Layouts for Improved Hierarchical Parallel Computations

Michael Hirsch[a], Shmuel T. Klein[b], Yair Toaff[a]

[a]*IBM – Diligent, Tel Aviv, Israel*
{hirschm,yairtoaff}@il.ibm.com
[b]*Computer Science Department, Bar Ilan University, Ramat Gan 52900, Israel*
tomi@cs.biu.ac.il

Abstract

New layouts for the assignment of a set of n parallel processors to perform certain tasks in several hierarchically connected layers are suggested, leading, after some initialization phase, to the full exploitation of all of the processing power all of the time. This framework is useful for a variety of string theoretic problems, ranging from modular arithmetic used, among others, in Karp-Rabin type rolling hashes, as well as in cryptographic applications, and up to data compression and error-correcting codes.

Keywords: Data compression, parallel processors, modular arithmetic.

1. Introduction

We consider an (unbounded) stream of character strings of fixed, given length k, called *chunks*, and wish to apply a certain operation on each of the elements of this stream. For the ease of description, we shall use the remainder operation modulo some large integer P as a running example, considering each chunk as representing an integer of size k bytes. The method, however, applies as well to a large variety of other associative operations: if one considers a chunk as a sequence of numbers, one could calculate their sum, product, maximum or minimum, or Boolean operations like AND, OR or XOR.

The motivation for the repeated application of the remainder operation stems from our work on a large deduplication system [1], whose technical details are not relevant here. For the present work, it suffices to know that we wish to evaluate the remainders, modulo a large prime number P, of an unbounded sequence of input chunks. Specifically, we shall: (1) identify a chunk B, which is a character string of fixed size k, with its ASCII encoding; (2) consider this encoding as the standard binary representation of a large $8k$-bit long integer; and (3) evaluate $h(B) = B \bmod P$.

The use of the remainder operation has many other applications, beside deduplication, like Karp and Rabin's probabilistic pattern matching algorithm [10], modular exponentiation in cryptographic methods, like El Gammal's scheme [5] or RSA [13].

The length k of the chunks may be 512 or more, so that the evaluation might put a serious burden on the processing time. This can be improved if we assume the availability of several processors working in parallel. We show below how to exploit a set of n hierarchically connected parallel processors to perform the needed operations in several layers, but keeping all the processors busy without idle time, after some initialization phase. The challenge is to design the transition protocols from one step to another in a way that can be repeated indefinitely.

Parallelism has been discussed in connection with accelerating hashes in deduplication systems in [9], which uses the cryptographic SHA hash function for collision resistant fingerprinting, and in [14], presenting a deduplication system called P-Dedupe that pipelines and parallelizes the processes. More generally, [6] study the use of hashing in storage systems when a Graphics Processing Unit (GPU) can be used to speed up the processing. In another application, [2] use GPUs to parallelize hashing for the detection of image fragments in an image database.

Other applications of the hierachical method we describe below, beside remainder calculations, are the compression of sparse bit-strings, as described in [3] in which the recursive operation is the bit-wise OR, and the evaluation of the parity bits in the Hamming Error-Correcting Code [7], where the recursive operation is summation modulo 2, or equivalently, bit-wise XOR. The general case of a parallel hierarchical evaluation of an associative operation has been studied in [12], and our basic scheme with $\log n$ phases for n elements is mentioned in [11], but without referring to the layout permitting to keep all processors busy. Parallel implementations for more specific operations (multiplication and addition modulo $(2^n \pm 1)$) appear in [15].

The pertinence of the current work to string manipulation methods is thus twofold: it is not restricted to the suggested solution itself, which assigns the processors on the basis of the binary representation of their indices, but extends also to a large body of potential string-theoretic applications, such as data compression, pattern matching, error-correcting codes, cryptography, and others.

In the next section, we introduce the notation used below followed by the details of the suggested layouts in Section 3, using the application to the evaluation of a modulus $B \bmod P$ as an underlying example, rather than giving a generic description of the method. Section 4 deals with the mathematical details of this application and Section 5 presents some experimental results.

2. Notation

Given a sequence of chunk $B^i = x_1^i x_2^i \cdots x_m^i$, where the x_j^i denote characters of an alphabet Σ, we wish to apply the classical hash function $h(B^i) = B^i \bmod P$ for some large prime number P. We assume the availability of several processors working in parallel. A simplistic solution of assigning a set of ℓ processors would be as follows. Suppose an (unbounded) stream of non-overlapping chunks B is given, we shall process them by subsets of ℓ elements. For each subset, the ℓ processors are assigned sequentially to the ℓ chunks. Once all of them have produced their output, the whole set of processors is reassigned to the following ℓ chunks, etc. We may assume that all the processors need roughly the same time for the processing of their respective chunks, since the execution time of the given operation, $B \bmod P$ in our example, is not data dependent. Therefore this way of processing the stream by subsets of ℓ chunks keeps all the processors busy all of the time. We call this the *basic* parallel method.

The drawback, on the other hand, is that results are produced by packets of ℓ every t time units, where t is the sequential time needed by a single processor for processing a single chunk. In a streaming mode, we would prefer to apply the combined power of several processors in parallel on a *single* chunk and thereby obtain the requested result in less than t time units.

The following strategy could thus be applied. Partition the chunk to be processed into n *blocks* of k/n bytes each. For example, a chunk of 512 bytes could be split into $n = 64$ blocks of 8 bytes each. In a first stage which we call Step 0, a set of n processors is used to work simultaneously on the n blocks of the chunk. In Step 1, only $n/2$ processors are used, each acting on two blocks evaluated in the previous step, and in general in Step i, only $n/2^i$ processors are used, each acting on two blocks evaluated in the previous step $i-1$. Finally, in Step $\log n$, only a single processor is used. While the overall work of all the processors together is not reduced relative to an equivalent sequential evaluation on a single processor, the total processing time, if one accounts only once for commands executed in parallel, is reduced from $t = O(n)$ to $t = O(\log n)$. We call this the *hierarchical* method.

However, only in the first stage is the set of processors fully exploited, and in fact, for reasonable choices of n, most of the processors remain idle for most of the time. The average number of occupied processors is

$$\frac{n + \frac{n}{2} + \frac{n}{4} + \cdots + 2 + 1}{1 + \log n} = \frac{2n - 1}{1 + \log n},$$

which means that for $n = 64$, only about 28% of the processors are busy on average. The present work addresses this waste of processing time, by grouping several tasks together so as to get full exploitation of the available processing power, thereby reducing the waste to zero. This optimal utilization of the n processors is achieved by means of a particular strategy and a special way, to be described below, of assigning processors to tasks; we shall refer to a specific processor assignment as a *layout*.

The main idea leading to the full exploitation of all of the processors all of the time, is to assign them in such a way that, after an initialization phase of $\log n$ steps, a sequence of $\log n$ consecutive chunks will be processed simultaneously in parallel. The challenge is therefore to design an appropriate layout, showing how to assign the available processors at each time step. In particular, this layout has to be consistent over time transitions from step i to step $i + 1$, while also complying with the hierarchical definition of the function to be evaluated. In other words, we are looking for a function from the set of indices of the processors to itself, showing how to assign the subtasks to the processors at each step, so that the chain of transitions can be continued indefinitely without wasting any processing power. This will be called the *interleaving hierarchical* method.

3. Processor layouts for hierarchical computations

We assume a task, such as evaluating $B \bmod P$, is given, which has to be executed in several layers by a set of parallel processors, as explained above. The number of processors needed for layer 0 is n, we then further suppose that $2n - 1$ processors are available. A typical value of n could be 64 or some larger power of 2. We wish to exploit the processing power of all the processors, but only for layer 0 would all the processors be active, while for the next layer this is true for only half of them, then for a quarter, etc.

We start, at time 0, by assigning n processors to the first chunk (chunk indexed 0), where they will perform layer 0 of the parallel evaluation algorithm. At the following step, at time 1, $n/2$ of the so far idle processors will perform layer 1 for chunk 0, while the first n processors are reassigned to perform layer 0 of chunk 1. At time 2, $n/4$ new processors will perform layer 2 for chunk 0, the $n/2$ processors working in the previous step on layer 1 for chunk 0 are reassigned to perform layer 1 for chunk 1, and the n processors working in the previous step on layer 0 for chunk 1 are reassigned to perform layer 0 for chunk 2. This is schematically drawn in Figure 1, in which solid lines indicate blocks of currently working processors, and broken lines recall processors that have been working on lower layers in earlier time steps.

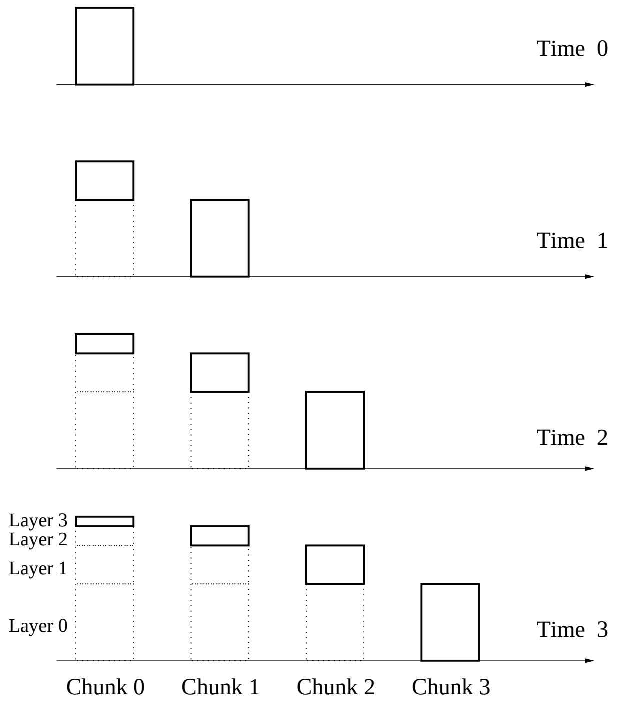

FIGURE 1: *Layout for the first few chunks*

In general, at time i, $i = 0, 1, \dots, \log n$, the set of available processors is partitioned into $i + 1$ uneven parts dealing with the first $i + 1$ chunks as follows: $n/2^i$ new processors will perform layer i of chunk 0, $n/2^{i-1}$ processors will perform layer $i-1$ of chunk 1,…, $n/2$ processors will perform layer 1 of chunk $i - 1$, and n processors will perform layer 0 of chunk i.

That is, only $\sum_{j=0}^{i} \frac{n}{2^j} = 2n - \frac{n}{2^i}$ processors are working at time step i for $i < \log n$, but after the initial $\log n - 1$ time steps, all the $2n - 1$ processors will be working. Figure 1 is the scenario for the initial steps. Following that, for $j = 1, 2, \dots,$ at time step $j + \log n$, one processor will perform layer $\log n$ of chunk $j - \log n$, two processors will perform layer $\log n - 1$ of chunk $j - \log n + 1$, four processors will perform layer $\log n - 2$ of chunk $j - \log n + 2$,…, $n/2$ processors will perform layer 1 of chunk $j - 1$, and n processors will perform layer 0 of chunk j. This can be summarized by:

For $j = 1, 2, \ldots$ and $i = 0, 1, \ldots, \log n$, at time $j + \log n$,
$\frac{n}{2^{\log n - i}}$ processors will perform layer $\log n - i$ of chunk $j - \log n + i$.

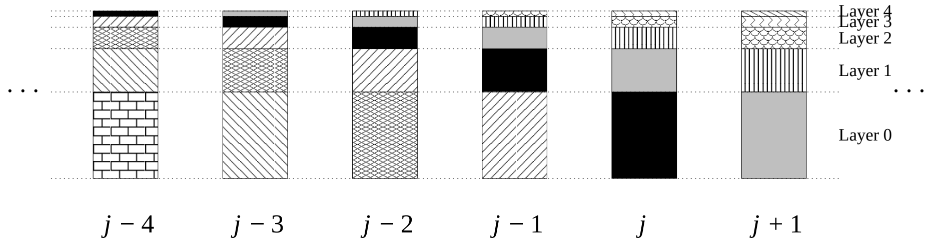

Figure 2: *Layout for the general case*

Figure 2 summarizes this layout for the general case, after the initial steps. While in Figure 1, there is a separate drawing for each time step, all these drawings appear overlaid in Figure 2. More precisely, each column in Figure 2 corresponds to one of the chunks (indexed here $j - 4$ to $j + 1$), and the time steps are characterized by the shading, that is, rectangles with identical fill patterns represent sets of processors working simultaneously. The solid black rectangles represent the set of processors working in parallel at time $j + \log n$: half of them on chunk j, a quarter on chunk $j - 1$, etc. The solid grey rectangles are the working processors in the following step, at time $j + \log n + 1$.

One can see in both Figures 1 and 2 that at the transition from one time step to the following one, all the processors move to the following chunk, but remain working on the same layer as before. Looking at a specific chunk, the transition from one time step to the following one corresponds to passing to the next higher layer and to reducing the number of processors working on this chunk by half.

As a result of this policy of assigning the processors to the chunks, no processor will stay idle after the initialization phase of $\log n$ time steps, after which all $2n - 1$ processors will work in parallel on $\log n$ consecutive chunks; moreover, the modulus for each of the processed chunks will be evaluated in layers within $\log n$ consecutive time steps.

There is still a certain degree of freedom for partitioning the processors into the subsets performing different tasks. We consider two possible scenarios. In the first, we ignore the time needed for each processor to read its necessary data, but consider the possibility of the use of some parameters which depend only on the index of the currently processed layer, and not on the particular chunk itself. This suggests a layout in which a processor is always assigned to perform a task at the same layer. In the second scenario, input operations are also being considered, which leads to a layout in which the assignment of new data to a processor is reduced to the possible minimum. In fact, a processor reads new data only after being done with the data that has been released.

3.1. **Assigning processors to layers**

Layer 0	Layer 1	Layer 2	Layer 3	Layer 4
0 0000**0**	1 000**01**	3 00**011**	7 0**0111**	15 **01111**
2 0001**0**	5 001**01**	11 01**011**	23 1**0111**	
4 0010**0**	9 010**01**	19 10**011**		
6 0011**0**	13 011**01**	27 11**011**		
8 0100**0**	17 100**01**			
10 0101**0**	21 101**01**			
12 0110**0**	25 110**01**			
14 0111**0**	29 111**01**			
16 1000**0**				
18 1001**0**				
20 1010**0**				
22 1011**0**				
24 1100**0**				
26 1101**0**				
28 1110**0**				
30 1111**0**				

TABLE 1: *Partition of the indices 0–30 into layers*

It would be easiest to design the layout such that the processors are divided into fixed sets of $n, \frac{n}{2}, \frac{n}{4}, \ldots, 2, 1$ processors, respectively. In that case, referring to Figure 3, the first subset, of n processors, will always work on layer 0, and generally, the subset of $n/2^j$ processors will always work on layer j, for $j = 0, 1, \ldots, \log n$. This could be an advantage if different constants are used for the different layers. For example, in the application to parallel remainder evaluation mentioned above, each processor acts on an input consisting of two data blocks. These data blocks are adjacent for layer 0, but for higher layers, the blocks are further apart, and the distance between the blocks depends on the index of the layer. This fact translates to using a constant C_i in the evaluation procedure performed by each of the processors, and this constant is the same for all processors acting within the same layer, but differs from layer to layer. If a given processor is thus always assigned to the same layer i, there is no need to update its constant C_i, which can be hardwired into it. A possible fixed partition of the indices of processors is given below in Table 1.

Suppose the processors are indexed from 0 to $2n-2$. The n processors acting on level 0 are those with the even indices, $\{0, 2, 4, 6, \ldots\}$. The $n/2$ processors acting on level 1 are those with indices that are of the form 1 + multiples of 4, $\{1, 5, 9, 13, \ldots\}$. The $n/4$ processors acting on level 2 are those with indices that are of the form 3 + multiples of 8, $\{3, 11, 19, 27, \ldots\}$, etc. In general, the $n/2^i$ processors acting on level i are those with indices that are of the form $2^i - 1+$ multiples of 2^{i+1}, $i = 0, 1, \ldots, \log n$. An equivalent way of describing this partition, which also has the advantage of showing that this way of numbering indeed induces a partition, that is, that all indices are accounted for and none of them appears twice, is by referring to the $(1 + \log n)$-bit standard binary representation of the numbers 0 to $2n-2$: the n even indices are those ending in 0, the indices of level 1 are those ending in 01, then 011, and generally, the indices of level i are the $n/2^i$ numbers whose $(1 + \log n)$-bit standard binary representation ends in $011 \cdots 1$, where the length of the string of 1s is i. Table 1 brings the partition for $n = 16$, the indices appearing in decimal and binary, with their suffixes emphasized.

3.2. **Assigning processors to chunks**

The drawback of the approach of assigning a given processor at each time step to work on the same layer is that all the processors would have to read new data, and the overhead caused by this input operation could void all the benefits of using parallelization in the first place.

Consider therefore the following more involved indexing scheme, assigning the processors according to their index in such a way that only n, that is, about half of the processors, have to read new data at each time step, which is the possible minimum as at each time step, a new data chunk is accessed. The remaining $n-1$ processors stay with the data they have read when they have been assigned to layer 0. This implies that there is no delay caused by input commands during the $\log n$ consecutive steps required to process the chunk in layers. The following explanation corresponds to the general case, not the initial $\log n$ chunks.

Let us this time index the $2n-1$ processors by the integers from 1 to $2n-1$, where we assume that n is a power of 2, say $n = 2^d$, and consider the (left 0-padded) $(d+1)$-bit standard binary representation of these indices. For example, for $d = 4$, the indices are $00001, 00010, \ldots, 11110$ and 11111. The processors are partitioned as follows: the n processors assigned to chunk

j are those with odd indices (in other words, those with indices equal to 1 modulo 2), the $n/2$ processors assigned to chunk $j-1$ are those with indices ending in 10 (in other words, those with indices equal to 2 modulo 4), and in general, the $n/2^r$ processors assigned to chunk $j-r$ are those with indices ending in $10\cdots0$ (1 followed by r zeros, in other words, those with indices equal to 2^r modulo 2^{r+1}). These blocks of processors can be seen in the upper part of Figure 3 below, where they are ordered, within each column, lexicographically. The fixed suffixes for each block, 1, 10, 100, etc, are boxed for emphasis.

Layer 0	Layer 1	Layer 2	Layer 3	Layer 4
1 0000**1**	2 000**10**	4 00**100**	8 0**1000**	16 **10000**
3 0001**1**	6 001**10**	12 01**100**	24 1**1000**	
5 0010**1**	10 010**10**	20 10**100**		
7 0011**1**	14 011**10**	28 11**100**		
9 0100**1**	18 100**10**			
11 0101**1**	22 101**10**			
13 0110**1**	26 110**10**			
15 0111**1**	30 111**10**			
17 1000**1**				
19 1001**1**				
21 1010**1**				
23 1011**1**				
25 1100**1**				
27 1101**1**				
29 1110**1**				
31 1111**1**				

TABLE 2: *New partition of the indices 1–31*

When passing from time step i to $i+1$, half of the processors working on each of the currently processed consecutive chunks $j, j-1, \ldots, j-\log n$ are reassigned to the new chunk to be processed, indexed $j+1$, while the other half remains with the chunk they started with and pass to a higher layer. More precisely, all the processors with indices $\geq n$, that is, whose binary representation starts with 1 (those in the bold rectangles in Figure 3), are assigned to the new chunk, while those with indices $< n$ remain with their earlier chunk. To get a consistent numbering, the following transformation is applied to each of the indices at the transition between time steps: the index B at time $i+1$ is obtained from the index A at time i by applying a cyclical shift by one bit left to the binary representation. Note that this is a bijection, so that starting with all the numbers between 1 and $2n-1$, we

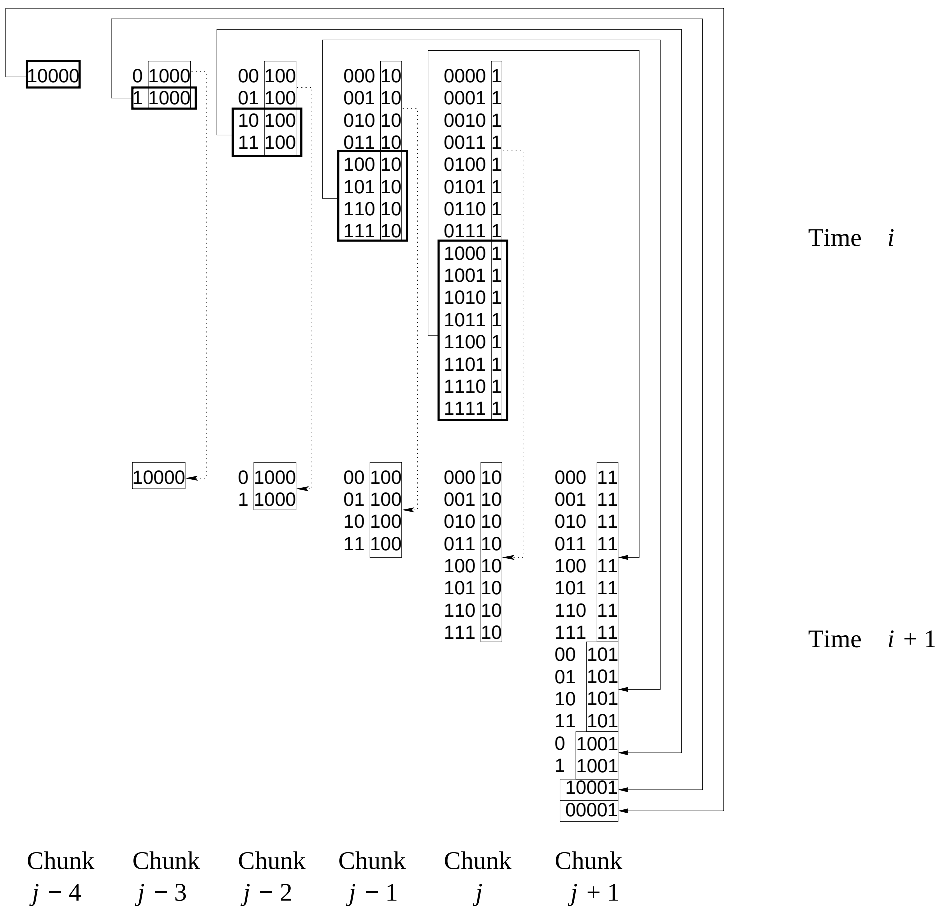

FIGURE 3: *Index layout for transition from time step i to $i+1$*

again get the same set after the transformation on all the elements of the initial set. For example, if $A = 11001 = 25$ then $B = 10011 = 19$, and if $A = 01010 = 10$ then $B = 10100 = 20$. In other words

$$B = \begin{cases} 2(A-n)+1 & \text{if } A \geq n \\ 2A & \text{if } A < n \end{cases}$$

After this transformation, all indices in the new chunk $j+1$ end in 1, all those in chunk j (which is now processing layer 1) end in 10, etc. As can be seen, the new layout is similar to the one we had in the previous time step. Indeed, the column of indices of Chunk t in the lower part of the figure, corresponding to time $i+1$, is identical to the column of indices of Chunk $t-1$ in the upper part of the figure, corresponding to time $i+1$, for $t = j, j-1, j-2$ and $j-3$. Note that the elements in the last column (chunk $j+1$ in the lower part of the figure) are not ordered lexicographically to emphasize their origin, but one can easily check that this column is just a permutation of the elements in the column of Chunk j of the upper part of the figure. Figure 3 summarizes this layout and shows the details of the transition from time step i to $i+1$ for $d = 4$, i.e., $n = 16$.

Table 2 above summarizes this new layout and the partition it induces in a similar way as done above for the previous partition in Table 1. An alternative way of interpreting the new partition is by noting a correspondence between Tables 1 and 2: the element indexed i in a certain position of Table 1 corresponds to the element indexed $i+1$ in the same position of Table 2.

4. Application

Though many applications for the above layouts are possible, we shall restrict our description to the example mentioned above that is related to compression and coding problems we are interested in, namely the evaluation of the remainder function [8]. Consider the input string B partitioned into n subblocks of d bits each, denoted $A[0], \ldots, A[n-1]$, where n is a power of 2, and d is a small integer, so that d bits can be processed as an indivisible unit, typically $d = 32$ or 64. Given also is a large constant number P of length up to d bits, that will serve as modulus. Typically, but not necessarily, P will be a prime number. For example, one could use $n = d = 64$. We would like to split the evaluation of $B \bmod P$ so as to make use of the possibility to evaluate functions of the $A[i]$ in parallel on n independent processors $p_0, p_1, \ldots, p_{n-1}$, which should yield a speedup.

Note then that if we have a string D of $2d$ bits and we want to evaluate $\overline{D} = D \bmod P$, then we can write $D = D_1 \times 2^d + D_2$, where D_1 and D_2 are the leftmost, respectively rightmost d bits of D. We get that $\overline{D} = \overline{\overline{D_1} \times \overline{2^d} + \overline{D_2}} = \overline{\overline{D_1} \times C + D_2}$, where $C = 2^d \bmod P$ is a constant that can be pre-computed.

This can be generalized to a hierarchical tree structure that exploits the parallelism repeatedly in $\log n$ layers, using the n available processors. In Step 0, the n processors are used to evaluate $A[i] \bmod P$, for $0 \le i < n$, in parallel. This results in n residues, which can be stored in the original place of the n blocks $A[i]$ themselves, since P is assumed to fit into d bits.

In Step 1, only $\frac{n}{2}$ processors are used and each of them works, in parallel, on two adjacent blocks. The work to be performed by each of these processors is what has been described earlier for the block D. Again, the results will be stored in-place, that is, right-justified in $2d$-bit blocks, of which only the rightmost d bits will be affected.

In Step 2, $\frac{n}{4}$ processors are used, and each of them is applied, in parallel, on two adjacent blocks of the previous stage. That is, we should have applied now the first processor on $A[0]A[1]$ and $A[2]A[3]$, but in fact we know that $A[0]$ and $A[2]$ contain only zeros, so we can simplify and apply the processor on $A[1]$ and $A[3]$, and in parallel apply the next processor on $A[5]$ and $A[7]$, etc. Again, the work to be performed by each of these processors is what has been described earlier for the block D since we are combining two blocks, with the difference that the new constant C should now be $2^{2d} \bmod P = \overline{C^2}$. The results will be stored right-justified in $4d$-bit blocks, of which, as before, only the rightmost d bits or less will be affected. Continuing with further steps will yield a single operation after $\log n$ iterations, and the final value $B \bmod P$ will be in $A[n-1]$.

```
for $i \longleftarrow 1$ to $\log n$ do
    for $k \longleftarrow 0$ to $\frac{n}{2^i} - 1$ do
        use the set of processors assigned to layer $i$ to evaluate, in parallel,
            $\ell \longleftarrow 2^i k + 2^i - 1$
            $A[\ell] \longleftarrow (A[\ell - 2^{i-1}] \times C[i] + A[\ell]) \bmod P$
```

FIGURE 4: *Hierarchical parallel evaluation of $B \bmod P$*

Summarizing, we first evaluate an array of constants $C[i] = \overline{C^{2^{i-1}}} =$

$\overline{2^{d\times 2^{i-1}}}$ to be used in layer i for $i = 1, 2, \ldots, \log n$. This is easily done noticing that $C[1] = C$ and $C[i+1] = \overline{C[i]^2}$ for $i \geq 1$. The parallel procedure for the higher layers is then given in Figure 4.

5. Experimental results

To get some empirical evaluation of the influence of the suggested layout, we ran the following tests. The tests were run on a GPU: a Nvidia GeForce GTX 465 graphics board, programmed in CUDA [4]. The input was 32MB of a video file, which in fact is not important, since the processing time does not depend on the specific input data given. The file was processed 1000 times, and the timing results averaged, excluding data copy time. The number of parallel processors was $n = 64$. As a baseline, to measure the GPU overhead, we ran a simple loop Xoring every fourth byte of the input. The intention was to force a data flow similar to the proposed algorithms, but which an optimizer could not eliminate. The time in milliseconds to process the whole file and the throughput in GB per second is given in the first column of Table 3.

For the second column of Table 3, we used the basic parallel method described in the introduction. The file has been processed in subsets of n consecutive strings of length 512 bytes, each of these strings being considered as a chunk, on which a single processor worked sequentially. Once all n chunks have been processed, the n processors were assigned to the following subset of n chunks.

	baseline	64 parallel processors	hierarchical single block	hierarchical interleave
Time	4.86	19.53	17.45	7.09
Throughput	6.43	1.60	1.79	4.40

TABLE 3: *Processing time in ms and throughput in GB/sec*

The next step was to apply the hierarchical method in which 64 processors were applied, in turn, in 7 layers, to the consecutive chunks. A new output value was produced each 7th time step, but most of the processors were idle on the average. The total time appears in the third column and shows an increase in throughput of about 10% relative to the previous method.

Finally, the timing for the interleaving hierarchical method appears in the last column, which corresponds to the layout of Table 2. We see that the interleaving was able to increase the throughput 2.5 fold relative to the simple hierarchical method, while yielding a streaming mode in which a new output value is produced at each time step.

Summarizing, we suggest an indexing mechanism for a set of parallel processors by means of which one may assign the processors to act on parts of chunks at various layers, according to the algorithm at hand. At each time step transition, a part of the processors is reassigned in such a way that the assignment of processors to chunks remains consistent with the earlier definition, which allows an unlimited sequence of transitions, while constantly keeping all the processors busy.

References

[1] Aronovich L., Asher R., Bachmat E., Bitner H., Hirsch M., Klein S.T., The Design of a Similarity Based Deduplication System, *Proc. SYSTOR'09*, Haifa, (2009) 1–14.

[2] Collange S., Dandass Y.S., Daumas M., Defour D., Using graphics processors for parallelizing hash-based data carving, *Proc. 42nd Hawaii Intern. Conf. on System Sciences*, Waikoloa (2009) 1–10.

[3] Choueka Y., Fraenkel A.S., Klein S.T., Segal E., Improved Hierarchical Bit-Vector Compression in Document Retrieval Systems, *Proc. 9-th ACM-SIGIR Conf.,* Pisa (1986) 88–97.

[4] Nvidia CUDA C Programming Guide, www.nvidia.com/cuda.

[5] El Gamal T., A public key cryptosystem and a signature scheme based on discrete logarithms, *Proc. CRYPTO'84 on Advances in cryptology* (1985) 10–18.

[6] Gharaibeh A., Al-Kiswany S., Gopalakrishnan S., Ripeanu M., A GPU accelerated storage system, *Proc. 19th ACM Intern. Symposium on High Performance Distributed Computing*, Chicago (2010) 167–178.

[7] Hamming R.W., *Coding and Information Theory*, 2nd ed., Prentice Hall (1986).

[8] Hirsch M., Klein S.T., Toaff Y., Improving deduplication techniques by accelerating remainder calculations, *Discrete Applied Mathematics* **163**(3) (2014) 307–315.

[9] Li X., Lilja D.J., A highly parallel GPU-based hash accelerator for a data deduplication system, *Proc. 21st IASTED Intern. Conf. Parallel and Distributed Computing and Systems*, Cambridge, USA (2009) 268–275.

[10] Karp R.M., Rabin M.O., Efficient Randomized Pattern-Matching Algorithms, *IBM Journal of Research and Development*, **31**(2) (1987) 249–260.

[11] Kruskal C.P., Rudolph L., Snir M., The power of parallel prefix, *IEEE Transactions on Computers* **C–34**(10) (1985) 965–968.

[12] Ladner R.E., Fischer M.J., Parallel prefix computation, *Journal of the ACM*, **27**(4) (1980) 831–838.

[13] Rivest R.L., Shamir A., Adleman L.M., A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, *Communications of the ACM* **21**(2) (1978) 120–126.

[14] Xia W., Jiang H., Feng D., Tian L., Fu M., Wang Z., P-Dedupe: exploiting parallelism in data deduplication systems, *Proc. 7th Intern. IEEE Conf. on Networking, Architecture and Storage*, Xiamen, Fujian (2012) 338–347.

[15] Zimmermann R., Efficient VLSI implementation of modulo $(2^n \pm 1)$ addition and multiplication, *Proc. 14th IEEE Symposium on Computer Arithmetic*, Adelaide, Australia (1999) 158–167.