Adaptive Loops with Kaapi on Multicore and Grid: Applications in Symmetric Cryptography*

Vincent Danjean[abcd] Vincent.Danjean@imag.fr
Roland Gillard[ebc] gillard@ujf-grenoble.fr
Serge Guelton[ad] Serge.Guelton@imag.fr
Jean-Louis Roch[acdf] Jean-Louis.Roch@imag.fr
Thomas Roche[g] Thomas.Roche@imag.fr

[a]Laboratoire d'Informatique de Grenoble, 51 av. Jean Kuntzmann, 38330 Montbonnot-Saint-Martin, France
[b]Universite Joseph Fourier
[c]Grenoble Universite
[d]INRIA Rhône-Alpes
[e]Institut Fourier, 100 rue des Maths, BP74 38402 St Martin d'Heres, France
[f]Institut National Polytechnique de Grenoble (INPG)
[g]CS, Communication&Systemes, 22 avenue Galilee, 92350 Le Plessis Robinson, France

ABSTRACT

The parallelization of two applications in symmetric cryptography is considered: block ciphering and a new method based on random sampling for the selection of basic substitution boxes (S-box) with good algebraic properties. While both consists mainly in loops with independent computations and possibly early termination, they are subject to changing computation loads and processor speeds which can be managed by distributed workstealing. To take benefit of workstealing, we propose in this paper a generic way to rewrite loops in a recursive way, involving three complementary levels of parallelism. Dealing with early termination is performed by an amortized control, original to our knowledge. Those schemes have been embedded in STL-like parallel algorithms implemented on top of KAAPI library that provides distributed workstealing on a wide range of platforms. Experiments and performances are reported on SMP (up to 16 processors) and grid architectures (up to 2120 processors) for benchmarks (e.g. STL `find_if`) and for the two target cryptography applications. These experiments exhibit the stability of the library and its usability by external users for effective applications.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms: Algorithms, Design, Experimentation

Keywords: Adaptive parallelism, grid computation, parallel STL, symmetric cryptography, workstealing

*This work is partially supported by ANR (French National Research Agency, project SAFESCALE-BGPR n. ANR-05-SSIA-005) and CS company.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PASCO'07, July 27–28, 2007, London, Ontario, Canada.
Copyright 2007 ACM 978-1-59593-741-4/07/0007 ...$5.00.

1. INTRODUCTION AND CONTEXT

Cryptography involves exact arithmetic computations that may take benefit of various level of parallelism. At fine grain, taking benefit of the available parallelism (multicore processors, co-processors such as FPU or GPU) may increase the performance of applications which require a high throughput. An example in cryptography is block ciphering where performance is a critical issue [1]; then, using multicore parallelism may reduce the delay between reading the source data and writing the ciphered data on the output (either a disk or the network) which is crucial for performance. Furthermore, at a coarser grain, other applications in cryptography may require a huge number of computations. Obvious examples include breaking a protocol or exhibiting its level of security. Yet, the design and analysis of secure protocols often relies on statistical computations, like selecting cryptographic S-boxes with good properties. For such computations, large scale architectures (grids) are mostly considered. In this paper, the parallelization of two real cryptographic applications is considered: multicore optimization of a block cipher and selection of S-boxes. For both real applications, sequential codes have been previously developed and optimized without taking into account any parallelization. The goal is to provide parallelizations that achieve effective performances with respect to the initial sequential codes which are expressed as the following generic loop:

Algorithm 1 Core Sequential Loop Algorithm

Data : An input initial value Acc_0;
Result: $Acc_n = Acc_0 \bigoplus f(1) \bigoplus \ldots \bigoplus f(n)$
while $!Terminated(n, Acc_n)$ **do**
  $x_{n+1} = f(n)$;
  $Acc_{n+1} = Acc_n \bigoplus x_{n+1}$;
  $n = n + 1$;
end

where $\bigoplus$ is an associative, but not necessarily commutative, operation. Such a loop is generic in many applications: with some restrictions to enable parallelism, it encompasses the C++ STL algorithms such as `accumulate`, `for_each`, `find_if` or `transform` [25]. For instance, generic modular

computations follow such a loop (eg iterative computation of the determinant of integer sparse matrices).

While both applications lead to a similar high level loop parallelism (with mainly independent tasks), the execution contexts (multicore SMP machine on one hand; heterogeneous grid on the other one) may suggest drastically different implementations. However, in both cases, resources are heterogeneous and must be efficiently used. Heterogeneity may be due to hardware design on the grid but also to different loads. For example, on a multi-user multicore SMP machine, we cannot always control the load of the different processors due to concurrent applications or concurrent components inside a single application. Then, we have to manage processors with different observable speeds. Workstealing is an on-line scheduling on heterogeneous processors [3, 8, 17] with provable performances. It is often used in symbolic computations where unpredictable recursive parallelism often appears at execution time due to input data characteristics.

But, to be efficient, the application should contain a high level of parallelism. Then it will work only if the application is correctly structured with regards to its parallelism so that the runtime is able to automatically and efficiently schedule the parallel tasks of the application. To run our experiments, we choose to use the KAAPI runtime. Implementing workstealing based on a work-first principle [15, 17], KAAPI is able to run parallel tasks on heterogeneous machines (cluster or grid), on SMP machines and it has a flexibility and easy to use C++ interface for programming recursive parallel tasks with data dependencies. Theoretical foundation of workstealing and the KAAPI software are described in the next section. Then, in Section 3, we introduce the three levels of parallelism used to parallelize the previous loop with applications to `find_if`. This theoretical development is applied in section 4 to the block cipher application on a SMP machine. Then, in Section 5, it is used for a real cryptographic application that aims at finding substitution boxes with good cryptographic invariants and no quadratic relations; the results obtained on a grid of 2120 processors are presented in Section 6.

Related works on adaptive loop parallelization

Parallelization of loops is a classical problem in parallelism. Parallel and distributed containers are provided in PSTL [18] and STAPL [5] without relying on workstealing. STAPL provides a framework for algorithm selection and tuning (FAST) which is performed at runtime based on a performance prediction model. Our approach gives a generic way to exhibit adaptive parallelism in a loop with provable performances on heterogeneous processors. Based on workstealing, the multi-core STL (MCSTL) [27], is restricted to multi-core architectures with a small number of processors. Moreover, the MCSTL does not deal with early termination.

2. KAAPI AND WORKSTEALING

KAAPI [24] is a C++ runtime library that allows one to execute multithreaded computation with data flow synchronization between threads. KAAPI is ported on a large class of parallel computing platforms: multicore and SMP architectures; clusters; network with heterogeneous processors; global computing platforms (grids and P2P platforms). Furthermore KAAPI supports fault-tolerance as well as run-time addition and resilience of resources [19]. Yet, KAAPI implements the application programming interface ATHAPASCAN [16] that enables to separate the expression of the parallelism in the application (described by the data flow graph which is related to the execution and unfold at runtime) from its scheduling (specified by code annotations). Athapascan supports dynamic recursive parallelism with data dependencies. It has been successfully used for parallelization of numerical and combinatorial optimization computations[17].

The default scheduling in KAAPI 2.2, called DDS and described in [17], is workstealing. It is based on the work-first principle [15]: the overhead of parallelism is transferred from local task creation to task migration. Actually, each processor maintains a local double-ended queue, called deque. When a processor performs a task creation `Fork<f>()(args)` instruction, it pushes the corresponding closure at the bottom of its deque which is accessed by non-blocking locks (e.g. compare-and-swap). Read-write data dependencies between tasks are chained according to a sequential order (depth-first called reference order); when a task completes, the next task (if any) at the bottom of its stack is ready and is popped for execution. If there is no ready task at the bottom of its deque, the processor becomes a stealer: it steals a ready task (the oldest one) on a randomly chosen victim processor (the topmost task on the front of the victim processor, i.e. its oldest ready task) and allocates a new deque for its execution. Most overhead due to computation of data dependencies is moved to this steal operation [17]. Indeed steal operations are very rare events as stated in [6, 8] on a grid when processors speeds ratios may vary only within a bounded interval.

ATHAPASCAN/KAAPI provides a language-based performance model using work and depth [4] to predict the execution time T on a distributed architecture. The work W is the total number of elementary (unit) operations performed; the depth D is the critical-path, i.e. the number of (unit) operations for an execution on an unbounded number of processors. Note that D accounts not only for data-dependencies among tasks but also for recursive task creations. The work (and depth) of an ATHAPASCAN parallel program includes both the sequential work (W_S) and the cost of task creations but without considering the scheduling overhead; similarly to a sequential function call, the cost of a task creation with n unit arguments is $\tau_{fork} + n\tau_{arg}$. If the cost of those task creations is negligible in front of W_s, then $W \simeq W_s$.

A distributed (heterogeneous) architecture with processors of changing speed is modelled as in [8]. Given p processors, let $\Pi_i(t)$ be the instantaneous speed of processor i that participates to the computation at time t, measured as the number of elementary operations per unit of time; let $\Pi_{ave} = \frac{\sum_{t=1}^{T}\sum_{i=1}^{p}\Pi_i(t)}{p.T}$ be the average speed of a processor for a computation with duration T. The next theorem from [8] makes explicit a bound on T with respect to Π_{ave}, p, D and W for recursive parallel creation of parallelism (fully-strict computation).

Theorem(see [8, 17]) *With high probability, the number of steal operations is $O(pD)$ and the execution time T is bounded by $T \leq \frac{W}{p\Pi_{ave}} + O\left(\frac{D}{\Pi_{ave}}\right)$.*

Thus, when $D \ll W$, the resulting time is close to the expected optimal one $\frac{W}{p\Pi_{ave}}$. Then, to fit to KAAPI and previous theorem, the following sections explicit the parallelization of the sequential loop (Algorithm 1) to ensure that its depth D is very small and its work W is close to W_s.

3. ADAPTIVE LOOP PARALLELIZATION

This section presents the parallelization of the sequential loop (Algorithm 1). To receive the benefit of workstealing, three levels are distinguished: micro-loop provides parallelization by recursing halving of the iteration (which is a fully strict computation); nano-loop amortizes the overhead introduced by micro-loop and workstealing; macro-loop enables to detect early termination with provable performances with respect to a sequential execution of the loop. The next section considers the restricted case where the size of the loop is fixed, while the following ones extend to the general case.

3.1 Recursive parallelism: parallel micro-loop

In this section, we consider a restricted version of the loop in Algorithm 1 where the number N of computations $f(i)$ to perform is an input. As a result, the `Terminated`(n, Acc_n) test reduces to the $n \leq N$ test. In all the sequel, we will assume that the associative operation $\bigoplus$ takes a constant time. Let W_s be the total number of unit operations performed by the loop. The number of operations $w_{f(n)}$ performed by $f(n)$ may depend on n; we denote by $D_f = \max_{i=1}^{n} w_{f(i)}$ the maximal number of operations performed by $f(i)$, $1 \leq i \leq N$. We assume that the degree of potential parallelism $\frac{W_s}{D_f}$ is large enough to motivate the use of p processors to compute the loop.

Then, to exploit recursive parallelism, the loop from i to j may be recursively split into two parallel parts of size $N/2$ and $n-N/2$ till the finest grain $f(i)$, each part corresponding to a task scheduled by workstealing. This recursive extraction is referred in the sequel as *micro-loop* since it extracts parallelism until fine grain tasks. Then the total arithmetic work of the computation is $W = W_s + \alpha N$; the overhead αN encompasses the $2N$ local task creations (for f and $\oplus$ computations). The critical depth of the computation, that dominates the number of steals, is $D = D_f + \beta \log N$; the overhead $\beta \log N$ encompasses the $\lfloor 1 + \log_2 N \rfloor \leq n_i \leq \lceil 1 + \log_2 N \rceil$ task creations on the path of length n_i from the leave task node $f(i)$ to the root $\bigoplus$. Then, from theorem 2, the time T of the computation is bounded by

$$T \leq \frac{W_s + \alpha N}{p\Pi_{ave}} + O\left(\frac{D_f + \beta \log N)}{\Pi_{ave}}\right).$$

Assuming $D \ll W$ and $p \ll N$, this bound is close to optimal if $W_s \gg \alpha.N$ but not if $W_s = \Theta(N)$. The next section extends it to this general case where $W_s = \Omega(N)$.

3.2 Local sequential nano loop

When $\beta \log N$ is large compared to D_f, the overhead of task creation may be hidden without increasing the depth of the computation. Indeed, the previous recursive splitting is stopped when the input interval contains less than $\frac{\beta \log N}{D_f}$ computations of f to perform: then it is computed by a single task that performs a sequential loop (similar to the initial loop in Algorithm 1). Let w be the maximal work performed by such a sequential loop: $D_f + \frac{\beta \log N}{D_f} \leq w \leq \beta \log N$. Since $\frac{D}{2} \leq w < D$, this work is critical: then the loop is called *nano-loop*. Considering both the micro-loop (recursive splitting) and the nano-loops, the resulting depth D' verifies $D \leq D' \leq 2\beta \log N = 2D$ and remains of the same order as D.

Note that the nano-loop hides the overhead due to task creations and access to the local deque. As a result, it decreases the work W' of the parallel program to the one performed by the reference sequential loop: $W' = W_s + \alpha \frac{N}{\log N}$. Considering W' and D', the corresponding execution time T is now bounded by

$$T \leq \frac{W_s}{p\Pi_{ave}} + \alpha \frac{N}{(\log N)\, p\Pi_{ave}} + O\left(\frac{D}{\Pi_{ave}}\right).$$

Yet, assuming $D \ll W$ and $p \ll N$ as before, $T \simeq \frac{W_s}{p\Pi_{ave}}$. in the general case where $W_s = \Omega(N)$.

Remark 1. Based on work-stealing, the extraction of parallelism (i.e. a fraction of the remaining work W_r) can be performed on line. Then each non-idle processor performs a local micro loop, each step of the micro-loop being a nano-loop with non preemptive execution. Then, the corresponding size of the nano loop in this case is self-tuned to $\Theta(\log W_r)$ since, in the best case, the depth to execute a work W_r on an unbounded number of processors is at least $\Omega(\log W_r)$.

3.3 Self-adaptive computation with early termination: global sequential macro loop

We now consider the general case of the loop Algorithm 1 where the number N of effective computations performed is not an input but decided on-line by `Terminated`(n, Acc_n). A typical example is the STL `find_if` algorithm: taking in input an iterator in the range $[f, l)$, it returns the first iterator $N \in [f, l)$ such that the predicate `pred`(*N) is true and l if no such predicate is found[1]. Then the work performed by the loop in Algorithm 1 is $W_s = \sum_{k=1}^{N} w_{f(i)}$.

To ensure that the work W performed by the parallel algorithm is close to W_s, we use the general scheme proposed in [7] based on an amortized technique inspired by Floyd's algorithm to detect periodicity in a sequence.

The global loop is broken into several *macro steps* each consisting in consecutive iterations of the loop. Let s_n be the number of iterations in the m-th macro-step and let $n_m = \sum_{i=1}^{m} s_i$. The m-th macro-step takes in input the value $x_{n_{m-1}}$ of the previous macro-step and computes x_{n_m} by the micro-loop/nano-loop previous scheme. Inside a macro-step, no termination test is performed. At the end of the macro-step, Acc_n is reconstructed using x_{n_m} and the termination test is performed. The way the termination test is computed depends on the `Terminated` function. The first value N and corresponding Acc_N that makes the termination test true can be recovered from three main schemes:

1. either directly by the $\bigoplus$ function: this is the case when for instance $\bigoplus$ reduces to a min computation like in `find_if`;
2. or by a dichotomic search when the termination test remains true for any n larger than N [7];
3. or, in the general case, by a parallel partial sum (prefix) computation [28] followed by a dichotomic search. Note that in this case, due to the nano-loop inside a macro-step, the number of prefix to compute in the m-th macro-step is $O(\frac{s_n}{\log s_n})$ that compares favorably to the arithmetic work $\Omega(s_n)$.

[1]To exhibit parallelism, the predicate `pred` is assumed to implement a fixed function of its input, not depending on the context.

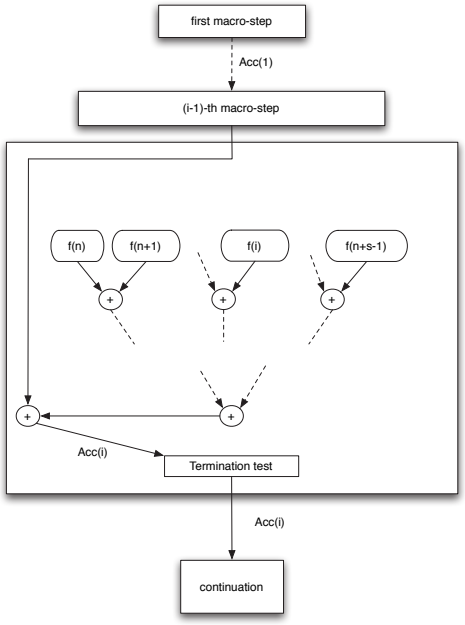


Figure 1: Scheme of the macro-step iteration.

In any case, the amount of potential extra-work is bounded by the size of the macro-steps. However, with parallelization in mind, the global number of macro-steps should be kept small: each macro-step involves a global synchronization; and the larger the size of a macro-step, the better the parallelism. When the termination test remains true for any $i \geq N$, various tradeoff are studied in [7], in particular $s_n = \rho^{\frac{n}{\log n}}$ with $1 < \rho < 2$. Table 1 explicits the extra-work for some values of s_n.

s_k	#macro-steps	extra work
constant	$O(N)$	$O(1)$
2^k	$O(\log N)$	$O(N)$
$\rho^{\frac{k}{\log k}}$	$O(\log N \log\log N)$	$O\left(\frac{N}{\log\log N}\right)$

Table 1: Tradeoffs for the size s_k of macro-step k.

3.4 STL interface: `find_if` experiments

The previous three levels of parallelism (micro, nano and macro loops) have been developed on top of KAAPI providing parallel implementation for the following generic STL algorithms: `for_each`, `transform`, `accumulate`, and `find_if`. Those parallel implementations are restricted to random iterators; when involved, a binary operator is not assumed commutative but associative; both unary and binary operators are assumed to return the same value for a given input whatever the order of evaluation is. We report in this paragraph experimental results for the `find_if` implementation which requires the three levels of parallelism. Experiments are performed on an AMD Opteron machine with sixteen 2200 MHz processors (8 dual-core). The time of `find_if` increases (almost proportionally) with respect to the smallest index N that makes the predicate true. For each experiment, T_s denotes the time of the the sequential `find_if` of the STL native implementation and T_p the one of our parallel implementation on p processors; the relative speed-up $\frac{T_s}{T_p}$ is exhibited. The native STL `find_if` is also called to perform the nano-loop. The size s_k chosen for the k-th macro-loop step is $s_k = \frac{1}{2}\sum_{i=1}^{k-1} s_i$ which ensures a theoretical average work overhead of 25% (with upper bound 50%).

Two classes A and B of experiments consisting in computing `find_if` on an array of n double floating point numbers are reported for a number of processors varying from 1 to 16. The size of the nanoloop has been tuned to obtain the best experimental time. In class A (fig. 2), $n = 10^6$ and the cost of the predicate is $\tau_{Pred} = 30.71\mu$s; each colour represents a specific value of $N \in \{10, 10^2, 10^3, 10^4, 10^5, 5.10^5, 10^6\}$ corresponding to a sequential reference computation time T_s from 0.3ms to 30s. In B, $n = 10000$ and $\tau_{Pred} = 30.02$ ms; $N \in \{1, 10, 100, 1000, 5000, 10000\}$ corresponding to a sequential reference computation time T_s from 30ms to 30s.

On both figures, we observe saw tooths in particular for small values of N. However, those saw tooths are indeed much less apparent in the speed-up $\frac{T_1}{T_p}$ relative to the execution of the parallel algorithm on 1 processor: yet, they are mainly due to the arithmetic overhead of the macro-loop with respect to the sequential execution, this overhead depending on N.

We observe speed-up only from $T_s \geq 3ms$ ($N \geq 10^4$ in A, $N \geq 1$ in B): this exhibits the minimal grain for our implementation combining the three loops on KAAPI. But, once speed-up is observed, it scales almost linearly. This exhibits the efficiency of the parallelization on 16 processors even for fine grain computations (less than 1 second) and unpredictable work.

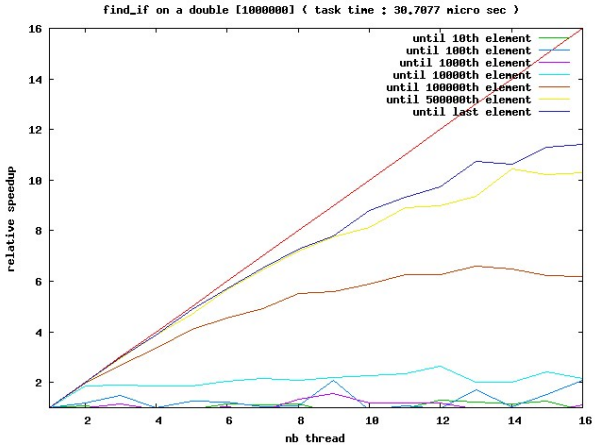


Figure 2: `find_if` – A: $n = 10^6, \tau_{Pred} = 30.71\mu$s

4. STREAM CIPHER

The previous adaptive parallel `transform` loop has been used to parallelize a stream cipher. A stream cipher is defined from a block cipher *Enc* that takes in input a secret key and a block M_i with fixed size (e.g. 64 bits in the Data Encryption Standard, DES) and returns an encrypted block C_i of M_i with the same size. There are different ways to encrypt/decrypt a data stream using a block cipher *Enc*. These methods are called *modes of operation* and were originally designed for DES in Federal Information Processing Standard (FIPS 81).In 2000 and 2001 the National Institute of Standards and Technology (NIST) sponsored an international workshop in order to give a list of modes of operations for symmetric bloc ciphers [2]). It came out five modes of operations for symmetric block ciphers: Electronic Code-Book (ECB), Cipher Block Chaining (CBC), Cipher Feed-

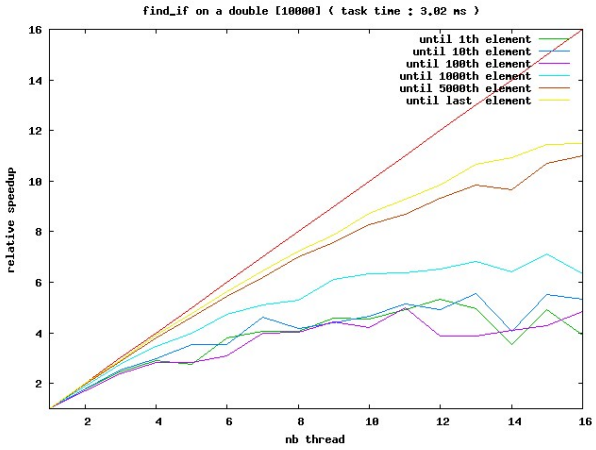


Figure 3: `find_if` – B: $n = 10^4, \tau_{Pred} = 3.02$ ms

Back (CFB), Output FeedBack (OFB) and Counter. As detailed in [29] each mode has its specificity in term of security, efficiency and fault tolerance.

The initial sequential application benchmark is based on CBC mode, each 64 bits block being ciphered by a black box. For the application context, high security against replay attacks was required and therefore Counter mode chosen [21]. Figure 4 shows the counter mode block diagram for

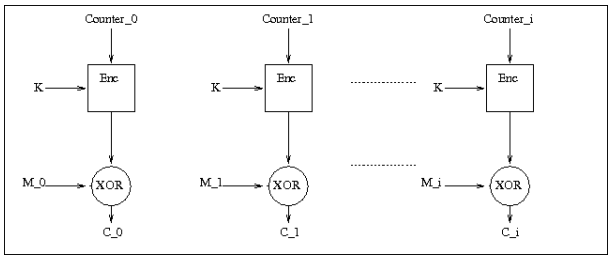


Figure 4: Counter Mode

encryption: M_i is the i^{th} block of the plaintext, its size corresponds to the block size of the symmetric block cipher; K is the secret key used for encryption; $Counter_0$ is an agreed upon value (does not need to be secret but must be different for each messages encrypted with a given secret key K); $\forall i, Counter_i = f(Counter_0, i)$ where f is a function easy to compute with a period on i very large, greater than the number of blocks in the whole plaintext, such as addition as used below; C_i is the i^{th} block of ciphertext (encrypted block); Enc represents the block cipher encryption function.

As one can see, the counter mode allows parallelism since each value $Counter_i$ is encrypted with the block cipher encryption function independently of each other. Furthermore, assuming that the value $Counter_0$ is used only once for a given key, this mode of operation ensures a total security against replay attacks.

Counter mode has been implemented on top of the previous parallel STL `transform` and measured on the same 16-way Opteron machine used on previous `find_if` benchmark. The Enc is the 64 bits blackbox used in the sequential initial benchmark. In Figure 5 the output is written in memory while in Figure 6 it is written to output. In both cases, parallelism improves performances: however, while speed-up is optimal with respect to arithmetic (Figure 5), it is limited by input-output bandwidth (Figure 6).

In this application, the macro-loop is used to avoid waiting for the *a priori* unknown end of the input stream before starting to encrypt/decrypt (similar to the early termination previously explained).

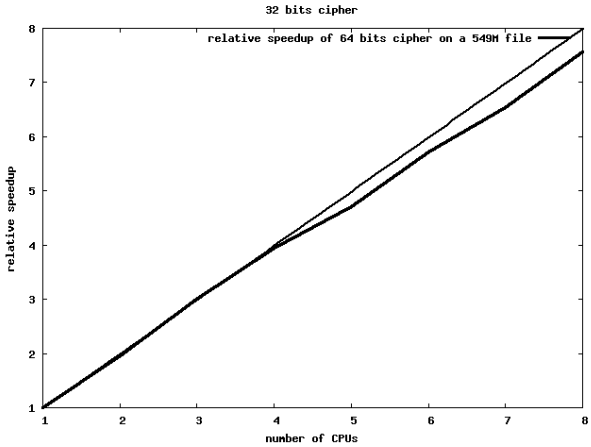


Figure 5: Counter stream cipher in memory (549MB)

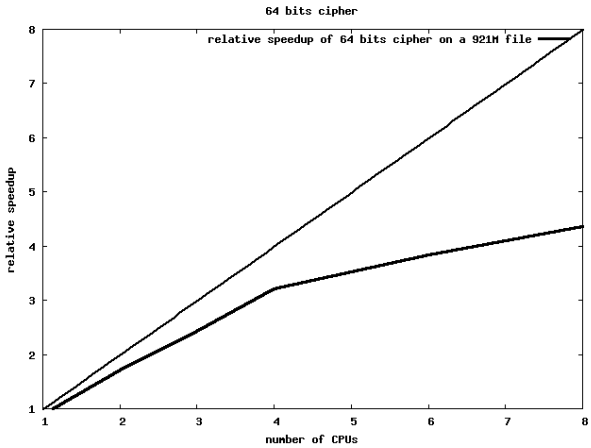


Figure 6: Counter stream cipher with I/O (921MB)

5. SELECTION OF INTERESTING CRYPTOGRAPHIC BOXES

The secret key cryptology has a well known standard — the Advanced Encryption Standard (AES) — which succeeded in 2002 to the Data Encryption Standard. Two of the main qualities of the AES are, first a very good diffusion, and second a substitution box with exceptionally good invariants[2] (δ, λ) coming from the algebraic inversion on a finite field. It was exhibited in [26] for these properties : $(\delta, \lambda) = (4, 16)$. Note that other ciphers have recently been proposed whose boxes exhibit really worse invariants than AES.

In [12] some algebraic attacks of the AES were developed using the algebraic definition of the substitution box. Indeed for this substitution there are 39 quadratic relations between the input bits and the output bits. These attacks can be expressed as the search for solutions of a system of low

[2]see 5.1.3 for the definitions

degree algebraic equations, translating the search of the key and the plain text as a system of 8 000 algebraic quadratic equations in 1 600 unknowns. We are far from being able to solve such a system because it comprises around 2^{10} monomials, but the progress in (parallel) computations may help in a near future. As a result, it is very important to develop research on new good boxes. The substitution boxes are a critical element of a symmetric cipher. Their study is still very sparse.

5.1 Sequential algorithm

One-byte substitution boxes is a one to one map for the $[|0; 255|]$ interval. There exist 256! (i.e. about 2^{2000}) different ones, too many to test them all. Instead, we propose to take random substitution boxes and test the two properties. This allows us to select the best boxes with regards to these two properties. For these selected boxes, we can then check if they have no quadratic relation. Note that [13] already studied 2 million such boxes finding only substitution boxes with $(\delta, \lambda) = (10, 30)$ or $(8, 32)$.

A simple sequential program as been written that tested about 2^{30} within 3 weeks on a standard PC. To get better statistics (and perhaps even better substitution boxes), we need an efficient parallel version that will be able to use as many processors as available. So, at first, we describe here the core (sequential) algorithm. Then, the next subsection (Section 5.2) will explain how it has been parallelized with KAAPI. Eventually the next section (Section 6) will present the results we got on the grid.

5.1.1 Core sequential algorithm

We summarize the sequential algorithm used as the basis of the grid computation. It begins by computing a random substitution on 256 elements then computes cryptographic invariants (δ, λ). Note that because computation of the invariant λ is much longer than the one of δ, we compute the first one only when δ is good enough. We'll briefly recall later the definition of δ and λ, as well as the standard algorithm for random substitutions.

Algorithm 2 Core Sequential Algorithm

```
Data  : Fix N, for example N = 2^30;
Result: a list of B with (δ, λ) = (8, 30) or better.
for i = 0; i < N; i++ do
    Compute a random substitution box B using the
    seed i;
    Compute its δ;
    Record statistics about δ;
    if δ ≤ 8 then
        Compute its λ;
        Record statistics about λ;
        if λ ≤ 30 or δ ≤ 6 then
            print i and B;
        end
    end
end
```

In addition to record good substitution boxes, we also record statistics about δ and λ (when it is computed for the second one). That is, we count how many boxes we find with a given value of δ and λ. As we are only interested in small values of δ and λ, we keep only one class for all values greater or equals to 40.

5.1.2 Random substitution

The core algorithm needs to get random substitution boxes. The easiest way providing random substitutions is explained in [20], see also [14]. We summarize it for substitutions on 256 elements:

Algorithm 3 Building a random permutation

```
build a 256 array T with T[i] = i , i = 0, ..., 255;
for i = 0; i < 255; i++ do
    swap i with a random j, with i ≤ j < 256
end
return T
```

To get uniform random substitution boxes, all we need with this algorithm is a uniform random choice of j. For getting these random numbers we use the famous Mersenne twistor algorithm, see [23], fast with a huge period: $2^{19937} - 1$. See the comments for the adaptation in Grid setting.

5.1.3 Cryptographic invariants for a substitution F

A substitution function $F : \mathbf{F}_2^n \longrightarrow \mathbf{F}_2^n$ used in a block cipher is the only non linear part in it.
If $x = (x_0, \ldots, x_{n-1}) \in \mathbf{F}_2^n$, let $y = (y_0, \ldots, y_{n-1}) = F(x)$.

We now recall the quantities δ_F and λ_F related to differential and linear cryptanalysis that measure the non-linearity of F. They should be small; the AES substitution is exceptionally good with $(\delta_F, \lambda_F) = (4, 16)$.

Differential invariant. The differential invariant δ_F [9, 10] measures the deviation from a linear function verifying $F(a \oplus x) = F(a) \oplus F(x)$ where $\oplus$ is the `xor` operator.
It is defined by $\delta_F = \max_{a,b \in \mathbf{F}_2^n, a \neq 0} \{\delta_F(a, b)\}$

with $\delta_F(a, b) = \#\{x \in \mathbf{F}_2^n : F(x) \oplus F(a \oplus x) = b\}$

Note that δ_F is an even number ≥ 2.

Besides, if F is linear, then $\forall x, y \quad F(x) \oplus F(a \oplus x) = F(a)$. So $\delta_F(a, b) = 2^n$ if $b = F(a)$. Otherwise $\delta_F(a, b) = 0$.

Linear invariant. The linear invariant λ_F indicates whether there exist linear relations between input bits and output bits. Only a likely relation would be useful for an attack.

It is defined by $\lambda_F = \max_{a,b \in \mathbf{F}_2^n, b \neq 0} \{\lambda_F(a, b)\}$ with

$$\lambda_F(a, b) = \left| -2^{n-1} + \#\{x \in \mathbf{F}_2^n; \langle a, x \rangle \oplus \langle b, F(x) \rangle = 0\} \right|$$

where $\langle \alpha, \beta \rangle = \sum_{i=0}^{n-1} \alpha_i \beta_i$ is the scalar product in $\mathbf{F}_2^n$.

5.2 Parallel algorithm and implementation

The sequential algorithm is a loop similar to Algorithm 1 we have to parallelize. We will see how this can be done efficiently with KAAPI and how such an application can be deployed on a grid.

5.2.1 KAAPI

As explain in Section 2, KAAPI can efficiently schedule tasks on a grid with respect to their dependencies. However, when splitting the outer loop of the core sequential algorithm in several tasks, we have to ensure that the cost of stealing and scheduling tasks, sending results, etc. will remain low. Creating one task for each loop would not allow efficient scheduling: each remote node would steal a task, run it, send its results and restart this again and again. There would be far to much steals and communications to

be efficient. So, the algorithm is parallelized with the three nested loops scheme proposed in Section 3.

The nano-loop ensures that each task does not have too much overhead over the sequential algorithm. This means that a KAAPI task will compute several boxes at once. Then task creation, locking and so on will be amortized by this loop. The size of this loop can be automatically computed and dynamically adjusted (see 3.2). However, for this experiment, as the inner loop is very regular, we choose a fixed size and adjust it so that such a task costs about[3] 30s.

Algorithm 4 Nano-loop

```
Data  : N1 and N2 such as N1 < N2
Result: Good boxes and statistics for all n ∈ [N1; N2[
List={};
for i = N1; i < N2; i++ do
    Compute a random substitution box B using the
    seed i;
    Compute its δ;
    Record statistics about δ;
    if δ ≤ 8 then
        Compute its λ;
        Record statistics about λ;
        if λ ≤ 30 or δ ≤ 6 then
            store i in List;
        end
    end
end
return List and statistics about δ and λ;
```

The micro-loop (Section 3.1) is the one responsible for an efficient work stealing by recursive splitting of the research space. This allows remote nodes to steal a large part of the remaining tasks. Since workstealing ensure a small number of steals, remote nodes can make lots of computation before needing to send back results and steal other tasks.

Algorithm 5 Micro-loop

```
Data  : N1 and N2 such as N1 < N2
Result: Good boxes and statistics for all n ∈ [N1; N2[
if N2 − N1 < threshold (nanoloop) then
    return nanoloop(N1, N2);
end
else
    size = (N2−N1)/2 ;
    (List1, Stat1) = Fork ⟨microloop(N1, N1 + size)⟩;
    (List2, Stat2) = Fork ⟨microloop(N1 + size, N2)⟩;
    return Fork ⟨Merge(List1, Stat1, List2, Stat2)⟩ ;
end
```

Dependencies between created KAAPI tasks will be automatically managed by the runtime. So, for example, a *Merge* task will not be scheduled before $List_1$, $Stat_1$, $List_2$, and $Stat_2$, are available, that is, before both previous micro-loops completed.

Eventually, the macro-loop is here to ensure some kind of fault tolerance and a correct management of the allocated time on the available resources. It deals with to early termination due to plate-form crash, network failure or change in machines' reservation. For example, we are able to stop and restart the instance if we need to free a part of the grid we use.

Algorithm 6 Macro-loop

```
Data  : N1 and N2 such as N1 < N2
Result: Good boxes and statistics for all n ∈ [N1; N2[
for i = N1; i < N2; i+= threshold(macroloop) do
    (L, St) = microloop(i, i + threshold(macroloop));
    Record(L, St);
end
```

Following Section 3.3, the threshold used for this loop can also be dynamic. For example, if we set it to twice its previous value at each loop, we ensure we do not lose more than half of the work done. However, as computers on the grid we used are allocated by chunks of hours, we use a fixed threshold of 2^{32}, i.e. we call the micro-loop with chunks of 2^{32} boxes to test.

5.2.2 Parallel random generator

To minimize communication costs and memory footprint of the intermediate results (selected boxes), when we find a good substitution box, we do not record the whole box (i.e. 256 integers) but only the seed of the random generator that allows to rebuild it (i.e. about 64 bits).

The Mersenne twistor library we used in the sequential program did not work 'as is' in the KAAPI version of the program. Indeed, it uses some global variables to track the series between each invocation. However, as KAAPI use several threads on multiprocessor machines to run several task in parallel, random boxes generated from a seed were not reproducible: random series were "corrupted" by calls made to the library by the other threads. But, as sources of the library were available, we made it thread safe: each library function was added a parameter. This new parameter is in fact a pointer to a structure that stores previous values of global variables. These structures are allocated by each threads. As a result, the library becomes thread-safe.

5.2.3 Grid deployment on Grid5000

Grid5000 is a French platform whose main purpose is to serve as an experimental testbed for research in Grid Computing. The physical platform consists of 9 local sites, each with 100 to a thousand PCs, connected by the RENATER Education and Research Network with a 10Gb/s link (or at least 1 Gb/s, when 10Gb/s is not available yet). This platform offers many facilities to create and deploy specific environments and to monitor experiments.

To deploy our software on the grid, we used the TakTuk[22] software. It allowed us to replicate our environment on all sites. The heterogeneity of speed between the different machines is directly handled by KAAPI's scheduling. To port the application on each site, we just needed to perform a recompilation to correctly link the application with the local used libraries. TakTuk also offered us a parallel launcher of our application. This parallel launcher takes a list of machines (the machines we reserved), detects dead nodes, and runs our KAAPI program only on good nodes. This feature turned out to be really useful as, each time we reserved hundreds of nodes, several of them were dead.

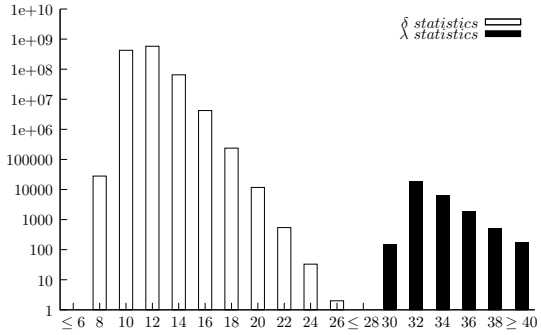
[3] This 30s time interval was chosen such that the depth $D = 30s + T_{recursive\ tasks\ creation}$ is negligible compared to the total cost of one macro-loop ($W \approx 10^6$ s for a chunk of 2^{32} boxes) and such that the cost of one task creation (< 0.1 s) is negligible compared to the cost of one nano-loop (i.e. 30 s).

δ	# instances	fraction of instances
≤ 6	0	0 %
8	28,027	0.0026 %
10	425,465,687	39.6 %
12	578,922,640	53.9 %
14	64,858,233	6.04 %
16	4,218,423	0.39 %
18	236,480	0.022 %
20	11,756	0.0011 %
22	544	0.000051 %
24	33	0.0000031 %
26	2	0.00000017 %

(a) δ statistics

λ	# instances	fraction of instances
≤ 28	0	0 %
30	153	0.55 %
32	19,181	68 %
34	6,115	22 %
36	1,895	6.8 %
38	504	1.8 %
≥ 40	179	0.64 %

(b) λ statistics



(c) cryptographic invariants histogram (log scale)

2^{30} boxes have been tested on 1 processor in 3 weeks. The number of boxes found for each value of δ in presented in Table 7a. For the 28,027 boxes with $\delta \leq 8$, the λ value has been computed (Table 7b). All these results are also presented in the Figure 7c (log scale).

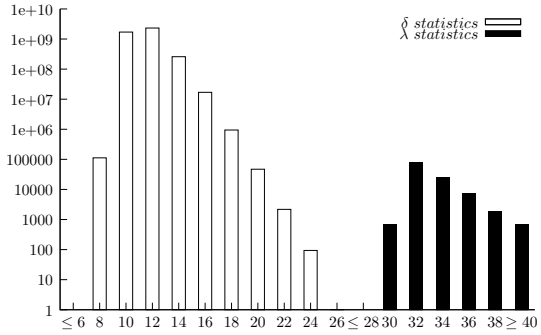
Figure 7: Sequential program

δ	# instances	fraction of instances
≤ 6	0	0 %
8	112,484	0.0026 %
10	1,701,987,184	39.6 %
12	2,315,610,867	53.9 %
14	259,395,688	6.04 %
16	16,867,689	0.39 %
18	944,112	0.022 %
20	47,008	0.0011 %
22	2,168	0.000051 %
24	94	0.0000022 %
26	1	0.000000023 %

(a) δ statistics

λ	# instances	fraction of instances
≤ 28	0	0 %
30	686	0.61 %
32	76,498	68 %
34	25,032	22 %
36	7,633	6.8 %
38	1,943	1.7 %
≥ 40	692	0.62 %

(b) λ statistics



(c) cryptographic invariants histogram (log scale)

2^{32} boxes have been tested on 200 processors in 2 hours. The number of boxes found for each value of δ in presented in Table 8a. For the 112,484 boxes with $\delta \leq 8$, the λ value has been computed (Table 8b). All these results are also presented in the Figure 8c (log scale).

Figure 8: KAAPI program on a cluster

6. VALIDATION ON LARGE SCALE GRIDS OF CLUSTERS WITH SMP NODES

Our experiments are described in Sections 6.1 and 6.2. Comments about the results we got are in Section 6.3.

6.1 Preliminary results

Preliminary tests have been conducted with the sequential program on a standard PC (AMD 2600+ 32 bits, 1GB RAM). We have been able to test 2^{30} substitution boxes within about 3 weeks. This shows us the need for an efficient parallel version of the program to get more interesting results.

153 substitution boxes with $(\delta, \lambda) = (8, 30)$ have been found. All statistics are presented in Figure 7.

Then, a first version of the KAAPI program has been written, without the macro-loop. This version has been run on 200 AMD processors 64bits. On this cluster, we have been able to test 2^{32} substitution boxes within about 2 hours.

686 substitution boxes with $(\delta, \lambda) = (8, 30)$ have been found. All statistics are presented in Figure 8.

6.2 Results on the grid

Eventually, we added the macro-loop and run the KAAPI program on Grid5000. We also optimized the computation of δ and λ with loop unrolls and partial pre-computation. In this case, we reserved 997 machines, that is 2120 processors (most machines are AMD64 bi-processors, some are quadri-processors) for a time between 24 hours and 65 hours (depending on other reservations already done on Grid5000). So we get 66280 hours of processor time. These machines were located on 10 clusters on 7 different sites in France. We have been able to test 264 chunks of 2^{32} substitution boxes.

6 substitution boxes with $(\delta, \lambda) = (8, 28)$ have been found. All statistics are presented in Figure 9.

6.3 Discussion about the results

6.3.1 Cryptographic results

It is interesting to note that the distribution is quite regular: we got around 670 substitution boxes fore each chunks of 2^{32} computations. We had 264 such chunks with a min of 548 and a max of 778. The average ratio is $1/7 \times 10^6$ and is pretty stable.

It was a good surprise to see $(\delta, \lambda) = (8, 28)$ boxes appearing. Although we obtained only 6 of them. We also wished $(6, 30)$ would have occurred but we would need even more boxes for that result!

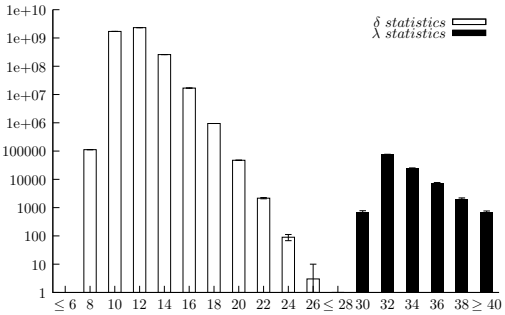
Note that NO substitution boxes our program selected

δ	# instances per chunk			fraction of all instances
	min	max	average	
≤ 6	0	0	0	0 %
8	111,078	113,197	112,070	0.0026 %
10	1,701,882,332	1,702,054,969	1,701,963,689	39.6 %
12	2,315,523,588	2,315,722,018	2,315,612,368	53.9 %
14	259,369,423	259,452,977	259,416,281	6.04 %
16	16,857,620	16,881,634	16,869,866	0.39 %
18	940,786	945,721	943,391	0.022 %
20	46,762	47,954	47,372	0.0011 %
22	2,051	2,267	2,160	5×10^{-05} %
24	68	112	90	2×10^{-06} %
26	0	10	3	8×10^{-08} %
28	0	2	0	3×10^{-09} %
30	0	1	0	9×10^{-11} %

(a) δ statistics for chunks

λ	# instances per chunk			fraction of all instances
	min	max	average	
28	0	1 (in 6 chunks)	0	2.04e-05 %
30	548	778	669	0.597 %
32	75,552	77,044	76,226	68 %
34	24,480	25,318	24,918	22.2 %
36	7,289	7,744	7,504	6.7 %
38	1,962	2,208	2,073	1.85 %
≥ 40	614	762	678	0.605 %

(b) λ statistics for chunks



(c) cryptographic invariants histogram (log scale)

Figure 9: KAAPI on Grid5000

2^{40} boxes have been tested on 2120 processors in a time between 24 hours and 65 hours (depending on the machines availability). The macro-loop had a threshold of 2^{32}. The number of boxes by chunk found for each value of δ in presented in Table 9a. For the boxes with $\delta \leq 8$, the λ value has been computed (Table 9b). The graph (Figure 9c) displays the average number of substitution boxes found in each tested chunk for each value of δ and λ when $\delta \leq 8$. The (rather small) line on top of each box of the graph shows the maximum and the minimum number of substitution boxes between all chunks.

had quadratic relations, a key advantage against algebraic attacks.

Note also that we checked our 30 000 substitution boxes were all different. Even more, their linear affine classes were different: we used the algorithm in [11] to check them all.

6.3.2 Security of Results

When executing large computation on numerous machines, one can be concerned about the correctness of the result they get. Indeed, some machine can have buggy libraries or buggy processors. Additionally, some machines can be compromised. Such concerns are very important for cryptographic works.

In this work, there are several ways to verify the results. The first one is the use of redundancy within the statistics we collect. For example, we count the number of boxes for each classes of δ, but we also count the total number of boxes we try. If the sum of classes does not equal the number of boxes, it means an error occurred. Actually, during the development process, such problem occurred. It was due to a program bug that was thus quickly tracked and resolved.

We can also be sure we got no false-positives. Indeed, once interesting boxes have been identified, it is easy and quick to recompute δ and λ. It was this check that revealed the Mersenne twistor library we used was not thread-safe: rebuild boxes gave wrong δ and λ (see Section 5.2.2). In fact, we need to recompute them among other computations to verify that there are no quadratic relations between the input bits and the output bits of the cryptographic substitution boxes. So, if some computers are compromised, they can hide some good boxes, but they cannot give bad boxes.

7. CONCLUSION AND FUTURE WORK

Many programs are structured around a loop with mostly independent computations. To efficiently parallelize such programs, three complementary levels of parallelism are needed. So, an efficient workstealing scheme can be built which is able to amortize most of the parallelization costs.

KAAPI library and its ATHAPASCAN interface allow to efficiently schedule a graph of tasks with dependencies on a set of machines. We succeeded in using KAAPI to parallelize programs structured around a loop. KAAPI offers some standard interfaces that can directly be used to parallelize a (part of a) program. For example, some operations of the `stl` library have been parallelized to encapsulate the three proposed levels. This enables to automatically and efficiently parallelize codes with KAAPI. For more complex programs, it is still required to use low level tools. The KAAPI/ATHAPASCAN interface is powerful enough to allow easy writing and customization of the three parallel loops with it. This has been done for a cryptographic application that has been launched on a grid with 2120 processors. The results showed that the application uses efficiently most of the available resources.

We are currently working on a new programming interface that would allow one to automatically parallelize this class of programs. This involves for instance dynamical thresholds management [5]. Thresholds will be automatically computed instead of using fixed ones such as our 30 s nano-loop and our chunk size of 2^{32} boxes. We are also adding a stable support in KAAPI to be fault tolerant [19]. This will allow us to continue a grid-application that has been interrupted without the need to restart all the computations. For grid-computing runtime, this is a requirement since interruptions due to resource failure or reallocation are frequent and inherent to grids. Our current preliminary developments in these domains already exhibit promising results.

8. REFERENCES

[1] *Final report of European project number IST-1999-12324, named New European Schemes for Signatures, Integrity, and Encryption.* Springer-Verlag, April 19, 2004.

[2] Report on the second modes of operation workshop. Technical report, National Institute of Standards and Technology (NIST), August 24, 2001.

[3] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theory Comput. Syst.*, 35(3):321–347, 2002.

[4] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theory Comput. Syst.*, 35(3):321–347, 2002.

[5] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. M. Amato, and L. Rauchwerger. STAPL: An Adaptive, Generic Parallel C++ Library. In *LCPC*, pages 193—208, 2001. `http://parasol.tamu.edu/groups/rwergergroup/research/stapl/`.

[6] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst.*, 34(2):115–144, 2001.

[7] O. Beaumont, E. Daoudi, N. Maillard, P. Manneback, and J.-L. Roch. Tradeoff to minimize extra-computations and stopping criterion tests for parallel iterative schemes. In *3rd International Workshop on Parallel Matrix Algorithms and Applications (PMAA04)*, CIRM, Marseille, France, October 2004.

[8] M. A. Bender and M. O. Rabin. Online scheduling of parallel programs on heterogeneous systems with applications to cilk. *Theory Comput. Syst.*, 35(3):289–304, 2002.

[9] E. Biham and A. Shamir. Differential cryptanalysis of DES-like cryptosystems. In *J. Cryptology*, volume 4 (1), pages 3–72, 1991.

[10] E. Biham and A. Shamir. Differential cryptanalysis of the full 16-round DES. In D. W. Davies, editor, *Advances in Cryptology — Crypto'92*, LNCS 740, pages 487–497. Springer-Verlag, 1992.

[11] A. Biryukov, C. de Cannière, A. Braeken, and B. Preenel. A Toolbox for Cryptanalysis: Linear and Affine Equivalence Algorithms. In *Advances in Cryptology - Eurocrypt 2003*, LNCS 2656, pages 33–50. Springer-Verlag, 2003.

[12] N. Courtois and J. Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In *Asiacrypt 2002*, pages 267–287, 2002. Extended version: `http://eprint.iacr.org/2002/044`.

[13] J. Daemen, L. Knudsen, and V. Rijmen. The Block Cipher Square.

[14] R. A. Fisher and F. Yates. *Statistical Tables.* London, 1938. example 12.

[15] M. Frigo, C. Leiserson, and K. Randall. The implementation of the cilk-5 multithreaded language. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[16] F. Galilée, J.-L. Roch, G. Cavalheiro, and M. Doreille. Athapascan-1: On-line Building Data Flow Graph in a Parallel Language. In IEEE, editor, *International Conference on Parallel Architectures and Compilation Techniques, PACT'98*, pages 88–95, Paris, France, October 1998.

[17] T. Gautier, J.-L. Roch, and F. Wagner. Fine grain distributed implementation of a dataflow language with provable performances. In *ICCS 2007 / PAPP 2007 4th Int. Workshop on Practical Aspects of High-Level Parallel Programming*, Beijing, China, May 2007.

[18] T. Gschwind. Pstl-a c++ persistent standard template library. In *COOTS, 6th USENIX Conference on Object-Oriented Technologies and Systems*, pages 147–158, 2001.

[19] S. Jafar, A. W. Krings, T. Gautier, and J.-L. Roch. Theft-induced checkpointing for reconfigurable dataflow applications. In IEEE, editor, *IEEE Electro/Information Technology Conference , (EIT 2005)*, Lincoln, Nebraska, May 2005. This paper received the EIT'05 Best Paper Award.

[20] D. E. Knuth. *The Art of Computer Programming*, volume 2, page 145. Addison-Wesley, Reading, Massachusetts, third edition, 1997.

[21] H. Lipmaa, P. Rogaway, and D. Wagner. Comments to NIST concerning AES-modes of operations: CTR-mode encryption. In *Symmetric Key Block Cipher Modes of Operation Workshop*, Baltimore, Maryland, USA, 2000.

[22] C. Martin, O. Richard, and G. Huard. Déploiement adaptatif d'applications parallèles. In *Techniques et Sciences Informatiques*, volume 24, 2005.

[23] M. Matsumoto and T. Nishimura. Mersenne Twistor: A 623-diminsionally equidistributed uniform pseudorandom number generator. In *ACM Transactions on Modeling and Computer Simulation*, volume 8, pages 3–30, Jan. 1998. `http://www.math.keio.ac.jp/~matumoto/emt.html`.

[24] MOAIS Team. KAAPI. `http://kaapi.gforge.inria.fr/`, 2005.

[25] D. R. Musser, G. J. Derge, and A. Saini. *STL tutorial and reference guide, second edition: C++ programming with the standard template library.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[26] K. Nyberg. Differentially Uniform Mappings for Cryptography. In *Eurocrypt 93*, LNCS 765, pages 55–64. Springer-Verlag, 1993.

[27] F. Putze, P. Sanders, and J. Singler. Mcstl: The multi-core standard template library (extended poster). In *ACM 2007 SIGPLAN Conference on Principles and Practice of Parallel Computing.* Mar 2007.

[28] J.-L. Roch, D. Traore, and J. Bernard. On-line adaptive parallel prefix computation. In L. . Springer-Verlag, editor, *EUROPAR'2006*, pages 843–850, Dresden, Germany, August 2006.

[29] B. Schneier. *Applied cryptography: protocols, algorithms, and source code in C.* Wiley, New York, 2nd edition, 1996, pages 201-226.