Adaptive and Hybrid Algorithms: classification and illustration on triangular system solving*

Van Dat Cung Vincent Danjean Jean-Guillaume Dumas
Thierry Gautier Guillaume Huard Bruno Raffin Christophe Rapine
Jean-Louis Roch Denis Trystram

Abstract

We propose in this article a classification of the different notions of hybridization and a generic framework for the automatic hybridization of algorithms. Then, we detail the results of this generic framework on the example of the parallel solution of multiple linear systems.

Introduction

Large-scale applications, software systems and applications 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 structured inputs. In this paper, we propose a classification of the different notions of hybridization and a generic framework for the automatic hybridization of algorithms. We illustrate our framework in the context of combinatorial optimizations and linear algebra, in a sequential environment as well as in an heterogeneous parallel one. In the sequel, we focus on hybrid algorithms with provable performance. Performance is measured in terms of sequential time, parallel time or precision.
After surveying, classifying and illustrating the different notions of hybrid algorithms in section 1, we propose a generic recursive framework enabling the automation of the process of hybridization in section 2. We then detail the process and the result of our generic hybridization on the example of solving linear systems in section 3.

1 A survey and classification of hybrid algorithms

1.1 Definitions and classification

In this section we propose a definition of *hybrid* algorithm, based on the notion of strategic choices among several algorithms. We then refine this definition to propose a classification

*This work is supported by the INRIA-IMAG project AHA: Adaptive and Hybrid Algorithms.

of hybrid algorithms according to the number of choices performed (*simple, baroque*) and the amount of inputs/architecture information used (*tuned, adaptive, introspective, oblivious, engineered*). Figure 1 summarizes this classification.

Definition 1.1 (Hybrid). An algorithm is *hybrid* (or a poly-algorithm) when there is a choice at a high level between at least two distinct algorithms, each of which could solve the same problem.

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. The following criterion on the number of choices to decide is used to make a first distinction among *hybrid* algorithms.

Definition 1.2 (Simple versus Baroque). A *hybrid* 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. Parallel divide&conquer algorithms illustrate this point in next section.
- *baroque*: the number of choices is not bounded: it depends on the input (e.g. its size).

While choices in a simple *hybrid* alogrithm may be defined statically before any execution, some choices in *baroque hybrid* algorithms are necessarily computed at run time.
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 [11] and irregular [1] problems, on sequential and parallel machine models [2]. 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.

Definition 1.3 (oblivious, tuned, engineered, adaptive, introspective). Considering the way choices are computed, we distinguishe the following class of *hybrid* algorithms:

- A *hybrid* algorithm is *oblivious*, if its control flow depends neither on the particular values of the inputs nor on static properties of the resources.
- A *hybrid* 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. A *hybrid* algorithm is *self-tuned* if the choices are automatically computed by an algorithm.
- A *hybrid* algorithm is *adaptive* if it avoids any machine or memory-specific parameterization. Strategic decisions are made based on resource availability or input data

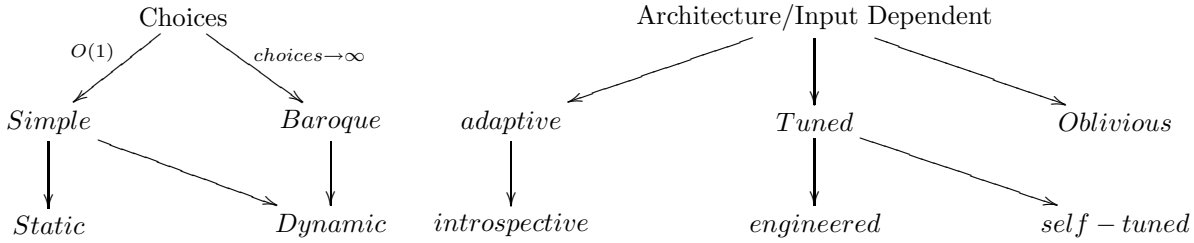


Figure 1: Classification of *hybrid* algorithms

properties, both discovered at run-time (such as idle processors).
An adaptive algorithm is *introspective* if a strategic decision is made based on assessment of the algorithm performance on the given input up to the decision point.

In [12], Ganek and Corbi defined *autonomic computing* to be the conjunction of self-configuring, self-healing, self-optimizing and self-protecting systems. Self-configuring relates to what we call adaptivity, self-optimizing to self-tuning. Autonomic computing thus adds fault-tolerance (self-healing) and security (self-protecting) to our notion of *hybrid* computing. Above definitions deliberately focus on a general characterization of adaptation in the algorithm. They consider neither implementation nor performance. To implement an *adaptive* algorithm, we may distinguish two approaches. Either the choices are included in the algorithm itself, or they may be inserted dynamically to change the software itself, or its execution environment. An algorithm is *evolutive* (or interactive) if a strategic choice is inserted dynamically. Reflexive languages enable to change the behavior of a program dynamically [19]. Polymorphism or template specialization is a way to optimize an algorithm. We view polymorphism and template mechanisms as a possible way to implement the different kinds of *hybrid* algorithm we propose.

1.2 Illustrations on examples

We illustrate the previous criteria on some examples of *hybrid* algorithms or libraries.

BLAS libraries. ATLAS [24] and GOTO [14] are libraries that implement basic linear algebra subroutines. Computation on matrices are performed by blocks. The block size and the sequential algorithm used for a basic block are chosen based on performance measures on the target architecture. The decisions are computed automatically at installation with ATLAS while they are provided only for some architectures with GOTO. ATLAS implements *self-tuned simple hybrid* algorithms and GOTO *simple engineered* ones.

Granularity in sequential divide&conquer algorithms. Halting recursion in divide&conquer to complete small size computations with another more efficient algorithm is

a classical technique to improve the practical performance of sequential algorithms. The resulting algorithm is a *simple hybrid* one. Often the recursion threshold is based on resource properties. This is the case for the GMP integer multiplication algorithm that successively couples four algorithms: Schönhage-Strassen $\Theta(n \log n \log\log n)$, Toom-Cook 3-way ($\Theta(n^{1.465})$), Karatsuba $\Theta(n^{log_2 3})$ and standard $\Theta(n^2)$ algorithms.

Linpack benchmark for parallel LU factorization. Linpack [6] is one milestone in parallel machines' power evaluation. It is the reference benchmark for the top-500 ranking of the most powerful machines. The computation consists in a LU factorization, with raw partial pivoting in the "right-looking" variant [6], the processors assumed being identical. To limit the volume of communication to $O(n^2\sqrt{p})$, a cyclic bidimensional block partitioning is used on a virtual grid of $q^2 = p$ processors. The block (i, j) is mapped to the processor of index $P(i, j) = (i \bmod q)q+(j \bmod q)$ and operations that modify block (i, j) are scheduled on processor $P(i, j)$. Linpack has a standard implementation on top of MPI with various parameters that may be tuned: broadcast algorithm (for pivot broadcasting on a line of processors), level of recursion in the "right-looking" decomposition algorithm and block size. The parallel architecture may also be tuned to improve the performance [22]. Linpack is an *engineered tuned simple hybrid* algorithm.

FFTW. FFTW [10] is a library that implements discrete Fourier transform of a vector of size n. We summarize here the basic principle of FFTW. For all $2 \leq q \leq \sqrt{n}$, the FFT Cooley-Tuckey recursive algorithm reduces to q FFT subcomputations of size $\left\lceil \frac{n}{q} \right\rceil$ and $\left\lceil \frac{n}{q} \right\rceil$ FFT subcomputations of size q, plus $O(n)$ additional operations. Hybridization in FFTW occurs at two levels:

- at installation on the architecture. For a given n_0 the best unrolled FFT algorithm for all $n \leq n_0$ is chosen among a set of algorithms by experimental performance measurements. This *hybrid* algorithm is *simple tuned*.
- at execution. FOr a given size n of the input vectors and for all $n_0 \leq m \leq n$, a planner precomputes the splitting factor q_m that will be further used for any recursive FFT with size m. This precomputation is performed by dynamic programming: it optimizes each sub-problem of size m locally, independently of the larger context where it is invoked. The planner adds a precomputation overhead. This overhead may be amortized by using the same plan for computing several FFTs of the same size n. FFTW3 also proposes heuristic algorithms to compute plans with smaller overhead than dynamic programming.

The number of choices in FFTW depends on the size n of the inputs. FFTW is a *self-tuned baroque hybrid* algorithm.

Granularity in parallel divide&conquer algorithms. Parallel algorithms are often based on a mix of two algorithms: a sequential one that minimizes the number of operations

T_1 and a parallel one that minimizes the parallel time T_∞. The cascading divide&conquer technique [17] is used to construct a *hybrid* algorithm with parallel time $O(T_\infty)$ while performing $O(T_1)$ operations. For instance, iterated product of n elements can be performed in parallel time $T_\infty = 2.\log n$ with $\frac{n}{\log n}$ processors by choosing a grain size of $\log n$. Even if this choice depends on the input size n, it can be computed only once at the beginning of the execution. The algorithm is a *simple hybrid* one.
Other examples of such parallel *simple hybrid* algorithms are: computation of the maximum of n elements in asymptotic optimal time $\Theta(\log\log n)$ on a CRCW PRAM with $\frac{n}{\log\log n}$ processors [17]; solving of a triangular linear system in parallel time $O(\sqrt{n}\log n)$ with $\Theta(n^2)$ operations [20]. In section 3 we detail an extended *baroque* hybridization for this problem, enabling a higher performance on a generic architecture.

Parallel *adaptive* algorithms by work-stealing - Kaapi. Cilk [18], Athapascan/Kaapi [16] and Satin [23] are parallel programming interfaces that support recursive parallelism and implement a work-stealing scheduling based on the *work first principle*. A program explicits parallelism and synchronization. While Cilk and Satin are restricted to serie-parallel tasks DAGs, Kaapi accepts any kind of dataflow dependencies. However, all are based on a sequential semantics: both depth first sequential search (DFS) and width (or breadth) first parallel search (BFS) are correct executions of the program. Then the program implements a parallel algorithm (BFS) that can also be considered as a sequential one (DFS). The (recursive) choices between both are performed by the scheduler. To save memory, depth-first execution (DFS) is always locally preferred. When a processor becomes idle, it steals the oldest ready task on a non-idle processor This stealing operation then corresponds to a breadth first execution (BFS). 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 a *hybrid baroque* algorithm. Because the choice does not depend on the input size but only on resource idleness, the algorithm is *adaptive*. In section 2.3 we detail a more general coupling for this problem.

2 Generic algorithmic schemes for *hybrid* algorithms

In this section we detail a generic scheme to control the time overhead due to choices in a *hybryd* algorithm, providing a proven upperbound for sequential and parallel *baroque* hybridization.

2.1 Basic representation

Let f be a problem with input set I and output set O. For the computation of f, a *hybrid* algorithm is based on the composition of distinct algorithms $(f_i)_{i=1,\dots,k}$, each solving the problem f. 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).

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. Hybridization then consists in choosing for each of those subcomputations the suited algorithm f_j to be used (fig. 2). This choice can be implemented in various ways. For instance, f may

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

Figure 2: Recursive description of a *hybrid* algorithm f_i.

be implemented as a pure virtual function, each of the f_i being an inherited specialization.

Scheme for decreasing overhead due to choices. 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. In the next section, we describe an original alternative scheme to decrease the overhead induced by the choices for each call to f in the previous algorithm. Generalization to various computations [5] (namely Branch&X computations and linear algebra) is based on the use of an exception mechanism. For a given subcomputation, a default given computation f_j is favored. However, 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 become negligible. We detail such a scheme in next section.

2.2 *Baroque* coupling of sequential and parallel algorithms

We presented the coupling of a sequential algorithm f_{seq} and a parallel one f_{par} that solve the same problem f. For the sake of simplicity, 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 [3], 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 [4] 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 can 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 [18] also valid for *Kaapi* [16]. It relies on Graham's bound (see Equation 2 in [18]). 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 2.1. *When the* hybrid *program is executed on a machine with m identical processors, the number of choices that result in a choice f_{par} for f instead of f_{seq} 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 *hybrid* algorithm is then $T_\infty^{(\text{par})}$. Then the 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 *hybrid 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 *hybrid* algorithm is *baroque* (non constant number of choices), the overhead in time due to choices in the *hybrid* 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}.

2.3 Application to the coupling of DFS/BFS for combinatorial optimization

The performance and overhead of the previous scheme were experimentally determined for the *Quadratic Assignment Problem* (for instance NUGENT 22[1]). This application implements a Branch&Bound algorithm: it recursively generates nodes in the search tree, which has 221938 nodes and a maximal depth of 22.
Locally, each processor implements by default a sequential algorithm f_{seq} that implements a depth first search (DFS) in the tree. It enables to save memory and also to optimize branching in the tree without copy (sons of a node n are sequentially created from the value of n with backtracking). To minimize critical time, the alternative f_{par} parallel algorithm implements a breadth first search (BFS) algorithm. When a processor becomes idle, it picks the oldest node of a randomly chosen non-idle processor (*ExtractPar*). This parallel algorithm introduces an overhead due to node copy.
The experiments were conducted on the iCluster2[2], a cluster of 104 nodes interconnected by a 100Mbps Ethernet network. Each node features two Itanium-2 processors (900 MHz) and 3 GB of local memory. The algorithm was parallelized using Kaapi. The degree of parallelism (threshold) can be adjusted: after a given depth, the subtree of a node is computed locally by f_{seq}. This thresold defined the minimum granularity and should be chosen such that the time of the local computation by f_{seq} is comparable to the time overhead of parallelism.

Figure 3: Impact of granularity

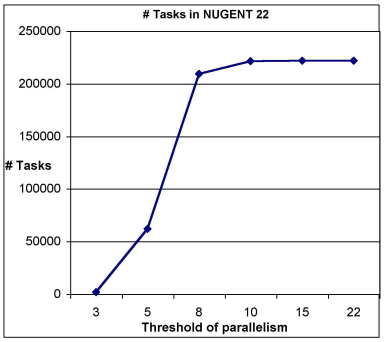
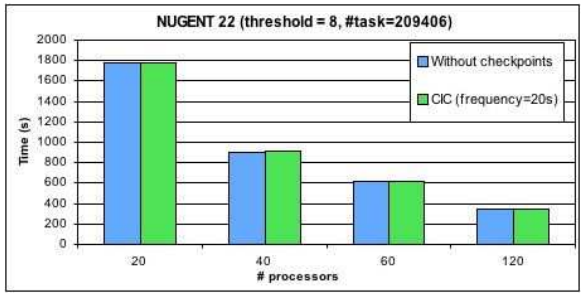


Figure 4: Execution time (sequential time: 34,695s)



The sequential execution time (C++ code without Kaapi) was 34,695 seconds. With Kaapi, at fine grain (threshold ≥ 10), the execution on a single processor generated 225,195 tasks and ran in 34,845 seconds. The impact of the degree of parallelism can be seen in Figure 3 that gives the number of parallel tasks generated for different thresholds. The degree of parallelism increases drastically for threshold 5 and approaches its maximum at threshold 10. Figure 4 shows that the application is scalable with a fine threshold (8, i.e. 209406 nodes). Since the critical time T_∞ is small, there are few successful steals and the overhead of hybridation between f_{seq} and f_{par} has small impact on efficiency.

[1] http://www.opt.math.tu-graz.ac.at/qaplib
[2] http://www.inrialpes.fr/sed/i-cluster2

Notice that Kaapi also includes a (*hybrid*) checkpoint/restart mechanism [16] to support the resilience and the addition of processors. This features makes the application itself *oblivious* to dynamic platforms. The overhead of this checkpoint mechanism appears negligible for this application (Figure 4).
In the next section, we detail various forms of hybridation on a single example, the solving of a triangular system.

3 Hybridization for triangular system solving

3.1 Triangular system solving with matrix right-hand side

Exact matrix multiplication, together with matrix factorizations, over finite fields can now be performed at the speed of the highly optimized numerical BLAS routines. This has been established by the FFLAS and FFPACK libraries [8, 9]. In this section we 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. Without loss of generality for the triangularization, we here consider only the case where the row dimension, m, of the the triangular system is less than or equal to the column dimension, n. 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. Therefore, 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: $R(m,k,n) \leq C_\omega min(m,k,n)^{\omega-2} max(mk,mn,kn)$ [15]. In the following subsections, we present the block recursive algorithm and two optimized implementation variants.

3.2 Scheme of the block recursive algorithm

The classical idea is to use the divide and conquer approach. Here, we consider the upper left triangular case without loss of generality, since any combination of upper/lower and left/right triangular cases are similar: if U is upper triangular, L is lower triangular and B is rectangular, we call `ULeft-Trsm` the resolution of $UX = B$. Suppose that we split the matrices into blocks and use the divide and conquer approach as follows:

$$\overbrace{\begin{bmatrix} A_1 & A_2 \\ & A_3 \end{bmatrix}}^{A} \overbrace{\begin{bmatrix} X_1 \\ X_2 \end{bmatrix}}^{X} = \overbrace{\begin{bmatrix} B_1 \\ B_2 \end{bmatrix}}^{B}$$

1. X_2 :=`ULeft-Trsm`(A_3, B_2);
2. $B_1 := B_1 - A_2 X_2$;
3. X_1 :=`ULeft-Trsm`(A_1, B_1);

With $m = n$ and classical matrix multiplication, the arithmetic cost of this algorithm is $TRSM(m) = m^3$ as shown e.g. in [9, Lemma 3.1].
We also now 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 via a block decomposition, one requires four recursive calls and two dense matrix-matrix multiplications. The cost is thus $TRMM(m) = 4TRMM(m/2) + 2MM(m/2)$. The latter is $TRMM(m) = m^3$ with classical matrix multiplication.
Now the inverse of a triangular matrix requires two recursive calls to invert A_1 and A_3. Then, the square block of the inverse is $-A_1^{-1}A_2A_3^{-1}$. The cost is thus $INVT(m) = 2INVT(m/2) + 2TRMM(m/2)$. The latter is $INVT(m) = \frac{1}{3}m^3$ with classical matrix multiplication.

3.3 Two distinct *hybrid* degenerations

3.3.1 Degenerating to the BLAS "dtrsm"

Matrix multiplication speed over finite fields was improved in [8, 21] by the use of the numerical BLAS[3] library: matrices were converted to floating point representations (where the linear algebra routines are fast) and converted back to a finite field representation afterwards. The computations remained exact as long as no overflow occurred. An implementation of `ULeft-Trsm` can use the same techniques. Indeed, as soon as no overflow occurs one can replace the recursive call to `ULeft-Trsm` by the numerical BLAS *dtrsm* routine. But one can remark that approximate divisions can occur. So we need to ensure both that only exact divisions are performed and that no overflow appears. However when the system is unitary (only 1's on the main diagonal) the division are of course exact and will even never be performed. Our idea is then to transform the initial system so that all the recursive calls to `ULeft-Trsm` are unitary. For a triangular system $AX = B$, it suffices to factor first the matrix A into $A = UD$, where U, D are respectively an upper unit triangular matrix and a diagonal matrix. Next the unitary system $UY = B$ is solved by any `ULeft-Trsm` (even a numerical one), without any division. The initial solution is then recovered over the finite field via $X = D^{-1}Y$. This normalization leads to an additional cost of $O(mn)$ arithmetic operations (see [9] for more details).
We now care for the coefficient growth. The use of the BLAS routine trsm is the resolution of the triangular system over the integers (stored as `double` for `dtrsm`). The restriction is the coefficient growth in the solution. Indeed, the k^{th} value in the solution vector is a linear combination of the $(n - k)$ already computed next values. This implies a linear growth in the coefficient size of the solution, with respect to the system dimension: for a given p, the dimension n of the system must satisfy $\frac{p-1}{2}\left[p^{n-1} + (p-2)^{n-1}\right] < 2^{m_a}$ where m_a is the size of the mantissa [9]. Then 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.

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

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. Also, `BLASTrsm` is the recursive block algorithm, switching to the BLAS resolution as soon as the splitting gives a block size lower than $S_{BLAS}(p)$.

3.3.2 Degenerating to delayed modulus

In the previous section we noticed that BLAS routines within `Trsm` are used only for small systems. An alternative is to change the cascade: instead of calling the BLAS, one could switch to the classical iterative algorithm: Let $A \in {\mathbb{Z}_{/p\mathbb{Z}}}^{m\times m}$ and $B, X \in {\mathbb{Z}_{/p\mathbb{Z}}}^{m\times n}$ such that $AX = B$, then $\forall i, X_{i,*} = \frac{1}{A_{i,i}}(B_{i,*} - A_{i,[i+1..m]}X_{[i+1..m],*})$ The idea is that the iterative algorithm computes only one row of the whole solution at a time. Therefore its threshold t is greater than the one of the BLAS routine, namely it requires only $t(p-1)^2 < 2^{m_a}$ for a $0..p-1$ unsigned representation, or $t(p-1)^2 < 2^{m_a+1}$ for a $\frac{1-p}{2}..\frac{p-1}{2}$ signed one. Now we focus on the dot product operation, base for matrix-vector product. According to [7], where different implementations of a dot product are proposed and compared on different architecture (Zech log, Montgomery, float, ...), the best implementation is a combination of a conversion to floating point representation with delayed modulus (for big prime and vector size) and an overflow detection trick (for smaller prime and vector size).
$\texttt{DelayTrsm}_t$ is the recursive block algorithm, switching to the delayed iterative resolution as soon as the splitting gives a block size lower than t (of course, t must satisfy $t \leq S_{BLAS}(p)$).

3.4 Tuning the "Trsm" implementation

3.4.1 Experimental tuning

As shown in section 3.2 the block recursive algorithm `Trsm` is based on matrix multiplications. This allows us to use the fast matrix multiplication routine of the FFLAS package [8]. This is an exact wrapping of the ATLAS library[4] used as a kernel to implement the `Trsm` variants. The following table results from experimental results of [9] and expresses which of the two preceding variants is better. `Mod<double>` is a field representation from [7] where the elements are stored as floating points to avoid one of the conversions. `G-Zpz` is a field representation from [13] where the elements are stored as small integers.

n	*400*	*700*	*1000*	*2000*	*5000*
`Mod<double>(5)`	`BLASTrsm`	`BLASTrsm`	`BLASTrsm`	`BLASTrsm`	`BLASTrsm`
`Mod<double>(32749)`	$\texttt{DelayTrsm}_{50}$	$\texttt{DelayTrsm}_{50}$	$\texttt{DelayTrsm}_{50}$	`BLASTrsm`	`BLASTrsm`
`G-Zpz(5)`	$\texttt{DelayTrsm}_{100}$	$\texttt{DelayTrsm}_{150}$	$\texttt{DelayTrsm}_{100}$	`BLASTrsm`	`BLASTrsm`
`G-Zpz(32749)`	$\texttt{DelayTrsm}_{50}$	$\texttt{DelayTrsm}_{50}$	$\texttt{DelayTrsm}_{50}$	$\texttt{DelayTrsm}_{50}$	$\texttt{DelayTrsm}_{50}$

Table 1: Best variant for `Trsm` on a P4, 2.4GHz

In the following, we will denote by $S_{Del}(n,p)$ the threshold t for which $\texttt{DelayTrsm}_t$ is the most efficient routine for matrices of size n. $S_{Del}(n,p)$ is set to 0 if e.g. the `BLASTrsm` routine

[4] `http://math-atlas.sourceforge.net`[24]

is better. The experiment shows that $S_{Del}(n,p)$ can be bigger or smaller than $S_{BLAS}(p)$ depending on the matrix size, the prime and the underlying arithmetic implementation.

3.4.2 *Hybrid tuned* algorithm

The experimental results of previous section, thus provide us with an *hybrid* algorithm where we can tune some static threshold in order to benefit from all the variants. Moreover, some choices have to be made for the splitting size k in order to reach the optimal complexity T_{opt}:

$$Topt(m) = Min_k\{Topt(k) + Topt(m-k) + R(m-k,k,n)\}.$$

Algorithm `ULeft-Trsm`(A,B)
Input: $A \in {\mathbb{Z}_{/p\mathbb{Z}}}^{m\times m}$, $B \in {\mathbb{Z}_{/p\mathbb{Z}}}^{m\times n}$.
Output: $X \in {\mathbb{Z}_{/p\mathbb{Z}}}^{m\times n}$ such that $AX = B$.

if $m \leq S_{Del}(m,p)$ **then** *// Hybrid modulus degeneration 3.3.2*
  $X :=$ `DelayTrsm`(A,B);
else if $m \leq S_{BLAS}(p)$ **then** *// Hybrid BLAS degeneration 3.3.1*
  $X :=$ `BLASTrsm`(A,B);
else *// Hybrid block recursive 3.2*
  $k := Choice(1..\lfloor \frac{m}{2} \rfloor)$;
  Split matrices into k and m−k blocks $\begin{bmatrix} A_1 & A_2 \\ & A_3 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$
  $X_2 :=$`ULeft-Trsm`(A_3, B_2);
  $B_1 := B_1 - A_2X_2$;
  $X_1 :=$`ULeft-Trsm`(A_1, B_1);
return X;

3.5 *Baroque hybrid* parallel Trsm

The previous algorithm takes benefit of parallelism at the level of Blas matrix product operations. However, using the scheme proposed in §2.2, 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 arithmetic 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 2.2. to estimate this value at runtime using the *hybrid* coupling of a "sequential" algorithm f_s with a parallel one f_p.

3.5.1 Parallel *adaptive* TRSM

We assume that the parallel *hybrid* TRSM is spawned by a high priority process. Then the parallel *hybrid* TRSM consists in computing concurrently in parallel (Figure 5):

- "sequential" computation (f_s) at high priority: bottom-up computation of $X = TRSM(A, B)$ till reaching k, implemented by BUT algorithm (Bottom-Up TRSM - §A.1); 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 - §A.2); all processes that participates in parallel TDTI are executed at low priority.

Algorithm `HybridParallelTrsm`$(A; B)$
Input: $A \in {\mathbb{Z}_{/p\mathbb{Z}}}^{m \times m}$, $B \in {\mathbb{Z}_{/p\mathbb{Z}}}^{m \times n}$.
Output: $X \in {\mathbb{Z}_{/p\mathbb{Z}}}^{m \times n}$ such that $AX = B$.

$k_{TDTI} := 0$; $k_{BUT} := m$;
Parallel {
  At high priority: $(X_2, B_1') := BUT(A, B)$;
  At low priority: $M := TDTI(\emptyset, A)$;
}
Here, BUT has stopped TDTI and $k_{BUT} \leq k_{TDTI}$.
Now, let $A_1^{'-1} = M_{1..k_{BUT},1..k_{BUT}}$;
$X_1 := A_1^{'-1}.B_1'$;

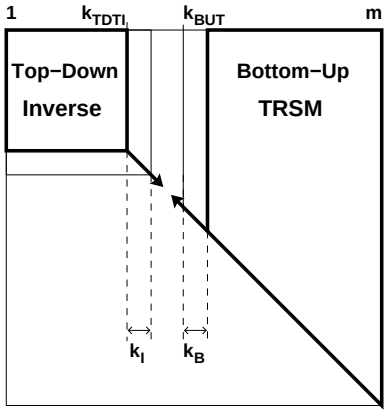


Figure 5: Parallel *adaptive* TRSM

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 5 and detailed subroutines BUT and TDTI in appendices). Note that the values of k_B and k_I may vary during the execution depending on the current state.

3.5.2 Definiton of parameters k_I and k_B

Parameters k_B (resp. k_I) corresponds to the *ExtractSeq* (resp. *ExtractPar*) operations presented in §2.2. 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 *hybrid* 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 *hybrid tuned* TRSM algorithm. Also, the algorithm is *oblivious* to the number of resources and their relative performance.

4 Conclusion

Designing efficient *hybrid* algorithms is the key to get most of the available resources and most of the structure of the inputs of numerous applications as we have shown e.g. for linear algebra or for combinatorial optimization Branch&X. In this paper, we have proposed a classification of the distinct forms of *hybrid* algorithms and a generic framework to express this adaptivity. On a single simple example, namely solving linear systems, we show that several of these "hybridities" can appear. This enables an effective hybridization 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 automatically generated in our simple framework. The requirements are just to provide recursive versions of the different methods. In the AHA group[5], such coupling are studied in the context of many examples: vision and adaptive 3D-reconstruction, linear algebra in general, and combinatorial optimization.

Acknowledgments. The authors gratefully acknowledge David B. Saunders for useful discussions and suggestions for the classification of *hybrid* algorithms.

[5] aha.imag.fr

References

[1] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. *SIAM J. Comput.*, 35(2):341–358, 2005.

[2] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. Concurrent cache-oblivious b-trees. In *SPAA'05: Proceedings of the 17th annual ACM symposium on Parallelism in algorithms and architectures*, pages 228–237, New York, NY, USA, 2005. ACM Press.

[3] R.S. Bird. *Logic of Programming and Calculi of Discrete Design*, chapter Introduction to the Theory of Lists. Springer-Verlag, 1987.

[4] M. Cole. Parallel Programming with List Homomorphisms. *Parallel Processing Letters*, 5(2):191–204, 1995.

[5] El-Mostafa Daoudi, Thierry Gautier, Aicha Kerfali, Rémi Revire, and Jean-Louis Roch. Algorithmes parallèles à grain adaptatif et applications. *Technique et Science Informatiques*, 24:1—20, 2005.

[6] F. D'Azevedo and J. Dongarra. The design and implementation of the parallel out-of-core scalapack lu, qr and cholesky factorization routines. Technical Report CS-97-347, University of Tenessee, january 1997. http://www.netlib.org.

[7] Jean-Guillaume Dumas. Efficient dot product over finite fields. In Victor G. Ganzha, Ernst W. Mayr, and Evgenii V. Vorozhtsov, editors, *Proceedings of the seventh International Workshop on Computer Algebra in Scientific Computing, Yalta, Ukraine*, pages 139–154. Technische Universität München, Germany, July 2004.

[8] Jean-Guillaume Dumas, Thierry Gautier, and Clément Pernet. Finite Field Linear Algebra Subroutines. In Teo Mora, editor, *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation, Lille, France*, pages 63–74. ACM Press, New York, July 2002.

[9] Jean-Guillaume Dumas, Pascal Giorgi, and Clément Pernet. FFPACK: Finite Field Linear Algebra Package. In Jaime Gutierrez, editor, *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation, Santander, Spain*, pages 119–126. ACM Press, New York, July 2004.

[10] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2), 2005. Special issue on "Program Generation, Optimization, and Adaptation".

[11] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285, Washington, DC, USA, 1999. IEEE Computer Society.

[12] Alan G. Ganek and Thomas A. Corbi. The Dawning of the Autonomic Computing Era. *IBM Systems Journal*, 42(1):5–18, 2003.

[13] Thierry Gautier, Gilles Villard, Jean-Louis Roch, Jean-Guillaume Dumas, and Pascal Giorgi. Givaro, a C++ library for computer algebra: exact arithmetic and data structures. Software, ciel-00000022, October 2005. `www-lmc.imag.fr/Logiciels/givaro`.

[14] Kazushige Goto and Robert A. van de Geijn. Anatomy of High-Performance Matrix Multiplication. *ACM Transactions on Mathematical Software*. Submitted.

[15] Xiaohan Huang and Victor Y. Pan. Fast rectangular matrix multiplications and improving parallel matrix computations. In ACM, editor, *PASCO '97. Proceedings of the second international symposium on parallel symbolic computation, July 20–22, 1997, Maui, HI*, pages 11–23, New York, NY 10036, USA, 1997. ACM Press.

[16] Samir Jafar, Thierry Gautier, Axel W. Krings, and Jean-Louis Roch. A checkpoint/recovery model for heterogeneous dataflow computations using work-stealing. In LNCS Springer-Verlag, editor, *EUROPAR'2005*, Lisboa, Portugal, August 2005.

[17] J. Jájá. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, Massachussets, 1992.

[18] K. H. Randall M. Frigo, C. E. Leiserson. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223. ACM Press, 1998.

[19] Frederic Ogel, Bertil Folliot, and Ian Piumarta. On Reflexive and Dynamically Adaptable Environments for Distributed Computing. In *ICDCS Workshops*, pages 112–117. IEEE Computer Society, 2003.

[20] Victor Y. Pan and Franco P. Preparata. Work-preserving speed-up of parallel matrix computations. *SIAM Journal on Computing*, 24(4), 1995.

[21] Clément Pernet. Implementation of Winograd's matrix multiplication over finite fields using ATLAS level 3 BLAS. Technical report, Laboratoire Informatique et Distribution, July 2001. `www-id.imag.fr/Apache/RR/RR011122FFLAS.ps.gz`.

[22] B. Richard, P. Augerat, N. Maillard, S. Derr, S. Martin, and C. Robert. I-cluster: Reaching top500 performance using mainstream hardware. Technical Report HPL-2001-206 20010831, HP Laboratories Grenoble, August 2001.

[23] Rob V. van Nieuwpoort, Jason Maassen, Thilo Kielmann, and Henri E. Bal. Satin: Simple and efficient java-based grid programming. *Scalable Computing: Practice and Experience*, 6(3):19–32, September 2005.

[24] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–35, January 2001. `www.elsevier.nl/gej-ng/10/35/21/47/25/23/article.pdf`.

Van Dat Cung and Christophe Rapine
GILCO Laboratory, ENSGI-INPG, H building, Office H123.
46, avenue Félix-Viallet, 38031 Grenoble, FRANCE.
{Van-Dat.Cung,Christophe.Rapine}@gilco.inpg.fr,
gilco.inpg.fr/~{cung,rapine}.

Vincent Danjean, Thierry Gautier, Guillaume Huard, Bruno Raffin, Jean-Louis Roch and Denis Trystram
Laboratoire Informatique et Distribution, ENSIMAG - antenne de Montbonnot
ZIRST 51, avenue Jean Kuntzmann, 38330 Montbonnot Saint Martin, FRANCE.
FirstName.LastName@imag.fr, www-id.imag.fr/Membres.

Jean-Guillaume Dumas
Laboratoire de Modélisation et Calcul, Université Joseph Fourier, Grenoble I
51, av. des Mathématiques, BP 53X, 38041 Grenoble, FRANCE.
Jean-Guillaume.Dumas@imag.fr,
www-lmc.imag.fr/lmc-mosaic/Jean-Guillaume.Dumas

A Appendix

A.1 Bottom-up TRSM

We need to group the last recursive `ULeft-Trsm` call and the update of B_1. The following algorithm thus just computes these last two steps ; the first step being performed by the work stealing as shown afterwards.

Algorithm `BUT`
Input: $(A_2; A_3; B)$.
Output: X_2, k_{BUT}.

Mutual Exclusion section {
  if ($k_{TDTI} \geq k_{BUT}$) Return;
  $k_B := Choice(1..(k_{BUT} - k_{TDTI}))$.
  Split remaining columns into $k_{TDTI}..(k_{BUT}-k_B)$ *and* $(k_{BUT}-k_B)..k_{BUT}$

$$\begin{bmatrix} A_{2,1} & A_{2,2} \\ A_{3,1} & A_{3,2} \\ & A_{3,3} \end{bmatrix} \begin{bmatrix} X_{2,1} \\ X_{2,2} \end{bmatrix} = \begin{bmatrix} B_1 \\ B_{2,1} \\ B_{2,2} \end{bmatrix}$$

  $k_{BUT} := k_{BUT} - k_B$;
}
$X_{2,2}$:=`ULeft-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`$\left(A_{2,1}; A_{3,1}; \begin{bmatrix} B_1 \\ B_{2,1} \end{bmatrix}\right)$

A.2 Top down triangular inversion of A_1

Algorithm `TDTI`
Input: $\left(A_1^{-1}; A_2; A_3\right)$.
Output: A^{-1}, k_{TDTI}.

Mutual Exclusion section {
if ($k_{TDTI} \geq k_{BUT}$) 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* $(k_{TDTI} + k_I)..k_{BUT}$

$$\begin{bmatrix} A_{2,1} & A_{2,2} \\ A_{3,1} & A_{3,2} \\ & A_{3,3} \end{bmatrix}$$

}
Parallel {
$A_{3,1}^{-1}$:=`Inverse(`$A_{3,1}$`)`;
$T := A_1^{-1}.A_{2,1}$
}
$A'_{2,1} = -T.A_{3,1}^{-1}$
Now, 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 section {
$k_{TDTI} := k_{TDTI} + k_I$;
}
$A_{3,3}^{-1}$:=`TDTI(`$A_1^{'-1}$`;`A'_2`;`$A_{3,3}$`)`;