Optimization of Tree Modes for Parallel Hash Functions : A case study

Kevin Atighehchi, Robert Rolland

OPTIMIZATION OF TREE MODES FOR PARALLEL HASH FUNCTIONS

KEVIN ATIGHEHCHI AND ROBERT ROLLAND

ABSTRACT. This paper focuses on parallel hash functions based on tree modes of operation for a compression function. We discuss the various forms of optimality that can be obtained when designing such parallel hash functions. The first result is a scheme which optimizes the tree topology in order to decrease at best the running time. Then, without affecting the optimal running time we show that we can slightly change the corresponding tree topology so as to decrease at best the number of required processors as well. Consequently, the resulting scheme optimizes in the first place the running time and in the second place the number of required processors. The present work is of independent interest if we consider the problem of parallelizing the evaluation of an expression where the operator used is neither associative nor commutative.

Keywords. Hash functions, Hash tree, Merkle tree, Parallel algorithms

1. INTRODUCTION

A hash function is an algorithm (or a mode of operation in the cryptographic terminology) iterating (operating) a function having a fixed input size (a compression function or a block cipher) in order to process messages of arbitrary lengths. Such a function must satisfy the usual properties of pre-image resistance (given a digest value, it is hard to find any pre-image producing this digest value), second pre-image resistance (given a message m_1, it is hard to find a second message m_2 which produces the same digest value), and collision resistance (it is hard to find two distinct messages which produce the same digest value). A sequential (or serial) hash function can only use Instruction-Level Parallelism (ILP) and SIMD instructions [1, 2]. A cryptographic hash function has numerous applications, the main one is its use in a signature algorithm to compress a message before signing it.

The most well known sequential hashing mode is the Merkle-Damgård [3, 4] construction which can only take advantage of the fine-grained parallelism of the operated compression function. If such a low-level "primitive" can benefit from the Instruction-Level Parallelism, by using also SIMD instructions, the outer algorithm iterating this building block could benefit from a coarse-grained parallelism. This parallelism can be employed in multithreaded implementations. Let suppose that we have a collision-free hash (or compression) function taking as input a fixed-size data, $f : \{0,1\}^{2N} \to \{0,1\}^N$. By using a balanced binary tree structure, Merkle and Damgård [3, 5] show that we can extend the domain of this function so that the

new outer function, denoted $H : \{0,1\}^* \to \{0,1\}^N$, has an arbitrary sized domain and is still collision-resistant.

A construction using a balanced binary tree allows simultaneous processing of multiple parts of data at a same level of the tree, reducing the running time to hash the message from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$ if we have $\mathcal{O}(n)$ processors [3, 5]. If we want to further reduce the amount of resources involved, we can use one of the following rescheduling techniques:

- Each processor is assigned the processing of a subtree (in the data structure sense) having $\log n$ leaves. There are approximatively $n/\log n$ such subtrees. The processing of the remaining ancestor nodes, at each remaining level of the tree, is distributed as fairly as possible between the processors. An example is depicted in Figure 1.

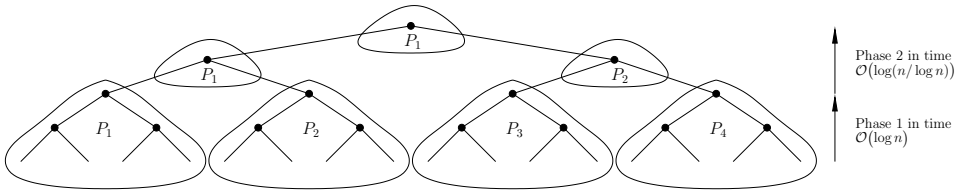


FIGURE 1. Example of the computation of the root node in $\mathcal{O}(\log n)$ time using $\mathcal{O}(n/\log n)$ processors. The message to hash is of size $n = 16$. In Phase 1, the computation of each hash subtree containing $4 = \log_2 16$ leaves is assigned to each processor. The first subtree is assigned to processor P_1, the second one to processor P_2, the third one to processor P_3 and the last one to processor P_4. A fine-grained allocation is then performed in Phase 2.

- An alternative solution is, at each level of the tree, to distribute as fairly as possible the node computations among $\mathcal{O}(n/\log n)$ processors.

The number of processors is then reduced by a factor $\log n$ and the asymptotic running time is conserved (with, nevertheless, a multiplicative factor 2). In this paper we are not interested in tradeoffs between the amount of used resources and the running time but instead we study optimal algorithms in finite distance. More precisely, we determine the hash tree structures which give the best concrete (parallel) time complexity for finite message lengths.

A tree structure is notably used in parallel hashing modes of Skein [6], Blake2 [7] or MD6 [8]. To give some examples, Skein uses a tree whose topology is controlled by the user thanks to three parameters: the arity of base level nodes which is a power of two; the arity of other inner nodes, which is also a power of two, and a last parameter limiting the height of the tree. MD6 uses a full (but not necessarily

perfect) quaternary tree, in the sense that an inner node has always four children. Some fictive leaves or nodes padded with 0 are added so that a rightmost node has the correct number of children. Like Skein, MD6 offers a parameter which serves to limit the height of the tree.

Some proposals [9, 10, 11] consider that a tree covering all the message blocks is not a good thing, because the number of processors should not grow with the size of the message. For instance, the domain extension parallel algorithm from Sarkar et al. [9, 10, 11] uses a perfect binary tree of processors, of fixed size. This perfect binary tree of compression/hash function calls can be seen as a big compression function, sequentially iterated over large parts of the message. In other words only the nodes computations performed in the tree can be done in parallel. The number of usable processors is a system parameter chosen by the issuer of the cryptographic form when hashing the message. The value of this parameter has to be reused by the recipients, for instance when verifying a signature. Thus, this one limits the scalability and the potential speedup. In this paper we consider that the scalability and the potential speedup should be independent of the characteristics (configuration) of the transmitting computer.

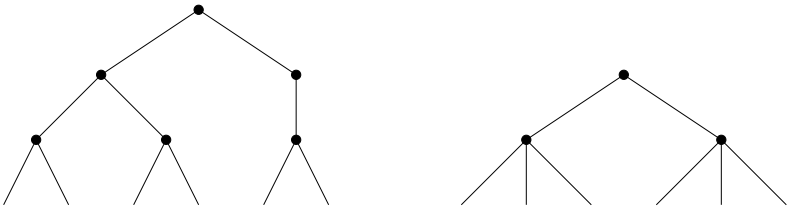
Bertoni *et al.* [12, 13] give sufficient conditions for a tree based hash function to ensure its indifferentiability from a random oracle. They propose several tree hashing modes for different usages. For example we can make use of a tree of height 2, defined in the following way: we divide the message in as many parts (of roughly equal size) as there are processors so that each processor hashes each part, and then the concatenation of all the results is sequentially hashed by one processor. To divide the message in parts of roughly equal size, the algorithm needs to know in advance the size of the message. Bertoni *et al.* use an idea from Gueron [14] to propose a variant which still makes use of a tree having two levels and a fixed number of processors, but this one interleaves the blocks of the message. This interleaving has several advantages. It allows an efficient parallel hashing of a streamed message, a roughly equal distribution of the data processed by each processor in the first level of tree (without prior knowledge of the message size), and finally a correct alignment of the data in the processors' registers. This kind of solution is suitable for multithreaded and SIMD implementations. In this paper we study theoretically optimal speedups, and, as a consequence, the message to hash is supposed to be already available.

The aim of this work is to show that we can improve the performance of a hash tree mode of operation by reworking the tree structured circuit topology. In particular, we are interested in minimizing the depth (parallel time) of the circuit and the width (number of processors involved). To the best of our knowledge, it is the first time that the problem of optimizing hash trees is addressed. The main interest of this paper is the methodology provided. The results are the followings:

- The first result is an algorithm which optimizes the tree topology in order to decrease at best the depth. We first show that a node arity greater than 5 is not possible and then we prove that we can construct such an optimal tree using exclusively levels of arity 2 and 3.

- Without affecting this depth, we show that we can change the corresponding tree topology in order to decrease at best the width. This width is optimal for trees having all their leaves at the same level. In particular, we show that for some message lengths l, the width can be decreased to $\lceil l/5 \rceil$.
- Observations are made about trees having their leaves at different levels, indicating that if our previous algorithm does not produce optimal solutions for this kind of trees, it probably produces near-optimal solutions.
- For trees having all their leaves at the same level, we also provide an algorithm which optimizes the number of processors at each step of the hash computation.

Suppose that the processing of one block of the message by the compression function costs one unit of time. A binary tree is not necessarily the structure which gives the best running time. Figure 2 shows two different tree topologies for hashing a 6-block message. The binary tree depicted in (2a) gives a (parallel) running time of 6 units while the rightmost one with a different arity at each level, depicted in (2b), gives a running time of 5 units. Furthermore, one may note that for messages of length less that 5 blocks, the use of the topology (2a) has no utility compared to a purely sequential mode (*i.e.* a completely degenerated binary tree).



(A) Non optimal tree (B) Optimal tree

FIGURE 2. Tree hashing with a 6-block message. The hash tree on the left requires 2 units of time to process each level, while the one on the right requires 3 units of time to process the base level and 2 units of time to process the root node.

In what follows, we suppose the use of a multitude of different compression functions, namely $f_x : \{0,1\}^{xN} \to \{0,1\}^N$ for $x \geq 2$. For each node, the choice of the function to use to compress its children depends on their number x (the node arity). We also assume that a compression function which compresses x blocks of size N bits has a computational cost of x units. In other words, if we consider a tree of calls of this compression function, the computation of a node having k children (*i.e.* k blocks) has a cost of k units. For instance, the UBI compression function, used in the hash function family Skein [6], performs x calls to the tweakable block cipher Threefish to compress a data of length x blocks. Assuming a hash tree of height h and x_i the arity of level i (for $i = 1 \dots h$), we define the parallel running time to obtain the root node value as being $\sum_{i=1}^{h} x_i$.

The paper is organized in the following way. In Section 2 we give background information and definitions. In Section 3 we describe the approach to reduce at

best the running time of a hash function. Then, in Section 4, we give an algorithm to construct a hash tree topology which achieves the same optimal running time while requiring a near-optimal number of processors. We also show how we can use this algorithm to optimize the number of processors at each step of the hash computation. Finally, in Section 5, we conclude the paper and discuss future works.

2. Background information and definitions

Throughout this paper we use the convention[1] that a node is the result of a function called on a data composed of the node's children. We call a base level node a node at level 1 pointing to the leaves representing message data parts. The leaves (or leaf nodes) are then at level 0. We define the arity of a level in the tree as being the greatest node arity in this level.

A k-ary tree is a tree where the nodes are of arity at most k. For instance a tree with only one node of arity k is said to be a k-ary tree. A full k-ary tree is a tree where all nodes have exactly k children. A perfect k-ary tree is a full k-ary tree where all leaves have the same depth.

We also define other "refined" types of tree. We say that a tree is of arities $\{k_1, k_2, \ldots, k_n\}$ (we can call it a $\{k_1, k_2, \ldots, k_n\}$-aries tree) if it has n levels (not counting level 0) whose nodes at the first level are of arity at most k_1, nodes at level 2 are of arity at most k_2, and so on. We say that such a tree is full if all nodes at the first level have exactly k_1 children, all nodes at level 2 have exactly k_2 children, and so on. As before, we say that such a tree is perfect if it is full and if all the leaves are at the same depth.

3. Optimization of hash trees for parallel computing

3.1. **Minimizing the running time.** In order to optimize the running time of a tree mode, we make a certain degree of flexibility on the choices of node arities. We can note straightaway that allowing different node arities in a same level of the tree provides no efficiency gains. Worse, the running time may be less interesting since a tree level processing running time is bounded by the running time to process the node having the highest arity. This observation suggests that, in order to hope a reduction of the tree processing running time, node arities at the same level need to be set to the same value[2] while allowing arities to vary from one level to another. Therefore our strategy allows a different arity at each level of the tree.

Let us denote l the block-length of a message. The problem is to find a tree height h and integer arities x_1, x_2, ..., x_h such that $\sum_{i=1}^{h} x_i$ is minimized. Any solution to the problem must necessarily satisfy the following constraints:

$$\prod_{i=1}^{h} x_i \geq l \text{ and } \left(\prod_{i=1}^{h} x_i\right)/x_j < l \quad \forall\, j \in [\![1, h]\!]. \tag{1}$$

[1] This corresponds to the convention used to describe Merkle trees. An other convention is to define a node as being the list of inputs of a function and not its result.

[2] Except maybe the rightmost node which may be of smaller arity

A solution to this problem is a multiset of arities. First, we show that, in a non-asymptotic setting, a perfect ternary tree comes closer to optimality than a perfect binary tree. Then we examine the case of trees having different arities at each level.

First of all, we can start by considering the h and x_i (for $i = 1 \dots h$) as real numbers. Thus, we have to minimize the summation of x_i subject to the constraint that their product is l. We know that the minimum is reached when the x_i are equal to the same number, which we will denote x. So we have $x^h = l$, that is $x = l^{\frac{1}{h}}$. We must now determine h so that $hl^{\frac{1}{h}}$ is minimized. The calculation of a derivative shows that this minimum is reached for $h = \ln(l)$, which implies $x = e$. Consequently, we can wonder what is the best solution between a perfect binary tree and a perfect ternary tree. The comparison of these two cases is done in A and shows that beyond a certain message size l ($l = 2^{28}$), a perfect ternary tree gives a better running time than a perfect binary tree. In fact, as the present general study shows, a tree having different level arities can give better results.

Let us remind that node arities are not allowed to vary in a same level (same stage) of the tree. A level of the tree is said to be of arity a when all nodes at this level are of arity at most a. Given an optimal tree (in the sense of the running time) for hashing, we can ask what the possible arities are for its levels. We have the following Theorem:

Theorem 1. *For a hash tree whose running time is optimal, the followings hold:*

- *It can be comprised of levels of arity* 2*,* 3*,* 4*, or* 5*. Higher arities are not possible.*
- *It can be constructed using only levels of arity* 2 *and* 3*.*

Proof. We first show that levels of arity a with $a \geq 7$ lead to trees having a suboptimal running time. Indeed, any node of arity $a \geq 7$ can be replaced by a tree of arity 2 having a better running time. We simply have to note that $2\lceil \log_2 a \rceil < a$ for all $a > 6$, meaning that a a-ary tree of height 1 can be advantageously replaced by a binary tree of height $\lceil \log_2 a \rceil$. In contrast, for all nodes of arity a with $a \in [\![3, 6]\!]$ and for all $i \in [\![2, 5]\!]$ we have $i\lceil \log_i a \rceil \geq a$. Finally, a node of arity 6 can be replaced by a $\{3, 2\}$-aries tree, since $2 \cdot 3 = 6$, thereby reducing the running time to $2 + 3 = 5$ units. As regards the second assertion, a node of arity 5 can be replaced by a tree of arities $\{3, 2\}$, since $2 \cdot 3 = 6 > 5$. This transformation does not change the running time since $2 + 3 = 5$. Finally, a node of arity 4 can be replaced by a binary tree of height 2 for a running time which is still unchanged. □

An optimal tree has not necessarily a single topology. Firstly, a solution satisfying constraints (1) can be defined as a multiset of arities since we can permute them. For instance, suppose a tree has three levels with the first level of arity 3, the second one of arity 2 and the last one (that is, the root node) of arity 3. We can permute these arities so that the first level is of arity 2 and the latter two levels of arity 3. If this new tree has the same running time, its topology has however changed. Secondly, we can find examples where different multiset of arities lead to trees of optimal running time. For instance, if we consider a 7-block message, the multisets of arities $\{2, 2, 2\}$, $\{3, 3\}$ and $\{4, 2\}$ allow the construction of trees having the

optimal running time. We can, however, construct optimal trees by restricting the set of possible arities. We have the following theorem:

Theorem 2. *Let a message of length l blocks and let i be the lowest integer such that $3^i \geq l$. Let us note $x \in [\![0, 2]\!]$ the value which minimizes the product $3^{i-x}2^x$ under the constraint $3^{i-x}2^x \geq l$. There exists an optimal tree (in the sense of optimal running time) which has $i-x$ levels of arity 3 and x levels of arity 2. More precisely, we can state the followings:*

- *If $l \leq 3^i < \frac{3l}{2}$ then a ternary hash tree is optimal for a running time of $3i$.*
- *If $\frac{3l}{2} \leq 3^i < \frac{9l}{4}$ then an optimal hash tree has $i-1$ levels of arity 3 and one level of arity 2, for a running time of $2 + 3(i-1)$.*
- *Otherwise $\frac{9l}{4} \leq 3^i < 3l$, and then an optimal hash tree has $i-2$ levels of arity 3 and 2 levels of arity 2, for a running time of $4 + 3(i-2)$.*

Such an optimal tree maximizes the number of levels of arity 3.

Proof. If we have at least 3 levels of arity 2 then we can replace these 3 levels by 2 levels of arity 3 ($3^2 = 9 > 2^3 = 8$). The running time to process 3 levels of arity 2 or 2 levels of arity 3 is 6. Therefore, it is always possible to construct optimal trees with at most 2 levels of arity 2. The three assertions follow immediately. □

Remark 1. Let i be such that $3^i \geq l$. Its is not difficult to see that the sought solution corresponds to the highest value $x \in [\![0, 2]\!]$ such that $3^{i-x}2^x \geq l$.

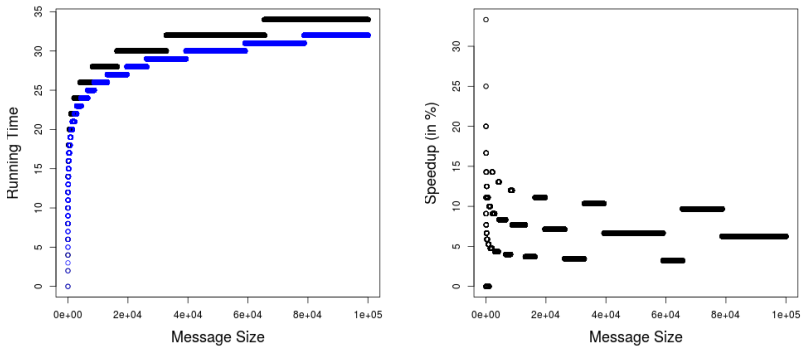
Algorithm 1. To determine the levels arities of an optimal tree, we first compute $i = \lceil \log l / \log 3 \rceil$ and then $x = \left\lfloor \log(l/3^i) / \log(2/3) \right\rfloor$. The $i-x$ first levels are of arity 3 and the last x levels of arity 2.

Examples. For messages of lengths $l = 4, 5$ and 10 blocks respectively, Algorithm 1 returns the multisets of arities $\{2, 2\}$, $\{3, 2\}$ and $\{3, 2, 2\}$ respectively. The number of processors is not optimized here. This aspect is addressed in the following section.

Why is it possible to minimize the running time with a tree whose leaves are at the same depth? Let us suppose that we have, for a given message length, an optimal tree whose leaves are not at the same depth. Then, for each leaf located at a level greater than zero, we can create descendants in order to complete the tree so that all leaves are at level 0. It is possible to perform this while keeping a tree of same height and respecting the level arities. The result is a tree whose the number of leaves is greater than the message length (the tree is said to be *perfect* since, on the one hand, nodes at a same level are all of same arity, and, on the other hand, all the leaves are at the same depth). It is possible to prune some right branches to remove this surplus of leaves. Consequently, there exists necessarily a tree having the same height, the same multiset of arities and a lower number of leaves corresponding to the message length. In the rest of the paper, we refer to a truncated $(x_1, x_2, \ldots, x_h)$-aries tree to speak about a tree having a number of leaves equal to the message length and where the nodes of the base level are of arity at most x_1, nodes at the second level are of arity at most x_2 and so on.

As a last remark, since the hash function must be deterministic, the multiset of arities must also be chosen deterministically as a function of the message size. For instance, we can arrange in descending order the elements of the multiset of arities. The solution to the problem of minimizing the running time is then uniquely determined as an ordered multiset.

Performance improvements. We have seen that for a message of 6 blocks (see Figure (2)), the performance gain of an optimal tree compared to a binary tree is 20%. Figure (3a) shows the running times of an optimal tree and a binary tree as functions of the message size varying from 1 to 10^5 blocks. Figure (3b) shows the speed gain obtained with an optimal tree. The gain in time (or speedup gain) is computed as $100(T_b/T_o - 1)$ where T_b is the running time of a binary tree and T_o the running time of an optimal tree. As we can see, the gain differs from one message size to another. The gain can be greater than 30% for very short messages but decreases quickly, to cap at 10%. As regards the message size, although the diagram does not cover a sufficiently long range, one can note a slight downward slope.



(A) Running time of an optimal tree (shown in blue) compared to a binary tree (in black)

(B) Speed gain of an optimal tree compared to a binary tree

FIGURE 3. Performance comparison between an optimal tree and a binary tree

3.2. **Minimizing the number of processors.** In this section we look into how to reduce at best the number of required processors to obtain the optimal running time. We have two cases to study, the trees having all leaves at the same depth and the others. We fully treat the first case and we make a few observations regarding the second type of tree, which we intuitively sense to further reduce the number of required processors.

At the outset, one may be interested in the maximum possible number of levels of arity 5 or 4. We have the following Lemma:

Lemma 1. *In a tree having an optimal running time there can at most be* 1 *level of arity* 5 *and* 6 *levels of arity* 4*.*

Proof. Suppose that the tree has 2 levels of arity 5. We replace these 2 levels by 3 levels of arity 3 since $3^3 = 27 > 5^2 = 25$. The running time is improved since $3 \cdot 3 = 9 < 2 \cdot 5 = 10$. We can then state that 2 levels of arity 5 lead to a tree having a sub-optimal running time. Now, let us look for a pair of minimum integers (i, j) satisfying $3^i > 4^j$ and $3 \cdot i < 4 \cdot j$. The first pair which satisfies these constraints is $i = 9$ and $j = 7$. We can then replace 7 levels of arity 4 by 9 levels of arity 3 in order to decrease the running time. □

3.2.1. *Trees having all leaves at the same level.* We have seen that it is possible to construct a tree optimizing the running time by using only levels of arity 2 and 3. In what follows, we show how to deduce an optimal tree minimizing the number of involved processors. Let us suppose that level arities x_1, x_2 ..., x_h are noted in (no strictly) decreasing order so that x_1 is the arity of the base level and x_h the arity of the last level, *i.e.* the arity of the root node. The trees optimizing the running time, defined above, are not necessarily full in the sense that a rightmost node at a given level can be of arity strictly lower than the arity of this level. First, we note that for the trees constructed with Algorithm 1, the number of required processors is equal to $\lceil l/3 \rceil$ in the best case, and equal to $\lceil l/2 \rceil$ when there are only levels of arity 2. Moreover, according to Theorem 1 we know that a level arity cannot be greater than 5. This means that in the best case, after optimization, the number of required processors could be reduced to $\lceil l/5 \rceil$. Thus, we could in the best case decrease the number of processors by a factor of about $5/2$.

Given an optimal tree for the running time, the intent is to increase the arity of the first level (base level) while decreasing arities of the following levels so that the sum of the levels arities remains constant and their product remains greater than or equal to l. To solve this problem we propose in B two solutions (Algorithm 2a or 2b). However, as will be discussed below, we can further optimize hash trees.

According to Theorem 1, a level arity of a tree minimizing the running time cannot exceed 5. Thus, Algorithm 2a (or Algorithm 2b) of B allows us to substitute any sub-multiset A for another one, denoted A', whose the sum of arities remains the same, and by trying to increase the arity of the base level up to 5. Consider, for instance, a message of size $l = 95$ blocks. With such a message size, Algorithm 1 returns the multiset of arities $A_0 = \{3, 3, 3, 2, 2\}$ which defines a tree structure involving 32 processors. By applying Algorithm 2, we obtain the multiset $A_1 = \{4, 3, 3, 3\}$ which reduces the number of involved processors to 24 while leaving the running time unchanged.

What if the arity of each level is increased? As much as possible? We just saw that we can increase the arity of the first level. It would also be preferable to increase the arity of each level of the tree in order to free up the highest number of processors at each step of the computation. An example is depicted in Figure 4.

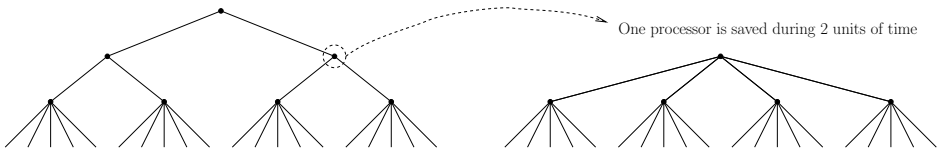


FIGURE 4. Two trees compressing a 20-block message, optimized both for the running time and the number of involved processors. Nevertheless, we note that the right tree is the best choice. Indeed, the one on the left needs 4 processors during 5 units of time, then 2 processors during 2 units of time, and finally one processor during 2 units of time. The one on the right needs 4 processors during 5 units of time and then one processor during 4 units of time.

While we propose an iterative algorithm in B to construct an optimal tree maximizing the arity of each level, we also enumerate all possible cases in the following Theorem:

Theorem 3. *For any integer $l \geq 2$ there is an unique ordered multiset A of h_5 arities 5, h_4 arities 4, h_3 arities 3 and h_2 arities 2 such that the corresponding tree covers a message size l, has a minimal running time and has first h_5 as large as possible, then h_4 as large as possible, and then h_3 as large as possible. More precisely, if i is the lowest integer such that $l \leq 3^i < 3l$, this ordered multiset is such that:*

$$\begin{cases}
|A| = i, h_5 = 0, h_4 = 0, h_3 = i, h_2 = 0, & \text{if } l \leq 3^i < \frac{9l}{8}, \\
|A| = i, h_5 = 0, h_4 = 1, h_3 = i-2, h_2 = 1, & \text{if } \frac{9l}{8} \leq 3^i < \frac{81l}{64}, \\
|A| = i-1, h_5 = 0, h_4 = 3, h_3 = i-4, h_2 = 0, & \text{if } \frac{81l}{64} \leq 3^i < \frac{27l}{20}, \\
|A| = i-1, h_5 = 1, h_4 = 1, h_3 = i-3, h_2 = 0, & \text{if } \frac{27l}{20} \leq 3^i < \frac{729l}{512}, \\
|A| = i-1, h_5 = 0, h_4 = 4, h_3 = i-6, h_2 = 1, & \text{if } \frac{729l}{512} \leq 3^i < \frac{3l}{2}, \\
|A| = i, h_5 = 0, h_4 = 0, h_3 = i-1, h_2 = 1, & \text{if } \frac{3l}{2} \leq 3^i < \frac{81l}{50}, \\
|A| = i-1, h_5 = 1, h_4 = 1, h_3 = i-4, h_2 = 1, & \text{if } \frac{81l}{50} \leq 3^i < \frac{27l}{16}, \\
|A| = i-1, h_5 = 0, h_4 = 2, h_3 = i-3, h_2 = 0, & \text{if } \frac{27l}{16} \leq 3^i < \frac{9l}{5}, \\
|A| = i-1, h_5 = 1, h_4 = 0, h_3 = i-2, h_2 = 0, & \text{if } \frac{9l}{5} \leq 3^i < \frac{243l}{128}, \\
|A| = i-1, h_5 = 0, h_4 = 3, h_3 = i-5, h_2 = 1, & \text{if } \frac{243l}{128} \leq 3^i < \frac{2187l}{1024}, \\
|A| = i-2, h_5 = 0, h_4 = 5, h_3 = i-7, h_2 = 0, & \text{if } \frac{2187l}{1024} \leq 3^i < \frac{9l}{4}, \\
|A| = i-1, h_5 = 0, h_4 = 1, h_3 = i-2, h_2 = 0, & \text{if } \frac{9l}{4} \leq 3^i < \frac{81l}{32}, \\
|A| = i-1, h_5 = 0, h_4 = 2, h_3 = i-4, h_2 = 1, & \text{if } \frac{81l}{32} \leq 3^i < \frac{27l}{10}, \\
|A| = i-1, h_5 = 1, h_4 = 0, h_3 = i-3, h_2 = 1, & \text{if } \frac{27l}{10} \leq 3^i < \frac{729l}{256}, \\
|A| = i-2, h_5 = 0, h_4 = 4, h_3 = i-6, h_2 = 0, & \text{if } \frac{729l}{256} \leq 3^i < 3l,
\end{cases}$$

where the number h_3 is at least 1 in the first case and can be 0 in the other cases.

Proof. Let us start from the 3 cases of Theorem 2 which maximize the number of levels of arity 3. For a given message length l, we consider the corresponding optimal tree (in the sense of the running time). We denote by a the initial number of levels of arity 2 and by $i-a$ the initial (maximized) number of levels of arity 3. We want to transform this tree in order to increase the arity of each level as much as possible, while leaving the running time unchanged. According to Lemma 1, there are at most one level of arity 5 and at most six levels of arity 4. Since we want to maximize the number of levels of arity 4 after having maximized the number of levels of arity 5, there cannot be more than one level of arity 2. Thus, $h_5 \in [\![0,1]\!]$, $h_4 \in [\![0,6]\!]$ and $h_2 \in [\![0,1]\!]$, meaning there shall be at most 28 cases. Note that among these 28 cases, many may not be valid solutions. The aim is to transform the initial product $2^a 3^{i-a}$ into a product $2^w 3^{i-a-b} 4^v 5^u$ where b is the number of levels of arity 3 that we have transformed and u, v, w the number of levels of arity 5, 4, 2 respectively. For each triple $(h_5 = u, h_4 = v, h_2 = w)$ with $u \in [\![0,1]\!]$, $v \in [\![0,6]\!]$ and $w \in [\![0,1]\!]$, we can easily verify that there is an unique solution (a,b) with a an integer in $[\![0,2]\!]$ and b a positive integer such that $3b+2a = 5u+4v+2w$. Indeed, let suppose a second solution (a',b'). Since $3b+2a = 3b'+2a'$, we have $3(b'-b) = 2(a'-a)$, meaning that 3 divides $(a'-a)$. This is impossible, unless $a' = a$. Such a solution must satisfy $3^{i-a-b}2^w 4^v 5^u \geq l$, that is $3^i \geq 3^{a+b} l/(2^w 4^v 5^u)$. According to Theorem 2, we have $(3/2)^a l \leq 3^i < \min(3l, (3/2)^{a+1} l)$. Consequently, if we have

$$\frac{3^{a+b} l}{2^w 4^v 5^u} > \min\left(3l, \left(\frac{3}{2}\right)^{a+1} l\right),$$

$$\text{where } \min\left(3l, \left(\frac{3}{2}\right)^{a+1} l\right) = \begin{cases} 3l & \text{if } a = 2 \\ (3/2)^{a+1} l & \text{if } a = 0, 1 \end{cases},$$

this solution does not exist. Among the 28 cases, we observe that 13 of them are not valid solutions. Thus, we have 15 solutions, denoted (u,v,w,a,b), for which we compute and sort the values $3^{a+b} l/(2^w 4^v 5^u)$ so that we can establish their domains of validity. □

For the purpose of minimizing the number of processors at each step of the computation, we apply Theorem 3. There are 15 possible cases and we would like to estimate their distribution. The following theorem helps us to calculate the proportions:

Theorem 4. *Let a message size l drawn uniformly at random from the set $[\![2, L]\!]$ where L is a fixed positive integer. Let $k = \lceil \log_3 L \rceil$ and α, β two real numbers such that $1 \leq \alpha < \beta \leq 3$. The probability that $3^{\lceil \log_3 l \rceil}$ is in the interval $[\alpha l, \beta l[$ is equal to*

$$P(E) = \frac{1}{L-1}\left(\sum_{i=1}^{k-1}\left(\left\lfloor \frac{3^i}{\alpha} \right\rfloor - \left\lfloor \frac{3^i}{\beta} \right\rfloor\right) + \mu\right),$$

where

$$\mu = \begin{cases} 0 & \text{if} \quad L \leq \dfrac{3^k}{\beta}, \\ \left(\min\left(L, \left\lfloor \frac{3^k}{\alpha} \right\rfloor\right) - \left\lfloor \frac{3^k}{\beta} \right\rfloor\right) & \text{if} \quad L > \dfrac{3^k}{\beta}. \end{cases}$$

Proof. Let E be the event: "l is such that $3^{\lceil \log_3 l \rceil} \in [\alpha l, \beta l[$". For any i such that $1 \leq i \leq k-1$ let E_i be the event: "l is such that $3^{i-1} < l \leq 3^i$" and E_k the event: "l is such that $3^{k-1} < l \leq L$". The conditional probability $P(E \mid E_i)$ is given by the following:

(1) Case $i \leq k-1$. The event E is realized if and only if $\alpha l \leq 3^i < \beta l$, namely if and only if

$$\frac{3^i}{\beta} < l \leq \frac{3^i}{\alpha}.$$

As l must be an integer, this condition is equivalent to

$$l \in \left[\!\left[\left\lfloor \frac{3^i}{\beta} \right\rfloor, \left\lfloor \frac{3^i}{\alpha} \right\rfloor \right]\!\right].$$

Hence

$$P(E \mid E_i) = \frac{1}{3^i - 3^{i-1}} \times \left(\left\lfloor \frac{3^i}{\alpha} \right\rfloor - \left\lfloor \frac{3^i}{\beta} \right\rfloor \right).$$

(2) Case $i = k$.

(a) If $L \leq \dfrac{3^k}{\beta}$ then $P(E \mid E_i) = 0$.

(b) If $L > \dfrac{3^k}{\beta}$ then

$$P(E \mid E_i) = \frac{1}{L - 3^{k-1}} \times \left(\min\left(L, \left\lfloor \frac{3^k}{\alpha} \right\rfloor\right) - \left\lfloor \frac{3^k}{\beta} \right\rfloor \right).$$

As the E_i are disjoint, we can compute $P(E)$ by the following formula:

$$P(E) = \sum_{i=1}^{k} P(E_i) P(E|E_i),$$

which gives the expected result. □

Remark 2. In the previous theorem we can write

$$\left\lfloor \frac{3^i}{\alpha} \right\rfloor - \left\lfloor \frac{3^i}{\beta} \right\rfloor = 3^i \left(\frac{1}{\alpha} - \frac{1}{\beta} \right) + u_i$$

where $|u_i| \leq 1$. Then

$$P(E) = \frac{1}{L-1} \left(\frac{3}{2} \left(\frac{1}{\alpha} - \frac{1}{\beta} \right) \left(3^{k-1} - 1 \right) + \sum_{i=1}^{k-1} u_i + \mu \right). \tag{2}$$

Let us now consider trees whose the number of leaves is equal to the message length. Having a multiset of arities arranged in descending order, that we denote $A = \{x_1, x_2, \ldots, x_{|A|}\}$, the number of nodes of level i is $\lceil l/(x_1 x_2 \ldots x_i) \rceil$. One important thing is the number of nodes of the base level. We have the following Theorem:

Theorem 5. *Let the message size be $l \geq 2$ and let i be the lowest integer such that $3^i \geq l$. The number of processors required to process such a message is:*

- $\lceil l/3 \rceil$ *if* $3^i \in [l, \frac{9l}{8}[\cup [\frac{3l}{2}, \frac{81l}{50}[$,
- $\lceil l/4 \rceil$ *if* $3^i \in [\frac{9l}{8}, \frac{27l}{20}[\cup [\frac{729l}{512}, \frac{3l}{2}[\cup [\frac{27l}{16}, \frac{9l}{5}[\cup [\frac{243l}{128}, \frac{27l}{10}[\cup [\frac{729l}{256}, 3l[$,
- $\lceil l/5 \rceil$ *if* $3^i \in [\frac{27l}{20}, \frac{729l}{512}[\cup [\frac{81l}{50}, \frac{27l}{16}[\cup [\frac{9l}{5}, \frac{243l}{128}[\cup [\frac{27l}{10}, \frac{729l}{256}[$.

Proof. These results follow immediately from Theorem 3. □

The following theorem gives the proportions of messages sizes for which $\lceil l/c \rceil$ processors (with $c = 3, 4, 5$) are required:

Theorem 6. *Let a message size $l \geq 2$ bounded by $L = 3^j$ be drawn randomly. When j tends to infinity, the number of required processors is $\lceil l/3 \rceil$ with probability approaching $13/54$ ($\approx 24\%$), $\lceil l/4 \rceil$ with probability approaching $16/27$ ($\approx 59\%$), and $\lceil l/5 \rceil$ with probability approaching $1/6$ ($\approx 17\%$).*

Proof. Remark that if $L = 3^j$, Formula (2) becomes

$$P(E) = \frac{1}{L-1}\left(\frac{3}{2}\left(\frac{1}{\alpha} - \frac{1}{\beta}\right)\left(3^j - 1\right) + \sum_{i=1}^{j} u_i\right). \tag{3}$$

Now we apply Theorem 4 and Remark 2 with α and β given by the intervalls occuring in Theorem 5. These α and β are of the form $\dfrac{3^s}{u}$ where $s \leq 6$ and u integer. Then for $i \geq 6$ the numbers $\dfrac{3^i}{\alpha}$ and $\dfrac{3^i}{\beta}$ are integers. Thus, for $i \geq 6$ we have $u_i = 0$ and the following formula holds:

$$\begin{aligned} P(E) &= \frac{1}{L-1}\left(\frac{3}{2}\left(\frac{1}{\alpha} - \frac{1}{\beta}\right)\left(3^j - 1\right) + \sum_{i=1}^{5} u_i\right) \\ &= \frac{3}{2}\left(\frac{1}{\alpha} - \frac{1}{\beta}\right) + \frac{1}{L-1}\sum_{i=1}^{5} u_i. \end{aligned}$$

When j tends to infinity, $P(E)$ has a limit which is

$$\frac{3}{2}\left(\frac{1}{\alpha} - \frac{1}{\beta}\right).$$

This formula applied to the intervalls given in Theorem 5 gives the expected results. □

Remark 3. When L approaches infinity, the approached proportions for the fifteen cases of Theorem 3 can be estimated similarly. These proportions are depicted in Figure 5.

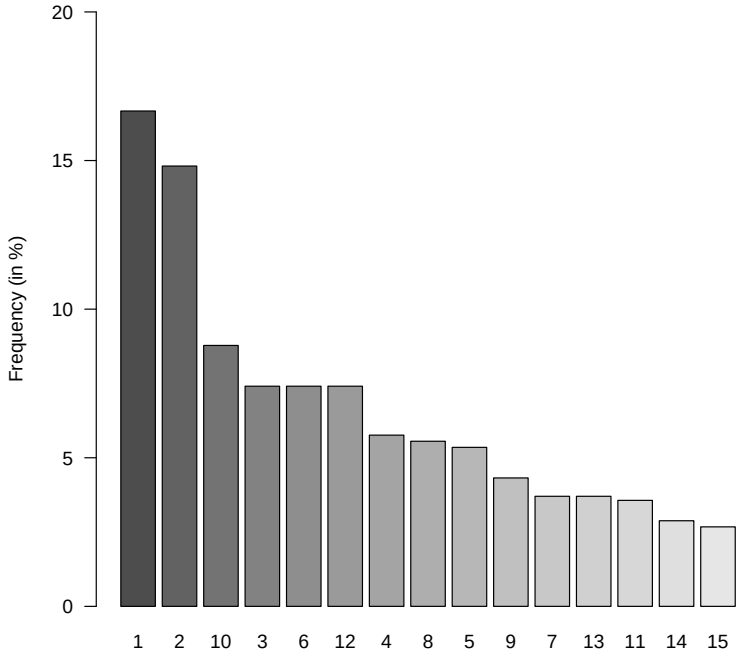


FIGURE 5. Proportions for the fifteen cases of Theorem 3. The bars are drawn in decreasing order of frequency.

An other important thing is the minimization of the amortized number of processors per unit of time (for the processing of a single message). Since we are interested in hash trees having an optimal running time, we only need to concern ourselves with the number of inner nodes of the tree. We first apply Theorem 3 which gives, for a message size l, a tree having a minimal running time and using a minimal number of processors at each step of the computation. For a perfect $(x_1, x_2, \ldots, x_h)$-aries tree constructed thanks to this theorem, the total number of inner nodes is:

$$N_l = 1 + x_h + x_h x_{h-1} + x_h x_{h-1} x_{h-2} + \cdots + x_h x_{h-1} \ldots x_2.$$

We notive that $\lceil \lceil \cdots \lceil \lceil l/x_1 \rceil / x_2 \rceil \cdots \rceil / x_i \rceil = \lceil l/(x_1 x_2 \cdots x_i) \rceil$ for (strictly) positive integer $(x_j)_{j=1\ldots i}$. For a $(x_1, x_2, \ldots, x_h)$-aries truncated tree constructed thanks to this theorem, the total number of inner nodes is:

$$N_{tr,l} = \lceil l/x_1 \rceil + \lceil l/(x_1 x_2) \rceil + \cdots + \lceil l/(x_1 x_2 \ldots x_h) \rceil.$$

For the trees produced by Theorem 2, we denote by $N'_{tr,l}$ the number of inner nodes of a truncated tree and by N'_l the number of inner nodes of a perfect tree.

Considering truncated trees, the expected gain when using Theorem 3 is

$$G = \frac{N'_{tr,l}}{N_{tr,l}}.$$

Let us denote x_1, x_2, ..., x_h the level arities of the optimal tree given by Theorem 3. We want to give a lower bound for the cases of this theorem. To achieve this, we first need to upperbound $N_{tr,l}$ by the number of inner nodes of a perfect $\{x_1, \ldots, x_h\}$-aries tree, that we denote N_l. Then, we need a sufficiently tight lower bound for $N'_{tr,l}$. Let us now consider the optimal solution of Theorem 2, denoted (i, a), where $i = \lceil \log_3 l \rceil$ is the height of the tree and a is the number of levels of arity 2. We assume that this is this solution which produces $N'_{tr,l}$ inner nodes in the tree. We have to remark that trees produced by this theorem 2 can be arranged in increasing order of running time:

- if $a = 2$ then $(i' = i - 1, a' = 0)$ is the preceding (non optimal) solution,
- if $a = 1$ then $(i' = i, a' = 2)$ is the preceding (non optimal) solution,
- if $a = 0$ then $(i' = i, a' = 1)$ is the preceding (non optimal) solution.

One can easily verify that for $l' = 3^{i'-a'}2^{a'}$, the number N_{sub} of inner nodes of the non optimal tree of parameters (i, a) and having exactly l' leaves is greater than or equal to $N'_{l'}$. Since $l > l'$, we necessarily have $N'_{tr,l} \geq N_{sub}$. Thus, we have:

$$N'_{tr,l} \geq N'_{l'},$$

and we can lower bound G as:

$$G \geq \max\left(\frac{N'_{l'}}{N_l}, 1\right),$$

where

$$l' = 3^{i-\lfloor (a+1)/3 \rfloor - ((a+1) \bmod 3)} 2^{(a+1) \bmod 3}.$$

3.2.2. *Other balanced trees.* If we can move up (lift up) some leaves in the tree so that all leaves are not at the same depth, then we can reduce the number of processors. Let us suppose that we merely use full binary trees. Whatever the message length is, it is always possible to construct a full binary tree. This allows a reduction of the number of processors. Indeed, let us consider a compression function f taking as input two blocks and returning one block. If l is the number of blocks of the message, we compute the root node in $l - 1$ evaluations of the function f as follows: the blocks of the message are paired consecutively and f is applied on each pair. The possibly remaining block (if l is odd) is not processed. We then consider the list of resulting blocks by f with the possibly remaining block and we repeat the process again and again until there is a single remaining block. The height of the resulting tree is $\lceil \log_2(l) \rceil$ and the number of saved processors is $\lceil l_f/2 \rceil$ where l_f is the number of leaves located at a level greater than 0. Note that in the best case $l_f = l_L/4$ where l_L is the number of leaves of the largest perfect binary subtree.

Let us consider our variant of the definition of a full tree, when arities differ from one level to another. The question that arises is: is it always possible to construct a full tree minimizing the number of processor for an optimal running time ? The

answer is no. To show that it suffices to exhibit an example in which a tree which is not full further minimizes the number of processors compared to a full tree. For the sake of concreteness, let us take the example of a message of length 26 blocks. The optimal running time for such a message length is 9 and the only multiset of arities which allows deriving it is $\{3, 3, 3\}$. It can be noted that a rightmost node at the base level is of arity 2, showing that the minimum cannot be obtained with a full tree.

If we consider trees without any structural constraint, there is still scope for reducing the number of processors, although marginally. Let us take a message of 56 blocks. An optimal set of arities for a tree with all leaves at the same level is $\{5, 4, 3\}$. We can note that this tree has a rightmost node at the base level having only one child. This child can take the place of its parent node in order to save one more processor. We also notice that the resulting tree is full in the sense of our new definition.

4. Conclusion

In this paper, we have shown, for a given message length, how to construct a hash tree minimizing the running time. For a hash tree having its leaves at the same depth, we have shown how to decrease at best the number of processors allowing such a minimized running time. We have also seen that it is possible to slightly decrease the number of processors by considering other types of trees. Analysis on few small message sizes have revealed that, in the best case, we can save one more processor by using a tree which does not have all its leaves at the same depth. Further work is necessary to adequately specify to what extent the amount of resources can actually be decreased.

References

[1] S. Gueron, V. Krasnov, Parallelizing message schedules to accelerate the computations of hash functions, J. Cryptographic Engineering 2 (4) (2012) 241–253.

[2] S. Gueron, V. Krasnov, Simultaneous hashing of multiple messages, J. Information Security 3 (4) (2012) 319–325.

[3] I. Damgård, A design principle for hash functions, in: CRYPTO '89: Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology, Springer-Verlag, London, UK, 1990, pp. 416–427.

[4] R. C. Merkle, Secrecy, authentication, and public key systems., Ph.D. thesis, Stanford, CA, USA (1979).

[5] R. C. Merkle, Protocols for public key cryptosystems, in: Proceedings of the 1980 IEEE Symposium on Security and Privacy, 1980, pp. 122–134.

[6] N. Ferguson, S. L. Bauhaus, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, J. Walker, The skein hash function family (version 1.2) (2009).

[7] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, C. Winnerlein, Blake2: Simpler, smaller, fast as md5, in: Proceedings of the 11th International Conference on Applied Cryptography and Network Security, ACNS'13, Springer-Verlag, Berlin, Heidelberg, 2013, pp. 119–135.

[8] R. L. Rivest, B. Agre, D. V. Bailey, C. Crutchfield, Y. Dodis, K. Elliott, F. A. Khan, J. Krishnamurthy, Y. Lin, L. Reyzin, E. Shen, J. Sukha, D. Sutherland, E. Tromer, Y. L. Yin, The md6 hash function: A proposal to nist for sha-3 (2008).

[9] P. Sarkar, P. J. Schellenberg, A parallel algorithm for extending cryptographic hash functions, in: Progress in Cryptology - INDOCRYPT 2001, Second International Conference on Cryptology in India, Chennai, India, December 16-20, 2001, Proceedings, 2001, pp. 40–49.
[10] P. Sarkar, P. J. Schellenberg, A parallelizable design principle for cryptographic hash functions, IACR Cryptology ePrint Archive 2002 (2002) 31.
[11] P. Pal, P. Sarkar, PARSHA-256- - A new parallelizable hash function and a multithreaded implementation, in: Fast Software Encryption, 10th International Workshop, FSE 2003, Lund, Sweden, February 24-26, 2003, Revised Papers, 2003, pp. 347–361.
[12] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, Sufficient conditions for sound tree and sequential hashing modes, Cryptology ePrint Archive, Report 2009/210 (2009).
[13] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, Sakura: A flexible coding for tree hashing 8479 (2014) 217–234.
[14] S. Gueron, Parallelized hashing via j-lanes and j-pointers tree modes, with applications to SHA-256, IACR Cryptology ePrint Archive 2014 (2014) 170.

APPENDIX A. COMPARISON BETWEEN A PERFECT BINARY TREE AND A PERFECT TERNARY TREE

Let $l \geq 2$ an integer. Let h_2 the lowest integer such that $2^{h_2} \geq l$ and h_3 the lowest integer such that $3^{h_3} \geq l$. We assume that we use a perfect binary (or ternary) tree as in the original Merkle (and Damgård) hash tree mode, *i.e.* the message is padded to obtain a message size which is a power of 2 (or 3). The problem is to compare $2h_2$ and $3h_3$.

Any l can be uniquely written

$$l = 2^k + u,$$

where u is an integer such that $0 \leq u < 2^k$. Then

$$l = 2^k(1 + a) \text{ where } a = \frac{u}{2^k}.$$

If $a = 0$ then $h_2 = k$ else $h_2 = k + 1$.

A.1. **The case $a = 0$.** In this case

$$l = 2^k, h_2 = k, h_3 = \left\lceil \frac{k \log(2)}{\log(3)} \right\rceil.$$

Then

$$h_3 = \frac{k \log(2)}{\log(3)} + \alpha,$$

where $0 < \alpha < 1$. We must compare $3h_3$ with $2h_2$, namely

$$3\frac{k \log(2)}{\log(3)} + 3\alpha \text{ with } 2k,$$

or

$$3\frac{\log(2)}{\log(3)} + 3\frac{\alpha}{k} \text{ with } 2.$$

As α is bounded by 1 and $3\dfrac{\log(2)}{\log(3)} < 2$, for k sufficiently large we have $3h_3 < 2h_2$. More precisely if $k \geq 28$ then $3h_3 < 2h_2$, meaning that a perfect ternary tree gives

a better running time than a perfect binary tree. When $2 \leq k \leq 27$, we compute the 27 values

$$T = \frac{3}{k}\left\lceil \frac{k\log(2)}{\log(3)} \right\rceil - 2$$

and we look at the sign of the result:

- For $k = 3s$ ($s = 1, \cdots, 9$), a perfect binary tree and a perfect ternary tree give the same result ($T = 0$).
- For $k = 11, 14, 17, 19, 20, 22, 23, 25, 26$, a perfect ternary tree is better ($T < 0$).
- For $k = 2, 4, 5, 7, 8, 10, 13, 16$, a perfect binary tree is better ($T > 0$).

A.2. **The case $a \neq 0$.** In this case $h_2 = k + 1$ and

$$h_3 = \left\lceil \frac{k\log(2)}{\log(3)} + \frac{\log(1+a)}{\log(3)} \right\rceil.$$

We must compare $3h_3$ to $2h_2$. But:

$$\frac{3h_3}{k} \leq \frac{3\log(2)}{\log(3)} + \frac{3\log(2)}{k\log(3)} + \frac{3}{k}$$

and

$$\frac{2h_2}{k} = 2 + \frac{2}{k}.$$

As $\dfrac{3\log(2)}{\log(3)} < 2$, for k sufficiently large we have $3h_3 < 2h_2$. More precisely for $k \geq 27$ then $3h_3 < 2h_2$, meaning that a perfect ternary tree gives a better running time than a perfect binary tree. For any $2 \leq k \leq 26$ and any u such that $1 \leq u < 2^k$ we must compute the sign of

$$R = 3\left\lceil \frac{k\log(2)}{\log(3)} + \frac{\log\left(1 + \frac{u}{2^k}\right)}{\log(3)} \right\rceil - 2k - 2.$$

As R is an increasing function of u, it is sufficient to determine for any $k < 27$ the value of u where the sign changes. This can be done by dichotomy. Results are in Table 1.

APPENDIX B. ALGORITHMS FOR REDUCING THE NUMBER OF PROCESSORS

B.1. **Reducing the number of processors at the base level.** We propose two (different) algorithms to construct an optimal tree (in the sense of the running time) which covers exactly l blocks (the tree is not necessarily perfect) and increases as much as possible the arity of the base level. The first solution consists to check if there exists an optimal tree having a level of arity 5 or 4.

$k = 2$	$Sign = 0$ for any u
$k = 3$	$Sign < 0$ for $u = 1$ and $Sign > 0$ for $u > 1$
$k = 4$	$Sign < 0$ for $u \leq 11$ and $Sign > 0$ for $u > 11$
$k = 5$	$Sign = 0$ for any u
$k = 6$	$Sign < 0$ for $u \leq 17$ and $Sign > 0$ for $u > 17$
$k = 7$	$Sign < 0$ for $u \leq 115$ and $Sign > 0$ for $u > 115$
$k = 8$	$Sign = 0$ for any u
$k = 9$	$Sign < 0$ for $u \leq 217$ and $Sign > 0$ for $u > 217$
$k = 10$	$Sign < 0$ for any u
$k = 11$	$Sign < 0$ for $u \leq 139$ and $Sign = 0$ for $u > 139$
$k = 12$	$Sign < 0$ for $u \leq 2465$ and $Sign > 0$ for $u > 2465$
$k = 13$	$Sign < 0$ for any u
$k = 14$	$Sign < 0$ for $u \leq 3299$ and $Sign = 0$ for $u > 3299$
$k = 15$	$Sign < 0$ for $u \leq 26281$ and $Sign > 0$ for $u > 26281$
$k = 16$	$Sign < 0$ for any u
$k = 17$	$Sign < 0$ for $u \leq 46075$ and $Sign = 0$ for $u > 46075$
$k = 18$	$Sign < 0$ for any u
$k = 19$	$Sign < 0$ for any u
$k = 20$	$Sign < 0$ for $u \leq 545747$ and $Sign = 0$ for $u > 545747$
$k = 21$	$Sign < 0$ for any u
$k = 22$	$Sign < 0$ for any u
$k = 23$	$Sign < 0$ for $u \leq 5960299$ and $Sign = 0$ for $u > 5960299$
$k = 24$	$Sign < 0$ for any u
$k = 25$	$Sign < 0$ for any u
$k = 26$	$Sign < 0$ for $u \leq 62031299$ and $Sign = 0$ for $u > 62031299$

TABLE 1. Comparison between a perfect binary tree and a perfect ternary tree. If $Sign < 0$ a perfect ternary tree has a better running time. If $Sign = 0$ the two trees give the same running time. Otherwise a perfect binary tree is better.

Algorithm 2a. This algorithm takes as inputs a message length l, a multiset of arities (arranged in descending order) minimizing the running time, denoted $A = \{x_1, x_2, ..., x_{|A|}\}$, and returns a multiset of arities (still sorted in descending order) minimizing the number of processors while leaving unchanged the running time. Let t_l the optimal running time for a message of size l, *i.e.* the sum of arities of A. The algorithm proceeds as follows:

(1) Use Algorithm 1 to construct a tree for a message length $l' = \lceil l/5 \rceil$ and denote by A' the corresponding ordered multiset of arities. If $t_l = t_{l'} + 5$ then return the multiset $A'' = \{5, A'\}$, otherwise go to the following step.
(2) Use Algorithm 1 to construct a tree for a message length $l' = \lceil l/4 \rceil$ and denote by A' the corresponding ordered multiset of arities. If $t_l = t_{l'} + 4$ then return the multiset $A'' = \{4, A'\}$, otherwise go to the following step.

(3) Return A (which cannot be further optimized).

The second approach uses the following hints:

Hints. Let us note that if $k > 0$, then $a > b \iff (a-k)b > a(b-k)$. Moreover, if $b \leq a$ then $(b-1)(a+1) \leq ab$. This suggests that a product of several numbers, whose the sum is constant, is maximized when these numbers are as close together as possible. In order to decrease the product of arities as slowly as possible we use the fact that if $c \geq b \geq a$ we have $(c+1)(b-1)a \geq (c+1)b(a-1)$.

Algorithm 2b. This algorithm takes as inputs a message length l, a multiset of arities (arranged in descending order) minimizing the running time, denoted $A = \{x_1, x_2, ..., x_{|A|}\}$, and returns a multiset of arities (still sorted in descending order) minimizing the number of processors while leaving unchanged the running time. The algorithm proceeds as follows:

- We start by replacing in A each pair of arities 2 by an arity 4 (leaving possibly only one arity 2 in A). We sort A in descending order.
- We repeat at most two times the following routine to determine the solution:
  - Case $|A| = 1$: we return A.
  - Case $|A| = 2$:
    - Case $x_1 = 5$: we return A.
    - Case $x_1 \geq 3, x_2 \geq 3$: if $(x_1+1)(x_2-1) \geq l$ then $A = \{x_1+1, x_2-1\}$, otherwise we return A.
    - Case $x_1 = 4, x_2 = 2$: we return A.
    - Case $x_1 = 3, x_2 = 2$: if $5 \geq l$ then $A = \{5\}$. We return A.
  - Case $|A| \geq 3$:
    - Case $x_1 = 5$: we return A.
    - Case $x_1 \geq 3, x_2 \geq 3, x_3 \geq 2$: if $(x_1+1)(x_2-1)\prod_{i=3}^{|A|} x_i \geq l$ then we perform the following operations: (i) we add 1 to x_1 and we subtract 1 to x_2; (ii) we replace a possible pair of arities 2 by an arity 4; (iii) we reorder A. If either the check fails or $x_1 = 5$ then we return A.

B.2. **Reducing the number of processors at all the levels.** The following algorithm uses Algorithm 1 and 2 in order to compute a multiset of arities (sorted in descending order) minimizing the running time and the required number of processors at each step of the computation.

Algorithm 3. Let $A_0 = \{x_1, x_2, ..., x_{|A_0|}\}$ be the multiset of arities returned by Algorithm 1. We then use Algorithm 2 with a message of length l and the multiset A_0 to compute the multiset of arities $A_1 = \{x'_1, x'_2, ..., x'_{|A_1|}\}$. The rest of the algorithm proceeds iteratively as follows:

- We apply Algorithm 2 on inputs $l' = \lceil l/x'_1 \rceil$ and $A'_1 = \{x'_2, ..., x'_{|A_1|}\}$ to compute the multiset $A'_2 = \{x''_2, ..., x''_{|A'_1|}\}$. We set $n = 1$.
- As long as one of the following termination conditions is not met, namely (i) $A^{(n)}_{n+1} = A^{(n)}_n$; (ii) the highest number of levels of arity 4 has been reached (see Lemma 1); or (iii) $A^{(n)}_{n+1} = \varnothing$, we set

$n = n + 1$ and apply Algorithm 2 with the inputs $l^{(n)} = \left\lceil l^{(n-1)}/x_n^{(n)} \right\rceil$ and $A_n^{(n)} = \{x_{n+1}^{(n)}, \ldots, x_{|A_n^{(n-1)}|}^{(n)}\}$ to compute the multiset $A_{n+1}^{(n)} = \{x_{n+1}^{(n+1)}, \ldots, x_{|A_n^{(n)}|}^{(n+1)}\}$.

The resulting multiset of arities $A_r = \{x_1', x_2'', \ldots, x_n^{(n)}, x_{n+1}^{(n+1)}, \ldots, x_{|A_n^{(n)}|}^{(n+1)}\}$ minimizes the number of required processors at each step of the computation.

AIX-MARSEILLE UNIVERSITÉ, LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE MARSEILLE, CASE 901, F13288 MARSEILLE CEDEX 9, FRANCE
E-mail address: kevin.atighehchi@univ-amu.fr

AIX-MARSEILLE UNIVERSITÉ, INSTITUT DE MATHÉMATIQUES DE MARSEILLE, CASE 907, F13288 MARSEILLE CEDEX 9, FRANCE
E-mail address: robert.rolland@acrypta.fr