Parallelizing the Big Prime Berlekamp Algorithm with Distributed Maple*

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
`Wolfgang.Schreiner@risc.uni-linz.ac.at`
`http://www.risc.uni-linz.ac.at`

July 26, 2001

Abstract

We report on a parallel implementation of the Big Prime Berlekamp algorithm with Distributed Maple. This implementation is based on three parallel subalgorithms with different parallelization strategies and dynamic behaviors. The experimental results on a cluster and a multiprocessor system yield superlinear speedups due to a performance problem of the sequential Maple implementation and demonstrate varying efficiencies for the parallel subalgorithms.

*Supported by the FWF projects P11160-TEC (HySaX) und SFB F013/F1304.

1 Introduction

We report on the design and implementation of a parallel variant of the Big Prime Berlekamp algorithm in Distributed Maple. Distributed Maple is a Java-based environment for parallel computer algebra which embeds kernels of the computer algebra system Maple and allows to write parallel programs in the Maple language [9, 10]. The programming model is based on principles from functional and dataflow languages and hides from the programmer low level details such as task scheduling and communication. The system has been used for the implementation of various parallel algorithms from computer algebra and algebraic geometry [14, 13, 15, 12].

The problem of polynomial factorization algorithms has a long tradition. The modern era on this subject starts with the work of Berlekamp [1, 2] on the factorization of univariate polynomials over finite fields to which the more general factorization problems can be reduced. While asymptotically faster algorithms have been developed [17, 16, 7], variants such as the "Big Prime Berlekamp Algorithm" [3] are still standard implementations in systems such as Maple. For a general introduction into the problem domain, see e.g. [4].

Also the development of parallel variants of these algorithms has been undertaken by various researchers. Wang has developed a parallel variant of the Big Prime Berlekamp algorithm for shared memory multiprocessors with a small number of processors [18, 19]. Kaltofen and Loba have developed a parallel version of an asymptotically fast algorithm for factoring sparse high degree polynomials over moderately sized fields; the sequential Maple system uses an external parallel linear system solver running on a computer cluster as a "black box" [6]. [5] sketches a parallel implementation of Shoup's algorithm for polynomial factorization in the MuPad system.

In the following, we develop a parallel variant of the Big Prime Berlekamp algorithm suitable for an implementation in Distributed Maple. Section 2 describes the mathematical preliminaries and outlines the sequential algorithm. Section 3 introduces the parallel algorithm which is a combination of three parallel sub-algorithms. Section 4 sketches the implementation in Distributed Maple and gives benchmark results on a cluster and a massively parallel multiprocessor. Section 5 concludes the paper.

2 The Big Prime Berlekamp Algorithm

We are going to present the mathematical preliminaries of the Berlekamp algorithm [1, 2] in the variation devised by Cantor and Zassenhaus [3]. This "Big Prime Berlekamp Algorithm" is especially efficient for large coefficient fields; for mathematical details, see [4].

2.1 The Basic Algorithm

Let $p > 2$ be prime[1] and $a(x) \in \mathbb{Z}_p[x]$, i.e., a is a univariate polynomial whose coefficients can be represented by the integers $0 \ldots p-1$. Our problem is to find all those polynomials in $\mathbb{Z}_p[x]$ that divide $a(x)$ (i.e., that are *factors* of $a(x)$) and that are only divided by 1 and themselves (i.e., that are *irreducible*). We assume that $a(x)$ is square-free, i.e., does not have repeated factors[2].

Let n be the degree of $a(x)$ and define

$$W := \{v(x) \in \mathbb{Z}_p(x) : \deg(v(x)) < n \wedge v(x)^p = v(x) \bmod a(x)\}.$$

One can show that each $v(x) \in W$ has a factor $v'(x) := v(x)^{\frac{p-1}{2}}$ and that, with probability greater than or equal $4/9$, $a(x)$ and $v'(x)$ have a non-trivial common factor. Furthermore, define the *residue class* $[v(x)]_{a(x)} := \{w(x) : v(x) = w(x) \bmod a(x)\}$; then the set of residue classes

$$W' := \{[v(x)]_{a(x)} : v(x) \in W\}$$

is a vector space over Z_p whose dimension k equals the number of the irreducible factors of $a(x)$.

Based on these results, the Big Prime Berlekamp Algorithms randomly chooses candidates $v(x)$ from W to split factors from $a(x)$ until k such factors have been found (or no more candidate exists, in which case the algorithm fails). The algorithm is depicted in Figure 1.

2.2 The Refined Algorithm

For implementing the Big Prime Berlekamp algorithm, we need a representation of W with efficient algorithms for the various required operations. We define the $n \times n$ matrix Q over $\mathbb{Z}_p$ such that

$$x^{pi} = Q_{i,0} + Q_{i,1}x + \ldots Q_{i,n-1}x^{n-1} \bmod a(x),$$

i.e., row i of Q contains the coefficients of the remainder of x^{pi} and $a(x)$ (Q is indexed as $Q_{1..n,0..n-1}$). One can then show that

$$W = \{\sum_{i=0}^{n-1} v_i x^i : v(Q - I) = \vec{0}\}$$

where I is the $n \times n$ identity matrix and $\vec{0}$ is the zero vector of length n. In other words, W is the solution space of a system of n equations in n unknowns determined by the coefficient matrix $Q - I$.

Furthermore, assume that Q is in triangular idempotent form (i.e., Q has only 0 entries above the main diagonal and only 0 or 1 entries on the main diagonal)

[1] The algorithm can be generalized to the case $p = q^m$ for prime q and $m \geq 1$.

[2] There exists an efficient algorithm for the problem of *square-free factorization* which reduces the general factorization problem to this special case

```
F := BigPrimeBerlekamp(a(x)):
  W := MakeW(a(x))
  k := dim(W')
  F := {a(x)}
  while |F| < k do
    if W = ∅ then fail end
    choose v(x) ∈ W; W := W\{v(x)}
    G := F; F := ∅
    while G ≠ ∅ do
      choose f(x) ∈ G; G := G\{f(x)}
      g(x) := gcd(f(x), v'(x))
      if g(x) = 1 ∨ g(x) = f(x) then
        F := F ∪ {f(x)}
      else
        F := F ∪ {g(x), f(x)/g(x)}
      end
    end
  end
end
```

Figure 1: Big Prime Berlekamp Algorithm

and denote by B all k rows of $I-Q$ that have a non-zero entry (for some $k \in \mathbb{N}$). Then k is the dimension of W' and the set

$$\{\sum_{i=0}^{n-1} v_i x^i : v \in B\}$$

forms a basis for W.

Based on these results, the Big Prime Berlekamp algorithm operates on matrix Q rather than on set W. By a process similar to Gaussian eliminiation, it transforms Q into triangular idempotent form and thus determines a basis B of W (as a sequence of lines) and the dimension k of W' (as the length of this sequence). The enumeration of the elements $v(x) \in W$ is then simulated by a random combination of the elements of basis B. The refined algorithm is depicted in Figure 2 ("random(p)" returns a non-negative pseudo-random number less than p).

A Note on Termination Compared with the basic algorithm, above refinement has the following theoretical problems:

1. Not necessarily all elements of W are enumerated, because not necessarily all possible combinations of basis elements are computed (if the period of the random number generator divides k). The algorithm may therefore not suceed for all inputs where the basic algorithm succeeds.

2. The algorithm omits the "fail" check. Instead of failing, the algorithm runs into an infinite loop (because used candidates $v(x)$ are chosen again).

```
F := BigPrimeBerlekamp(a(x)):
  Q := MakeQ(a(x))
  B := Solve(Q − I)
  k := length(B)
  F := {a(x)}
  while |F| < k do
    v(x) := Σ_{i=0}^{k−1} (random(p) * Σ_{j=0}^{n−1} B(i)_j x^j)
    G := F; F := ∅
    while G ≠ ∅ do
      choose f(x) ∈ G; G := G\{f(x)}
      g(x) := gcd(f(x), v'(x))
      if g(x) = 1 ∨ g(x) = f(x) then
        F := F ∪ {f(x)}
      else
        F := F ∪ {g(x), f(x)/g(x)}
      end
    end
  end
end
```

Figure 2: Big Prime Berlekamp Algorithm (Refined)

The refined algorithm may therefore, however with extremely low probability, run into non-termination. We could easily introduce an upper bound for the number of iterations (the size of W) and thus force termination. However, this is not of practical relevance and neglected in literature [4] as well as in implementation practice. We will therefore also ignore this deficiency although it actually turns the basic algorithm into a semi-algorithm.

2.3 The Subalgorithms

It remains to be shown how to construct the matrix Q from input $a(x)$ and how to compute a basis for the solution space of $Q - I$, i.e., how to implement the functions "MakeQ" and "Solve" of Figure 2.

Let us consider the first question, i.e., how to compute, for every $1 \leq i \leq n$, row i of matrix Q such that

$$x^{pi} = Q_{i,0} + Q_{i,1}x + \ldots Q_{i,n-1}x^{n-1} \bmod a(x).$$

The first row of Q, i.e., the coefficients of x^p, may be efficiently computed by *binary powering* which iteratively determines $x \bmod a(x)$, $x^2 \bmod a(x)$, $x^4 \bmod a(x)$, $\ldots$ (we omit the description). Each further row is computed by multiplying the previous line with x^p. The algorithm is depicted in Figure 3.

The computation of the basis of the null space of matrix $Q - I$ proceeds by transforming the matrix into a triangular idempotent form M such that the solution space is preserved. We then compute $I - M$ and determine the basis of the solution space. The algorithm is depicted in Figure 4 ($M_{*,j}$ denotes column j of M).

Q := MakeQ(a(x)):
  $n := \deg(a(x))$
  $Q_{1,*} := \text{coeffs}(x^p \bmod a(x))$
  for i **from** 2 **to** n **do**
    $Q_{i,*} := \text{coeffs}(x^p * \text{poly}(Q_{i-1,*}, x) \bmod a(x))$
  end
end

Figure 3: Q Matrix Computation

B := Solve(M):
  $n := \dim(M)$
  for i **from** 1 **to** n **do**
    $j := i - 1$; **while** $j < n \wedge M_{i,j} = 0$ **do** $j := j + 1$ **end**
    if $j < n$ **then**
      $M_{*,j} := M_{*,j} * M_{i,j}^{-1} \bmod p$
      $M_{*,j}, M_{*,i-1} := M_{*,i-1}, M_{*,j}$
      for j **from** 0 **to** $n - 1$ **do**
        if $i - 1 \neq j$ **then**
          $M_{*,j} := M_{*,j} - M_{*,i-1} * M_{i,j} \bmod p$
        end
      end
    end
  end
  for i **from** 1 **to** n **do** $M_{i,i-1} := M_{i,i-1} - 1 \bmod p$ **end**
  $B := [\sum_{j=0}^{n-1} M_{i,j} * x^j : \exists j : M_{i,j} \neq 0]$
end

Figure 4: Null Space Basis Computation

F := BigPrimeBerlekampPar(a(x)):
$Q :=$ MakeQPar$(a(x))$
$B :=$ SolvePar$(Q - I)$
$F :=$ SplitPar$(a(x), B)$
end

Figure 5: Big Prime Berlekamp Algorithm (Parallel)

Asymptotic Complexity The cost of determining Q is $O(n^2 \log(q) + n^3)$ operations in field $\mathbb{Z}_p$. Determining the basis of the solution space of $Q - I$ adds another $O(n^3)$ field operations. The total complexity of the splitting step is $O(n^2 \log q \log k)$ field operations. Thus the total asymptotic complexity of the algorithm is $O(n^2 \log q \log k + 2n^3)$ field operations.

3 The Parallel Algorithm

The Big Prime Berlekamp Algorithm operates in three steps of

1. constructing the matrix Q,
2. determining the basis B of the null space of $Q - I$,
3. using B to split the factors of the polynomial.

Our parallel version of the algorithm combines parallel versions of these three steps (see Figure 5) which we are going to describe in the following subsections.

3.1 Parallel Q Matrix Computation

For computing matrix Q in parallel, we apply a strategy that was formerly developed for an implementation of this algorithm in a parallel functional language [8]. We note that, once that $x^p \bmod a(x)$ (i.e., the first matrix row) has been computed, each further $x^{ip} \bmod a(x)$ (i.e., matrix row i, for $2 \leq i \leq n$) can be computed as follows:

- if i is even: $x^{ip} \bmod a(x) = (x^{\frac{i}{2}p} \bmod a(x)) * (x^{\frac{i}{2}p} \bmod a(x))$, i.e., row i can be computed from row $\frac{i}{2}$,
- if i is odd: $x^{ip} \bmod a(x) = (x^{\frac{i-1}{2}p} \bmod a(x)) * (x^{(\frac{i-1}{2}+1)p} \bmod a(x))$, i.e., row i can be computed from rows $\frac{i-1}{2}$ and $\frac{i-1}{2} + 1$.

From this observation, the following tree of data dependencies emerges; each tree level denotes those lines that can be simultaneously computed from the lines in the previous level:

Q := MakeQPar(a(x)):
  $n :=$ degree($a(x)$)
  $r :=$ [data() : $1 \leq i \leq n$]
  put(r_1, x^p mod $a(x)$)
  for i **from** 2 **to** n
    start(MakeQTask, r, i, $a(x)$)
  end
  for i **from** 1 **to** n
    $Q_{i,*} :=$ coeffs(**get**(r_i))
  end
end

MakeQTask(r, i, a(x)):
  if i is even **then**
    $p1 :=$ **get**($r_{i/2}$); $p2 := p1$
  else
    $p1 :=$ **get**($r_{(i-1)/2}$); $p2 :=$ **get**($r_{(i-1)/2+1}$)
  end
  put(r_i, $p1 * p2$ mod $a(x)$)
end

Figure 6: Q Matrix Computation (Parallel)

```
1
↓                   ↘
2                    3
↓      ↘             ↓      ↘
4        5           6        7
↓ ↘      ↓ ↘         ↓ ↘      ↓ ↘
8   9    10  11      12  13   14  15
.............................................
```

The parallel algorithm based on this dataflow pattern is depicted in Figure 6. In this algorithm, a vector of shared data is created by calls of procedure "data". These data are initially in the state empty. A call of "get" on such an empty object blocks the current task until its value has been made available by a call of "put" issued by another task. The actual implementation differs from this algorithm in that a task does not just compute a single matrix line but a block of consecutive lines. The number of lines per block is adjusted such that the number of tasks is a small multiple of the number of processors; thus we have a sufficient amount of parallelism but the system is not flooded with tasks.

3.2 Parallel Null Space Basis Computation

For computing the triangular form of matrix M in parallel, we distribute the rows of M cyclically among all np processes available. In iteration j of the elimination process, the process in charge of row i broadcasts this row to the other processes. Each process then independently performs the necessary elimination steps on its set of rows and proceeds with the next iteration. The algorithm is depicted in Figure 7 (all column operations $M_{*,j}$ in "SolveParTask" now refer only to the rows of the matrix stored in the process).

3.3 Parallel Factor Splitting

The parallel version of the third step proceeds by decomposing in parallel all factors $a_i(x)$ determined up to now and also by, for each $a_i(x)$, speculatively investigating multiple candidates that may have a common factor with $a_i(x)$. We apply, however, speculative parallelism only, if processors would idle otherwise, i.e., when there are fewer factors than processors (especially in the early phases of the decomposition process).

The algorithm maintains a set of factors and, for each factor, a set of tasks that speculatively investigate the decomposition of this factor (using different candidates). When one of these tasks is successful, the other tasks are aborted, the old factor is removed from the set and the new factors are added. Whenever the total number of tasks falls below the number of processors, a new candidate is chosen and, for each factor, a new task is started which investigates the decomposition of the factor by this candidate. This process terminates when k factors have been found. The algorithm is depicted in Figure 8.

4 Experimental Results

We have implemented the parallel algorithm in Distributed Maple based on the function `mod/Berlekamp` of the Maple standard library; we use this sequential function as the basis of our speedup comparisons. The structure of the implementation essentially corresponds to the pseudo-code given in the previous sections where the parallel constructs **start**, **data**, **put**, **get**, **select**, and **stop** are implemented by the Distributed Maple operations `dist[process]`, `dist[data]`, `dist[put]`, `dist[get]`, `dist[select]` and `dist[delete]`.

Since the matrix Q can become very big, we put the rows individually into shared data and only pass the corresponding data handles to tasks. Tasks are created by `dist[process]`, which exclusively allocates a kernel for each task such that the use of shared data cannot cause deadlocks in task scheduling. In the computation of the null space basis of $Q - I$, we explicitly place the tasks to processors using the call `dist[place]` to override the otherwise non-optimal placement of the automated load balancing mechanim.

We have systematically benchmarked the algorithm with four randomly generated dense polynomials with various degrees n and sizes of prime p. The benchmarks have been executed in two environments:

- A cluster of 16 Linux PCs with Pentium III processors running at various clock speeds, from 450 Mhz to 735 Mhz (average roughly 590 Mhz); the machines were linked by 100Mbit point-to-point switched fast Ethernet.
- A Silicon Graphics Origin 3800 multiprocessor with 32 (out of 128) R12000 processors running at 450 Mhz.

```
Q := SolvePar(M):
  n := dim(M)
  r := [data() : 1 ≤ i ≤ n]
  for k from 1 to np do
    start(MakeQTask, r, k, [M_{i*np+k,*} : 0 ≤ i ≤ ⌊(n − k)/np⌋])
  end
  for i from 1 to n do
    M_{i,*} := get(r_i)
    M_{i,i−1} := M_{i,i−1} − 1 mod p
  end
  B := [∑_{j=0}^{n−1} M_{i,j} * x^j : ∃j : M_{i,j} ≠ 0]
end

MakeQTask(r, k, R)
  for i from 0 to ⌊(n − k)/np⌋ do
    M_{i*np+k,*} := R_i
  end
  for i from 1 to n do
    if i − k mod np = 0 then
      put(r_i, M_{i,*})
    else
      M_{i,*} := get(r_i)
    end
    j := i − 1; while j < n ∧ M_{i,j} = 0 do j := j + 1 end
    if j < n then
      M_{*,j} := M_{*,j} * M_{i,j}^{−1} mod p
      M_{*,j}, M_{*,i−1} := M_{*,i−1}, M_{*,j}
      for J from 0 to n − 1 do
        if i − 1 ≠ j then
          M_{*,j} := M_{*,j} − M_{*,i−1} * M_{i,j} mod p
        end
      end
    end
  end
end
```

Figure 7: Null Space Basis Computation (Parallel)

Q := SplitPar(a(x), B):
  $n := \deg(a(x))$
  $k := \text{length}(B)$
  $F := \{a(x)\}$
  $T := \emptyset$
  $C_{a(x)} := \emptyset$
  while $|F| < k$ **do**
    $v(x) := \sum_{i=0}^{k-1}(\text{random}(p) * \sum_{j=0}^{n-1} B(i)_j x^j)$
    $G := F$
    while $G \neq \emptyset$ **do**
      choose $f(x) \in G;\ G := G \backslash \{f(x)\}$
      $t :=$ **start**$(\text{SplitParTask}, f(x), v(x))$
      $T := T \cup \{t\}$
      $C_{f(x)} := C_{f(x)} \cup \{t\}$
    end
    while $|T| \geq np \wedge |F| < k$ **do**
      $t :=$ **select**$(T);\ T := T \backslash \{t\}$
      for $u \in C_{t_0} \backslash \{t\}$ **do**
        stop(T)
      end
      $T := T \backslash C_{t_0}$
      $C_{t_0} := \emptyset$
      if $|t_1| > 1$ **then**
        $F := F \backslash \{t_0\} \cup t_1$
        for $b(x) \in t_1$ **do**
          $t :=$ **start**$(\text{SplitParTask}, b(x), v(x))$
          $T := T \cup \{t\}$
          $C_{b(x)} := \{t\}$
        end
      end
    end
  end
end

F := SplitParTask(f(x), v(x)):
  $g(x) := \gcd(f(x), v'(x))$
  if $g(x) = 1 \vee g(x) = f(x)$ **then**
    $F := \langle f(x), \{f(x)\} \rangle$
  else
    $F := \langle f(x), \{g(x), f(x)/g(x)\} \rangle$
  end
end

Figure 8: Splitting Factors (Parallel)

Neither cluster nor Origin were exclusively dedicated to the computation but both environments ran in normal multi-user operation during the benchmarks (however usually lightly loaded). In the cluster, sequential execution time was measured on the fastest processor. The execution times and speedup curves of the benchmarks in both environments are depicted in Figures 9 and 10.

The most striking effect in these benchmarks is that the parallelization of the null space basis computation gives drastically superlinear speedups which also results in superlinear speedups of the whole factorization algorithm. Some experiments indicate that this effect is due to a problem with the Maple V.5 implementation of "arrays" respectively "matrices" as hash tables whose keys are the array indices. These tables apparently do not adapt their sizes to the number of entries such that the element access time significantly increases with large matrix dimensions[3]. Since in the parallel implementation of the null space basis computation each process has only a fraction of the matrix, it solves the problem much faster than can be expected from the sequential execution time. To isolate this effect, we have computed a "distilled" speedup which takes as the sequential computation time 1/4 of the parallel computation time with 4 processors. The corresponding diagrams show how the subalgorithm actually scales with increasing number of processors. On the cluster, the distilled speedup gets about 12 with degree 300 and 400 and practically linear with degree 500; on the Origin, the distilled speedup gets up to 12 with degree 300, 16 with degree 400 and degree 20–24 with degree 500.

The other two parallel algorithms scale much less than the null space computation. The parallel Q matrix computation achieves a speedup of about 6 on the cluster and about 10 on the Origin. The parallel factor splitting achieves a speedup of about 3 on the cluster and on the Origin. These results can be explained from the sequential startup and the limited parallelism in the early stages of the Q matrix computation, respectively from the small number of factors of the random polynomials and the small amount of speculation speedup (less than 2, since in almost 50% of the cases a candidate splits a factor). Especially for large p (Examples 1 and 4), this limits the overall speedup to half the speedup of the null space basis computation.

As for the comparison of both execution environments, we see that the cluster scales (for the null space basis computation) well up to 16 processors, while the speedup on the Origin does not improve much beyond 24 processors. Within the same processor range, the Origin seems to scale less well than the cluster which confirms earlier observations with Distributed Maple in these environments [11].

5 Conclusions

Our parallel variant of the Big Prime Berlekamp algorithm combines various parallel subalgorithms with different parallelization strategies and different dynamic behaviors. The parallel computation of the Q matrix is based on a dataflow style of parallelism; the parallel null space basis computation applies

[3]Maple V.6 introduces a distinct datatype "Matrix" which seems to overcome this problem. However, the basic Maple algorithms such as Berlekamp factorization still use the old representation.

Execution Times (s)						
Example	Algorithm	S	4	8	12	16
1 ($n = 300, \log_2 p \approx 128$)	Total	3770	926	668	538	486
	MakeQ	185	82	58	40	34
	Solve	3019	511	231	190	169
	Split	515	281	328	256	231
2 ($n = 400, \log_2 p \approx 64$)	Total	27756	1234	647	555	444
	MakeQ	209	82	55	47	39
	Solve	27305	1033	505	419	316
	Split	220	89	67	68	69
3 ($n = 500, \log_2 p \approx 16$)	Total	13647	2171	1027	771	622
	MakeQ	188	89	60	53	48
	Solve	13417	2057	940	694	551
	Split	27	20	22	19	19
4 ($n = 500, \log_2 p \approx 128$)	Total	39979	5155	2760	2398	2304
	MakeQ	866	367	195	202	141
	Solve	36849	3895	1669	1294	933
	Split	2127	752	754	760	1088

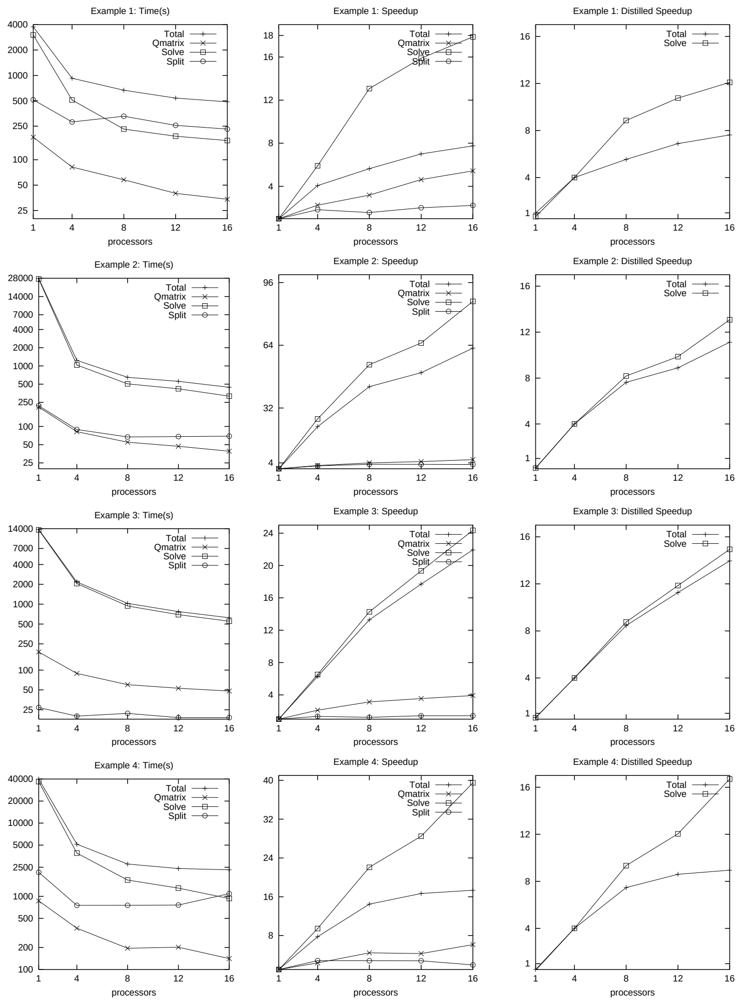

Figure 9: Experimental Results (Linux Cluster)

Execution Times (s)								
Example	Algorithm	S	4	8	12	16	24	32
1 ($d = 300, \log_2 p \approx 128$)	Total	4224	1446	862	1061	967	839	996
	MakeQ	295	114	65	48	41	30	28
	Solve	3021	537	306	250	214	174	187
	Split	828	712	411	681	629	652	701
2 ($d = 400, \log_2 p \approx 64$)	Total	12567	1442	776	607	621	543	512
	MakeQ	337	142	75	54	46	36	37
	Solve	11817	1079	557	418	369	287	297
	Split	359	189	110	103	174	187	146
3 ($d = 500, \log_2 p \approx 16$)	Total	13895	1976	933	676	580	449	403
	MakeQ	323	133	76	63	56	59	60
	Solve	13497	1976	933	676	580	449	403
	Split	45	36	38	43	46	61	77
4 ($d = 500, \log_2 p \approx 128$)	Total	40367	5759	4008	3364	2498	2214	2264
	MakeQ	1381	549	295	212	170	127	126
	Solve	35362	3802	1716	1156	927	667	686
	Split	3407	1187	1778	1779	1184	1201	1231

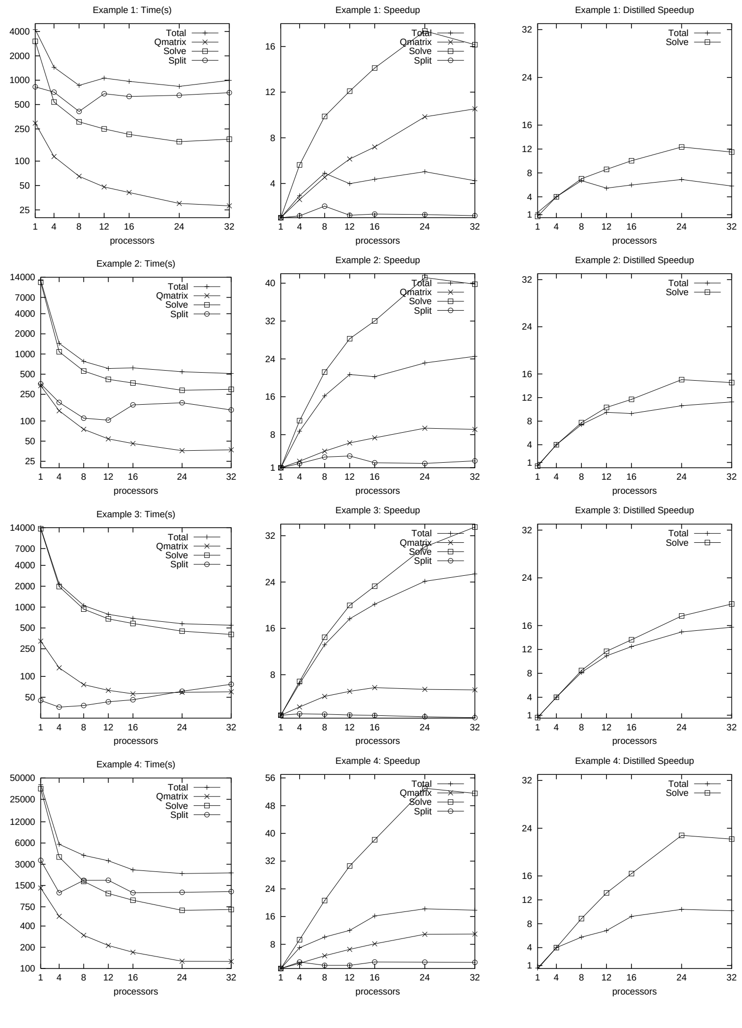

Figure 10: Experimental Results (Origin 3800)

data parallelism by persistent tasks that iteratively broadcast intermediate results to the other tasks; the factor splitting phase applies a combination of data parallelism and speculative parallelism to saturate processors with work.

Due to an unexpected deficiency of Maple's matrix representation which slows down the builtin sequential implementation of the algorithm, the null space basis computation achieves drastically superlinear speedups which also gives superlinear speedups for the whole algorithm. After isolation of this effect, it turns out that this subalgorithm scales very well up to 24 processors, while the other subalgorithms give for algorithmic reasons only moderate speedups (6–10 in one case and 3 in the other case). In total, the parallel algorithm gives reasonable speedups for polynomial degrees greater than 400 and scales well up to 16 processors. The larger the size of prime p becomes, the more the moderate speedups of Q matrix computation and factor splitting play a role; for 128 bit polynomial coefficients the overall speedup is limited to half the speedup of the null space basis computation.

References

[1] E. R. Berlekamp. Factoring Polynomials over Finite Fields. In *Bell System Technical Journal*, volume 46, pages 1853–1859, 1967.

[2] E. R. Berlekamp. Factoring Polynomials over Large Finite Fields. In *Mathematics of Computation*, volume 24, pages 713–735, 1970.

[3] D. Cantor and H. Zassenhaus. A New Algorithm for Factoring Polynomials over Finite Fields. In *Mathematics of Computation*, volume 36, pages 587–592, 1981.

[4] K.O. Geddes, S.R. Czapor, and G. Labahn. *Algorithms for Computer Algebra.* Kluwer Academic Publishers, Boston, Massachusetts, 1992.

[5] Christian Heckler, Torsten Metzner, and Paul Zimmermann. Progress Report on Parallelism in MuPad. In *Poster at the PASCO'97 International Symposium on Parallel Symbolic Computation*, Maui, Hawaii, USA, 1997. http://ww.mupad.de/BIB/pasco97.html.

[6] Erich Kaltofen and Austin Lobo. Factoring High-Degree Polynomials by the Black Box Berlekamp Algorithm. In *ISSAC'94 International Symposium on Symbolic and Algebraic Comutation*, pages 90–98, Oxford, England, July 20–22, 1994. ACM Press, New York.

[7] Erich Kaltofen and Victor Shoup. Fast Polynomial Factorization over High Algebraic Extensions of Finite Fields. In *ISSAC'97 International Symposium on Symbolic and Algebraic Computation*, pages 184–188, Kihei, Maui, Hawaii, USA, July 21–23, 1997. ACM Press, New York.

[8] Wolfgang Schreiner. A Para-Functional Programming Interface for a Parallel Computer Algebra Package. *Journal of Symbolic Computation*, 21:593–614, 1996.

[9] Wolfgang Schreiner. Distributed Maple — User and Reference Manual. Technical Report 98-05, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1998. http://www.risc.uni-linz.ac.at/software/distmaple.

[10] Wolfgang Schreiner. Developing a Distributed System for Algebraic Geometry. In Barry H.V. Topping, editor, *EURO-CM-PAR'99 Third Euro-conference on Parallel and Distributed Computing for Computational Mechanics*, pages 137–146, Weimar, Germany, March 20-25, 1999. Civil-Comp Press, Edinburgh.

[11] Wolfgang Schreiner. Distributed Maple: Parallel Computer Algebra in Networked Environments. Submitted for publication, 2001.

[12] Wolfgang Schreiner. Manager-Worker Parallelism versus Dataflow in a Distributed Computer Algebra System. In *PaCT'2001, Sixth International Conference on Parallel Computing Technologies*, Lecture Notes in Computer Science, Novosibirsk, Russia, September 3–7, 2001. Springer, Berlin.

[13] Wolfgang Schreiner, Christian Mittermaier, and Franz Winkler. Analyzing Algebraic Curves by Cluster Computing. In *Distributed and Parallel Systems – From Instruction Parallelism to Cluster Computing, Proceedings of DAPSYS'2000, 3rd Austrian-Hungarian Workshop on Distributed and Parallel Systems*, pages 175–184, Balatonfüred, Lake Balaton, Hungary, September 10–13, 2000. Kluwer Academic Publishers, Boston.

[14] Wolfgang Schreiner, Christian Mittermaier, and Franz Winkler. On Solving a Problem in Algebraic Geometry by Cluster Computing. In *Euro-Par 2000, 6th International Euro-Par Conference*, volume 1900 of *Lecture Notes in Computer Science*, pages 1196–1200, Munich, Germany, August 29 - September 1, 2000. Springer, Berlin.

[15] Wolfgang Schreiner, Christian Mittermaier, and Franz Winkler. Plotting Algebraic Space Curves by Cluster Computing. In Xiao-Shan Gao and Dongming Wang, editors, *4th Asian Symposium on Computer Mathematics*, pages 49–58, Chiang Mai, Thailand, December 17-21, 2000. World Scientific Publishers, Singapore/River Edge.

[16] Victor Shoup. A New Polynomial Factorization Algorithm and its Implementation. *Journal of Symbolic Computation*, 20(4):363–398, 1995.

[17] Joachim von zur Gathen and Victor Shoup. Computing Frobenius Maps and Factoring Polynomials. *Computational Complexity*, 2:187–224, 1992.

[18] Paul S. Wang. Parallel Univariate Polynomial Factorization on Shared-Memory Multiprocessors. In *ISSAC'90 International Symposium on Symbolic and Algebraic Computation*, pages 145–151, Tokyo, Japan, August 20–24, 1990. ACM Press, New York.

[19] Paul S. Wang. Parallel Polynomial Operations: a Progress Report. In Hoon Hong, editor, *PASCO'94 International Symposium on Parallel Symbolic Computation*, pages 394–404, Hagenberg, Austria, September 26–28, 1994. World Scientific Publishers.