A Modified Parallel F4 Algorithm for Shared and Distributed Memory Architectures

Severin Neumann

Fakultät für Mathematik und Informatik
Universität Passau, D-94030 Passau, Germany
`neumans@fim.uni-passau.de`

Abstract

In applications of symbolic computation an often required but complex procedure is finding Gröbner bases for polynomial ideals. Hence it is obvious to realize parallel algorithms to compute them. There are already flavours of the F4 algorithm like [4] and [13] using the special structure of the occurring matrices to speed up the reductions on parallel architectures with shared memory. In this paper we start from these and present modifications allowing efficient computations of Gröbner bases on systems with distributed memory. To achieve this we concentrate on the following objectives: decreasing the memory consumption and avoiding communication overhead. We remove not required steps of the reduction, split the columns of the matrix in blocks for distribution and review the effectiveness of the `SIMPLIFY` function. Finally we evaluate benchmarks with up to 256 distributed threads of an implementation being available at `https://github.com/svrnm/parallelGBC`.

1 Introduction

Parallelization is one of the most used methods to improve the performance of existing software. But it should be introduced systematically. One chooses the most time-consuming segments of a program and tries to improve these first. In applications of symbolic computation an often required but complex procedure is finding Gröbner bases for polynomial ideals. Therefore it is obvious to realize parallel versions of algorithms to compute them. Although it is possible to use Buchberger's algorithm [14], we favour Faugère's algorithm F4 [3] since the reduction is done due to matrix transformation which is known for being well parallelizable.

There are already parallel flavours of F4 like [4] and [13] using the special structure of the occurring matrices to speed up the reduction. Both approaches have been developed for multicore and multiprocessor systems with shared memory. For these systems the number of parallel processing units is limited and currently there are not many platforms serving more than 64 processors. Compared with that clusters of several suchlike computers can have an theoretical unlimited number of processors. This advantage comes with the downside of distributed memory: realizing algorithms requires to consider that data has to be transfered. By that memory duplication and communication overhead are introduced.

In the following we start from the mentioned developments for shared memory systems and present modifications which will finally allow efficient computation of Gröbner bases also for systems with distributed memory. To achieve this we have concentrated on the following objectives: reducing the memory consumption and avoiding communication overhead. We remove needless steps of the reduction, split the columns of the matrix in blocks for distribution and review the effectiveness of the `SIMPLIFY` function.

At the end we will evaluate benchmarks of an implementation using eight distributed nodes having 32 processor cores each. The source code is available at `https://github.com/svrnm/parallelGBC`.

2 Notations

In the following we will use the notation of [11]. Hence $\mathbb{K}$ denotes a field and $\mathbb{K}[x_1, \ldots, x_n]$ the polynomial ring over $\mathbb{K}$ in n indeterminantes. $\mathbb{T}^n$ is defined as the set of all terms $x_1^{\alpha_1} \cdot \ldots \cdot x_n^{\alpha_n}$. σ is a term ordering on this set. For a polynomial $f \in \mathbb{K}[x_1, \ldots, x_n]$ with $f = \sum_{i=1}^{m} c_i \cdot t_i$, where $c_i \in \mathbb{K}$ and $t_i \in \mathbb{T}^n$ for all i, we call the set $Supp(f) := \{t_1, \ldots, t_m\}$ support of f. Then the leading term of f is defined as $LT_\sigma(f) := t_k = max_\sigma(Supp(f))$ and the leading coefficient is $LC_\sigma(f) := c_k$.

Let $\mathcal{F} := \{f_1, \ldots, f_s\} \in \mathbb{K}[x_1, \ldots, x_n] \setminus \{0\}$ be a set of polynomials. Then $I := \langle \mathcal{F} \rangle$ is the ideal generated by $\mathcal{F}$. The elements of the set $\mathbb{B} := \{(i, j) \mid 1 \leq i < j < s\}$ are called critical pairs. For each critical pair exists an S-polynomial $S_{i,j} := \frac{1}{LC_\sigma(f_i)} \cdot t_{i,j} \cdot f_i - \frac{1}{LC_\sigma(f_j)} \cdot t_{j,i} \cdot f_j$ with $t_{i,j} := \frac{lcm(LT_\sigma(f_i), LT_\sigma(f_j))}{LT_\sigma(f_i)}$ and $t_{j,i} := \frac{lcm(LT_\sigma(f_j), LT_\sigma(f_i))}{LT_\sigma(f_j)}$.

The set $\mathcal{F}$ is a σ-Gröbner basis of I if for all critical pairs the S-polynomials can be reduced to zero by the elements of $\mathcal{F}$ with respect to σ.

If not stated otherwise we will use the degree reverse lexicographic termordering (`DegRevLex`) and the finite field with 32003 elements ($\mathbb{F}_{32003}$) for examples and benchmarks.

3 Preliminaries

The F4 algorithm was introduced by Faugère in 1999 [3]. By using the special structure of the matrix and column-based parallelism Faugère and Lachartre introduced a parallelization improving the performance remarkable [4]. We have presented another modification [13] for F4 speeding up the computation by row-based parallelization.

In the following we give a quick summary about Gröbner bases computation and the mentioned developments. For a given set of polynomials $\mathcal{F} := \{f_1, \ldots, f_s\} \in \mathbb{K}[x_1, \ldots, x_n] \setminus \{0\}$ generating an ideal $I = \langle \mathcal{F} \rangle$ a σ-Gröbner basis $\mathcal{G} = \{g_1, \ldots, g_t\} \in \mathbb{K}[x_1, \ldots, x_n]$ can be computed using the F4 algorithm by the following four steps:

1. Create and update the set of critical pairs $\mathbb{B}$ using Gebauer's and Möller's `UPDATE` function [6]. This function guarantees that the leading terms $t_{i,j} \cdot f_i = t_{j,i} \cdot f_j$ of the minuend and subtrahend of the S-polynomial is unique among the critical pairs.

2. Select a subset of all critical pairs for reduction using the normal or the sugar cube strategy [7]. At this point the details are not relevant. For our benchmarks we chose the sugar cube strategy. Although we have to mention that our implementation supports the normal strategy and for some of the polynomial systems we used as example this strategy might perform better.

3. Symbolic preprocess the selected pairs to obtain a coefficient matrix representing the S-polynomials. The matrix is additionally filled with rows representing reduction polynomials for each term which can be reduced using elements of the partial computed Gröbner basis $\mathcal{G}$. The `SIMPLIFY` method can be applied to reuse results of previous reductions.

4. Reduce the generated matrix until row-echelon form. This is equivalent to top-reducing the S-polynomials. All non-zero rows of this matrix having leading terms which are not divisible by any element of $\mathcal{G}$ are inserted into $\mathcal{G}$. As long as new elements are found the algorithm continues with step 1.

The most time-consuming step is the reduction of the coefficient matrix. The mentioned parallelizations of Lachartre and Faugère as well as our own approach concentrate on speeding up this step. We want to show how to use these improvements also for distributed parallel architectures, therefore we have to review them.

Starting after symbolic preprocessing we obtain a coefficient matrix M which has to be reduced to a row-echelon form. M has n rows and m columns with $m \geq n$ and can be decomposed in submatrices:

1. Submatrices S_1 and S_2 arise from the subtrahends and minuends of the S-polynomials. Since the `UPDATE` function guarantees that all polynomials have distinct leading terms the representing rows have unique pivot columns. Therefore S_1 and S_2 are upper-triangular.
2. Submatrix R represents the reduction polynomials. They were computed during symbolic preprocessing. For each reducible term there is only one reduction polynomial. Because of that R is upper-triangular.
3. Finally these submatrices can be split up in pivot and non-pivot columns. The pivots of R and S_1 are forming the submatrix A and the non-pivots the submatrix B. The submatrix S_2 is split into C containing its pivots and D containing the non-pivot columns. This notation was introduced in [4].

The following example demonstrates the construction of a matrix occurring during Gröbner bases computations.

Example 1. *Let $K := \mathbb{K}_{32003}$, $P := K[x_1, x_2, x_3]$ and let the polynomials $g_1 = x_1^2 + x_2^2$, $g_2 = x_1 \cdot x_2 + x_2^2 + x_2 \cdot x_3$ and $g_3 = x_2^2 + x_3^2 + x_3$ form the ideal $I := \langle g_1, g_2, g_3 \rangle$ for which a Gröbner basis with respect to `DegRevLex` should be computed. Using `UPDATE` we get the critical pairs $(2,3)$ and $(1,2)$. They have the same sugar degree $sugar(g_2, g_3) = 3 = sugar(g_1, g_2)$. So we reduce the following polynomials:*

$$\begin{aligned}
f_{1,2} &= (x_1^2 + x_2^2) \cdot x_2 = x_1^2 \cdot x_2 + x_2^3 \\
f_{2,1} &= (x_1 \cdot x_2 + x_2^2 + x_2 \cdot x_3) \cdot x_1 = x_1^2 \cdot x_2 + x_1 \cdot x_2^2 + x_1 \cdot x_2 \cdot x_3 \\
f_{2,3} &= (x_1 \cdot x_2 + x_2^2 + x_2 \cdot x_3) \cdot x_2 = x_1 \cdot x_2^2 + x_2^3 + x_2^2 \cdot x_3 \\
f_{3,2} &= (x_2^2 + x_3^2 + x_3) \cdot x_1 = x_1 \cdot x_2^2 + x_1 \cdot x_3^2 + x_1 \cdot x_3
\end{aligned}$$

and as set of reduction polynomials we obtain:

$$\begin{aligned}
r_1 &= g_3 \cdot x_2 = x_2^3 + x_2 \cdot x_3^2 + x_2 \cdot x_3 \\
r_2 &= g_2 \cdot x_3 = x_1 \cdot x_2 \cdot x_3 + x_2^2 \cdot x_3 + x_2 \cdot x_3^2 \\
r_3 &= g_3 \cdot x_3 = x_2^2 \cdot x_3 + x_3^3 + x_3^2
\end{aligned}$$

At the end we get the following matrix M:

$$\begin{array}{c}
f_{1,2} \\ f_{2,3} \\ r_1 \\ r_2 \\ r_3 \\ f_{2,1} \\ f_{3,2}
\end{array}
\left(\begin{array}{ccccc|cccccc}
\mathbf{1} & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \mathbf{1} & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \mathbf{1} & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & \mathbf{1} & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 1 & 0 & 0 & 1 \\
\hline
1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0
\end{array}\right)$$

One can easily see the mentioned structure of the matrix. The upper left submatrix is A and its pivots are highlighted. Below is C and the non-pivot columns are forming B and D.

Using this decomposition the (full) reduction of the matrix can be computed using the following algorithm:

1. Primary reduction: Reduce B to $A^{-1} \cdot B$.
2. Secondary reduction: Reduce D to $D - C \cdot A^{-1} \cdot B$.
3. Final reduction: Reduce D to its reduced row-echelon form using Gaussian elimination.
4. Optional reduction: Reduce B using the pivots of D.

Especially the primary reduction can be efficiently parallelized since the matrix is in upper triangular form and so all pivots are known. The primary and secondary reduction can be computed in parallel because a reduced row of B can be applied independently on rows of D.

After the final reduction all non-zero rows of D are elements of the Gröbner basis and can be used to compute further critical pairs. Matrix (AB) does not contain any required elements because by construction the pivots of A are equivalent to leading terms which are already contained in the Gröbner basis. Hence the fourth step is optional and only required if `SIMPLIFY` is used.

There are two different approaches to achieve parallelization: Column-based as suggested by Faugère and Lachatre and row-based as suggested by us. Actually these two are not mutually exclusive and we will use this property to advance the parallelism on architectures with distributed memory. In the following we will use the row-based approach for shared memory parallelism but one can easily see that all modifications are also applicable to the column-based approach of Faugère and Lachatre.

4 Matrix Distribution

In the case of the matrix reduction during Gröbner basis computation we have to consider how to distribute the coefficient matrices with least duplication and communication. Like any other parallel and distributed algorithm for transforming matrices into row echelon form we can choose between row- and column-based distribution or a combination of both.

If row-based or combined distribution is chosen the algorithm scatters rows or blocks among the computation nodes and these need to send rows from one to each other if they are required for further reductions. To reduce the communication overhead this needs to use a strategy which decides how the matrix is distributed. Since we use rows-based parallelization for the shared memory level and since the column-based distribution does play well with the special properties of Gröbner basis computations we didn't analyse the effectiveness of the row-based or combined approach.

If column-based distribution is chosen the pivot matrices A and C have to be send to all nodes and the non-pivot matrices B and D can be distributed without duplication. Since all pivots are preset for primary and secondary reduction this approach does only require communication before and after all reductions are done.

Furthermore the matrices are sparse and by using appropriate matrix formats the communication overhead can be reduced even more. We chose to store the off-diagonal elements of A and C in a coordinate list, i.e. there is one list storing tuples of row, column and value. This format allows reordering in parallel executable sets to improve the row-based parallelization as presented in [13]. For B and D a modified list of lists format is used storing pairs of row index and value for each column. This allows to distribute columns independently. As suggested in [8, Chapter 6] the columns are distributed round-robin to optimize load balancing. Afterwards each node can use row-based parallelism on the local shared memory. The following shows the distribution of the matrix constructed in example 1:

Example 2. *Using two parallel nodes the matrix M of the previous example is decomposed into one coordinate list and two lists of lists. Afterwards the pivots are sent to both nodes and the two sets of non pivot columns are distributed among the nodes. Finally the two nodes are holding the following data:*

Node #1

(A C)		
(4,1,1)	(4,3,1)	(2,0,1)
(2,1,1)	(3,5,1)	
(0,5,1)		
(1,5,1)		
(1,6,1)		

(B D)	
5	(6,1)
7	(4,1)
9	(4,3)

Node #2

(A C)		
(4,1,1)	(4,3,1)	(2,0,1)
(2,1,1)	(3,5,1)	
(0,5,1)		
(1,5,1)		
(1,6,1)		

(B D)		
6	(2,1)	(3,1)
8	(6,1)	-
10	(4,1)	-

Table 1 shows that the final reduction can be computed on a single node because the submatrix D consumes only a tiny fraction of the whole matrix. One can easily check that this is true in general: the submatrices S_1 and S_2 have s rows each and the submatrix R has r rows. There are p pivot columns and n non-pivot columns. Hence the coefficient matrix has $2 \cdot s + r$ rows and $p + n$ columns. After primary and secondary reduction the final reduction only requires the submatrix D which has s remaining rows and n columns. Therefore all reduced columns can be gathered on one node which executes the Gaussian elimination on D.

Polynomial System	Sugar degree	primary matrix	final matrix	ratio (%)
Gametwo 7 [10]	17	7731 x 5818	2671x2182	13
Cyclic 8 [1]	14	3819 x 4244	612x1283	4.8
Cyclic 9	16	70251 x 75040	4014x10605	0.8
Katsura 12 [9]	9	12141 x 11490	2064x2155	3.2
Katsura 13	9	26063 x 23531	4962x4346	3.5
Eco 12 [12]	9	12547 x 10937	2394x1269	2.2
Eco 13	10	25707 x 24686	2883x2310	1.0
F966 [5]	9	60274 x 63503	4823x9058	1.1

Table 1: Ratio of primary matrix and final matrix during reduction for selected problems.

The optional reduction is an obstacle for the effectiveness of the algorithm since it requires that the reduced matrix D has to be distributed again to be applied on B. Because of that we examine if the optional reduction is useful at all. Even more we put to test, if the `SIMPLIFY` method and distributed computation play well together.

5 Simplifying the F4 Algorithm

Removing the optional reduction may scale down the required communication overhead and the reduction time. Even more, if `SIMPLIFY` isn't used during symbolic preprocessing, the distributed matrix has not to be gathered completely since the columns of the reduced matrix B are not required after primary and secondary reduction.

Algorithm 1: SIMPLIFY

```
Input: $t \in \mathbb{T}^n$ a term, $f \in \mathbb{K}[x_1, \ldots, x_n]$ a polynomial, $\mathcal{F} \in (F_k)_{k=1,\ldots,d-1)}$, where $F_k$ is
       finite subset of $\mathbb{K}[x_1, \ldots, x_n]$
Output: a non evaluated product, i.e. an element of $\mathbb{T}^n \times \mathbb{K}[x_1, \ldots, x_n]$
1 foreach $u \in$ list of divisors of $t$ do
2     if $\exists j (1 \leq j < d)$ such that $(u \cdot f) \in F_j$ then
3         $\tilde{F_j}$ is the row echelon form of $F_j$ w.r.t. $\sigma$ ;
4         there exists a (unique) $p \in \tilde{F_j}^+$ such that $\mathrm{LT}_\sigma(p) = \mathrm{LT}_\sigma(u \cdot f)$ ;
5         if $u \neq t$ then return SIMPLIFY($\frac{t}{u}, p, \mathcal{F}$) else return $(1, p)$;
6     end
7 end
```

The SIMPLIFY method does not guarantee that a reduction is improved by reusing previous results. Actually SIMPLIFY proves it effectiveness mostly by application. Hence we require a best possible implementation of the method. Algorithm 1 is a repetition of the original SIMPLIFY [3]. It leaves some details open and there are some possibilities for optimization:

- Searching through all divisors of t is time consuming regarding the fact that only some divisors of u will satisfy the condition $(u, f) \in F_j$ for $j \in (1 \leq j < d)$.
- To check if there exists a j with (u, f) in F_j it is important how F_j is stored. A naive implementation will require to loop over all rows of F_j until the condition is satisfied.
- This algorithm does not check if there is another $j' \neq j$ satisfying the condition and provides a better replacement.

To address these issues we propose the following modifications:

- All rows of $\tilde{F}_1 \ldots, \tilde{F}_{d-1}$ are stored in a two-dimensional map. The first key is a polynomial f and the second is a term u. This allows to update (u, f) if a *better* replacement is found. By that only one value per pair is stored. This will decrease the required memory.
- The first map should provide a fast random access to the value of key f and the second map should have an ordering to allow a decreasing iteration of the possible terms. In this way the search space for possible replacements is reduced to the multiples of f only.
- The value of a map element is the reduced form of $f \cdot u$.

We call this data structure SimplifyDB. We can provide an improved version of SIMPLIFY as shown by algorithm 2. Additionally we have to introduce a function to update the SimplifyDB. Therefore after reduction we execute algorithm 3 for each row of $\tilde{F_j}^+$. The insertion step allows us to compare candidates for the SimplifyDB. In our implementation for two candidates the *better* is the one which has less elements in its support. This is equivalent to the first weighted length function $wlen(p) = \#Supp(p)$ in [2]. At this point we didn't try any other suggested weighted length function.

SIMPLIFY can be improved even more. During symbolic preprocessing it is called twice: once for the selected critical pair (m, f) with $m \in \mathbb{T}^n$ and $f \in \mathcal{G}$ and once to create reduction

Algorithm 2: `SIMPLIFY` with `SimplifyDB`

```
Input: $t \in \mathbb{T}^n$ a term, $f \in \mathbb{K}[x_1, \ldots, x_n]$ a polynomial, SimplifyDB
Output: a non evaluated product, i.e. an element of $\mathbb{T}^n \times \mathbb{K}[x_1, \ldots, x_n]$
1 foreach $u \leq t \in$ SimplifyDB$[f]$ do
2     if $u$ divides $t$ then
3         $p =$ SimplifyDB$[f][u]$ ;
4         if $u \neq t$ then return SIMPLIFY($\frac{t}{u}$, $p$, SimplifyDB) else return $(1,p)$ ;
5     end
6 end
```

Algorithm 3: Insertion into `SimplifyDB`

```
Input: $t \in \mathbb{T}^n$ a term, $f, p \in \mathbb{K}[x_1, \ldots, x_n]$ polynomials, SimplifyDB
Output: SimplifyDB
1 if $\exists u \in$ SimplifyDB[f] : $t = u$ then
2     Replace SimplifyDB$[f][u]$ with $p$ if it is better ;
3 else
4     SimplifyDB$[f][t] = p$ ;
5 end
```

polynomials $m' \cdot f$ with $m' \in \mathbb{T}^n$ and $f' \in \mathcal{G}$. As one can easily see the second input parameter is always an element of $\mathcal{G}$. So it stands to reason to restrict `SIMPLIFY`:

$$\texttt{SIMPLIFY}(t, i), \text{ where } i \text{ is the index of } g_i \in \mathcal{G}$$

Still there is the recursive call of `SIMPLIFY` taking a polynomial p as parameter which might not be an element of $\mathcal{G}$. Recall that p was found in the `SimplifyDB` using f and a term $u = \frac{t}{z}$, with $z \in \mathbb{T}^n$. With the knowledge, that $f = g_i \in \mathcal{G}$ we can write $p = f \cdot u - r = g_i \cdot u - r$, where $r \in \mathbb{K}[x_1, \ldots, x_n]$ is the sum of all reductions applied on f during a previous reduction step. The recursive call is looking for another polynomial p' which simplifies $z \cdot p = z \cdot (g_i \cdot u - r)$. If such a polynomial p' exists we can write $p' = z \cdot (g_i \cdot u - r) - r' = z \cdot u \cdot g_i - (z \cdot r - r')$. Hence if the `SimplifyDB` stores p' at index $(g_i, z \cdot u)$ instead of (p, z) the recursion is obsolete and `SIMPLIFY` can be replaced with algorithm 4 being recursion free.

Algorithm 4: New `SIMPLIFY` algorithm

```
Input: $t \in \mathbb{T}^n$ a term, $i$ is the index of $g_i \in \mathcal{G}$, SimplifyDB
Output: a non evaluated product, i.e. an element of $\mathbb{T}^n \times \mathbb{K}[x_1, \ldots, x_n]$
1 foreach $u \leq t \in$ SimplifyDB[f] do
2     if $u$ divides $t$ then return ($\frac{t}{u}$, SimplifyDB$[f][u]$) ;
3 end
```

Before benchmarking different variants of the distributed versions of the algorithm a first look at timings using only shared memory will thin out the number of combinations. For the following computations we used a system with 48 AMD Opteron™ 6172 cores having 64 gigabyte of main memory. The implementation of the parallelization for the shared memory is presented in [13] and in its latest version it is realized using Intel® Threading Building Blocks (TBB) and Open Multi-Processing (OpenMP) for parallelization.

Table 2 shows that the optional reduction is mostly useless and can be left out during Gröbner basis computation. Table 3 shows that our new recursion-free `SIMPLIFY` is also in practice faster and even more memory efficient.

Finally `SIMPLIFY` is improving performance by increasing memory usage and so for solving larger polynomial systems it might be possible to find a solution within the bounds of available memory by disabling `SIMPLIFY`. Consequently we will compare computations with and without the improved `SIMPLIFY` for the distributed parallel implementation in the following section.

Polynomial systems	without optional reduction		with optional reduction	
	max. matrix size	runtime	max. matrix size	runtime
Gametwo 7	7808 x 5895	28.46	7731 x 5818	29.85
Cyclic 8	3841 x 4295	11.18	3819 x 4244	12.15
Cyclic 9	70292 x 75080	606.7	70251 x 75040	628.3
Katsura 12	12142 x 11491	34.49	12141 x 11490	36.66
Katsura 13	26063 x 23531	152.2	26063 x 23531	169.2
Eco 12	12575 x 11349	25.22	12547 x 10937	28.36
Eco 13	25707 x 24686	91.76	25707 x 24686	103.7
F966	60898 x 63942	98.32	60274 x 63503	118.5

Table 2: Comparison of computations with and without optional reduction. The new `SIMPLIFY` algorithm was used in both cases.

Polynomial	with new `SIMPLIFY`		with original `SIMPLIFY`		without `SIMPLIFY`	
systems	runtime (s)	memory	runtime	memory	runtime	memory
Gametwo 7	**28.46**	459 MB	28.82	721 MB	29.97	262 MB
Cyclic 8	**11.18**	262 MB	14.43	393 MB	12.28	131 MB
Cyclic 9	606.7	15.6 GB	649.2	24.0 GB	**542.9**	2.10 GB
Katsura 12	**34.49**	655 MB	39.28	1.57 GB	43.74	328 MB
Katsura 13	**152.2**	2.56 GB	187.2	7.27 GB	207.1	917 MB
Eco 12	**25.22**	328 MB	38.75	1.38 GB	50.87	262 MB
Eco 13	**91.76**	1.14 GB	136.4	5.77 GB	218.2	852 MB
F966	**98.32**	2.56 GB	218.2	11.3 GB	212.5	1.25 GB

Table 3: Comparison of original and new `SIMPLIFY`

6 Benchmarks

For the following benchmarks of our distributed implementation we used a cluster of up to eight systems having 16 Intel® Xeon® E5-2670 cores with hyperthreading and 64 GB of RAM allowing us using up to 256 parallel threads. For data distribution we use the Message Passing Interface (MPI) and the Boost.MPI wrapper.

Table 4 compares once again runtimes for computations with and without `SIMPLIFY` using one, two or four nodes. For all polynomial input systems the computation time goes up and the distribution has no positive effect. However we accomplished to speed-up the reduction time. This is not an inconsistency. The overall runtime goes up due to the communication overhead and the possibility to improve the performance is limited to decreasing the reduction time. Hence the problem is the *small* size of the given polynomial systems. Figure 1 illustrates this issue for Cyclic 9.

Table 5 adds further and more difficult examples. Especially for Katsura 15 and Cyclic 10 the distributed parallelization takes effect providing a speed-up of 2.3 and 2.7 for the overall computation time and 5.7 and 5.0 for the reduction using four respectively eight nodes. Excluding the Eco 14 system the computation is most effective without `SIMPLIFY` and in particular for Cyclic 10 computations were not even possible with it due to memory overflows.

Eco 14 is an counterexample because it is not possible to decrease the computation time using a distributed system. This is caused by the effectiveness of `SIMPLIFY` for the problem instance.

The crashed computations for Katsura 15 and for Cyclic 10 without `SIMPLIFY` with two nodes are caused by a not yet solved overflow in the MPI implementation if more than 512 MB have to be transfered in one operation. This does not happen with four or eight nodes because the matrix size for each node decreases by factor two respectively four.

At the end no distributed computation with `SIMPLIFY` is faster and hence it should be removed in further developments of distributed parallel algorithms for Gröbner bases computations.

Polynomial systems	with `SIMPLIFY`			without `SIMPLIFY`		
# nodes	1	2	4	1	2	4
Gametwo 7	29.85	34.55	56.03	**28.48**	38.53	28.88
	(5.549)	(6.709)	(6.010)	(6.630)	(4.607)	(**3.427**)
Cyclic 8	7.798	11.12	9.520	**7.497**	14.06	20.78
	(1.327)	(1.707)	(1.953)	(1.332)	(1.473)	(**1.168**)
Cyclic 9	506.0	518.2	494.7	**330.6**	361.7	427.6
	(222.2)	(195.2)	(138.8)	(149.9)	(122.4)	(**73.73**)
Katsura 12	42.00	46.36	55.23	**28.59**	39.97	66.85
	(6.083)	(8.220)	(7.762)	(6.699)	(4.535)	(**3.362**)
Katsura 13	187.4	190.0	186.4	**139.6**	158.8	176.0
	(40.46)	(46.61)	(36.82)	(50.03)	(31.07)	(**18.15**)
Eco 12	**21.35**	29.40	41.49	24.98	41.08	88.02
	(**2.564**)	(3.760)	(3.953)	(6.458)	(3.664)	(3.097)
Eco 13	**91.78**	106.5	120.9	111.4	157.5	212.6
	(**12.10**)	(16.32)	(14.62)	(38.65)	(23.10)	(14.67)
F966	**91.88**	106.2	132.1	93.60	139.2	185.1
	(20.86)	(26.29)	(23.91)	(38.59)	(26.32)	(**18.70**)

Table 4: Distributed computation (and reduction) times for 1,2 or 4 nodes

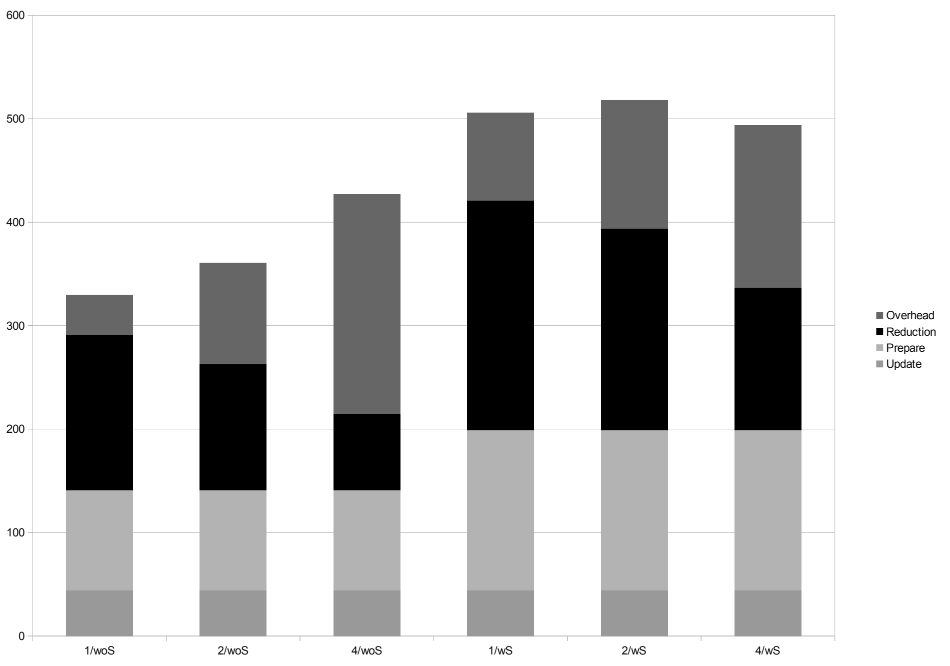

Figure 1: Computation time in detail for Cyclic 9 with and without `SIMPLIFY`

Poly. sys.	with `SIMPLIFY`				without `SIMPLIFY`			
# nodes	1	2	4	8	1	2	4	8
Katsura 14	957.5	831.0	748.9	744.8	775.6	**659.9**	675.4	991
	(313.9)	(263.4)	(201.2)	169.1	(398.4)	(224.4)	(135.1)	(**86**)
Katsura 15	5719	*crashed*	3440	3252	5615	3857	**3252**	3372
	(2571)	(-)	1272	1008	(3781)	(1960)	(1094)	(**666**)
Cyclic 10	*crashed*	*crashed*	*crashed*	*crashed*	30400	*crashed*	13270	**11280**
	(-)	(-)	(-)	(-)	(25240)	(-)	(7862)	(**5010**)
Eco 14	**509.8**	519.4	552.6	626	800.5	787.0	996.7	1660
	(110.6)	(117.7)	(98.02)	(89.38)	(411.3)	(224.0)	(127.7)	(**78.24**)

Table 5: Distributed computation (and reduction) times for larger input systems.

7 Conclusion

We have shown that the parallelization of the F4 algorithm can be expanded form a shared memory system to a cluster of distributed nodes providing a multiple of the computation power of a single system. This was primarily achieved by using column-based parallelism, removing the optional reduction and the `SIMPLIFY` function. It allowed us to solve larger input systems like Katsura 15 or Cyclic 10 almost three times faster using four or respectively eight distributed nodes. Some work is still required to optimize the speed-up and the communication of the nodes has to be improved to make solving of even larger problem instances possible.

References

[1] Göran Björck and Uffe Haagerup. All cyclic p-roots of index 3, found by symmetry-preserving calculations, 2008.

[2] Michael Brickenstein. Slimgb: Gröbner bases with slim polynomials. *Revista Matemática Complutense*, 23(2):453–466, 2010.

[3] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases (F4). *Journal of Pure and Applied Algebra*, 139(1–3):61–88, June 1999.

[4] Jean-Charles Faugère and Sylvain Lachartre. Parallel Gaussian Elimination for Gröbner bases computations in finite fields. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, pages 89–97, New York, USA, July 2010. ACM.

[5] Jean-Charles Faugère, Moreau Francois De Saint Martin, and Fabrice Rouillier. Design of regular nonseparable bidimensional wavelets using Groebner basis techniques. *IEEE Transactions on Signal Processing*, 46(4):845–856, 1998.

[6] Rüdiger Gebauer and H. Michael Möller. On an installation of buchberger's algorithm. *Journal of Symbolic Computation*, 6:275–286, December 1988.

[7] Alessandro Giovini, Teo Mora, Gianfranco Niesi, Lorenzo Robbiano, and Carlo Traverso. "One sugar cube, please" or selection strategies in the buchberger algorithm. In *Proceedings of the 1991 international symposium on Symbolic and algebraic computation*, ISSAC '91, pages 49–54, New York, USA, 1991. ACM.

[8] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 1996.

[9] Shigetoshi Katsura, Wataru Fukuda, Sakari Inawashiro, Nahomi Fujiki, and Rüdiger Gebauer. Distribution of effective field in the ising spin glass of the $\pm j$ model at $t = 0$. *Cell Biochemistry and Biophysics*, 11:309–319, 1987.

[10] David M. Kreps and Robert Wilson. Sequential equilibria. *Econometrica*, 50(4):863–94, July 1982.

[11] Martin Kreuzer and Lorenzo Robbiano. *Computational Commutative Algebra 1.* Computational Commutative Algebra. Springer, 2000.

[12] Alexander Morgan. *Solving polynomial systems using continuation for engineering and scientific problems.* Classics in applied mathematics. SIAM, 2009.

[13] Severin Neumann. Parallel reduction of matrices in Gröbner bases computations. In Vladimir P. Gerdt, Wolfram Koepf, Ernst W. Mayr, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing*, volume 7442 of *Lecture Notes in Computer Science*, pages 260–270. Springer Berlin Heidelberg, 2012.

[14] Kurt Siegl. A parallel factorization tree gröbner basis algorithm. In *Parallel Symbolic Computation PASCO, 1994: Proceedings of the First International Symposium*, River Edge, USA, 1994. World Scientific Publishing Co., Inc.