fig. 2, where the lowest threshold is that of Tail cut-off, whereas the preceding ones refer to Head cut-off. If a trajectory reaches the Tail cut-off threshold with no previous crossing of the staircase, then the delay computation completes successfully and the function yields the computed delay value, otherwise the computation stops at an earlier stage.

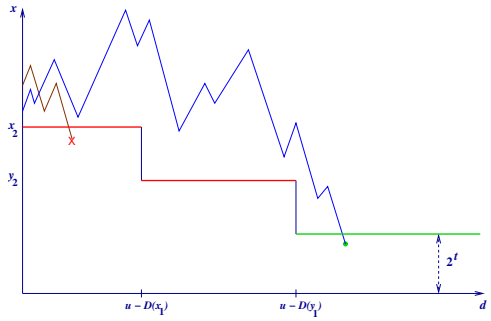

Fig. 2: Head and Tail cut-off of 3x+1 trajectories

An interesting virtue of the Head cut-off technique is that it inherently features *progressive self-optimization* of the search algorithm, in two respects. First, the progress of the CR search eventually leads to discovery of the CR for the lowest delay class with unknown CR; this event enables one to raise the value of u, thereby getting a higher cut-off rate (the staircase in the picture gets a horizontal stretching). Second, albeit less frequently, the progressive rise of u eventually enables the admission of a new, higher pair of consecutive DR's, thereby getting earlier cut-offs (the staircase is extended on the left by a new, topmost threshold).

131 DR's are known to date, but not all 130 pairs of consecutive DR's prove useful to effective Head cut-off. In the first place, the highest DR's have higher delays than u, which is the delay of the lowest delay class with unknown CR, and thus yield a negative upper bound on d. Furthermore, the Head cut-off test has a computational cost of its own, whereby not all pairs of consecutive DR's pay this off by an actual performance gain.

Actually, in the present case, the effectiveness of the selection of consecutive DR pairs for Head cut-off is also affected by the concurrent adoption of other optimization techniques, viz. Acceleration and smoothing, which make parsimonious sizing of the threshold sequence more cost-effective, as it is argued later below.

In conclusion, careful selection of the most effective pairs is a basically empirical task, also because the effectiveness of any given selection of DR pairs for Head cut-off varies with the search interval at hand. Intervals currently searched by the Cometa Grid lie in the neighbourhood of 2^{57}, and a sequence of seven pairs of consecutive DR's is employed for Head cut-off, see Section V below.

D. Acceleration

A small acceleration of delay computation comes from replacing the function f with T, defined as f on the even numbers, whereas for odd x one has

$$Tx = \tfrac{3x+1}{2} \quad = x + \lceil \tfrac{x}{2} \rceil \tag{1}$$

Clearly, if the T-trajectory from x to 1 has O applications of this rule and E applications of the halving rule, then

$$D(x) = 2O \ + E \tag{2}$$

The interest in T comes from the existence of a permutation of the residues (mod 2^k), defined by the k-prefix of the so-called *parity vector* of T-trajectories, viz. the binary sequence $\mathbf{v}(x) = (v_i(x) \ | i \in \mathbb{N})$ defined by

$$v_i(x) = (T^i x) \bmod 2 \tag{3}$$

The k-prefix of $\mathbf{v}(x)$ only depends on x mod 2^k, hence so does the selection of the applicable rule in the first k steps of the T-trajectory starting at x. Thus one may define 2^k distinct k-step composites of T, and decide which applies to x by only looking at x mod 2^k. It takes little effort to see that all such composites are linear functions of x. For odd x, which is the case of interest here, let

$$r = \lfloor \tfrac{x}{2} \rfloor \bmod 2^{k-1} \tag{4}$$

It is computationally convenient to express T-derivatives of x as follows, because intermediate computation values never exceed the final value:

$$T^k x = 3^{t_1(k,r)} \lfloor \tfrac{x}{2^k} \rfloor + t_0(k,r) \tag{5}$$

where $t_1(k,r)$ is the number of applications of rule 1 in the first k steps of the T-trajectory starting at x whenever equation 4 holds, and for $0 \le r < 2^{k-1}$, $t_1(k,r)$, $t_0(k,r)$ are defined by induction on k as follows:

$k = 1$:

$$t_1(1,0) = 0 \tag{6}$$
$$t_0(1,0) = 2 \tag{7}$$

$k > 0$: for $0 \le r < 2^{k-1}$, let $r' = r + 2^{k-1}$:

if $t_0(k,r) \equiv 1 \pmod 2$

then

$$t_1(k+1,r) = t_1(k,r) + 1 \tag{8}$$
$$t_0(k+1,r) = t_0(k,r) + \tfrac{t_0(k,r)+1}{2} \tag{9}$$
$$t_1(k+1,r') = t_1(k,r) \tag{10}$$
$$t_0(k+1,r') = \tfrac{3^{t_1(k,r)}+t_0(k,r)}{2} \tag{11}$$

else

$$t_1(k+1,r) = t_1(k,r) \tag{12}$$
$$t_0(k+1,r) = \tfrac{t_0(k,r)}{2} \tag{13}$$
$$t_1(k+1,r') = t_1(k,r) + 1 \tag{14}$$
$$t_0(k+1,r') = \tfrac{3(3^{t_1(k,r)}+t_0(k,r))+1}{2} \tag{15}$$

fi

An efficient implementation of this technique consists in fixing an optimal value for the acceleration parameter k, and then storing the pre-computed values of $t_0(k,r)$ and $t_1(k,r)$ in 2^k-sized arrays, as well as the few needed powers of 2 and 3 in constant arrays, so that the value of $T^k x$ may be computed at runtime just by quick table look-up, using the value of r as index to the arrays in question, and by the few simple operations thereafter involved in the computation of $T^k x$ by means of Equation 5.

The value of r also determines the delay increment $\delta_k(r)$ to be ascribed to a k-accelerated delay computation step. Another 2^k-sized array to store this is not needed, though, since it may be simply computed as follows:

$$\delta_k(r) = k + t_1(k,r) \tag{16}$$

An upper bound to the value of k comes from the machine word size, but for a 64-bit architecture the optimal value for k is usually well below that, as it is determined by the cache size, and may be easily determined by empirical testing. In the present Cometa implementation of the algorithm, k = 14 proves optimal. Furthermore, just as it happens with ordinary mechanical vehicles, truly optimal driving through the ups and downs of the subject trajectories needs *brakes* besides acceleration, as we are going to see.

E. Interference and smoothing

Head cut-off and Acceleration improve the time performance of the algorithm by acting on orthogonal aspects of the computation; the former drastically decreases the number of trajectories whose delay is eventually computed, the latter decreases the delay computation time for any given trajectory. So, one might expect that optimal integration of these techniques ought to result from their combination with parameter values that are respectively optimal for each of them considered in isolation. At a closer look, however, this expectation proves ill-founded, because of actual interference between the techniques in question. Fig. 3 illustrates the problem, which arises whenever the trajectory gets across the Head cut-off threshold in between the start and target points of an accelerated computation step, both of which stay above the threshold. Clearly, in such cases the cut-off opportunity is missed, hence the computation goes ahead—hopeless albeit fast.

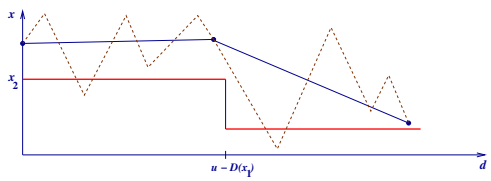

Fig. 3: Interference of Acceleration on Head cut-off

Basically, the problem arises owing to the loss of information brought by Acceleration; one may recover that much of this which is needed in order not to harm the effectiveness of Head cut-off, by suitably smoothing the Acceleration mechanism. The idea is to shorten the Acceleration extent for those composites which may cause interference, and only when the start point of the accelerated step is close enough to the Head cut-off threshold. In the smoothened variant, the Acceleration parameter k is to be taken as the maximum value that the acceleration extent may take, rather than as a fixed one—as it happens in the unsmoothened variant.

A fine grain discrimination between those composites that may interfere with Head cut-off, and those which may never do so, is a basic smoothing tool. The discrimination criterion is whether or not does the "local" trajectory, that is in between the start and end points of an accelerated step, ever fall below the step starting point x. This only depends on x mod 2^k, hence, for odd x, on the value of the r index as specified in the previous section. Static discrimination is thus feasible. Note that the present discussion also applies to Tail cut-off, therefore we shall use the generic "cut-off threshold" term to refer to both cases.

The next aspect of effective smoothing is a decision criterion about when, precisely, should one consider the start point of an accelerated step to lie "close enough" to the cut-off threshold as to apply smoothing. This is a somewhat more complicated question, since the answer depends not only on the r index but also on some "distance" between the local trajectory and the cut-off threshold. Equation 5, together with the subsequent inductive definition of its parameters help the effective solution of this question, as follows.

Let h be the number of halvings that occur in the local trajectory from a start point x, of which only the r index as given by Equation 4 need be known, to a chosen local minimum point in it below the start point (the choice of the local minimum, when more than one exist in the local trajectory, will be discussed in a short while; for the time being, assume that there is a criterion to make such a choice). Thus, h only depends on r, hence so does $t_1(h,r)$ as in Equation 5, with h substituted for k.

The definition of T entails $T^h x > 3^{t_1(h,r)} \frac{x}{2^h}$, so one may check this lower bound against the cut-off threshold, and thereby consider the local trajectory to be close enough to that threshold, say ϑ, whenever $3^{t_1(h,r)} \frac{x}{2^h} \leq \vartheta$. This may be rewritten as the condition

$$x \leq \frac{2^h \vartheta}{3^{t_1(h,r)}} \tag{17}$$

where the expression at the right hand side only depends on r and the cut-off threshold ϑ. It may thus be pre-computed and coded into an array of constants, to speed-up the runtime check. Some care is needed in the implementation of this mechanism, for these constants need two 64-bit words, hence the subject expression is to be evaluated in 2-word arithmetic.

Finally, a minor yet interesting design choice arises about the shortening of the acceleration extent when the local trajectory of unsmoothened acceleration features more than one local minimum points below the start point. Two options naturally surface: either *strong* smoothing, which means shortening to the first local minimum below the start point, or *weak* smoothing, that is, shortening to the lowest such minimum. The former favours Head cut-off (no loss of potentially useful information), the latter favours Acceleration. The difference in

terms of performance is likely to be hardly significant, yet the question is to be settled. The present algorithm features weak smoothing, according to the following analysis.

Assume the local trajectory features a local minimum below the start point, and a subsequent lowest minimum point. If both of them fall below the cut-off threshold, then there is no performance difference between the two smoothing options, whereas if both of them fall above the cut-off point, then weak smoothing affords the obvious performance gain of saving a computation step. Assume now that only one of the two aforementioned minimum points falls below the cut-off threshold. This may happen in two distinct ways. First, the former point lies above the cut-off threshold whereas the latter falls below it. In this case, the loss of information by weak smoothing is harmless, since the cut-off takes place at the lowest minimum anyway. The performance gain is again that of saving a computation step to reach this point. In the second case weak smoothing incurs in a performance penalty; this happens because strong smoothing stops the accelerated step at the first local minimum, but the additional delay between this and the lowest minimum (that where weak smoothing would put the stop) causes an edge fall-out of the (Head) cut-off threshold itself, with loss of the cut-off opportunity. Such a case thus seems to be rare, because of the limited extent of acceleration, which entails that the situation just described is to happen in a very narrow region around Head cut-off edges—which are not that many, by the way.

V. Performance Figures

The contribution by Tail cut-off to performance becomes lower and lower with the upward shift of search intervals, since the fraction of trajectories that manage to reach the lowest threshold becomes smaller and smaller, and it is of no significance at the current search level, which just reached 2^{57}. Tail cut-off cannot be disposed of, though, because it plays a useful role to ensure correctness of delay computation in the presence of Acceleration. As a matter of fact, careless application of the latter technique to a very low number might yield an excessive delay increment, corresponding to the case where the multi-step trajectory of the accelerated derivation step would go through the bottom number 1 more than once. Tail cut-off prevents this from possibly happening, provided its threshold is set to an adequately high value. With smoothing, the situation is a bit different. Strong smoothing provides an alternative solution to the problem, hence Tail cut-off could be disposed of. With weak smoothing, on the contrary, the problem persists, hence Tail cut-off is kept in. If k is the Acceleration factor (*i.e.*, the maximum number of halving steps in an accelerated computation step), then the safety Tail cut-off boundary is 2^{k-3} (only delays of odd numbers are stored in the tail). The lowering of the Tail cut-off boundary to the minimum needed for safe Acceleration frees cache memory, which may be utilized to get computation speed-up by an optimal combination of higher Acceleration factor and coarser Head cut-off, that is, smaller set of cut-off points. Why does the latter yield speed-up may be explained as follows.

The by far most memory-hungry data structures are those for acceleration smoothing; with acceleration factor k, if w is the number of head cut-off points, then two arrays of constants are needed, each of size $2^{k-1}\times(w+1)$ entries (since the tail cut-off point is also to be taken into account), respectively holding the lower and upper word of 2-word constants for the smoothing check expressed by condition 17. Reducing the size of the cut-off set thus delivers speed-up because it saves cache memory. The following 7-sized cut-off set, with k = 14, has emerged as optimal from extensive experimentation:

$$\{1958, 1919, 1874, 1662, 1443, 1255, 1050\}$$

where each cut-off point is actually represented by the delay of the lower DR in the pair of consecutive DR's that determines the cut-off point, as already explained.

The very near-to-linear progression of the optimal cut-off point delays is quite striking. The relatively closer distance of the highest points can be explained by their being fairly close to the current search numbers—a kind of "optical distortion", so to say. In the (very) long run, it is natural to expect the two highest points to get replaced by the 2090 DR, and later supplemented by the 2254 DR, both fairly aligned with the linear progression of the previous five DR delays in the set.

Table I summarizes the outcomes of a performance testing experiment, made with a search slice of size 2^{34} placed near 3×2^{55}. For some of the listed values of the cut-off set size w, a choice of various cut-off sets of that size were tested, and the performance figure reported is the best outcome out of that choice. Each row in the table refers to a fixed value of the Acceleration factor k. Performance figures are the measured CPU time in seconds, rounded up to the next integer. Scaled up to a 2^{44}-size search slice, which is the current typical size in the Cometa search, the optimal (k,w) combination measure from this experiment yields an expected CPU time of approximately 69 hours.

TABLE I: Sample performance figures

$k \setminus w$	16	15	11	9	8	7	6
13	273	255	253	253	253	250	262
14	278	292	260	257	250	243	254
15	335	473	283	280	256	246	255

Current search intervals, however, have reached 2^{57}, and the average CPU time measured is higher than that. Parallel search intervals of size 2^{49} are currently explored, and for each of them the average CPU time spent by the 32 processes exploring its 2^{44}-size slices is recorded. Over the latest 41 search intervals completed, these average values lie in the range of 85 to 103 hours. Although well above the expected, these values are still significantly below those measured before running the experiment, typically above 130 hours, with Acceleration factor 10 and a set of 16 cut-off points.

Table II summarizes measured statistics, viz. average CPU time and standard deviation ranges, from the 3x+1 CR search carried out in the Cometa Grid to date. The following conventions are adopted. Each row refers to a group of parallel search intervals processed by the same program version and with the same optimization parameter values. The program version is displayed in the first column, for later reference. The

subsequent two columns tell which intervals do the statistics refer to, as follows. Each parallel search interval s lies in search space $[2^a,\ 2^{a+1}[$, with a displayed in the first of the two columns. Two search spaces are considered, each partitioned into 128 equally sized search intervals, thus of size 2^{48} for $a=55$, whereas of size 2^{49} for $a=56$. Parallel search intervals are numbered starting with 0, and these identification number ranges are displayed in the second of the two columns. Regardless of the difference in interval size, the statistics refer to the same slice size in both search spaces, that is 2^{44}. The average CPU time and its standard deviation are measured, in seconds, over the slices of each interval, and their ranges are reported in the fourth and fifth column, respectively.

TABLE II: CPU TIME PERFORMANCE STATISTICS

v	*a*	*s*	average	s.d.
1.14	55	72–80	[569952–576030]	[56–1744]
2.01	55	81–90	[572932–586710]	[94–2299]
2.02	55	91–127	[541567–570129]	[196–2437]
3.01	56	0–3	[530877–531419]	[521–733]
3.02	56	4–40	[445088–476148]	[398–3465]
6.01	56	41–48	[288959–342423]	[7050–43040]
6.02	56	49–86	[310918–449557]	[15308–167909]
6.03	56	87–127	[304797–369236]	[20203–69636]

The progressive self-optimization afforded by Head cut-off is apparent in the downward trend of the average CPU time statistics, from each version to the next one. This is best appreciated if it is taken into account that, for each group of parallel search intervals sharing the same program version, the average CPU time actually features an upward trend, with the upward shift of search intervals; proper comparison of its values for different, adjacent groups should relate its highest value in the lower group with its lowest value in the higher group. This is also apparent from the following case.

Most program versions referred to in Table II only differ in their Head cut-off parameter values, most often just the u parameter value, that is the delay of the lowest delay class with unknown CR. In one case it happened that the same parallel search interval was explored twice, by different program versions; viz. search interval 4 for $a=56$ was explored with program versions 3.01 and 3.02, which differ both by the value of the u parameter (1964 vs. 1985) and by the introduction of a new, topmost Head cut-off point (with delay 1958 for the lower DR in the new pair), that nearly doubled the highest Head cut-off threshold. Table III lists the CPU time data by the two program versions, for all 32 slices in the subject interval.

A major performance breakthrough, visible in Table II, is brought by the version switch from 3.02 to 6.01. This was preceded by extensive performance testing, of which the sample performance figures reported in Table I are an outcome out of many. Besides algorithm optimization, program optimization techniques were deployed. However, the performance gain obtained in terms of average CPU time is accompanied with an increase of its standard deviation by an order of magnitude. This poses a serious management problem, for a low-variance distribution of job execution times proves much easier to handle than a high-variance one. In practice, the risk is that one has got to trade off job management time for CPU time, which is not necessarily an optimal state of affairs. Research is under way [9] to explore means of, at least partially, automating the job management task in such conditions.

TABLE III: CPU TIME BY DIFFERENT PROGRAM VERSIONS

slice	*v.* 3.01	*v.* 3.02
56/4/00	531385	443589
56/4/01	531573	445001
56/4/02	531360	444016
56/4/03	531187	444668
56/4/04	531577	443721
56/4/05	531253	444551
56/4/06	532025	443778
56/4/07	531321	443921
56/4/08	531214	445401
56/4/09	532452	444108
56/4/10	532710	444306
56/4/11	532672	445743
56/4/12	533140	446347
56/4/13	532595	444971
56/4/14	532893	445274
56/4/15	533559	444745
56/4/16	532555	445647
56/4/17	532387	444211
56/4/18	532481	444729
56/4/19	532438	444380
56/4/20	532468	444905
56/4/21	533638	445218
56/4/22	532413	445549
56/4/23	532946	446429
56/4/24	532229	445786
56/4/25	532388	446885
56/4/26	532547	445735
56/4/27	532369	446192
56/4/28	532894	445847
56/4/29	532642	445716
56/4/30	532309	445841
56/4/31	532334	445616

VI. STATE OF THE SEARCH AND RESULTS

The CR search up to 2^{57} has been just completed while polishing this paper. The CR search by the Cometa Grid started on 25 september 2007, from $2^{55} + 72 \cdot 2^{48}$, that is where the distributed search [6] had arrived at, at the time. As of today, after five months, the explored search space has thus been more than doubled. Here is a summary of the search outcomes.

51 new CR's have been found, listed in Table IV together with their discovery dates. Commas are inserted in the CR values in Table IV, to improve readability. Notes in the rightmost column apply to a few of these CR's. Two of them also are Delay Records, and one of these is even a Strength Record (SR), which is a very rare occurrence; indeed, this is the fifth nontrivial SR known to date, see [5] for the definition of this concept. Finally, three CR's are marked as "surprise!",

meaning that each of them is lower than the best candidate known, for that delay class, until the CR discovery.

TABLE IV: CLASS RECORDS BY THE COMETA SEARCH

date	*delay*	*CR*	*note*
26/09/2007	2036	57464,478811,199374	
30/09/2007	1974	58895,046121,751303	
03/10/2007	2005	59962,029694,389913	
06/10/2007	2049	60538,710187,930201	
06/10/2007	1925	60614,485209,289967	
14/10/2007	2018	61364,655839,889179	
14/10/2007	1987	62098,509011,513287	
06/11/2007	2031	64647,538662,599297	
10/11/2007	1969	65925,324859,522411	
10/11/2007	2000	67457,283406,188652	
16/11/2007	2044	68106,048961,421476	
16/11/2007	2013	69035,237819,875326	
25/11/2007	1951	69520,368888,759751	
25/11/2007	2088	70178,545269,601727	
25/11/2007	2057	71749,582444,954311	
25/11/2007	1982	72035,683012,370407	
29/11/2007	1995	72230,523319,648959	surprise!
30/11/2007	2026	72728,480995,424210	
01/12/2007	1964	74165,990466,962713	
08/12/2007	2008	77664,642547,359743	
09/12/2007	2083	79243,106410,450031	
09/12/2007	2052	80718,280250,573601	
24/12/2007	2021	81819,541119,852238	
28/12/2007	1990	82798,012015,351049	
28/12/2007	2065	85036,542156,982887	
28/12/2007	2034	86196,718216,799062	
01/01/2008	2003	89537,042605,047835	
01/01/2008	1985	90712,689753,202855	
01/01/2008	2047	90808,065281,895300	
06/01/2008	2016	91687,036586,197659	surprise!
07/01/2008	2091	93571,393692,802302	DR #130
09/01/2008	2060	95666,109926,605759	
10/01/2008	2029	96971,307993,898946	
14/01/2008	2055	103144,236075,381231	
14/01/2008	2011	103147,916159,472367	
14/01/2008	2254	104899,295810,901231	DR #131, SR #5
14/01/2008	2086	105267,817904,402591	
15/01/2008	2024	109092,721493,136316	
04/02/2008	2068	111123,844639,969023	surprise!
10/02/2008	2006	116496,963821,039615	
17/02/2008	2081	118864,659615,675047	
17/02/2008	1988	120950,253004,270473	
17/02/2008	2050	121077,420375,860401	
18/02/2008	2019	122729,311679,778358	
18/02/2008	2094	124761,858257,069737	
20/02/2008	2063	127554,813235,474331	
22/02/2008	2032	129080,227221,222247	
25/02/2008	2107	131436,360962,180463	
26/02/2008	2014	135001,531283,304447	
26/02/2008	2089	140357,090539,203454	
26/02/2008	2058	142798,650831,013915	

VII. FUTURE PERSPECTIVES AND CONCLUSIONS

With the CR finding for delay class 1988, all CR's with delay below 2000 have been found. The current lowest delay class with unknown CR is 2009, and the best candidate known for it to date is 154721,874239,208551, not very far away from current search intervals. Thus, another small self-optimization instance of the CR search algorithm is in sight. It will be small because of the subsequent lowest class with unknown CR is 2017, with 181276,570919,731903 as best candidate (pretty farther away); so, unless some surprise would show up, it will take a while before further self-optimization, and in the meantime performance is expected to gradually get worse.

In the medium-long term, a program optimization challenge is in sight. The current implementation of the CR search algorithm exploits the fact that, for numbers below 2^{59}, peak values in their trajectories are guaranteed to stay within the representability boundaries of two 64-bit words, see [8] for details, hence 2-word arithmetic suffices. The challenge consists of checking numbers above 2^{59} for possible need of trajectory computation by 3-word arithmetic, and then computing that way. Recent findings, reported by the same cited source, may enable one to raise the 2^{59} boundary up to 2^{61}. This would defer the challenge from medium to long term.

Finally, the last (but not least!) motivation mentioned in Section II sets the ground for the concluding remark out of this experience. While the algorithm implementation has been extensively tested, and its mathematical background firmly established, its documentation is not yet sufficiently complete to allow one to publish the software, especially for educational purposes. This is of utmost importance, however. The present paper is a first step in this direction, and progress toward this goal is planned for the near future.

ACKNOWLEDGMENTS

This work makes use of results produced by the PI2S2 Project managed by the Consorzio COMETA, a project co-funded by the Italian Ministry of University and Research (MIUR) within the Programma Operativo Nazionale "Ricerca Scientifica, Sviluppo Tecnologico, Alta Formazione" (PON 2000-2006). More information is available at http://www.pi2s2.it and http://www.consorzio-cometa.it.

REFERENCES

[1] J. Lagarias, The 3x+1 problem and its generalizations, *Amer. Math. Monthly* **92** (1985) 3–23, http://www.cecm.sfu.ca/organics/papers/lagarias.
[2] J. Lagarias, The 3x+1 Problem: An annotated bibliography (1963–2000), preprint on *arXiv*, v. 9, http://arxiv.org/abs/math/0309224v9.
[3] J. Lagarias, The 3x+1 Problem: An Annotated Bibliography, II (2001-), preprint on *arXiv*, v. 2, http://arxiv.org/abs/math/0608208v2.
[4] G.T. Leavens and M. Vermeulen, 3x+1 search programs, *Computers Math. Applic.* **24**:11 (1992) 79–99.
[5] E. Roosendaal, *On the 3x+1 problem*, http://www.ericr.nl/wondrous.
[6] E. Roosendaal, *The 3x+1 class record search*, http://www.ericr.nl/wondrous/search.html.
[7] E. Roosendaal, *Technical details*, http://www.ericr.nl/wondrous/techpage.html.
[8] E. Roosendaal, *3x+1 Path Records*, http://www.ericr.nl/wondrous/pathrecs.html.
[9] A. Sortino, *Metodi di controllo dei job nelle griglie computazionali*, Graduation Thesis, University of Catania, CdL Informatica Applicata (2008).