Adaptive triangular system solving

Jean-Guillaume Dumas[1], Clément Pernet[1] and Jean-Louis Roch[2]

[1] Université Joseph Fourier, Grenoble I.
Laboratoire de Modélisation et Calcul - IMAG
51, av. des Mathématiques, BP 53X, 38041 Grenoble, FRANCE.
{Jean-Guillaume.Dumas, Clement.Pernet}@imag.fr,
www-lmc.imag.fr/lmc-mosaic/{Jean-Guillaume.Dumas, Clement.Pernet}
[2] Laboratoire Informatique et Distribution, ENSIMAG.
ZIRST 51, avenue Jean Kuntzmann, 38330 Montbonnot Saint Martin, FRANCE.
Jean-Louis.Roch@imag.fr, www-id.imag.fr/∼jlroch

Abstract. We propose a new adaptive algorithm for the exact simultaneous resolution of several triangular systems over finite fields: it is composed of several practicable variants solving these systems (a pure recursive version, a reduction to the numerical dtrsm routine and a delaying of the modulus operation). Then a cascading scheme is proposed to merge these variants into an adaptive sequential algorithm.
We then propose a parallelization of this resolution by a coupling of the sequential algorithm and of the parallel one in order to get the best performances on any number of processors. The resulting cascading is then an adaptation to resources.
This shows that the same process has been used both for adaptation to data and to resources. We thus propose a generic framework enabling automatic adaptation of algorithms using recursive cascading.

1 Introduction

Large-scale applications and software systems are getting increasingly complex. To deal with this complexity, those systems must manage themselves in accordance with high-level guidance from humans. Adaptive and hybrid algorithms enable this self-management of resources and of structured inputs.
In this paper, we first propose a classification of the different notions of adaptivity. For us, an algorithm is *adaptive* when there is a choice at a high level between at least two distinct algorithms, each of which could solve the same problem [1]. The choice is strategic, not tactical. It is motivated by an increase of the performance of the execution, depending on both input/output data and computing resources.
An adaptive algorithm may be

– *simple*: $O(1)$ choices are performed whatever the input (e.g. its size) is. Notice that, while only a constant number of choises are done, each choice can be used several times (an unbounded number of times) during the execution.

[0] This work is supported by the INRIA-IMAG project AHA, aha.imag.fr.

– *baroque*: the number of choices is not bounded: it depends on the input (e.g. its size).

While choices in a simple adaptive alogrithm may be defined statically before any execution, some choices in *baroque adaptive* algorithms are necessarily computed at run time or pre-computed.
The choices may be performed based on machine parameters. But there exist efficient algorithms that do not base their choices on such parameters. For instance, cache-oblivious algorithms have been successfully explored in the context of regular [2] and irregular [3] problems, on sequential and parallel machine models [4]. They do not use any information about memory access times, or cache-line or disk-block sizes. This motivates a second distinction based on the information used:

– An *adaptive* algorithm is *oblivious*, if its control flow depends neither on the particular values of the inputs nor on static properties of the resources.
– An *adaptive* algorithm is *tuned*, if a strategic decision is made based on static resources such as memory specific parameters or heterogeneous features of processors in a distributed computation.
A *tuned* algorithm is *engineered* if a strategic choice is inserted based on a mix of the analysis and knowledge of the target machine and input patterns.
An *adaptive* algorithm is *self-tuned* if the choices are automatically computed by an algorithm.
– An *adaptive* algorithm is *introspective* if it avoids any machine or memory-specific parameterization. Strategic decisions are made based on resource availability or input data properties, both discovered at run-time (such as idle processors). For instance, a strategic decision is made based on assessment of the algorithm performance on the given input up to the decision point.

In this paper we want to use adaptive algorithms to improve exact linear algebra performances.
Indeed, exact matrix multiplication, together with matrix factorizations, over finite fields can now be performed at the speed of the highly optimized numerical BLAS routines[5, 6]. This has been established by the FFLAS and FFPACK libraries [7, 8]. We now discuss the implementation of exact solvers for triangular systems with matrix right-hand side (or equivalently left-hand side). This is also the simultaneous resolution of n triangular systems. The resolution of such systems is e.g. the main operation in block Gaussian elimination. For solving triangular systems over finite fields, the block algorithm reduces to matrix multiplication and achieves the best known algebraic complexity.
After recalling the block recursive scheme of triangular system solving we detail in section 2 two distinct sequential adaptive degenerations. The first one is a reduction to the numerical routine `dtrsm` and the second one is a delaying of the modulus operation. Then we compare their respective behaviors in practice and propose a new self-tuned cascading sequential triangular solver. Then, in section 3, we propose to adapt the work stealing parallel scheduler to obtain a generic baroque coupling of sequential and parallel algorithms and use this scheme to

propose a parallel triangular solver. As a conclusion we then propose a generic recursive framework enabling the automation of the process of adaptation in section 4.
A more detailed classification of adaptive and hybrid algorithms as well as some preliminary results on the sequential variants of triangular solving have been presented in [1]. We here propose a new self-tuned cascading sequential algorithms and more experimental results.

2 Data tuned exact triangular system solving

2.1 Triangular system solving with matrix right-hand side

In the following, we consider without loss of generality that the system to solve is upper triangular and the unknown is a matrix at its right-hand side. Thus we consider the system $AX = B$ where A is a $m \times m$ non singular upper triangular matrix and X and B are $m \times n$ matrices. We consider the case $m \leq n$.
From now on we will denote by ω the exponent of square matrix multiplication (e.g. from 3 for classical, to 2.375477 for Coppersmith-Winograd). Moreover, we can bound the arithmetical cost of a $m \times k$ by $k \times n$ rectangular matrix multiplication (denoted by $R(m, k, n)$) as follows: $R(m, k, n) \leq C_\omega \left\lceil \frac{m}{z} \right\rceil \left\lceil \frac{n}{z} \right\rceil \left\lceil \frac{k}{z} \right\rceil z^\omega$ where $z = \min(m, k, n)$ [9].
This system resolution can be reduced to matrix multiplication by a block recursive algorithm. In the following, we first recall it and then present two variants improving its efficiency. At last, we combine them into a cascade algorithm taking benefit of each of their advantage.

2.2 Scheme of the block recursive algorithm

The recursive algorithm 1 is based on a recursive splitting of the matrices into blocks and a divide and conquer approach.
With $m = n$ and classical matrix multiplication, the arithmetic cost of this algorithm is $TRSM(m) = m^3$ as shown e.g. in [8, Lemma 3.1].
We now also give the cost of the triangular matrix multiplication, TRMM, and of the triangular inversion, INVT, as we will need them in the following sections.
To perform the multiplication of a triangular matrix by a dense matrix, one can use a similar block decomposition:

1. $X_1 :=$`trmm` (A_1, B_1);
2. $B_1 := B_1 + A_2 X_2$;
3. $X_2 :=$`trmm` (A_3, B_2);

The cost is thus $TRMM(m, n) = TRMM(m/2, n) + R(m/2, m/2, n)$. The latter is $TRMM(m, n) = m^2 n$ with classical matrix multiplication.
Now the inverse of the matrix $\begin{bmatrix} A_1 & A_2 \\ & A_3 \end{bmatrix}$ is the matrix $\begin{bmatrix} A_1^{-1} & -A_1^{-1} A_2 A_3^{-1} \\ & A_3^{-1} \end{bmatrix}$, so the cost of its computation is $INVT(m) = 2INVT(m/2) + 2TRMM(m/2, m/2)$. The latter is $INVT(m) = \frac{1}{3} m^3$ with classical matrix multiplication.

Algorithm 1: `trsm` : recursive algorithm

```
Data: $A \in \mathbb{Z}_{/p\mathbb{Z}}^{m\times m}$, $B \in \mathbb{Z}_{/p\mathbb{Z}}^{m\times n}$
Result: $X \in \mathbb{Z}_{/p\mathbb{Z}}^{m\times n}$ s.t. $AX = B$
begin
    if $m=1$ then
        $X := A_{1,1}^{-1} \times B$
    else
        /* (splitting of the matrices into blocks of size $\lfloor \frac{m}{2} \rfloor$
           and $\lceil \frac{m}{2} \rceil$)                                        */
        $$\overbrace{\begin{bmatrix} A_1 & A_2 \\ & A_3 \end{bmatrix}}^{A} \overbrace{\begin{bmatrix} X_1 \\ X_2 \end{bmatrix}}^{X} = \overbrace{\begin{bmatrix} B_1 \\ B_2 \end{bmatrix}}^{B}$$
        $X_2$ :=trsm($A_3, B_2$)
        $B_1 := B_1 - A_2X_2$
        $X_1$ :=trsm($A_1, B_1$)
    return $X$
end
```

2.3 A tuned cascading algorithm

Degenerating to the BLAS "dtrsm" Matrix multiplication speed over finite fields was improved in [7, 10] by the use of the numerical BLAS[1] library: the elements of the input matrices are converted to floating point representations, thus enabling the use of the efficient numerical linear algebra subroutines, and the result is converted back to a finite field representation afterwards. This computation corresponds to an injection of the finite field into $\mathbb{Z}$. The conversion back to the finite field is made possible if the computation only involved additions and multiplications and if now overflow of the integer representation has occured. Therefore this technique can be applied to the `trsm` routine under the following conditions:

- the triangular matrix must have a unit diagonal to avoid the n divisions by each of the diagonal entries of the triangular matrix
- all intermediate integral result must be representable in the floating point mantissa, so the growth of the coefficients must be controlled.

For the first point, it suffices to factorize the triangular matrix $A = AD$, where U is unit triangular and D is diagonal (D is exactly the diagonal of A). After solving the unit diagonal triangular system $UY = B$ (without division), there only remains to compute $X = D^{-1}Y$ by n divisions over the finite field. This normalization leads to an additional cost of $O(mn)$ arithmetic operations (see [8] for more details).

[1] `www.netlib.org/blas`

We now deal with the control of the coefficients growth. The kth row of the solution matrix X is a linear combination of the $(n-k)$ last rows of X, already computed. This imply an exponential growth of the value of the coefficients, or equivalently, a linear growth in their size, with respect to the system dimension. More precisely, if the finite fields elements are represented by integers between 0 and $p-1$, the system must satisfy

$$\frac{p-1}{2}\left[p^{n-1}+(p-2)^{n-1}\right] < 2^{m_a} \tag{1}$$

where m_a is the size of the mantissa [8]. Under this condition, the resolution over the integers using the BLAS trsm routine is exact. For instance, with a 53 bits mantissa, this gives quite small matrices, namely at most 55×55 for $p = 2$, at most 4×4 for $p \leq 9739$, and at most $p = 94906249$ for 2×2 matrices. Nevertheless, this technique is speed-worthy in many cases.
In the following, we will denote by $S_{BLAS}(p)$ the maximal matrix size for which the BLAS resolution is exact. We can now build the first cascade algorithm, `BLASTrsm`: the recursive block algorithm is used until the block size is lower than $S_{BLAS}(p)$, where the BLAS resolution is then applied.

Interleaving delayed modulus The block recursive algorithm consist in several matrix multiplications of different sizes. Over a finite field, this imply numerous unnecessary modular reductions, after each block multiplication. Therefore, we propose to delay these reductions: the matrix multiplications are performed over $\mathbb{Z}$, and the reduction is only applied when necessary. Over $\mathbb{Z}$, the matrix multiplication remains correct as long as

$$k(p-1)^2 < 2^{m_A}, \tag{2}$$

where k is the common dimension of the two matrices to be multiplied.

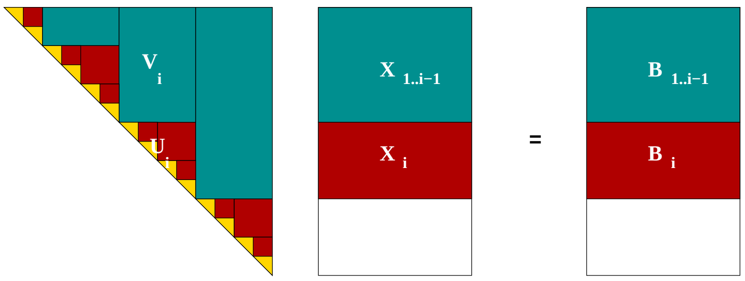

Fig. 1. Splitting for the double cascade `trsm` algorithm

This equation defines a second threshold S_{DEL}: for $k < S_{DEL}$, the matrix multiplication can be performed without modular reductions. To combine it with the previous threshold S_{BLAS}, we define S_{SPLIT} as the larger multiple of S_{BLAS} lower than S_{DEL}. From these two thresholds, one can build the splitting of the system, shown on figure 1.

Algorithm 2: `trsm-blas-del` : Recursive Blas Delayed

```
Data: $A \in \mathbb{Z}_{/p\mathbb{Z}}^{m\times m}$, $B \in \mathbb{Z}_{/p\mathbb{Z}}^{m\times n}$
Result: $X \in \mathbb{Z}_{/p\mathbb{Z}}^{m\times n}$ s.t. $AX = B$
begin
    Compute $S_{\text{BLAS}}(p)$ from equation (1)
    Compute $S_{\text{DEL}}$ from equation (2)
    $S_{\text{SPLIT}} \leftarrow \text{Min}(m/2, S_{\text{DEL}})$
    /* Adjusting $S_{\text{SPLIT}}$ to a multiple of $S_{\text{BLAS}}$ */
    $S_{\text{SPLIT}} \leftarrow (S_{\text{SPLIT}}/S_{\text{BLAS}})S_{\text{BLAS}}$
    foreach block column of $A$ of dimension $m \times S_{SPLIT}$ of the form $\begin{bmatrix} V_i \\ U_i \\ 0 \end{bmatrix}$ do
        $X_i = \texttt{trsm\_del}(U_i, B_i)$
        $X_i = X_i \mod p$
        $B_{1\dots i-1} = B_{1\dots i-1} - V_i X_i$
        $B_{1\dots i-1} = B_{1\dots i-1} \mod p$
    return $X$
end
```

Algorithm 2 is a loop on every block of column dimension S_{DEL}. For each of them, a triangular system is solved using algorithm 3, a modular reduction is performed and the solution is updated by a matrix multiplication. Algorithm 3 is simply the adaptive algorithm, formed by the block recursive algorithm 1 and the BLAS resolution with `dtrsm`. The matrix multiplication are delayed, since the dimension is supposed to satisfy equation (2). So the only modular reductions are performed after the call to `dtrsm`.

2.4 Experiments

We now compare four implementations of the `trsm` routine:

Pure recursive (`pure-rec`): simply algorithm 1,
Recursive-BLAS (`rec-BLAS`): the cascade algorithm formed by the recursive algorithm and the BLAS routine `dtrsm`,
Recursive-Delayed (`rec-delayed`) : the recursive algorithm, with delayed matrix multiplication (`rec-BLAS-delayed` without the cascade to `dtrsm`)
Recursive-BLAS-Delayed (`rec-BLAS-delayed`): the double cascade algorithm (algorithm 2)

We compare these four variants on finite fields with different characteristics, so as to make the parameters S_{BLAS} and S_{DEL} vary as in the following table:

Algorithm 3: trsm_del

```
Data: $A \in {\mathbb{Z}_{/p\mathbb{Z}}}^{m\times m}$, $B \in {\mathbb{Z}_{/p\mathbb{Z}}}^{m\times n}$, $m$ must be lower than $n_{\mathtt{DEL}}$
Result: $X \in {\mathbb{Z}_{/p\mathbb{Z}}}^{m\times n}$ s.t. $AX = B$
begin
    if $m \leq n_{BLAS}$ then
        $X = \mathtt{dtrsm}(A, B)$ ;                          /* the BLAS routine */
        $X = X \mod p$
    else
        /* (splitting of the matrix into blocks of dimension $\lfloor \frac{m}{2} \rfloor$ and
           $\lceil \frac{m}{2} \rceil$)                                                      */
        $\overbrace{\begin{bmatrix} A_1 & A_2 \\ & A_3 \end{bmatrix}}^{A} \overbrace{\begin{bmatrix} X_1 \\ X_2 \end{bmatrix}}^{X} = \overbrace{\begin{bmatrix} B_1 \\ B_2 \end{bmatrix}}^{B}$
        $X_2 :=$trsm_del($A_3, B_2$)
        $B_1 := B_1 - A_2X_2$ ;                    /* without modular reduction */
        $X_1 :=$trsm_del($A_1, B_1$)
    return $X$
end
```

p	$\lceil \log_2 p \rceil$	S_{BLAS}	S_{DEL}
5	3	23	2 147 483 642
1 048 583	20	2	8190
8 388 617	23	2	126

On the experiments of figure 2, the matrix B is square ($m = n$). One can first notice the gain provided by the use of `dtrsm` by comparing the curves `rec-BLAS` and `pure-rec` for $p = 5$. This advantage shrinks when the characteristic gets larger, since $S_{\mathrm{BLAS}} = 2$ for $p = 1\,048\,583$ or $p = 8\,388\,61$.
Now the delaying of the modular reduction improves by 500Mfops the speed of computation. This gain similar for $p = 5$ and $p = 1\,048\,583$ since in both cases $n < S_{\mathrm{DEL}}$ and there is therefore no modular reduction between the matrix multiplications.
Lastly for $p = 8\,388\,617$, the splitting in blocks of size S_{DEL} is much faster than the variants `rec-delayed` and `rec-blas-delayed`. The variants `pure-rec` and `rec-BLAS` are penalised by their dichotomic splitting, creating too many modular reductions after each matrix multiplication.
But we still can not explain the reason why the variant `rec-blas-delayed` is faster than `rec-delayed` for $p = 8\,388\,617$, whereas it is slightly slower for $p = 5$ or $p = 1\,048\,583$.

3 Resource introspective exact triangular system solving

3.1 Parallel *adaptive* algorithms by work-stealing

Cilk [11], Athapascan/Kaapi [12] and Satin [13] are parallel programming interfaces that support nested parallelism and implement a work-stealing scheduling

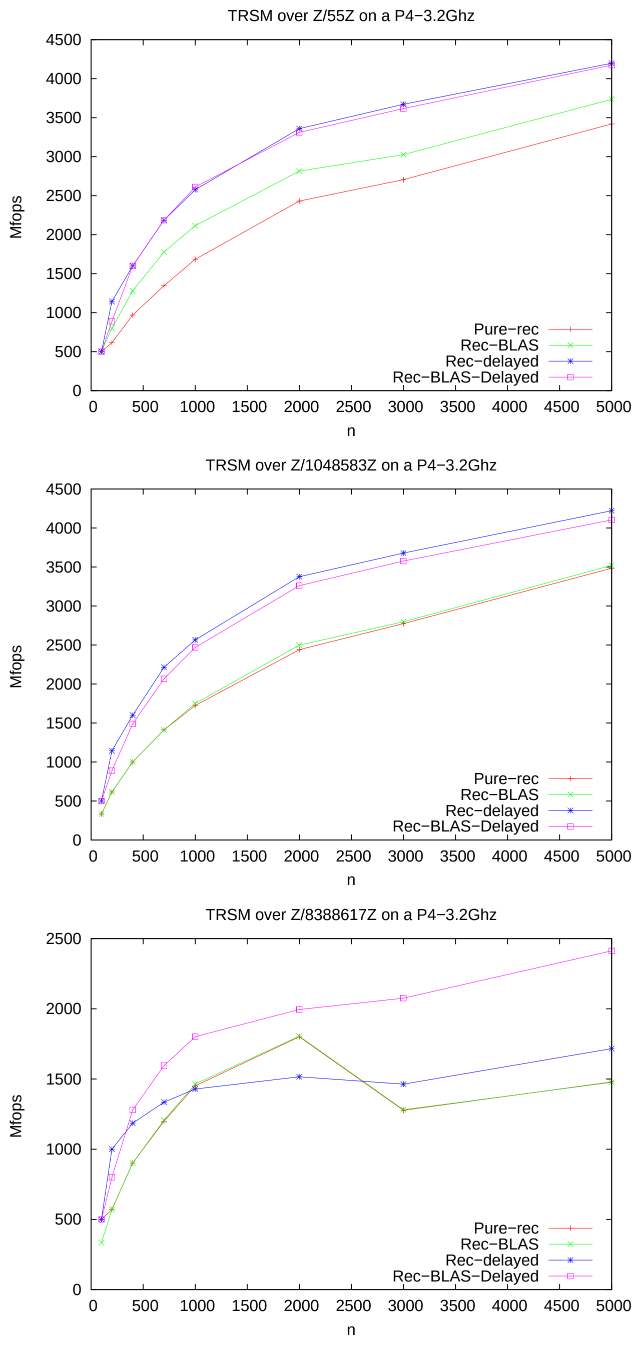

Fig. 2. Comparison of the `trsm` variants for $p = 5, 1\,048\,583, 8\,388\,617$, on a Pentium4-3, 2Ghz-1Go

[14, 15] based on the *work first principle.*. A program explicits parallelism by recursively forking tasks; a task is a procedure call that can be performed on any computational resource. In Athapascan/Kaapi (resp. Cilk), tasks are crated using a `fork`(resp. `spawn`) instruction: `fork f(` *args* `)`, where f is the name of the function and *args* its effective parameters. Due to recursion, at execution, `fork`instructions define a *fork tree* T, similarly to the procedure calls tree in a sequential program.

The tree T describes only task creations, not synchronization and data dependencies that exist between tasks, that define another DAG G, also unfolded at run time, which in general is not a tree. Any schedule that respect dependencies in G is valid; among all parallel schedules, a parallel width (or breadth) first parallel schedule (BFS) consists in executing tasks according to a breadth first trasversal of the tree T but additionally respecting synchronizations defined by G. While in Cilk (and Satin), G is restricted to be serie-parallel, Kaapi enables to program more general dependencies graph.

However, both are based on a sequential lexicographic semantics defined only by T: a depth first sequential execution (DFS), that executes tasks sequentially according to a depth first traversal of the tree T, is a valid schedule. Indeed, when a process creates a task, this task creation can be interpreted: either locally as a sequential function call according to DFS, with small overhead with respect to a direct function call and no synchronization overhead; or as a new thread creation on another process that will perform the task, with then synchronization overhead between both processes to migrate the task and to merge its results. Then the program implements a parallel algorithm (BFS) that can also be executed as a sequential one (DFS). The (recursive) choices between both are performed by the work-stealing schedule.

In a work-stealing schedule [14, 15], each process manages a local double-ended queue (deque) where it stores the tasks it has created, at the bottom of the deque. When a processor completes a task or blocks on a synchronization, two cases arise. Either its local deque contains some ready tasks: then it executes the first one corresponding to a depth-first sequential execution (DFS), by popping from the bootom of its deque. Or its local deque contains no ready task, then the process becomes a thief: it randomly scans the deque of the other processors (steal request), until finding one that contains ready tasks; then it steals a ready task from the top of this victim deque, according to BFS order in the local tree of task on the victim. This stealing operation then corresponds to a breadth first execution (BFS). The work stealing hybrids two algorithms, a DFS schedule and a BFS one. Since each parallel task creation can be performed either by a sequential call (DFS algorithm) or by creation of a new thread (BFS algorithm) depending on resource idleness, any parallel program with non-fixed degree of parallelism is an *adaptive baroque* algorithm. Because the choice does not depend on the input size but only on resource idleness, the algorithm is *adaptive*.

Even if the workstealing is an hybrid of both a sequential (DFS) and parallel (BFS) schedule, the algorithm (i.e. the tree T and the data dependencies in G) remains the same, since a single algorithm is executed. In the next section, we extend this model to hybrid two distinct algorithms, one sequential the other parallel, based on the workstealing.

3.2 *Baroque* coupling of sequential and parallel algorithms

We present the coupling, based on workstealing, of a sequential algorithm f_{seq} and a parallel one f_{par} that both solve the same problem f. Following [16, 17], we assume that the sequential algorithm performs a first part of the sequential computation (called *ExtractSeq*) and then performs a recursive terminal call to f to complete the computation. Besides, we assume that the sequential algorithm is such that at any time of its execution, the sequence of operations that completes the algorithm f_{seq} can be performed by another parallel recursive (fine grain) algorithm f_{par}. The operation that consists in extracting the last part of the sequential computation in progress to perform it in parallel with f_{par} is called *ExtractPar*. After completion of f_{par}, the final result is computed by merging both the result of the first part computed by f_{seq} (not affected by *ExtractPar*) and the result of the *ExtractPar* part computed by f_{par}.
More precisely, given a sequential algorithm f_{seq} (resp. parallel f_{par}), the result r of its evaluation on an input x is denoted $f_{\text{seq}}(x)$ (resp. $f_{\text{par}}(x)$). We assume that x has a list structure with a concatenation operator $\sharp$ and that there exists an operator $\oplus$ (not necessarily associative) for merging the results. At any time of evaluation of $f_{\text{seq}}(x)$, x can be split into $x_1\sharp x_2$, due to either an *ExtractSeq* or an *ExtractPar* operation on x. The result computed by the parallel algorithm is then $f_{\text{par}}(x) = f(x_1) \oplus f(x_2)$. We assume that both results $f_{\text{seq}}(x)$ and $f_{\text{par}}(x)$ are equivalents with respect to a given measure. In the restricted framework of list homomorphism [18], this hypothesis can be written as $f(x\sharp y) = f(x) \oplus f(y)$. However, it is possible to provide parallel algorithms for problems that are not list homomorphisms [19] at the price of an increase in the number of operations.
To decrease overhead related to choices for f between f_{seq} and f_{par}, f_{seq} is the default choice used. Based on a workstealing scheduling, f_{par} is only chosen when a processor becomes idle, which leads to an *ExtractPar* operation.
This exception mechanism may be implemented by maintaining during any execution of $f_{\text{seq}}(x)$ a lower priority process ready to perform an *ExtractPar* operation on x resulting in an input x_2 for f_{par} only when a processor becomes idle.
Then the overhead due to choices is only related to the number of *ExtractPar* operations actually performed.
To analyze this number, we adopt the simplified model of Cilk-5 [11] also valid for *Kaapi* [12]; it relies on the bound on the number of steals requests performed by the workstealing (see theorems 9 in [14]). Let $T_1^{(\text{seq})}$ (resp. $T_1^{(\text{par})}$) be the execution time on a sequential processor (i.e. work) of f_{seq} (resp. f_{par}), and let $T_\infty^{(\text{par})}$ be the execution time of f_{par} on an unbounded number of identical processors.

Theorem 1. *When the* adaptive *program is executed on a machine with m identical processors, the expected number of choices f_{par} instead of f_{seq} for f is bounded by* $(m-1).T_\infty^{(\text{par})}$

Proof. On an infinite number of processors, all the computation is performed by f_{par}; the parallel time of the *adaptive* algorithm is then $T_\infty^{(\text{par})}$. The execution is based on the workstealing schedule. From theorem 9 in [14], the expected number of steal requests is bounded by $T_\infty^{(\text{par})}$ on each processor (Graham's bound), except for the one running f_{seq}. The latter only executes the sequential algorithm, but is subject to ExtractPar, due to steal requests from the others. This is true for any execution of such *adaptive baroque* algorithm.

The consequence of this theorem is that for a fine grain parallel algorithm that satisfies $T_\infty^{(\text{par})} \ll T_1^{(\text{seq})}$, even if the *adaptive* algorithm is *baroque* (non constant number of choices), the overhead in time due to choices in the *adaptive* algorithm is negligible when compared to the overall work.

Remark. The overhead due to the default call to *ExtractSeq* can also be reduced. Ideally, *ExtractSeq* should extract a data whose computations by f_{par} would require a time at least $T_\infty^{(\text{par})}$, which is the critical time for f_{par}.
In the next section we apply the above coupling of a sequential and a parallel algorithm to the `TRSM` algorithm.

3.3 *Baroque introspective* parallel Trsm

The previous algorithm takes benefit of parallelism at the level of Blas matrix product operations. However, using the scheme proposed in §3.2, it is possible to obtain an algorithm with more parallelism in order to decrease the critical time when more processors are available. Furthermore, this also improves the performance of the distributed work-stealing scheduler.
Indeed, while X_2 and B_1 are being computed, additional idle processors may proceed to the parallel computation of A_1^{-1}. Indeed, X_1 may be computed in two different ways:

i. $X_1 = TRSM(A_1, B_1)$: the arithmetic cost is $T_1 = k^3$ and $T_\infty = k$;
ii. $X_1 = TRMM(A_1^{-1}, B_1)$: the cost is the same $T_1 = k^3$ but $T_\infty = \log k$.

Indeed the version (ii) with TRMM is more efficient on a parallel point of view: the two recursive calls and the matrix multiplication in (ii) (TRMM) are independent. They can be performed on distinct processors requiring less communications than TRSM.
Since precomputation of A_1^{-1} increases the whole arithmetic cost, it is only performed if there are extra unused processors during the computation of X_2 and B_1; the latter has therefore higher priority.
The problem is to decide the size k of the matrix A_1 that will be inverted in parallel. With the optimal value of k, the computation of A_1^{-1} completes

simultaneously with that of X_2 and B_1. This optimal value of k depends on many factors: number of processors, architecture of processors, subroutines, data. The algorithm presented in the next paragraph uses the *oblivious adaptive* scheme described in 3.2. to estimate this value at runtime using the *introspective* coupling of a "sequential" algorithm f_s with a parallel one f_p.

Parallel *introspective* TRSM Th parallel *introspective* TRSM algorithm 4 consists in computing concurrently in parallel (Figure 3):

- "sequential" computation (f_s) at high priority: bottom-up computation of $X = TRSM(A, B)$ till reaching k, implemented by the parallel BUT algorithm (Bottom-Up TRSM - §5); all processes that perform parallel BLAS operations in BUT are executed at high priority;
- parallel computation (f_p) at low priority: parallel top-down inversion of A till reaching k, implemented by TDTI algorithm (Top Down Triangular Inversion - §6); all processes that participates in parallel TDTI are executed at low priority.

Algorithm 4: AdaptiveParallelTrsm($A; B$)

```
Data: $A \in \mathbb{Z}_{/p\mathbb{Z}}^{m\times m}$, $B \in \mathbb{Z}_{/p\mathbb{Z}}^{m\times n}$.
Result: $X \in \mathbb{Z}_{/p\mathbb{Z}}^{m\times n}$ such that $AX = B$.
begin
    $k_{TDTI} := 0$ ; $k_{BUT} := m$;
    Parallel
        At high priority: $(X_2, B_1') := BUT(A, B)$;
        At low priority: $M := TDTI(\emptyset, A)$;
    endpar
    /* Now, BUT has stopped TDTI and $k_{BUT} \leq k_{TDTI}$ */
    Let $A_1'^{-1} = M_{1..k_{BUT},1..k_{BUT}}$ and compute $X_1 := A_1'^{-1}.B_1'$;
    return $\begin{bmatrix} X_1 \\ X_2 \end{bmatrix}$
end
```

At each step, the sequential bottom-up BUT algorithm (resp. the parallel top-down TDTI) performs an ExtractSeq (resp. ExtractPar) operation on a block of size k_B (resp. k_I) (Figure 3 and detailed subroutines BUT and TDTI are following). Note that the values of k_B and k_I may vary during the execution depending on the current state.

Bottom-up TRSM We need to group the last recursive `trsm` call and the update of B_1. Algorithm 5 thus just computes these last two steps ; the first step being performed by the work stealing as shown afterwards.

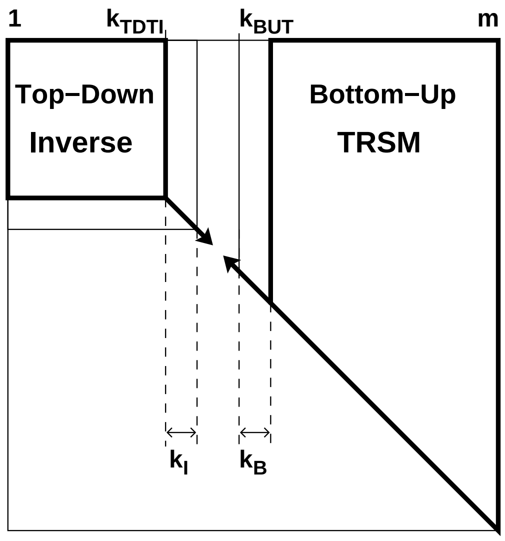

Fig. 3. Parallel *introspective* TRSM

```
Algorithm 5: BUT: Bottom-up trsm
Data: $(A_2; A_3; B)$
Result: $X_2$, $k_{BUT}$.
begin
    Mutual Exclusion
        if $k_{TDTI} \geq k_{BUT}$ then
            Return;
        $k_B := Choice(1..(k_{BUT} - k_{TDTI}))$;
        Split remaining columns into $k_{TDTI}..(k_{BUT} - k_B)$ and
        $(k_{BUT} - k_B)..k_{BUT}$:
            $$\begin{bmatrix} A_{2,1} & A_{2,2} \\ A_{3,1} & A_{3,2} \\ & A_{3,3} \end{bmatrix} \begin{bmatrix} X_{2,1} \\ X_{2,2} \end{bmatrix} = \begin{bmatrix} B_1 \\ B_{2,1} \\ B_{2,2} \end{bmatrix}$$
        $k_{BUT} := k_{BUT} - k_B$;
    endmutex
    $X_{2,2}$ :=trsm $(A_{3,3}, B_{2,2})$;
    $B_1 := B_1 - A_{2,2}X_{2,2}$;
    $B_{2,1} := B_{2,1} - A_{3,2}X_{2,2}$;
    $X_{2,1}$ :=BUT$\left(A_{2,1}; A_{3,1}; \begin{bmatrix} B_1 \\ B_{2,1} \end{bmatrix}\right)$ ;
end
```

Top down triangular inversion of A_1 Algorithm 6 is a block inversion of the top part of the triangular matrix. It is fully parallel as it computes the next triangular inverse while updating the upper rectangular part.

Algorithm 6: `TDTI`: Top-down triangular inverse

```
Data: $(A_1^{-1}; A_2; A_3)$.
Result: $A^{-1}$, $k_{TDTI}$.
begin
    Mutual Exclusion
        if $k_{TDTI} \geq k_{BUT}$ then
            Return;
        $k_I := Choice(1..(k_{BUT} - k_{TDTI}))$;
        Split remaining columns of $A_2$ and $A_3$ into
        $k_{TDTI}..(k_{TDTI} + k_I)$ and $(k_{TDTI} + k_I)..k_{BUT}$;
            $\begin{bmatrix} A_{2,1} & A_{2,2} \\ A_{3,1} & A_{3,2} \\ & A_{3,3} \end{bmatrix}$
    endmutex
    Parallel
        $A_{3,1}^{-1}$ :=Inverse($A_{3,1}$);
        $T := A_1^{-1}.A_{2,1}$;
    endpar
    $A'_{2,1} = -T.A_{3,1}^{-1}$;
    Let $A_1'^{-1} = \begin{bmatrix} A_1^{-1} & A'_{2,1} \\ & A_{3,1}^{-1} \end{bmatrix}$ and $A'_2 = \begin{bmatrix} A_{2,2} \\ A_{3,2} \end{bmatrix}$;
    Mutual Exclusion
        $k_{TDTI} := k_{TDTI} + k_I$;
    endmutex
    $A_{3,3}^{-1}$ :=TDTI($A_1'^{-1}$;$A'_2$;$A_{3,3}$);
end
```

Definiton of parameters k_I and k_B Parameters k_B (resp. k_I) corresponds to the *ExtractSeq* (resp. *ExtractPar*) operations presented in §3.2. The choice of their values is performed at each recursive step, depending on resources availability. This section analyzes this choice in the case where only one system is to be solved, i.e. $n = 1$.
Let $r = k_{BUT} - k_{TDTI}$.

– On the one hand, to fully exploit parallelism, k_B should not be larger than the critical time T_∞ of TDTI, i.e. $k_B = \log^2 r$.

– On the other hand, in order to keep an $O(n^2)$ number of operations if no more processors become idle, the number of operations $O(k_I^3)$ required by $TDTI$ should be balanced by the cost of the update, i.e. $k_I.r$, which leads to $k_I = \sqrt{r}$.

With those choices of k_I and k_B, and assuming that there are enough processors, the number of choices for k_I (and so k_B) will then be $O(\sqrt{r})$; the cost of the resulting *adaptive* algorithm becomes $T_1 = O(n^2)$ and $T_\infty = O(\sqrt{n}\log^2(n))$, a complexity similar to the one proposed in [20] with a fine grain parallel algorithm, while this one is coarse grain and dynamically adapts to resource idleness. Notice that if only a few processors are available, the parallel algorithm will be executed at most on one block of size $\sqrt{n}$. The BUT algorithm will behave like the previous *tuned* TRSM algorithm. Also, the algorithm is *oblivious* to the number of resources and their relative performance.

4 Generic framework for adaptive algorithms

The previous general adaptive algorithm for `Trsm` is based on the coupling of various recursive algorithms self-adapting to input data and processor load. In this section we generalize this coupling in order to provide a generic scheme for *baroque adaptive* algorithms based on recursive choices performed at runtime.

4.1 Recursive representation

Let f be a problem with input set I and output set O. For the computation of f, an *adaptive* algorithm is based on the composition of distinct algorithms $(f_i)_{i=1,\ldots,k}$, each solving the problem f. The sequential and parallel algorithms presented in the previous sections are all instances of distinct f_i for the computation of f=`Trsm`. Since an algorithm is finite, the number $k \geq 2$ of algorithms is finite; however, each of those algorithms may use additional parameters, based on the inputs, outputs or machine parameters (e.g. number of processors).
Generalizing previous algorithms for `Trsm`, we assume that each of those algorithms is written in a recursive way: to solve a given instance of f, algorithm f_i reduces it to subcomputations instances of f of smaller sizes. Adaptivity then consists in choosing for each of those subcomputations the suited algorithm f_j to be used (fig. 4). This choice can be implemented in various ways. For instance, f may be implemented as a pure virtual function, each of the f_i being an inherited specialization.
The benefits of this recursive approach is that the work already performed before any choice is automatically reused by the new variant taking over. In this way the overhead of the adaptive strategy is only that of the performed tests. In the TRSM case some of these tests can even be statically pre-computed.
Furthermore, this recursive scheme can also give way to complexity improvements: a fast practicable algorithm with bad worst-case complexity can be used to perform the first part of the work ; then a proven better algorithm with slower practicable performances can be used in a smaller part of the problem. This has been successfully used e.g. in the case of the integer determinant [21].

```
Algorithm $f_i$ ( I n, input, O output, ... ) {
  ...
  f( n-1, ... ) ;
  ...
  f( n / 2, ... ) ;
  ...
};
```

Fig. 4. Recursive description of an *adaptive* algorithm f_i.

4.2 Overhead due to choices and workstealing

For *baroque* algorithms the choices between the different f_i's are performed at runtime . Therefore an important problem is related to reducing the overhead related to the computation of each choice with respect to the arithmetic cost of the computation. Also, for a given subcomputation, a default given algorithm f_1 is favored; but this choice may be changed under some exceptional circumstances depending on values or machine parameters. Then, if the total number of such exceptions is small with respect to the total number of subcomputations, the overhead due to choices may become negligible.
For instance, in the case of workstealing schedule, a sequential execution is the default choice. We assume that this default choice corresponds to the algorithm f_1. Based on theorem 1, the expected number of choices different from f_1 is bounded with respect to the critical time T_∞ of the execution on an unbounded number of processors. Also, we assume that any of the (possibly many) algorithms f_i that may be chosen by the workstealing when a processor becomes idle (*ExtractPar* operation) have a critical time bounded by T_∞ on an infinite number of processors. Then the expected number of choices different from f_1 on a finite number p of processor is $O(T_\infty)$ on any processor, whatever the value of p and the whole number of operations performed. In the case of workstealing, the overhead due to choices is then bounded.

5 Conclusion

Designing efficient *adaptive* algorithms is the key to get most of the available resources and most of the structure of the inputs. We have shown e.g. for exact linear algebra or for combinatorial optimization Branch&X [1], that this is true for numerous applications. From a classification of the distinct forms of *adaptive* algorithms, we have proposed in this paper a generic framework to express this adaptivity. On a single simple example, namely solving linear systems, we show that several of these “adaptivities” can appear. This enables an effective adaptation of the algorithm and a nice way to adapt automatically its behavior, independent of the execution context. This is true in a parallel context where coupling of algorithms is critical to obtain a high performance.

The resulting algorithm is quite complex but can be nearly automatically generated in our simple framework. The requirements are just to provide recursive versions of the different methods.
In the AHA group, such coupling are studied in the context of many examples: vision and adaptive 3D-reconstruction, linear algebra in general, and combinatorial optimization.

References

1. Cung, V.D., Danjean, V., Dumas, J.G., Gautier, T., Huard, G., Raffin, B., Rapine, C., Roch, J.L., Trystram, D.: Adaptive and hybrid algorithms: classification and illustration on triangular system solving. [22]
2. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science, Washington, DC, USA, IEEE Computer Society (1999) 285
3. Bender, M.A., Demaine, E.D., Farach-Colton, M.: Cache-oblivious b-trees. SIAM J. Comput. **35** (2005) 341–358
4. Bender, M.A., Fineman, J.T., Gilbert, S., Kuszmaul, B.C.: Concurrent cache-oblivious b-trees. In: SPAA'05: Proceedings of the 17th annual ACM symposium on Parallelism in algorithms and architectures, New York, NY, USA, ACM Press (2005) 228–237
5. Goto, K., van de Geijn, R.A.: Anatomy of High-Performance Matrix Multiplication. ACM Transactions on Mathematical Software (2006) Submitted.
6. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated Empirical Optimizations of Software and the ATLAS Project. Parallel Computing **27** (2001) 3–35 `www.elsevier.nl/gej-ng/10/35/21/47/25/23/article.pdf`.
7. Dumas, J.G., Gautier, T., Pernet, C.: Finite Field Linear Algebra Subroutines. In Mora, T., ed.: Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation, Lille, France, ACM Press, New York (2002) 63–74
8. Dumas, J.G., Giorgi, P., Pernet, C.: FFPACK: Finite Field Linear Algebra Package. In Gutierrez, J., ed.: Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation, Santander, Spain, ACM Press, New York (2004) 119–126
9. Huang, X., Pan, V.Y.: Fast rectangular matrix multiplications and improving parallel matrix computations. In ACM, ed.: PASCO '97. Proceedings of the second international symposium on parallel symbolic computation, July 20–22, 1997, Maui, HI, New York, NY 10036, USA, ACM Press (1997) 11–23
10. Pernet, C.: Implementation of Winograd's matrix multiplication over finite fields using ATLAS level 3 BLAS. Technical report, Laboratoire Informatique et Distribution (2001) `www-id.imag.fr/Apache/RR/RR011122FFLAS.ps.gz`.
11. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the cilk-5 multithreaded language. In: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, ACM Press (1998) 212–223
12. Jafar, S., Gautier, T., Krings, A.W., Roch, J.L.: A checkpoint/recovery model for heterogeneous dataflow computations using work-stealing. In Springer-Verlag, L., ed.: EUROPAR'2005, Lisboa, Portugal (2005)
13. van Nieuwpoort, R.V., Maassen, J., Kielmann, T., Bal, H.E.: Satin: Simple and efficient java-based grid programming. Scalable Computing: Practice and Experience **6** (2005) 19–32

14. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. Theory Comput. Syst. **34** (2001) 115–144
15. Acar, U.A., Blelloch, G.E., Blumofe, R.D.: The data locality of work stealing. Theory Comput. Syst. **35** (2002) 321–347
16. Daoudi, E.M., Gautier, T., Kerfali, A., Revire, R., Roch, J.L.: Algorithmes parallèles à grain adaptatif et applications. Technique et Science Informatiques **24** (2005) 1—20
17. Roch, J.L., Traore, D., Bernard, J.: On-line adaptive parallel prefix computation. In Springer-Verlag, L., ed.: EUROPAR'2006 Conference. (2006)
18. Bird, R.: Introduction to the Theory of Lists. In: Logic of Programming and Calculi of Discrete Design. Springer-Verlag (1987)
19. Cole, M.: Parallel Programming with List Homomorphisms. Parallel Processing Letters **5** (1995) 191–204
20. Pan, V.Y., Preparata, F.P.: Work-preserving speed-up of parallel matrix computations. SIAM Journal on Computing **24** (1995)
21. Dumas, J.G., Urbanska, A.: An introspective algorithm for the determinant. [22]
22. Dumas, J.G., ed.: TC'2006. Proceedings of Transgressive Computing 2006, Granada, España. In Dumas, J.G., ed.: Proceedings of Transgressive Computing 2006, Granada, España. (2006)