Solving Quadratic Equations with XL on Parallel Architectures

extended version

Chen-Mou Cheng[1], Tung Chou[2], Ruben Niederhagen[2], and Bo-Yin Yang[3]

[1] Department of Electrical Engineering
National Taiwan University, Taipei, Taiwan
doug@crypto.tw
[2] Department of Mathematics and Computer Science
Eindhoven University of Technology, The Netherlands
blueprint@crypto.tw, ruben@polycephaly.org
[3] Institute of Information Science
Academia Sinica, Taipei, Taiwan
by@crypto.tw

Abstract. Solving a system of multivariate quadratic equations (MQ) is an NP-complete problem whose complexity estimates are relevant to many cryptographic scenarios. In some cases it is required in the best known attack; sometimes it is a generic attack (such as for the multivariate PKCs), and sometimes it determines a provable level of security (such as for the QUAD stream ciphers).
Under reasonable assumptions, the best way to solve generic MQ systems is the XL algorithm implemented with a sparse matrix solver such as Wiedemann's algorithm. Knowing how much time an implementation of this attack requires gives us a good idea of how future cryptosystems related to MQ can be broken, similar to how implementations of the General Number Field Sieve that factors smaller RSA numbers give us more insight into the security of actual RSA-based cryptosystems.
This paper describes such an implementation of XL using the block Wiedemann algorithm. In 5 days we are able to solve a system with 32 variables and 64 equations over $\mathbb{F}_{16}$ (a computation of about $2^{60.3}$ bit operations) on a small cluster of 8 nodes, with 8 CPU cores and 36 GB of RAM in each node. We do not expect system solvers of the F_4/F_5 family to accomplish this due to their much higher memory demand. Our software also offers implementations for $\mathbb{F}_2$ and $\mathbb{F}_{31}$ and can be easily adapted to other small fields. More importantly, it scales nicely for small clusters, NUMA machines, and a combination of both.

Keywords: XL, Gröbner basis, block Wiedemann, sparse solver, multivariate quadratic systems

1 Introduction

Some cryptographic systems can be attacked by solving a system of multivariate quadratic equations. For example the symmetric block cipher AES can be attacked by solving a system of 8000 quadratic equations with 1600 variables over $\mathbb{F}_2$ as shown by Courtois and Pieprzyk in [5] or by solving a system of 840 sparse quadratic equations and 1408 linear equations over 3968 variables

This work was in part supported by National Science Council (NSC), National Taiwan University, and Intel Corporation under Grants NSC 100-2911-I-002-001 and 101R7501. The authors would also like to thank partial sponsorship from NSC under Grants 100-2218-E-001-002 and 100-2628-E-001-004-MY3, as well as from Academia Sinica including a Career Award to the fourth author. Furthermore, this work was supported by the European Commission under Contract ICT-2007-216676 ECRYPT II. Part of this work was done when the second and third authors were employed at Academia Sinica. This paper is an extended version of a paper from CHES 2012. The Springer version is available at 10.1007/978-3-642-33027-8_21. Permanent ID of this document: 0d518cdf5f001b64aaec7a0bf93bf9b2. Date: 2016.04.26.

of $\mathbb{F}_{256}$ as shown by Murphy and Robshaw in [17]. Multivariate cryptographic systems can be attacked naturally by solving their multivariate quadratic system; see for example the analysis of the QUAD stream cipher by Yang, Chen, Bernstein, and Chen in [22].

We describe a parallel implementation of an algorithm for solving quadratic systems that was first suggested by Lazard in [11]. Later it was reinvented by Courtois, Klimov, Patarin, and Shamir and published in [4]; they call the algorithm *XL* as an acronym for *extended linearization*: XL *extends* a quadratic system by multiplying all equations with appropriate monomials and *linearizes* it by treating each monomial as an independent variable. Due to this extended linearization, the problem of solving a quadratic system turns into a problem of linear algebra.

XL is a special case of Gröbner basis algorithms (shown by Ars, Faugère, Imai, Kawazoe, and Sugita in [1]) and can be used as an alternative to other Gröbner basis solvers like Faugère's F_4 and F_5 algorithms (introduced in [7] and [8]). An enhanced version of F_4 is implemented for example in the computer algebra system Magma, and is often used as standard benchmark by cryptographers.

There is an ongoing discussion on whether XL-based algorithms or algorithms of the $\mathrm{F}_4/\mathrm{F}_5$-family are more efficient in terms of runtime complexity and memory complexity. To achieve a better understanding of the practical behaviour of XL for generic systems, we describe a parallel implementation of the XL algorithm for shared-memory systems, for small computer clusters, and for a combination of both. Measurements of the efficiency of the parallelization have been taken at small clusters of up to 8 nodes and shared-memory systems of up to 64 cores. A previous implementation of XL is PWXL, a parallel implementation of XL with block Wiedemann described in [15]. PWXL supports onl $\mathbb{F}_2$, while our implementation supports $\mathbb{F}_2$, $\mathbb{F}_{16}$, and $\mathbb{F}_{31}$. Furthermore, our implementation is modular and can be extended to other fields. Comparisons on performance of PWXL and our work will be shown in Section 6.3. We are planning to make our implementation available to the public.

This paper is structured as follows: The XL algorithm is introduced in Section 2. The parallel implementation of XL using the block Wiedemann algorithm is described in Section 5. Section 6 gives runtime measurements and performance values that are achieved by our implementation for a set of parameters on several parallel systems as well as comparisons to PWXL and to the implementation of F_4 in Magma.

2 The XL algorithm

The original description of XL for multivariate quadratic systems can be found in the paper [4]; a more general definition of XL for systems of higher degree is given in [3]. The following gives an introduction of the XL algorithm for quadratic systems; the notation is adapted from [24]:

Consider a finite field $K = \mathbb{F}_q$ and a system $\mathcal{A}$ of m multivariate quadratic equations $\ell_1 = \ell_2 = \cdots = \ell_m = 0$ for $\ell_i \in K[x_1, x_2, \ldots, x_n]$. For $b \in \mathbb{N}^n$ denote by x^b the monomial $x_1^{b_1} x_2^{b_2} \ldots x_n^{b_n}$ and by $|b| = b_1 + b_2 + \cdots + b_n$ the total degree of x^b.

XL first chooses a $D \in \mathbb{N}$ as $D := \min\{d : ((1-\lambda)^{m-n-1}(1+\lambda)^m)[d] \leq 0\}$ (see [23, Eq. (7)], [13,6]), where $f[i]$ denotes the coefficient of the degree-i term in the expansion of a polynomial $f(\lambda)$ e.g., $(\lambda+2)^3[2] = (\lambda^3 + 6\lambda^2 + 12\lambda + 8)[2] = 6$. XL extends the quadratic system $\mathcal{A}$ to the system $\mathcal{R}^{(D)} = \{x^b \ell_i = 0 : |b| \leq D-2, \ell_i \in \mathcal{A}\}$ of maximum degree D by multiplying each equation of $\mathcal{A}$ by all monomials of degree less than or equal to $D-2$. Now, each monomial $x^d, |d| \leq D$ is considered a new variable to obtain a linear system $\mathcal{M}$. Note that the system matrix of $\mathcal{M}$ is sparse since each equation has the same number of non-zero coefficients as the corresponding equation of the quadratic system $\mathcal{A}$. Finally the linear system $\mathcal{M}$ is solved, giving solutions for all monomials and particularly for $x_1, x_2, \ldots, x_n$. Note that the matrix corresponding to the linear system $\mathcal{M}$ is the Macaulay matrix of degree D for the polynomial system $\mathcal{A}$ (see [12], e.g., defined in [9]).

2.1 The Block Wiedemann algorithm

The computationally most expensive task in XL is to find a solution for the sparse linear system $\mathcal{M}$ of equations over a finite field. There are two popular algorithms for that task, the block Lanczos

algorithm [16] and the block Wiedemann algorithm [2]. The block Wiedemann algorithm was proposed by Coppersmith in 1994 and is a generalization of the original Wiedemann algorithm [21]. It has several features that make it powerful for computation in XL: From the original Wiedemann algorithm it inherits the property that the runtime is directly proportional to the weight of the input matrix. Therefore, this algorithm is suitable for solving sparse matrices, which is exactly the case for XL. Furthermore, big parts of the block Wiedemann algorithm can be parallelized on several types of parallel architectures. The following paragraphs give a brief introduction to the block Wiedemann algorithm. For more details please refer to [18, Section 4.2] and [2].

The basic idea of Coppersmith's block Wiedemann algorithm for finding a solution $\bar{x} \neq 0$ of $B\bar{x} = 0$ for $B \in K^{N \times N}$, $\bar{x} \in K^N$ (where B corresponds to the system matrix of $\mathcal{M}$ when computing XL) is the same as in the original Wiedemann algorithm: Assume that the characteristic polynomial $f(\lambda) = \sum_{0 \leq i} f[i]\lambda^i$ of B is known. Since B is singular, it has an eigenvalue 0, thus $f(B) = 0$ and $f[0] = 0$. We have:

$$f(B)\bar{z} = \sum_{i>0} f[i]B^i\bar{z} = B\sum_{i>0} f[i]B^{i-1}\bar{z} = 0,$$

for any vector $\bar{z} \in K^N$. Therefore, $\bar{x} = \sum_{i>0} f[i]B^{i-1}\bar{z}$, $\bar{z} \neq 0$ is a (hopefully non-zero) kernel vector and thus a solution of the linear equation system. In fact it is possible to use any *annihilating* polynomial $f(\lambda)$ of B, i.e., a polynomial $f(\lambda) \neq 0$ such that $f(B) = 0$.

Wiedemann suggests to use the Berlekamp–Massey algorithm for the computation of $f(\lambda)$. Given a linearly recurrent sequence $\{a^{(i)}\}_{i=0}^{\infty}$, the algorithm computes $c_1, \ldots, c_d$ for some d such that $c_1a^{(d-1)} + c_2a^{(d-2)} + \cdots + c_da^{(0)} = 0$. Choosing $a^{(i)} = \bar{x}^T BB^i\bar{z}$ with random vectors $\bar{x}$ and $\bar{z}$ (as delegates for BB^i) as input and $f[i] = c_{d-i}$, $0 \leq i < d$ as output returns $f(\lambda)$ as an annihilating polynomial of B with high probability.

Coppersmith [2] proposed a modification of the Wiedemann algorithm that makes it more suitable for modern computer architectures by operating in parallel on a *block* of $\tilde{n}$ column vectors $\bar{z}_i, 0 \leq i < \tilde{n}$, of a matrix $z \in K^{N \times \tilde{n}}$. His block Wiedemann algorithm computes kernel vectors in three steps which are called BW1, BW2, and BW3 for the remainder of this paper. The *block sizes* of the block Wiedemann algorithm are the integers $\tilde{m}$ and $\tilde{n}$. They can be chosen freely for the implementation such that they give the best performance on the target architecture for matrix and vector operations, e.g., depending on the size of cache lines or vector registers. Step BW1 computes the first $N/\tilde{m} + N/\tilde{n} + \mathrm{O}(1)$ elements of a sequence $\{a^{(i)}\}_{i=0}^{\infty}$, $a_i = \left(x \cdot (B \cdot B^i z)\right)^T \in K^{\tilde{n} \times \tilde{m}}$ using random matrices $x \in K^{\tilde{m} \times N}$ and $z \in K^{N \times \tilde{n}}$. This sequence is the input for the second step BW2, a block variant of the Berlekamp–Massey algorithm. It returns a matrix polynomial $f(\lambda)$ with coefficients $f[j] \in K^{\tilde{n} \times \tilde{n}}$, that is used by step BW3 to compute up to $\tilde{n}$ solution vectors in a blocked fashion similar as described above for the original Wiedemann algorithm.

3 The block Berlekamp–Massey algorithm

This section first introduces a tweak that makes it possible to speed up computations of Coppersmith's variant of the Berlekamp–Massey algorithm. Later the parallelization of the algorithm is described.

3.1 Reducing the cost of the block Berlekamp–Massey algorithm

The j-th iteration of Coppersmith's block Berlekamp–Massey algorithm requires a matrix $P^{(j)} \in K^{(m+n)\times(m+n)}$ such that the first n rows of $P^{(j)}H^{(j)}$ are all zeros. The main idea of this tweak is to make $P^{(j)}$ have the form

$$P^{(j)} = \begin{pmatrix} I_n & * \\ 0 & I_m \end{pmatrix} E^{(j)},$$

Algorithm 1 Gaussian elimination in Coppersmith's Berlekamp–Massey algorithm

function ELIMINATE($H^{(j)} \in K^{(m+n)\times m}$, a list of nominal degrees $d^{(j)}$)
  $M \leftarrow H^{(j)}$, $P \leftarrow I_{m+n}$, $E \leftarrow I_{m+n}$
  sort the rows of M by the nominal degrees in decreasing order
  apply the same permutation to $P^{(j)}$ and $E^{(j)}$
  for $k = 1 \to m$ **do**
    for $i = (m + n + 1 - k) \to 1$ **do**
      if $M_{i,k} \neq 0$ **then**
        $v_{(M)} \leftarrow M_i$,
        $v_{(P)} \leftarrow P_i$,
        $v_{(E)} \leftarrow E_i$
      end if
    end for
    for $l = i + 1 \to (m + n + 1 - k)$ **do**
      $M_{l-1} \leftarrow M_l$,
      $P_{l-1} \leftarrow P_l$,
      $E_{l-1} \leftarrow E_l$
    end for
    $M_{(m+n+1-k)} \leftarrow v_{(M)}$, $P_{(m+n+1-k)} \leftarrow v_{(P)}$, $E_{(m+n+1-k)} \leftarrow v_{(E)}$
    for $l = 1 \to (m + n - k)$ **do**
      if $M_{l,k} \neq 0$ **then**
        $M_l \leftarrow M_l - v_{(M)} \cdot (M_{l,k}/v_{(M)k})$
        $P_l \leftarrow P_l - v_{(P)} \cdot (M_{l,k}/v_{(M)k})$
      end if
    end for
  end for
  $P^{(j)} \leftarrow P$
  $E^{(j)} \leftarrow E$
  return ($P^{(j)} \in K^{(m+n)\times(m+n)}$, $E^{(j)} \in K^{(m+n)\times(m+n)}$)
end function

where $E^{(j)}$ is a permutation matrix corresponding to a permutation $\phi^{(j)}$ (the superscript of $\phi^{(j)}$ will be omitted in this section). Therefore, the multiplication $P^{(j)}f^{(j)}$ takes only $\deg(f^{(j)})\cdot\mathrm{Mul}(n, m, n)$ field operations (for the upper right submatrix in $P^{(j)}$).

The special form of $P^{(j)}$ also makes the computation of $H^{(j)}$ more efficient: The bottom m rows of each coefficient are simply permuted due to the multiplication by $P^{(j)}$, thus

$$(P^{(j)}f^{(j)}[k])_i = (f^{(j)}[k])_{\phi(i)},$$

for $n < i \leq m + n,\ 0 < k \leq \deg(f^{(j)})$. Since multiplication by Q corresponds to a multiplication of the bottom m rows by λ, it does not modify the upper n rows of the coefficients. Therefore, the bottom m rows of the coefficients of $f^{(j+1)}$ can be obtained from $f^{(j)}$ as

$$(f^{(j+1)}[k])_i = (QP^{(j)}f^{(j)}[k-1])_i = (f^{(j)}[k-1])_{\phi(i)},$$

for $n < i \leq m + n,\ 0 < k \leq deg(f^{(j)})$. Since the bottom right corner of $P^{(j)}$ is the identity matrix of size m, this also holds for

$$((f^{(j+1)}a)[j+1])_i = ((QP^{(j)}f^{(j)}a)[j+1])_i = ((f^{(j)}a)[j])_{\phi(i)}.$$

Thus, $H_i^{(j+1)}$ for $n < i \leq m + n$ can be computed as

$$H_i^{(j+1)} = ((f^{(j+1)}a)[j+1])_i = ((QP^{(j)}f^{(j)}a)[j+1])_i = ((f^{(j)}a)[j])_{\phi(i)} = H_{\phi(i)}^{(j)}.$$

This means the last m rows of $H^{(j+1)}$ can actually be copied from $H^{(j)}$; only the first n rows of $H^{(j+1)}$ need to be computed. Therefore the cost of computing any $H^{(j>j_0)}$ is reduced to $\deg(f^{(j)})\cdot\mathrm{Mul}(n, n, m)$.

The matrix $P^{(j)}$ can be assembled as follows: The matrix $P^{(j)}$ is computed using Algorithm 1. In this algorithm a sequence of row operations is applied to $M := H^{(j)}$. The matrix $H^{(j)}$ has rank m for all $j \geq j_0$. Therefore in the end the first n rows of M are all zeros. The composition of all the operations is $P^{(j)}$; some of these operations are permutations of rows. The composition of these permutations is $E^{(j)}$:

$$P^{(j)}(E^{(j)})^{-1} = \begin{pmatrix} I_n & * \\ 0 & F^{(j)} \end{pmatrix} \iff P^{(j)} = \begin{pmatrix} I_n & * \\ 0 & F^{(j)} \end{pmatrix} E^{(j)}.$$

The algorithm by Coppersmith requires that the first n rows of $P^{(j)}H^{(j)}$ are all zero (see [2, 7]); there is no condition for the bottom m rows. However, the first n rows of $P^{(j)}H^{(j)}$ are all zero independently of the value of $F^{(j)}$. Thus, $F^{(j)}$ can be replaced by I_m without harming this requirement.

3.2 Parallelization of the block Berlekamp–Massey algorithm

The parallel implementation of the block Berlekamp–Massey algorithm on c nodes works as follows: In each iteration step, the coefficients of $f^{(j)}(\lambda)$ are equally distributed over the computing nodes; for $0 \leq i < c$, let $S_i^{(j)}$ be the set containing all indices of coefficients stored by node i during the j-th iteration. Each node stores a copy of all coefficients of $a(\lambda)$.

Due to the distribution of the coefficients, the computation of

$$H^{(j)} = (f^{(j)}a)[j] = \sum_{l=0}^{j} f^{(j)}[l]a[j-l]$$

requires communication: Each node i first locally computes a part of the sum using only its own coefficients $S_i^{(j)}$ of $f^{(j)}$. The matrix $H^{(j)}$ is the sum of all these intermediate results. Therefore, all nodes broadcast their intermediate results to the other nodes. Each node computes $H^{(j)}$ locally; Gaussian elimination is performed on every node locally and is not parallelized over the nodes. Since only small matrices are handled, this sequential overhead is negligibly small.

Also the computation of $f^{(j+1)}$ requires communication. Recall that

$$f^{(j+1)} = QP^{(j)}f^{(j)}, \quad \text{for } Q = \begin{pmatrix} I_n & 0 \\ 0 & \lambda \cdot I_m \end{pmatrix}.$$

Each coefficient k is computed row-wise as

$$(f^{(j+1)}[k])_l = \begin{cases} ((P^{(j)}f^{(j)})[k])_l, & \text{for } 0 < l \leq n, \\ ((P^{(j)}f^{(j)})[k-1])_l, & \text{for } n < l \leq m+n. \end{cases}$$

Computation of $f^{(j+1)}[k]$ requires access to both coefficients k and $(k-1)$ of $f^{(j)}$. Therefore, communication cost is reduced by distributing the coefficients equally over the nodes such that each node stores a continuous range of coefficients of $f^{(j)}$ and such that the indices in $S_{i+1}^{(j)}$ always are larger than those in $S_i^{(j)}$.

Due to the multiplication by Q, the degree of $f^{(j)}$ is increased by at most one in each iteration. Therefore at most one more coefficient must be stored. The new coefficient obviously is the coefficient with highest degree and therefore must be stored on node $(c-1)$. To maintain load balancing, one node $i^{(j)}$ is chosen in a round-robin fashion to receive one additional coefficient; coefficients are exchanged between neighbouring nodes to maintain an ordered distribution of the coefficients.

Observe, that only node $(c-1)$ can check whether the degree has increased, i.e. whether $\deg(f^{(j+1)}) = \deg(f^{(j)}) + 1$, and whether coefficients need to be redistributed; this information needs to be communicated to the other nodes. To avoid this communication, the maximum nominal degree $\max(d^{(j)})$ is used to approximate $\deg(f^{(j)})$. Note that in each iteration all nodes can update a

iteration j	$S_3^{(j)}$	$S_2^{(j)}$	$S_1^{(j)}$	$S_0^{(j)}$	$\max(d^{(j)})$
0	$\emptyset$	$\emptyset$	{1}	{0}	1
1	$\emptyset$	{2}	{1}	{0}	2
2	{3}	{2}	{1}	{0}	3
3	{4}	{3}	{2}	{1,0}	4
4	{5}	{4}	{3,2}	{1,0}	5
5	{6}	{5,4}	{3,2}	{1,0}	6
6	{7,6}	{5,4}	{3,2}	{1,0}	7
...	...	...	...	...	...

Table 1. Example for the workload distribution over 4 nodes. Iteration 0 receives the distribution in the first line as input and computes the new distribution in line two as input for iteration 1.

local list of the nominal degrees. Therefore, all nodes decide locally without communication whether coefficients need to be reassigned: If $\max(d^{(j+1)}) = \max(d^{(j)}) + 1$, the number $i^{(j)}$ is computed as

$$i^{(j)} = \max(d^{(j+1)}) \bmod c.$$

Node $i^{(j)}$ is chosen to store one additional coefficient, the coefficients of nodes i, for $i \geq i^{(j)}$, are redistributed accordingly.

Table 1 illustrates the distribution strategy for 4 nodes. For example in iteration 3, node 1 has been chosen to store one more coefficient. Therefore it receives one coefficient from node 2. Another coefficient is moved from node 3 to node 2. The new coefficient is assigned to node 3.

This distribution scheme does not avoid all communication for the computation of $f^{(j+1)}$: First all nodes compute $P^{(j)}f^{(j)}$ locally. After that, the coefficients are multiplied by Q. For almost all coefficients of $f^{(j)}$, both coefficients k and $(k-1)$ of $P^{(j)}f^{(j)}$ are stored on the same node, i.e. $k \in S_{(i)}^{(j)}$ and $(k-1) \in S_{(i)}^{(j)}$. Thus, $f^{(j+1)}[k]$ can be computed locally without communication. In the example in Figure 1, this is the case for $k \in \{0, 1, 2, 4, 5, 7, 9, 10\}$. Note that the bottom m rows of $f^{(j+1)}[0]$ and the top n rows of $f^{(j+1)}[\max(d^{(j+1)})]$ are 0.

Communication is necessary if coefficients k and $(k-1)$ of $P^{(j)}f^{(j)}$ are not on the same node. There are two cases:

- In case $k - 1 = \max(S_{i-1}^{(j+1)}) = \max(S_{i-1}^{(j)})$, $i \neq 1$, the bottom m rows of $(P^{(j)}f^{(j)})[k-1]$ are sent from node $i-1$ to node i. This is the case for $k \in \{6, 3\}$ in Figure 1. This case occurs if in iteration $j+1$ no coefficient is reassigned to node $i-1$ due to load balancing.

- In case $k = \min(S_i^{(j)}) = \max(S_{i-1}^{(j+1)})$, $i \neq 1$, the top n rows of $(P^{(j)}f^{(j)})[k]$ are sent from node i to node $i-1$. The example in Figure 1 has only one such case, namely for coefficient $k = 8$. This happens, if coefficient k got reassigned from node i to node $i-1$ in iteration $j+1$.

If $\max(d^{(j+1)}) = \max(d^{(j)})$, i.e. the maximum nominal degree is not increased during iteration step j, only the first case occurs since no coefficient is added and therefore reassignment of coefficients is not necessary.

The implementation of this parallelization scheme uses the Message Passing Interface (MPI) for computer clusters and OpenMP for multi-core architectures. For OpenMP, each core is treated as one node in the parallelization scheme. Note that the communication for the parallelization with OpenMP is not programmed explicitly since all cores have access to all coefficients; however, the workload distribution is performed as described above. For the cluster implementation, each cluster node is used as one node in the parallelization scheme. Broadcast communication for the computation of $H^{(j)}$ is implemented using a call to the `MPI_Allreduce` function. One-to-one communication during the multiplication by Q is performed with the non-blocking primitives `MPI_Isend` and `MPI_Irecv` to avoid deadlocks during communication. Both OpenMP and MPI can

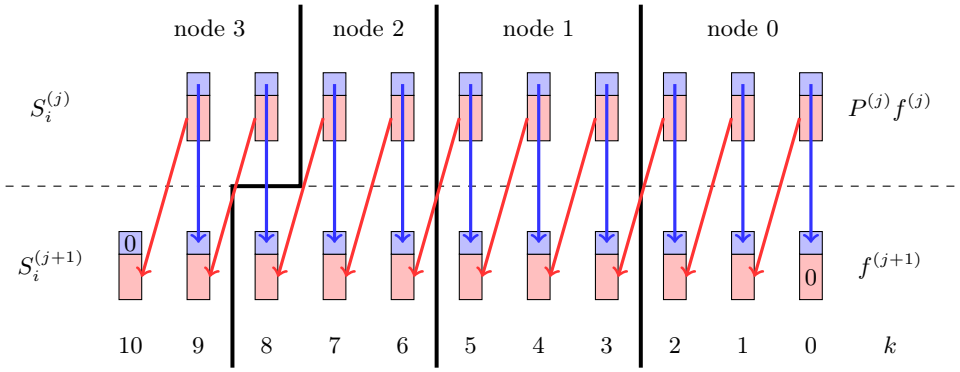

Figure 1. Example for the communication between 4 nodes. The top n rows of the coefficients are colored in blue, the bottom m rows are colored in red.

be used together for clusters of multi-core architectures. For NUMA systems the best performance is achieved when one MPI process is used for each NUMA node since this prevents expensive remote-memory accesses during computation.

The communication overhead of this parallelization scheme is very small. In each iteration, each node only needs to receive and/or send data of total size $O(n^2)$. Expensive broadcast communication is only required rarely compared to the time spent for computation. Therefore this parallelization of Coppersmith's Berlekamp–Massey algorithm scales well on a large number of nodes. Furthermore, since $f^{(j)}$ is distributed over the nodes, the memory requirement is distributed over the nodes as well.

4 Thomé's subquadratic version of the block Berlekamp–Massey algorithm

In 2002 Thomé presented an improved version of Coppersmith's variation of the Berlekamp–Massey algorithm [20]. Thomé's version is asymptotically faster: It reduces the complexity from $O(N^2)$ to $O(N \log^2(N))$ (assuming that m and n are constants). The subquadratic complexity is achieved by converting the block Berlekamp–Massey algorithm into a recursive divide-and-conquer process. Thomé's version builds the output polynomial $f(\lambda)$ of BW2 using a binary product tree; therefore, the main operations in the algorithm are multiplications of matrix polynomials. The implementation of Coppersmith's version of the algorithm is used to handle bottom levels of the recursion in Thomé's algorithm, as suggested in [20, Section 4.1].

The main computations in Thomé's version of the Berlekamp–Massey algorithm are multiplications of matrix polynomials. The first part of this section will take a brief look how to implement these efficiently. The second part gives an overview of the approach for the parallelization of Thomé's Berlekamp–Massey algorithm.

4.1 Matrix polynomial multiplications

In order to support multiplication of matrix polynomials with various operand sizes in Thomé's Berlekamp–Massey algorithm, several implementations of multiplication algorithms are used including Karatsuba, Toom–Cook, and FFT-based multiplications. FFT-based multiplications are the most important ones because they are used to deal with computationally expensive multiplications of operands with large degrees.

Different kinds of FFT-based multiplications are used for different fields: The field $\mathbb{F}_2$ uses the radix-3 FFT multiplication presented in [19]. For $\mathbb{F}_{16}$ the operands are transformed into polynomials over $\mathbb{F}_{16^9}$ by packing groups of 5 coefficients together. Then a mixed-radix FFT is applied using a primitive r-th root of unity in $\mathbb{F}_{16^9}$. In order to accelerate FFTs, it is ensured that r is a

number without large (≥ 50) prime factors. $\mathbb{F}_{16^9}$ is chosen because it has several advantages. First, by exploiting the Toom-Cook multiplication, a multiplication in $\mathbb{F}_{16^9}$ takes only $9^{\log_3 5} = 25$ multiplications in $\mathbb{F}_{16}$. Moreover, by setting $\mathbb{F}_{16} = \mathbb{F}_2[x]/(x^4 + x + 1)$ and $\mathbb{F}_{16^9} = \mathbb{F}_{16}[y]/(y^9 + x)$, reductions after multiplications can be performed efficiently because of the simple form of $y^9 + x$. Finally, $16^9 - 1$ has many small prime factors and thus there are plenty of choices of r to cover various sizes of operands.

4.2 Parallelization of Thomé's Berlekamp–Massey algorithm

Thomé's Berlekamp–Massey algorithm uses multiplication of large matrix polynomials and Coppersmith's Berlekamp–Massey algorithm as building blocks. The parallelization of Coppersmith's version has already been explained. Here the parallelization of the matrix polynomial multiplications is described on the example of the FFT-based multiplication.

The FFT-based multiplication is mainly composed of 3 stages: forward FFTs, point-wise multiplications, and the reverse FFT. Let f, g be the inputs of forward FFTs and f', g' be the corresponding outputs; the point-wise multiplications take f', g' as operands and give h' as output; finally, the reverse FFT takes h' as input and generates h.

For this implementation, the parallelization scheme for Thomé's Berlekamp–Massey algorithm is quite different from that for Coppersmith's: Each node deals with a certain range of rows. In the forward and reverse FFTs the rows of f, g, and h' are independent. Therefore, each FFT can be carried out in a distributed manner without communication. The problem is that the point-wise multiplications require partial f' but full g'. To solve this each node collects the missing rows of g' from the other nodes. This is done by using the function `MPI_Allgather`. Karatsuba and Toom-Cook multiplication are parallelized in a similar way.

One drawback of this scheme is that the number of nodes is limited by the number of rows of the operands. However, when the Macaulay matrix B is very large, the runtime of BW2 is very small compared to BW1 and BW3 since it is subquadratic in N. In this case using a different, smaller cluster or a powerful multi-core machine for BW2 might give a sufficient performance as suggested in [10]. Another drawback is, that the divide-and-conquer approach and the recursive algorithms for polynomial multiplication require much more memory than Coppersmith's version of the Berlekamp–Massey algorithm. Thus Coppersmith's version might be a better choice on memory-restricted architectures or for very large systems.

5 Implementation of XL

Stage BW1 of the block Wiedemann algorithm computes $a^{(i)} = \big(x \cdot (B \cdot B^i z)\big)^T$, $0 \leq i \leq N/\tilde{m} + N/\tilde{n} + \mathrm{O}(1)$. We do this efficiently using two sparse-matrix multiplications by making the random matrices x and z deliberately sparse. We compute a sequence $\{t^{(i)}\}_{i=0}^{\infty}$ of matrices $t^{(i)} \in K^{N \times n}$ defined as

$$t^{(i)} = \begin{cases} Bz & \text{for } i = 0 \\ Bt^{(i-1)} & \text{for } i > 0. \end{cases}$$

Thus, $a^{(i)}$ can be computed as $a^{(i)} = (xt^{(i)})^T$. In step BW3 we evaluate the annihilating polynomial $f(\lambda)$ by applying Horner's scheme, again using two sparse-matrix multiplications by computing

$$W^{(j)} = \begin{cases} z \cdot (f[\deg(f)]) & \text{for } j = 0, \\ z \cdot (f[\deg(f) - j]) + B \cdot W^{(j-1)} & \text{for } 0 < j \leq \deg(f). \end{cases}$$

For details on the steps BW1, BW2, and BW3 please refer to [18, Section 4.2].

Assuming that $\tilde{m} = c \cdot \tilde{n}$ for some constant $c \geq 1$, the asymptotic time complexity of step BW1 and BW2 can be written as $\mathrm{O}\big(N^2 \cdot w_B\big)$, where w_B is the average number of nonzero entries per row of B. Note that BW3 actually requires about half of the time of BW1 since it requires only about halve as many iterations. The asymptotic time complexity of Coppersmith's version of

the Berlekamp–Massey algorithm in step BW2 is $\mathrm{O}(N^2 \cdot \tilde{n})$. Thomé presents an improved version of Coppersmith's block Berlekamp–Massey algorithm in [20]. Thomé's version is asymptotically faster: It reduces the complexity of BW2 from $\mathrm{O}(N^2 \cdot \tilde{n})$ to $\mathrm{O}(N \cdot \log^2(N) \cdot \tilde{n})$. The subquadratic complexity is achieved by converting the block Berlekamp–Massey algorithm into a recursive divide-and-conquer process.

Since steps BW1 and BW3 have a higher asymptotic time complexity than Thomé's version of step BW2, we do not describe our implementation, optimization, and parallelization of Coppersmith's and Thomé's versions of step BW2 in detail in this paper for the sake of brevity. The interested reader is referred to [18, Chap. 4] for details. However, we discuss the performance of our implementations in Section 6.

Since the system matrix $\mathcal{M}$ has more rows than columns, some rows must be dropped randomly to obtain a square matrix B. Observe that due to the extension step of XL the entries of the original quadratic system $\mathcal{A}$ appear repeatedly in the matrix B at well-defined positions based on the enumeration scheme. Therefore, it is possible to generate the entries of B on demand spending a negligible amount of memory. However, the computation of the entry positions requires additional time; to avoid this computational overhead, we store the Macaulay matrix B in a compact memory format (see [18, Section 4.5.3]). This gives a significant speedup in the computation time—given that the matrix B fits into available memory.

5.1 Efficient matrix multiplication

All matrix multiplications of the shape $D = EF$ that we perform during XL are either multiplications of a sparse matrix by a dense matrix, or multiplications of a dense matrix by a dense matrix with matrices of a small size. For these cases, schoolbook multiplication is more efficient than the *asymptotically* more efficient Strassen algorithm or the Coppersmith–Winograd algorithm.

However, when computing in finite fields, the cost of matrix multiplications can be significantly reduced by trading expensive multiplications for cheap additions—if the field size is significantly larger than the row weight of E. This is the case for small fields like, for example, $\mathbb{F}_{16}$ or $\mathbb{F}_{31}$. We reduce the number of actual multiplications for a row r of E by summing up all row vectors of F which are to be multiplied by the same field element and performing the multiplication on all of them together. A temporary buffer $b_\alpha \in K^n, \alpha \in K$ of vectors of length n is used to collect the sum of row vectors that ought to be multiplied by α. For all entries $E_{r,c}$, row c of F is added to $b_{E_{r,c}}$. Finally, b can be reduced by computing $\sum \alpha \cdot b_\alpha, \alpha \neq 0, \alpha \in K$, which gives the result for row r of the matrix D.

With the strategy explained so far, computing the result for one row of E takes $w_E + |K| - 2$ additions and $|K| - 2$ scalar multiplications (there is no need for the multiplication by 0 and 1, and for the addition of 0). The number of actual multiplications can be further reduced by exploiting the distributivity of the scalar multiplication of vectors: Assume in the following that $K = \mathbb{F}_{p^k} = \mathbb{F}_p[x]/(f(x))$, with p prime and $f(x)$ an irreducible polynomial with $\deg(f) = k$. When $k = 1$, the natural mapping from K to $\{0, 1, \ldots, p-1\} \subset \mathbb{N}$ induces an order of the elements. The order can be extended for $k > 1$ by $\forall \beta, \gamma \in K : \beta > \gamma \iff \beta[i] > \gamma[i], i = \max(\{j : \beta[j] \neq \gamma[j]\})$. We decompose each scalar factor $\alpha \in K \setminus \{0, 1, x^1, \ldots, x^{k-1}\}$ of a multiplication $\alpha \cdot b_\alpha$ into two components $\beta, \gamma \in K$ such that $\beta, \gamma < \alpha$ and $\beta + \gamma = \alpha$. Starting with the largest α, iteratively add b_α to b_β and b_γ and drop buffer b_α. The algorithm terminates when all buffers b_α, $\alpha \in K \setminus \{0, 1, x^1, \ldots, x^{k-1}\}$ have been dropped. Finally, the remaining buffers b_α, $\alpha \in \{1, x^1, \ldots, x^{k-1}\}$ are multiplied by their respective scalar factor (except b_1) and summed up to the final result. This reduces the number of multiplications to $k - 1$. All in all the computation on one row of E (with row weight w_E) costs $w_E + 2(|K| - k - 1) + k - 1$ additions and $k - 1$ scalar multiplications. For example the computations in $\mathbb{F}_{16}$ require $w_E + 25$ additions and 3 multiplications per row of a matrix E.

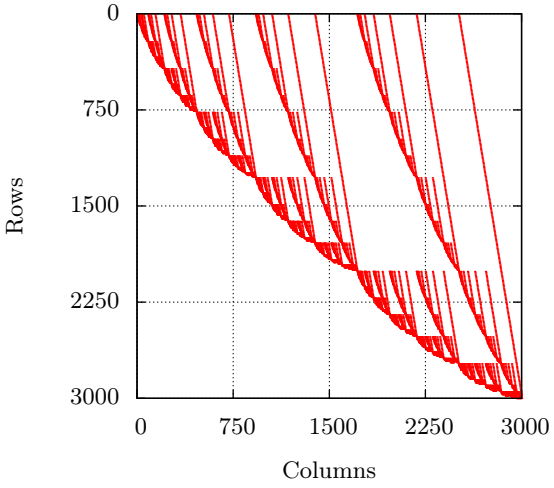

Figure 2. Plot of a Macaulay matrix for a system with 8 variables, 10 equations, using graded reverse lexicographical (grevlex) monomial order.

5.2 Parallel Macaulay matrix multiplication

The most expensive part in the computation of steps BW1 and BW3 of XL is a repetitive multiplication of the shape $t_{new} = B \cdot t_{old}$, where $t_{new}, t_{old} \in K^{N \times \tilde{n}}$ are dense matrices and $B \in K^{N \times N}$ is a sparse Macaulay matrix with an average row weight w_B.

For generic systems, the Macaulay matrix B has an expected number of non-zero entries per row of $(|K| - 1)/|K| \cdot \binom{n+2}{2}$. However, in our memory efficient data format for the Macaulay matrix we also store the zero entries from the original system. This results in a fixed row weight $w_B = |K| \cdot \binom{n+2}{2}$. This is highly efficient in terms of memory consumption and computation time for $\mathbb{F}_{16}$, $\mathbb{F}_{31}$, and larger fields (see [18, Chap. 4]). Since there is a guaranteed number of entries per row (i.e. the row weight w_B) we compute the Macaulay matrix multiplication in row order in a big loop over all row indices as described in the previous section.

The parallelization of the Macaulay matrix multiplication of steps BW1 and BW3 is implemented in two ways: On multi-core architectures OpenMP is used to keep all cores busy; on cluster architectures the Message Passing Interface (MPI) and InfiniBand verbs are used to communicate between the cluster nodes. Both approaches can be combined for clusters of multi-core nodes.

The strategy of the workload distribution is similar on both multi-core systems and cluster systems. Figure 2 shows an example of a Macaulay matrix. Our approach for efficient matrix multiplications (described in the previous section) trades multiplications for additions. The approach is most efficient, if the original number of scalar multiplications per row is much higher than the order of the field. Since the row weight of the Macaulay matrix is quite small, splitting the rows between computing nodes reduces the efficiency of our approach. Therefore, the workload is distributed by assigning blocks of rows of the Macaulay matrix to the computing units.

Parallelization for Shared-Memory Systems: We parallelize the data-independent loop over the rows of the Macaulay matrix using OpenMP with the directive "`#pragma omp parallel for`". The OpenMP parallelization on UMA systems encounters no additional communication cost although the pressure on shared caches may be increased. On NUMA systems the best performance is achieved if the data is distributed over the NUMA nodes in a way that takes the higher cost of remote memory access into account. However, the access pattern to t_{old} is very irregular due to the structure of the Macaulay matrix: In particular, the access pattern of each core does not necessarily fully cover memory pages. Furthermore, the same memory page is usually touched by several

cores. The same is true for t_{new}, since after each iteration t_{new} and t_{old} are swapped by switching their respective memory regions. Therefore, we obtained the shortest runtime by distributing the memory pages interleaved (in a round-robin fashion) over the nodes.

Parallelization for Cluster Systems: The computation on one row of the Macaulay matrix depends on many rows of the matrix t_{old}. A straightforward approach is to make the full matrix t_{old} available on all cluster nodes. This can be achieved by an all-to-all communication step after each iteration of BW1 and BW3. If B were a dense matrix, such communication would take only a small portion of the overall runtime. But since B is a sparse Macaulay matrix which has a very low row weight, the computation time for one single row of B takes only a small amount of time. In fact this time is in the order of magnitude of the time that is necessary to send one row of t_{new} to all other nodes during the communication phase. Therefore, this simple workload-distribution pattern gives a large communication overhead.

This overhead is hidden when communication is performed in parallel to computation. Today's high-performance network interconnects are able to transfer data via direct memory access (DMA) without interaction with the CPU, allowing the CPU to continue computations alongside communication. It is possible to split the computation of t_{new} into two column blocks; during computation on one block, previously computed results are distributed to the other nodes and therefore are available at the next iteration step. Under the condition that computation takes more time than communication, the communication overhead can almost entirely be hidden. Otherwise speedup and therefore efficiency of cluster parallelization is bounded by communication cost.

Apart from hiding the communication overhead it is also possible to totally avoid all communication by splitting t_{old} and t_{new} into independent column blocks for each cluster node. However, splitting t_{old} and t_{new} has an impact either on the runtime of BW1 and BW3 (if the block size becomes too small for efficient computation) or on the runtime of BW2 (since the block size has a strong impact on its runtime and memory demand).

We implemented both approaches since they can be combined to give best performance on a target system architecture. The following paragraphs explain the two approaches in detail:

a) Operating on Two Shared Column Blocks of t_{old} and t_{new}: For this approach, the matrices t_{old} and t_{new} are split into two column blocks $t_{old,0}$ and $t_{old,1}$ as well as $t_{new,0}$ and $t_{new,1}$. The workload is distributed over the nodes row-wise as mentioned before. First each node computes the results of its row range for column block $t_{new,0}$ using rows from block $t_{old,0}$. Then a non-blocking all-to-all communication is initiated which distributes the results of block $t_{new,0}$ over all nodes. While the communication is going on, the nodes compute the results of block $t_{new,1}$ using data from block $t_{old,1}$. After computation on $t_{new,1}$ is finished, the nodes wait until the data transfer of block $t_{new,0}$ has been accomplished. Ideally communication of block $t_{new,0}$ is finished earlier than the computation of block $t_{new,1}$ so that the results of block $t_{new,1}$ can be distributed without waiting time while the computation on block $t_{new,0}$ goes on with the next iteration step.

However, looking at the structure of the Macaulay matrix (an example is shown in Figure 2) one can observe that this communication scheme performs much more communication than necessary. For example on a cluster of four computing nodes, node 0 computes the top quarter of the rows of matrices $t_{new,0}$ and $t_{new,1}$. Node 1 computes the second quarter, node 2 the third quarter, and node 3 the bottom quarter. Node 3 does not require any row that has been computed by node 0 since the Macaulay matrix does not have entries in the first quarter of the columns for these rows. The obvious solution is that a node i sends only these rows to a node j that are actually required by node j in the next iteration step.

This communication pattern requires to send several data blocks to individual cluster nodes in parallel to ongoing computation. This can not be done efficiently using MPI. Therefore, we circumvent the MPI API and program the network hardware directly. Our implementation uses an InfiniBand network; the same approach can be used for other high-performance networks. We access the InfiniBand hardware using the InfiniBand verbs API. Programming the InfiniBand cards directly has several benefits: All data structures that are required for communication can

be prepared offline; initiating communication requires only one call to the InfiniBand API. The hardware is able to perform all operations for sending and receiving data autonomously after this API call; there is no need for calling further functions to ensure communication progress as it is necessary when using MPI. Finally, complex communication patterns using scatter-gather lists for incoming and outgoing data do not have a large overhead. This implementation reduces communication to the smallest amount possible for the cost of only a negligibly small initialization overhead.
This approach of splitting t_{old} and t_{new} into two shared column blocks has the disadvantage that the entries of the Macaulay matrix need to be loaded twice per iteration, once for each block. This gives a higher memory contention and more cache misses than when working on a single column block. However, these memory accesses are sequential. It is therefore likely that the access pattern can be detected by the memory logic and that the data is prefetched into the caches.

b) *Operating on Independent Column Blocks of t_{old} and t_{new}:* Any communication during steps BW1 and BW3 can be avoided by splitting the matrices t_{old} and t_{new} into independent column blocks for each cluster node. The nodes compute over the whole Macaulay matrix B on a column stripe of t_{old} and t_{new}. All computation can be accomplished locally; the results are collected at the end of the computation of these steps.
Although this is the most efficient parallelization approach when looking at communication cost, the per-node efficiency drops drastically with higher node count: For a high node count, the impact of the width of the column stripes of t_{old} and t_{new} becomes even stronger than for the previous approach. Therefore, this approach only scales well for small clusters. For a large number of nodes, the efficiency of the parallelization declines significantly. Another disadvantage of this approach is that since the nodes compute on the whole Macaulay matrix, all nodes must store the whole matrix in their memory. For large systems this is may not be feasible.

Both approaches for parallelization have advantages and disadvantages; the ideal approach can only be found by testing each approach on the target hardware. For small clusters approach b) might be the most efficient one although it loses efficiency due to the effect of the width of t_{old} and t_{new}. The performance of approach a) depends heavily on the network configuration and the ratio between computation time and communication time. Both approaches can be combined by splitting the cluster into independent partitions; the workload is distributed over the partitions using approach b) and over the nodes within one partition using approach a).

6 Experimental results

This section gives an overview of the performance and the scalability of our XL implementation for generic systems. Experiments have been carried out on two computer systems: a 64-core NUMA system and an eight node InfiniBand cluster. Table 2 lists the key features of these systems.

6.1 Impact of the block size

We measured the impact of the block size of the block Wiedemann algorithm on the performance of the implementation on a single cluster node (without cluster communication). We used a quadratic system with 16 equations and 14 variables over $\mathbb{F}_{16}$. In this case, the degree D for the linearization is 9. The input for the algorithm is a Macaulay matrix B with $N = 817190$ rows (and columns) and row weight $w_B = 120$. To reduce the parameter space, we fix $\tilde{m}$ to $\tilde{m} = \tilde{n}$.

Figure 3 shows the runtime for block sizes 32, 64, 128, 256, 512, and 1024. Given the fixed size of the Macaulay matrix and $\tilde{m} = \tilde{n}$, the number of field operations for BW1 and BW2 is roughly the same for different choices of the block size $\tilde{n}$ since the number of iterations is proportional

	NUMA	Cluster
CPU		
Name	AMD Opteron 6276	Intel Xeon E5620
Microarchitecture	Bulldozer Interlagos	Nehalem
Frequency	2300 MHz	2400 MHz
Number of CPUs per socket	2	1
Number of cores per socket	16 (2 x 8)	4
Level 1 data-cache size	16 × 48 KB	4 × 32 KB
Level 2 data-cache size	8 × 2 MB	4 × 256 KB
Level 3 data-cache size	2 × 8 MB	8 MB
Cache-line size	64 byte	64 byte
System Architecture		
Number of NUMA nodes	4 sockets × 2 CPUs	2 sockets × 1 CPU
Number of cluster nodes	—	8
Total number of cores	64	64
Network interconnect	—	InfiniBand MT26428
		2 ports of 4×QDR, 32 Gbit/s
Memory		
Memory per CPU	32 GB	18 GB
Memory per cluster node	—	36 GB
Total memory	256 GB	288 GB

Table 2. Computer architectures used for the experiments

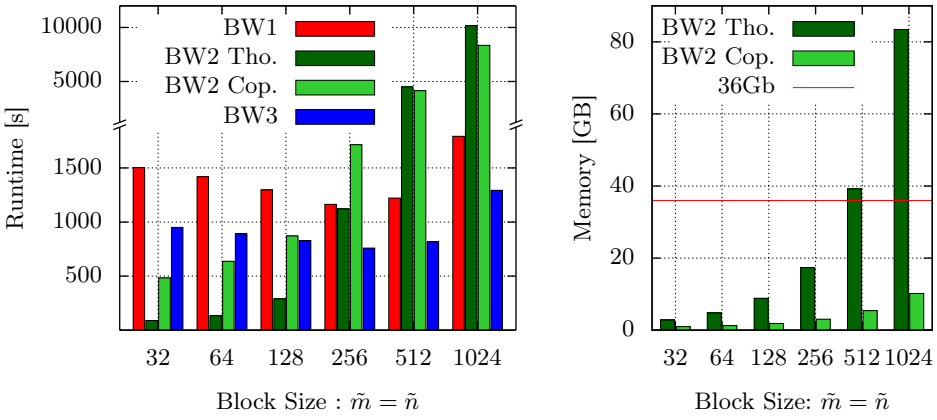

Figure 3. Runtime and memory consumption of XL 16-14 over different block sizes on a single cluster node with two CPUs (8 cores in total) and 36 GB RAM.

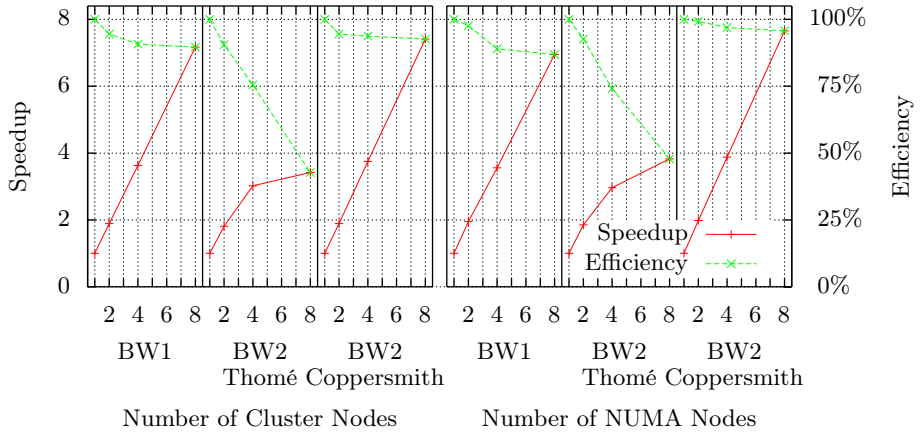

Figure 4. Speedup and efficiency of BW1 and BW2

to $1/\tilde{n}$ and number of field operations per iteration is roughly proportional to $\tilde{n}$. However, the runtime of the computation varies depending on $\tilde{n}$.

During the i-th iteration step of BW1 and BW3, the Macaulay matrix is multiplied with a matrix $t^{(i-1)} \in \mathbb{F}_{16}^{N \times \tilde{n}}$. For $\mathbb{F}_{16}$ each row of $t^{(i-1)}$ requires $\tilde{n}/2$ bytes of memory. In the cases $\tilde{m} = \tilde{n} = 32$ and $\tilde{m} = \tilde{n} = 64$ each row thus occupies less than one cache line of 64 bytes. This explains why the best performance in BW1 and BW3 is achieved for larger values of $\tilde{n}$. The runtime of BW1 and BW3 is minimal for block sizes $\tilde{m} = \tilde{n} = 256$. In this case one row of $t^{(i-1)}$ occupies two cache lines. The reason why this case gives a better performance than $\tilde{m} = \tilde{n} = 128$ might be that the memory controller is able to prefetch the second cache line. For larger values of $\tilde{m}$ and $\tilde{n}$ the performance declines probably due to cache saturation.

According to the asymptotic time complexity of Coppersmith's and Thomé's versions of the Berlekamp–Massey algorithm, the runtime of BW2 should be proportional to $\tilde{n}$. However, this turns out to be the case only for moderate sizes of $\tilde{n}$; note the different scale of the graph in Figure 3 for a runtime of more than 2000 seconds. For $\tilde{m} = \tilde{n} = 256$ the runtime of Coppersmith's version of BW2 is already larger than that of BW1 and BW3, for $\tilde{m} = \tilde{n} = 512$ and $\tilde{m} = \tilde{n} = 1024$ both versions of BW2 dominate the total runtime of the computation. Thomé's version is faster than Coppersmith's version for small and moderate block sizes. However, by doubling the block size, the memory demand of BW2 roughly doubles as well; Figure 3 shows the memory demand of both variants for this experiment. Due to the memory–time trade-off of Thomé's BW2, the memory demand exceeds the available RAM for a block size of $\tilde{m} = \tilde{n} = 512$ and more. Therefore, memory pages are swapped out of RAM onto hard disk which makes the runtime of Thomé's BW2 longer than that of Coppersmith's version of BW2.

6.2 Scalability experiments

The scalability was measured using a quadratic system with 18 equations and 16 variables over $\mathbb{F}_{16}$. The degree D for this system is 10. The Macaulay matrix B has a size of $N = 5\,311\,735$ rows and columns; the row weight is $w_B = 153$. Since this experiment is not concerned about peak performance but about scalability, a block size of $\tilde{m} = \tilde{n} = 256$ is used. For this experiment, the implementation of the block Wiedemann algorithm ran on 1, 2, 4, and 8 nodes of the cluster and on 1 to 8 CPUs of the NUMA system. The approach a) (two shared column blocks) was used on the cluster system for all node counts.

Given the runtime T_1 for one computing node and T_p for p computing nodes, the parallel efficiency E_p on the p nodes is defined as $E_p = T_1/pT_p$. Figure 4 shows the parallel speedup and

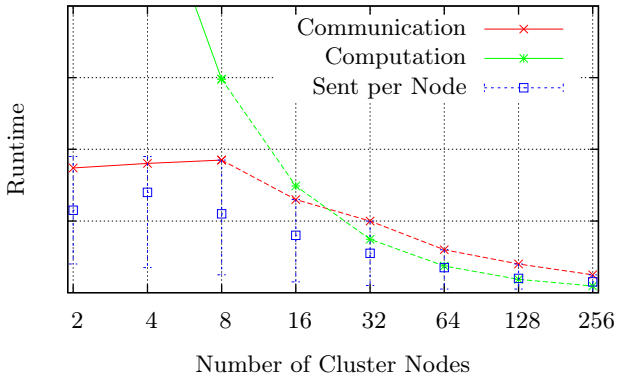

Figure 5. Estimation of computation time vs. communication time on a cluster system. The numbers for 2, 4, and 8 nodes are measurements, the numbers for larger cluster sizes are estimations. The amount of data sent per node varies; we show the maximum, minimum, and average.

the parallel efficiency of BW1 and BW2; the performance of BW3 behaves very similarly to BW1 and thus is not depicted in detail. These figures show that BW1 and Coppersmith's BW2 have a nice speedup and an efficiency of at least 90% on 2, 4, and 8 cluster nodes. The efficiency of Thomé's BW2 is only around 75% on 4 nodes and drops to under 50% on 8 nodes. In particular the polynomial multiplications require a more efficient parallelization approach. However, Thomé's BW2 takes only a small part of the total runtime for this system size; for larger systems it is even smaller due to its smaller asymptotic time complexity compared to steps BW1 and BW3. Thus, a lower scalability than BW1 and BW3 can be tolerated for BW2.

For this problem size, our parallel implementation of BW1 and BW3 scales very well for up to eight nodes. However, at some point the communication time is going to catch up with computation time: The computation time roughly halves with every doubling of the number of cluster nodes, while the communication demand per node shrinks with a smaller slope. Therefore, at a certain number of nodes communication time and computation time are about the same and the parallel efficiency declines for any larger number of nodes. We do not have access to a cluster with a fast network interconnect and a sufficient amount of nodes to measure when this point is reached, thus we can only give an estimation: Figure 5 shows the expected time of computation and communication for larger cluster sizes. We computed the amount of data that an individual node sends and receives depending on the number of computing nodes. We use the maximum of the outgoing data for the estimation of the communication time. For this particular problem size, we expect that for a cluster of around 16 nodes communication time is about as long as computation time and that the parallel efficiency is going to decline for larger clusters.

On the NUMA system, the scalability is similar to the cluster system. BW1 achieves an efficiency of over 85% on up to 8 NUMA nodes. The workload was distributed such that each CPU socket was filled up with OpenMP threads as much as possible. Therefore, in the case of two NUMA nodes (16 threads) the implementation achieves a high efficiency of over 95% since a memory controller on the same socket is used for remote memory access and the remote memory access has only moderate cost. When using more than one NUMA node, the efficiency declines to around 85% due to the higher cost of remote memory access between different sockets. Also on the NUMA system the parallelization of Thomé's BW2 achieves only a moderate efficiency of around 50% for 8 NUMA nodes. The parallelization scheme used for OpenMP does not scale well for a large number of threads. The parallelization of Coppersmith's version of BW2 scales almost perfectly on the NUMA system. The experiment with this version of BW2 is performed using hybrid parallelization by running one MPI process per NUMA node and one OpenMP thread per core. The overhead for

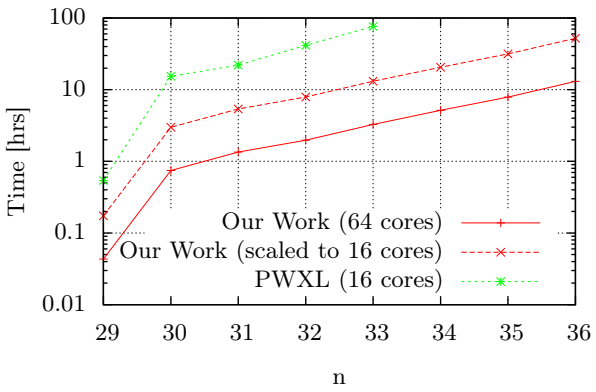

Figure 6. Comparison of the runtime of our work and PWXL, $m = n$, $\mathbb{F}_2$

communication is sufficiently small that it does not have much impact on the parallel efficiency of up to 8 NUMA nodes.

Our experiments show that the shape of the Macaulay matrix has a large impact on the performance and the scalability of XL. Currently, we are using graded reverse lexicographical order for the Macaulay matrix. However, as opposed to Gröbner basis solvers like F_4 and F_5, for XL there is no algorithmic or mathematic requirement for any particular ordering. In our upcoming research, we are going to examine if another monomial order or a redistribution of columns and rows of the Macaulay matrix has a positive impact on the performance of our implementation.

6.3 Comparison with PWXL and Magma F_4

To put our numbers into context, we compare our work with two other Gröbner basis solvers in this section: with PWXL, a parallel implementation of XL with block Wiedemann for $\mathbb{F}_2$ described in [15], and with the implementation of Faugère's F_4 algorithm [7] in the computational algebra system Magma.

Comparison with PWXL: Figure 6 compares the runtime of PWXL and our implementation for systems in $\mathbb{F}_2$ with $m = n$. We ran our XL implementation on our cluster system (see Table 2) while PWXL was running on a machine with four six-core AMD Opteron 8435 CPUs, running at 2.6 GHz.

Our implementation outperforms PWXL for the largest cases given in the paper, e.g., for $n = 33$ our implementation is 24 times faster running on 8 cluster nodes (64 CPU cores) and still 6 times faster when scaling to 16 CPU cores. This significant speedup may be explained by the fact that PWXL is a modification of the block-Wiedemann solver for factoring RSA-768 used in [10]. Therefore, the code may not be well optimized for the structure of Macaulay matrices. However, these numbers show that our implementation achieves high performance for computations in $\mathbb{F}_2$.

Comparison with F_4: Figure 7 compares time and memory consumption of the F_4 implementation in Magma V2.17-12 and our implementation of XL for systems in $\mathbb{F}_{16}$ with $m = 2n$. When solving the systems in Magma we coerce the systems into $\mathbb{F}_{256}$, because for $\mathbb{F}_{256}$ Magma performs faster than when using $\mathbb{F}_{16}$ directly. The computer used to run F_4 has an 8 core Xeon X7550 CPU running at 2.0 GHz; however, F_4 uses only one core of it. We ran XL on our NUMA system using all 64 CPU cores. For this comparison we use Coppersmith's version of BW2 since it is more memory efficient than Thomé's version.

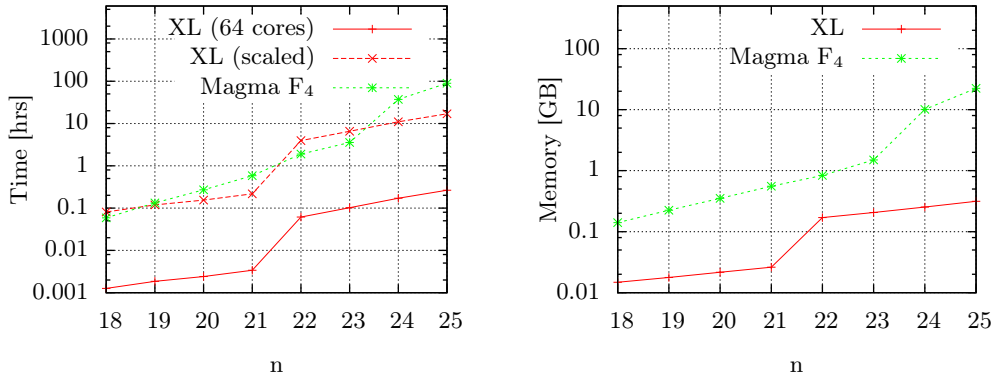

Figure 7. Comparison of runtime and memory demand of our implementation of XL and Magma's implementation of F_4, $m = 2n$

Note that there is a jump in the graph when going from $n = 21$ to $n = 22$ for XL our implementation, similarly when going from $n = 23$ to $n = 24$ for F_4. This is due to an increment of the degree D from 5 to 6, which happens earlier for XL. Therefore, F_4 takes advantage of a lower degree in cases such as $n = 22, 23$. Other XL-based algorithms like Mutant-XL [14] may be able to fill this gap. In this paper we omit a discussion of the difference between the degrees of XL and F_4/F_5. However, in cases where the degrees are the same for both algorithms, our implementation of XL is better in terms of runtime and memory consumption.

For $n = 25$, the memory consumption of XL is less than 2% of that of F_4. In this case, XL runs 338 times faster on 64 cores than F_4 on one single core, which means XL is still faster when the runtime is normalized to single-core performance by multiplying the runtime by 64.

6.4 Performance for computation on large systems

Table 3 presents detailed statistics of some of the largest systems we are able to solve in a moderate amount of time (within at most one week). In the tables the time (BW1, BW2, BW3, and total) is measured in seconds, and the memory is measured in GB. Note that for the cluster we give the memory usage for a single cluster node. While all the fields that we have implemented so far are presented in the table, we point out that the most optimization has been done for $\mathbb{F}_{16}$.

The system with $n = 32$ variables and $m = 64$ equations over $\mathbb{F}_{16}$ listed in Table 3 is the largest case we have tested. The system was solved in 5 days on the cluster using block sizes $\tilde{m} = 256$ and $\tilde{n} = 128$. With $n = 32$ and $D = 7$ we have $N = \binom{n+D}{D} = \binom{32+7}{7} = 15\,380\,937$ and $w_B = \binom{n+2}{2} = \binom{32+2}{2} = 561$. There are roughly $N/\tilde{n}+N/\tilde{m}$ iterations in BW1 and $N/\tilde{n}$ iterations in BW3. This leads to $2N/\tilde{n} + N/\tilde{m}$ Macaulay matrix multiplications, each takes about $N \cdot (w_B + 25) \cdot \tilde{n}$ additions and $N \cdot 3 \cdot \tilde{n}$ multiplications in $\mathbb{F}_{16}$ (see Section 5.2). Operations performed in BW2 are not taken into account, because BW2 requires only a negligible amount of time. Therefore, solving the system using XL corresponds to computing about $(2 \cdot 15\,380\,937/128 + 15\,380\,937/256) \cdot 15\,380\,937 \cdot (561 + 25) \cdot 128 \approx 2^{58.3}$ additions and about $2^{50.7}$ multiplications in $\mathbb{F}_{16}$. Since one addition in $\mathbb{F}_{16}$ requires 4 bit operations, this roughly corresponds to the computation of $4 \cdot 2^{58.3} \approx 2^{60.3}$ bit operations.

7 Source code

The Source code is available at http://www.polycephaly.org/xl/.

Field	Machine	m	n	D	Time in [sec] BW1	BW2	BW3	total	Memory in [GB]	Block Size $\tilde{m}$, $\tilde{n}$
$\mathbb{F}_2$	Cluster	32	32	7	3830	1259	2008	7116	2.4	512, 512
	Cluster	33	33	7	6315	2135	3303	11778	3.0	512, 512
	Cluster	34	34	7	10301	2742	5439	18515	3.8	512, 512
	Cluster	35	35	7	16546	3142	8609	28387	4.6	512, 512
	Cluster	36	36	7	26235	5244	15357	46944	5.6	512, 512
$\mathbb{F}_{16}$	NUMA	56	28	6	1866	330	984	3183	3.9	128,128
	Cluster				1004	238	548	1795	1.3	256,256
	NUMA	58	29	6	2836	373	1506	4719	4.6	128,128
	Cluster				1541	316	842	2707	1.6	256,256
	NUMA	60	30	7	91228	5346	64688	161287	68.8	256,128
	Cluster				53706	3023	38052	94831	10.2	256,128
	NUMA	62	31	7	145693	7640	105084	258518	76.7	256,128
	Cluster				89059	3505	67864	160489	12.1	256,128
	NUMA	64	32	7	232865	8558	163091	404551	100.3	256,128
	Cluster				141619	3672	97924	244338	15.3	256,128
$\mathbb{F}_{31}$	NUMA	50	25	6	1729	610	935	3277	0.3	64,64
	Cluster				1170	443	648	2265	0.7	128,128
	NUMA	52	26	6	2756	888	1483	5129	0.4	64,64
	Cluster				1839	656	1013	3513	0.9	128,128
	NUMA	54	27	6	4348	1321	2340	8013	0.5	64,64
	Cluster				2896	962	1590	5453	1.0	128,128
	NUMA	56	28	6	6775	1923	3610	12313	0.6	64,64
	Cluster				4497	1397	2458	8358	1.2	128,128
	NUMA	58	29	6	10377	2737	5521	18640	0.7	64,64
	Cluster				6931	2011	3764	12713	1.5	128,128

Table 3. Statistics of XL with block Wiedemann for $\mathbb{F}_2$ and $\mathbb{F}_{16}$ using Thomé's BW2, and $\mathbb{F}_{31}$ using Coppersmith's BW2

References

1. Ars, G., Faugère, J.C., Imai, H., Kawazoe, M., Sugita, M.: Comparison between XL and Gröbner basis algorithms. In: Lee, P. (ed.) Advances in Cryptology – ASIACRYPT 2004, Lecture Notes in Computer Science, vol. 3329, pp. 157–167. Springer (2004)
2. Coppersmith, D.: Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm. Mathematics of Computation 62(205), 333–350 (1994)
3. Courtois, N.T.: Higher order correlation attacks, XL algorithm and cryptanalysis of Toyocrypt. In: Lee, P., Lim, C. (eds.) Information Security and Cryptology – ICISC 2002, Lecture Notes in Computer Science, vol. 2587, pp. 182–199. Springer (2003)
4. Courtois, N.T., Klimov, A., Patarin, J., Shamir, A.: Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In: Preneel, B. (ed.) Advances in Cryptology – EUROCRYPT 2000, Lecture Notes in Computer Science, vol. 1807, pp. 392–407. Springer (2000)
5. Courtois, N.T., Pieprzyk, J.: Cryptanalysis of block ciphers with overdefined systems of equations. In: Zheng, Y. (ed.) Advances in Cryptology – ASIACRYPT 2002, Lecture Notes in Computer Science, vol. 2501, pp. 267–287. Springer (2002)
6. Diem, C.: The XL-algorithm and a conjecture from commutative algebra. In: Lee, P. (ed.) Advances in Cryptology – ASIACRYPT 2004. Lecture Notes in Computer Science, vol. 3329, pp. 146–159. Springer (2004)
7. Faugère, J.C.: A new efficient algorithm for computing Gröbner bases (F_4). Journal of Pure and Applied Algebra 139(1–3), 61–88 (1999)

8. Faugère, J.C.: A new efficient algorithm for computing Gröbner bases without reduction to zero (F_5). In: International Symposium on Symbolic and Algebraic Computation – ISSAC 2002. pp. 75–83. ACM (2002)
9. Faugère, J.C., Perret, L., Petit, C., Renault, G.: Improving the complexity of index calculus algorithms in elliptic curves over binary fields. In: Pointcheval, D., Johansson, T. (eds.) Advances in Cryptology – EUROCRYPT 2012, Lecture Notes in Computer Science, vol. 7237, pp. 27–44. Springer (2012)
10. Kleinjung, T., Aoki, K., Franke, J., Lenstra, A.K., Thomé, E., Bos, J.W., Gaudry, P., Kruppa, A., Montgomery, P.L., Osvik, D.A., Te Riele, H., Timofeev, A., Zimmermann, P.: Factorization of a 768-bit rsa modulus. In: Rabin, T. (ed.) Advances in Cryptology – CRYPTO 2010. Lecture Notes in Computer Science, vol. 6223, pp. 333–350. Springer (2010)
11. Lazard, D.: Gröbner-bases, Gaussian elimination and resolution of systems of algebraic equations. In: van Hulzen, J.A. (ed.) Computer Algebra – EUROCAL '83. Lecture Notes in Computer Science, vol. 162, pp. 146–156. Springer (1983)
12. Macaulay, F.S.: The Algebraic Theory of Modular Systems. No. 19 in Cambridge Tracts in Mathematics and Mathematical Physics, Cambridge University Press (1916)
13. Moh, T.T.: On the method of XL and its inefficiency to TTM. Cryptology ePrint Archive, Report 2001/047 (2001), http://eprint.iacr.org/2001/047
14. Mohamed, M.S.E., Mohamed, W.S.A.E., Ding, J., Buchmann, J.: MXL2: Solving polynomial equations over GF(2) using an improved mutant strategy. In: Buchmann, J., Ding, J. (eds.) Post-Quantum Cryptography – PQCrypto 2008. Lecture Notes in Computer Science, vol. 5299, pp. 203–215. Springer (2008)
15. Mohamed, W.S.A.E., Ding, J., Kleinjung, T., Bulygin, S., Buchmann, J.: PWXL: A parallel Wiedemann-XL algorithm for solving polynomial equations over GF(2). In: Cid, C., Faugère, J.C. (eds.) International Conference on Symbolic Computation and Cryptography – SCC 2010. pp. 89–100 (2010)
16. Montgomery, P.L.: A block Lanczos algorithm for finding dependencies over GF(2). In: Guillou, L., Quisquater, J.J. (eds.) Advances in Cryptology – EUROCRYPT '95, Lecture Notes in Computer Science, vol. 921, pp. 106–120. Springer (1995)
17. Murphy, S., Robshaw, M.J.B.: Essential algebraic structure within the AES. In: Yung, M. (ed.) Advances in Cryptology – CRYPTO 2002. Lecture Notes in Computer Science, vol. 2442, pp. 1–16. Springer (2002)
18. Niederhagen, R.: Parallel Cryptanalysis. Ph.D. thesis, Eindhoven University of Technology (2012), http://polycephaly.org/thesis/index.shtml
19. Schönhage, A.: Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. Acta Informatica 7(4), 395–398 (1977)
20. Thomé, E.: Subquadratic computation of vector generating polynomials and improvement of the block Wiedemann algorithm. Journal of Symbolic Computation 33(5), 757–775 (2002)
21. Wiedemann, D.H.: Solving sparse linear equations over finite fields. IEEE Transactions on Information Theory 32(1), 54–62 (1986)
22. Yang, B.Y., Chen, C.H., Bernstein, D.J., Chen, J.M.: Analysis of QUAD. In: Biryukov, A. (ed.) Fast Software Encryption, Lecture Notes in Computer Science, vol. 4593, pp. 290–308. Springer (2007)
23. Yang, B.Y., Chen, J.M.: All in the XL family: Theory and practice. In: Park, C., Chee, S. (eds.) Information Security and Cryptology – ICISC 2004, Lecture Notes in Computer Science, vol. 3506, pp. 32–35. Springer (2005)
24. Yang, B.Y., Chen, J.M., Courtois, N.T.: On asymptotic security estimates in XL and Gröbner bases-related algebraic cryptanalysis. In: Lopez, J., Qing, S., Okamoto, E. (eds.) Information and Communications Security – ICICS 2004. Lecture Notes in Computer Science, vol. 3269, pp. 281–286. Springer (2004)