On the Factor Refinement Principle and its Implementation on Multicore Architectures

2012 J. Phys.: Conf. Ser. 385 012015

(http://iopscience.iop.org/1742-6596/385/1/012015)

On the Factor Refinement Principle and its Implementation on Multicore Architectures

Md. Mohsin Ali, Marc Moreno Maza and Yuzhen Xie
University of Western Ontario, London, Ontario, Canada
E-mail: md.ali@anu.edu.au,moreno@csd.uwo.ca,yxie@maplesoft.com

Abstract.
We propose a divide and conquer adaptation of the factor refinement algorithm of Bach, Driscoll and Shallit. For an ideal cache of Z words, with L words per block, the original approach suffers from $O(n^2/L)$ cache misses, meanwhile our adaptation incurs $O(n^2/ZL)$ cache misses only. We have realized a multithreaded implementation of the latter using `Cilk++` targeting multicores. Our code achieves linear speedup on 16 cores for sufficiently large input data.

1. Introduction

The implementation of non-linear polynomial system solvers is a very active research area. It has been stimulated during the past ten years by two main progresses. Firstly, methods for solving such systems have been improved by the use of so-called modular techniques and asymptotically fast polynomial arithmetic. See the landmark textbook [11] for this research area. Secondly, the democratization of supercomputing, thanks to hardware acceleration technologies (multicores, general purpose graphics processing units) creates the opportunity to tackle harder problems.

One central issue in the implementation of polynomial system solvers is the elimination of redundant expressions that frequently occur during intermediate computations, with both numerical methods and computer algebra methods. Factor refinement, also known as coprime factorization, is a popular technique for removing repeated factors within a set of polynomials. Coprime factorization has numerous applications in number theory and polynomial algebra, see the papers [5, 6] and the web site `http://cr.yp.to/coprimes.html` by Bernstein.

Algorithms for coprime factorization have generally been designed with algebraic complexity as the complexity measure to optimize. However, with the evolution of computer architecture, parallelism and data locality are becoming major measures of performance, in addition to the traditional ones, namely serial running time and allocated space.

In this paper, we revisit the factor refinement algorithms of Bach, Driscoll and Shallit [4] based on quadratic arithmetic. We show that their *augment refinement principle* leads to a highly efficient algorithm in terms of parallelism and data locality. Our approach is inspired by the paper "Parallel Computation of the Minimal Elements of a Poset" [8]. Figure 1 illustrates the divide and conquer scheme of our algorithm. Once the original problem has been divided into sufficiently small sub-problems, those sub-problems are solved serially and then their solutions are merged in parallel. Here's the difficult point. As demonstrated in [3], a straightforward implementation of the merge phase can bring a bottleneck in terms of *cache complexity*; see [9, 12]

for this notion. The main contribution of this paper is to overcome this difficulty and exhibit a factor refinement algorithm which is efficient in terms of cache complexity and parallelism.

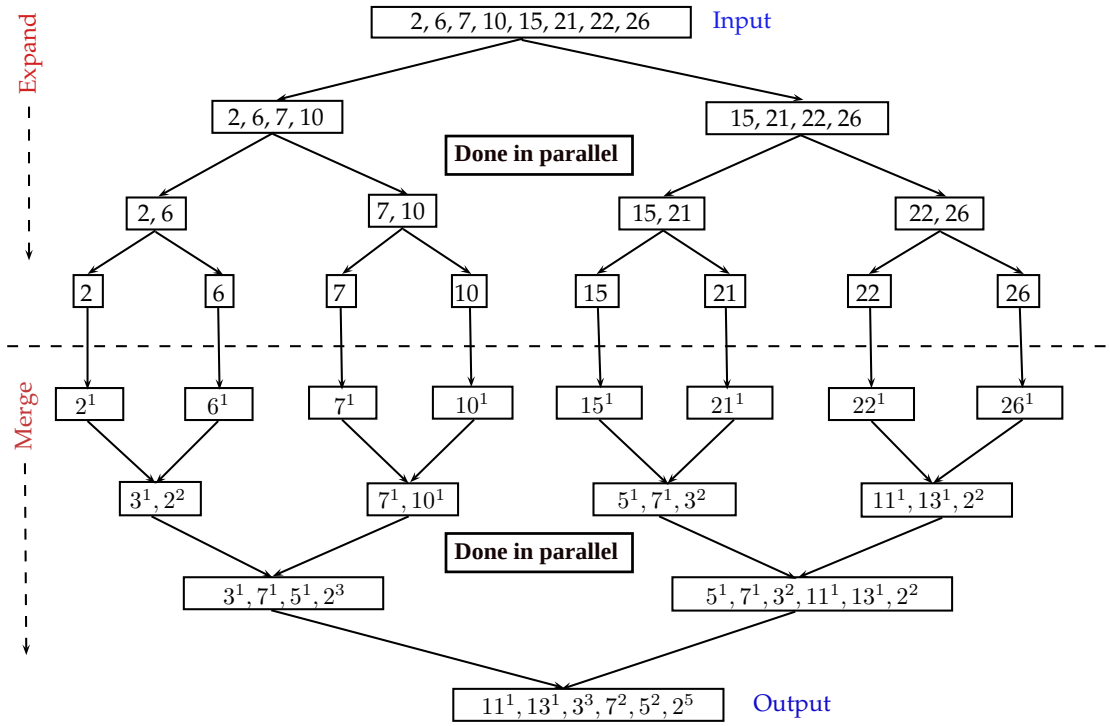

Figure 1. Example of algorithm execution.

On input data of size n, under the condition that each arithmetic operation (integer division and integer GCD computation) has a unit cost, this parallel algorithm features $O(n^2)$ work, $O(Cn)$ span, and thus $O(n/C)$ parallelism, where C is the size threshold below which serial code is run. Other approaches [5, 7] have a better work, but as shown in [3] they are not suitable for implementation on multicore architectures due to high parallelization overheads. This motivates the work reported in this paper.

For an ideal cache of Z words, with L words per cache-line, with C small enough, for any input of size n, the number of cache misses of our algorithm is

$$Q(n) = O(n^2/ZL + n^2/Z^2),$$

under the assumption that each input or output sequence is *packed*, and each sequence item is stored in *one machine word*. Moreover, under the tall cache assumption, the above estimate further simplifies to $Q(n) = O(n^2/ZL)$. All these assumptions hold in our implementation for the case of univariate polynomials over a small prime field. Moreover, our code enforces *balancing of sub-problems* for optimizing the share of computer resources among threads.

Our theoretical results are confirmed by an experimentation, based on a multicore implementation in the `Cilk++` concurrency platform [10, 13]. For problems on integers, we use the GMP library [1] while for problems on polynomials, we use the "Basic Polynomial Algebra Subroutines (BPAS)" library [14].

2. The Factor Refinement Principle

We review below the notions of factor refinement, coprime factorization and GCD-free basis. Let $n_1, \ldots, n_s$ be all integers or all univariate polynomials over a field $\mathbb{K}$. We say that $n_1, \ldots, n_s$ form a *GCD-free basis* whenever $\gcd(n_i, n_j) = 1$ for all $1 \leq i < j \leq s$. Let $m_1, \ldots, m_r$ be

other elements of the same type as the n_i's and let m be the product of the m_j's. Let $e_1, \ldots, e_s$ be positive integers. We say that the pairs $(n_1, e_1), \ldots, (n_s, e_s)$ form a *factor refinement* of $m_1, \ldots, m_r$ if the following conditions hold:

(*i*) $n_1, n_2, \ldots, n_s$ form a GCD-free basis,

(*ii*) for every $1 \leq i \leq r$ there exists non-negative integers $f_1, \ldots, f_s$ such that we have $\prod_{1 \leq j \leq s} n_j^{f_j} = m_i$,

(*iii*) $\prod_{1 \leq i \leq s} n_i^{e_i} = m$.

When this holds, we also say that $(n_1, e_1), \ldots, (n_s, e_s)$ is a *coprime factorization* of m. For instance, $5^1, 6^2, 7^1$ is a refinement of 30 and 42 while $5^1, 6^2, 7^1$ is a coprime factorization of 1260.

Given a partial factorization of an integer m, say $m = m_1 m_2$, one can compute $d = \gcd(m_1, m_2)$ and write

$$m = (m_1/d)(d^2)(m_2/d).$$

If this process is continued until all the factors of m are pairwise coprime, one obtains a coprime factorization of m within $O(\text{size}(m)^3)$ bit operations. In their landmark 1988 paper [4], Bach, Driscoll and Shallit observed that, by keeping track of the pairs (n_j, n_k) in an *ordered pair list* such that only elements adjacent in the list can have a nontrivial GCD, one computes a coprime factorization of m within $O(\text{size}(m)^2)$ bit operations. In their proof, they simply rely on plain (or quadratic) arithmetic. Using asymptotically fast algorithms for the case of univariate polynomials, D. Bernstein [5] on one hand and, Dahan, Moreno Maza, Schost, Xie [7] on another, have obtained an estimate of $O(d \log_2^4(d) \log_2(\log_2(d)))$, where d is the sum of the degrees of the input polynomials. We note that none of these previous works has considered the question of cache complexity for factor refinement or, equivalently for coprime factorization or GCD-free basis computation.

In the sequel of this paper, we focus on the case of univariate polynomials, for which an additional notion is essential. A univariate polynomial f with coefficients in a field $\mathbb{K}$ is said *squarefree* if for every non-constant polynomial $g \in \mathbb{K}[x]$, the polynomial g^2 does not divide f. Note that it is possible to compute a factor refinement $(f_1, e_1), \ldots, (f_s, e_s)$ of f such that the $f_1, \ldots, f_s$ are pairwise coprime squarefree polynomials and $e_i < e_j$ holds for all $1 \leq i < j \leq p$. This factor refinement is called a *squarefree factorization* of f; see [11, 15] for details.

3. A Divide and Conquer Approach

Algorithm 5 presented in this section is an efficient algorithmic solution in terms of data locality and parallelism for coprime factorization of integers or univariate polynomials. For an ideal cache of Z words, with L words per cache-line, for an input data of size of n, under the assumptions stated in the introduction, Algorithm 5 incurs $O(n^2/ZL)$ cache misses. In contrast, it is not hard to check, see [3], that the original algorithm by Bach, Driscoll and Shallit (based on ordered pair lists) suffers from $O(n^2/L)$ cache misses. Modern L1 caches typically match $L = 2^9$ and $Z = 2^{16}$ while for L3 caches, one can have $Z = 2^{26}$ or more.

This substantial gain of Algorithm 5 is obtained thanks to a cache efficient procedure (Algorithm 4) for merging two coprime factorizations. Before analyzing Algorithm 5, we describe its specifications and those of its subroutines, namely Algorithms 1, 2, 3, and 4:

- Algorithm 1 receives two square-free polynomials $a, b \in \mathbb{K}[x]$ and two positive integers e, f. Its output is a factor refinement of a^e, b^f.
- Algorithm 2 receives a square-free polynomial $a \in \mathbb{K}[x]$, a positive integer e, a sequence of square-free pairwise coprime polynomials $(b_1, b_2, \ldots, b_n)$ of $\mathbb{K}[x]$ and a sequence of positive integers $(f_1, f_2, \ldots, f_n)$. The output is a factor refinement of $a^e, {b_1}^{f_1}, \ldots, {b_n}^{f_n}$.

Algorithm 1: `PolyRefine`(a, e, b, f)

Input: $a, b \in \mathbb{K}[x]$ squarefree polynomials over a field $\mathbb{K}$ and e, f two positive integers.
Output: (c, u, G, V, d, w) where $c, d \in \mathbb{K}[x]$ and $u, w \in \mathbb{N}$ and G is a sequence $(g_1, \ldots, g_s)$ of polynomials of $\mathbb{K}[x]$ and V is a sequence $(v_1, \ldots, v_s)$ of positive integers such that $(c, u), (g_1, v_1), \ldots, (g_s, v_s), (d, w)$ is a factor refinement of $(a, e), (b, f)$.

$g \leftarrow \gcd(a, b)$; $a' \leftarrow a$ quotient g; $b' \leftarrow b$ quotient g;
if $g = 1$ **then**
  return $(a, e, \varnothing, \varnothing, b, f)$; `// Here` $\varnothing$ `designates the empty sequence`
else if $a = b$ **then**
  return $(1, 1, (a), (e + f), 1, 1)$
else
  $(\ell_1, e_1, G_1, V_1, r_1, f_1) \leftarrow$ `PolyRefine`$(a', e, g, e + f)$;
  $(\ell_2, e_2, G_2, V_2, r_2, f_2) \leftarrow$ `PolyRefine`(r_1, f_1, b', f);
  if $\ell_2 \neq 1$ **then**
    $G_2 \leftarrow G_2 + (\ell_2)$; `// Here + designates sequence concatenation`
    $V_2 \leftarrow V_2 + (e_2)$;
  return $(\ell_1, e_1, G_1 + G_2, V_1 + V_2, r_2, f_2)$;

Algorithm 2: `MergeRefinePolySeq`(a, e, B, F)

Input: A square-free polynomial $a \in \mathbb{K}[x]$, a positive integer e, a sequence of square-free pairwise coprime polynomials $(b_1, b_2, \ldots, b_n)$ of $\mathbb{K}[x]$ and a sequence of positive integers $(f_1, f_2, \ldots, f_n)$.
Output: (ℓ, m, Q, R, S, T) where $\ell \in \mathbb{K}[x]$, $m \in \mathbb{N}$, $Q = (q_1, \ldots, q_s)$ and $S = (s_1, \ldots, s_p)$ are 2 polynomial sequences of $\mathbb{K}[x]$, $R = (r_1, \ldots, r_s)$ and $T = (t_1, \ldots, t_p)$ are 2 positive integer sequences s. t. $(\ell, m), (q_1, r_1), \ldots, (q_s, r_s), (s_1, t_1), \ldots, (s_p, t_p)$ is a factor refinement of $a^e, {b_1}^{f_1}, \ldots, {b_n}^{f_n}$.

$\ell_0 \leftarrow a$; $m_0 \leftarrow e$; $Q \leftarrow \varnothing$; $R \leftarrow \varnothing$; $S \leftarrow \varnothing$; $T \leftarrow \varnothing$;
for i from 1 to n **do**
  $(\ell_i, m_i, G_i, V_i, d_i, w_i) \leftarrow$ `PolyRefine`$(\ell_{i-1}, m_{i-1}, b_i, f_i)$;
  $Q \leftarrow Q + G_i$;
  $R \leftarrow R + V_i$;
  if $d_i \neq 1$ **then**
    $S \leftarrow S + (d_i)$;
    $T \leftarrow T + (w_i)$;
return $(\ell_n, m_n, Q, R, S, T)$;

- Algorithm 3 takes two sequences of square-free pairwise coprime polynomials $(a_1, \ldots, a_n)$ and $(b_1, b_2, \ldots, b_r)$ of $\mathbb{K}[x]$ together with two sequences of positive integers $(e_1, e_2, \ldots, e_n)$ and $(f_1, f_2, \ldots, f_r)$. The output is a factor refinement of ${a_1}^{e_1}, \ldots, {a_n}^{e_n}, {b_1}^{f_1}, \ldots, {b_r}^{f_r}$.
- The specifications of Algorithm 4 are the same as those of Algorithm 3. The difference is that Algorithm 4 runs in parallel and uses Algorithm 3 as serial base case.
- Finally,Algorithm 5 takes a sequence of square-free polynomials $(m_1, m_2, \ldots, m_k) \in \mathbb{K}[x]$ as input and generates as output a sequence of univariate polynomials $(n_1, n_2, \ldots, n_s) \in \mathbb{K}[x]$

Algorithm 3: `MergeRefineTwoSeq`(A, E, B, F)

Input: Two sequences of square-free pairwise coprime polynomials $A = (a_1, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_r)$ of $\mathbb{K}[x]$ together with two sequences of positive integers $E = (e_1, e_2, \ldots, e_n)$ and $F = (f_1, f_2, \ldots, f_r)$.

Output: (L, M, Q, R, S, T) where $L = (\ell_1, \ldots, \ell_h)$, $Q = (q_1, \ldots, q_s)$ and $S = (s_1, \ldots, s_p)$ are three sequences of polynomials of $\mathbb{K}[x]$, $M = (m_1, \ldots, m_h)$, $R = (r_1, \ldots, r_s)$ and $T = (t_1, \ldots, t_p)$ are three sequences of positive integers such that $(\ell_1, m_1), \ldots, (\ell_h, m_h), (q_1, r_1), \ldots, (q_s, r_s), (s_1, t_1), \ldots, (s_p, t_p)$ is a factor refinement of $(a_1, e_1), \ldots, (a_n, e_n), (b_1, f_1), \ldots, (b_r, f_r)$.

2: $L \leftarrow \varnothing$; $M \leftarrow \varnothing$; $Q \leftarrow \varnothing$; $R \leftarrow \varnothing$; $S_0 \leftarrow B$; $T_0 \leftarrow F$;

4: **for** i from 1 to n **do**

6: $(\ell_i, m_i, Q_i, R_i, S_i, T_i) \leftarrow$ `MergeRefinePolySeq`$(a_i, e_i, S_{i-1}, T_{i-1})$;

8: $Q \leftarrow Q + Q_i$;

10: $R \leftarrow R + R_i$;

12: **if** $\ell_i \neq 1$ **then**

14: $L \leftarrow L + (\ell_i)$;

16: $M \leftarrow M + (m_i)$;

18: **return** (L, M, Q, R, S_n, T_n);

Algorithm 4: `MergeRefinementDNC`(A, E, B, F)

Input: Two sequences $A = (a_1, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_r)$ of square-free pairwise coprime polynomials in $\mathbb{K}[x]$ together with two sequences of positive integers $E = (e_1, e_2, \ldots, e_n)$ and $F = (f_1, f_2, \ldots, f_r)$.

Output: (L, M, Q, R, S, T) where $L = (\ell_1, \ldots, \ell_h)$, $Q = (q_1, \ldots, q_s)$ and $S = (s_1, \ldots, s_p)$ are three sequences of polynomials of $\mathbb{K}[x]$, $M = (m_1, \ldots, m_h)$, $R = (r_1, \ldots, r_s)$ and $T = (t_1, \ldots, t_p)$ are three sequences of positive integers such that $(\ell_1, m_1), \ldots, (\ell_h, m_h), (q_1, r_1), \ldots, (q_s, r_s), (s_1, t_1), \ldots, (s_p, t_p)$ is a factor refinement of $(a_1, e_1), \ldots, (a_n, e_n), (b_1, f_1), \ldots, (b_r, f_r)$.

2: **if** $n \leq$ BASESIZE or $r \leq$ BASESIZE **then**

4: **return** `MergeRefineTwoSeq`(A, E, B, F);

6: **else**

8: Divide A, E, B, and F into two halves called A_1, A_2, E_1, E_2, B_1, B_2, and F_1, F_2, respectively ;

10: $(L_1, M_1, Q_1, R_1, S_1, T_1) \leftarrow$ **spawn** `MergeRefinementDNC`(A_1, E_1, B_1, F_1) ;

12: $(L_2, M_2, Q_2, R_2, S_2, T_2) \leftarrow$ **spawn** `MergeRefinementDNC`(A_2, E_2, B_2, F_2) ;

14: **sync** ;

16: $(L_3, M_3, Q_3, R_3, S_3, T_3) \leftarrow$ **spawn** `MergeRefinementDNC`(L_1, M_1, S_2, T_2) ;

18: $(L_4, M_4, Q_4, R_4, S_4, T_4) \leftarrow$ **spawn** `MergeRefinementDNC`(L_2, M_2, S_1, T_1) ;

20: **sync** ;

22: **return** $\{L_3 + L_4, M_3 + M_4, Q_1 + Q_2 + Q_3 + Q_4, R_1 + R_2 + R_3 + R_4, S_3 + S_4, T_3 + T_4\}$;

Algorithm 5: `ParallelFactorRefinementDNC(`A`)`

Input: A sequence $A = (m_1, m_2, \ldots, m_k)$ of square-free polynomials of $\mathbb{K}[x]$.
Output: A sequence of square-free pairwise coprime polynomials $N = (n_1, n_2, \ldots, n_s) \in \mathbb{K}[x]$, and a sequence of positive integers $E = (e_1, e_2, \ldots, e_s)$ such that $(n_1, e_1), (n_2, e_2), \ldots, (n_s, e_s)$ is a factor refinement of $m_1, m_2, \cdots, m_k$.

```
if k < 2 then
    return (m_1), (1) ;
else
    Divide A into two subsequences called A_1 and A_2;
    (X_1, Y_1) ← spawn ParallelFactorRefinementDNC(A_1) ;
    (X_2, Y_2) ← spawn ParallelFactorRefinementDNC(A_2) ;
    sync;
    return MergeRefinementDNC(X_1, Y_1, X_2, Y_2);
```

and a sequence of positive integers $(e_1, e_2, \ldots, e_s)$ such that $((n_1, e_1), (n_2, e_2), \ldots, (n_s, e_s))$ is a refinement of $(m_1, m_2, \cdots, m_k)$. Moreover, $n_1, n_2, \ldots, n_s$ are square-free.

Proposition 1 analyzes the parallelism of Algorithm 5 for the fork-join parallelism model [10] under one simplification hypothesis, which we state below.

We assume that each polynomial operation (division or GCD computation) involved in Algorithm 5 or its subroutines has a unit cost. Obviously, this assumption does not seem realistic. However, if D is the maximum degree of an input polynomial and if each arithmetic operation (addition, multiplication, inversion) in $\mathbb{K}$ has a constant bit cost, then each subsequent polynomial operation performed by Algorithm 5 or its subroutines runs in $O(D^2)$ bit operations. Therefore, this assumption can still be seen as a first approximation of what really happens.

In order to analyze Algorithm 5 and its subroutines, one needs to choose a measure of the input data. Our assumption suggests the following choice. For each of Algorithms 1, 2, 3, 4 or 5, the input size is the total number of polynomials in the input.

Proposition 1 *With the above hypotheses, for a size n input, the work, span, and parallelism of Algorithm 5 are respectively $O(n^2)$, $O(Cn)$ and $O(n/C)$, where C is the threshold* BASESIZE.

The proof of Proposition 1 can easily be made by adapting the proof techniques of [8], see [3].

We turn now our attention to cache complexity and analyze the cache complexity of the serial versions of our algorithms. We start by specifying how data is layed out in memory. We assume that each input or output sequence in Algorithms 1, 2, 3 or 4 is *packed*, that is, its successive polynomials occupy consecutive memory slots. We also assume that each element of the field $\mathbb{K}$ can be stored in one machine word.

We specify a few helpful notations for establishing our cache complexity results. We denote by $|p|$ the degree of a non-zero univariate polynomial p over $\mathbb{K}$. For a finite polynomial sequence $P = (p_1, \ldots, p_n)$, we denote by $|P|$ the sum of the degrees of $p_1, \ldots, p_n$.

Nest, we make a second assumption: each integer in any input or output sequence of positive integers is stored in one machine word; moreover, any polynomial in any input or output sequence of Algorithms 1, 2, 3 or 4 is non-constant and monic[1]. Consequently, any polynomial p in any input or output sequence of Algorithms 1, 2, 3 or 4 can be stored within $|p|$ machine words.

[1] As they are currently stated, Algorithms 1, 2, 3 or 4 do not meet this latter assumption However, enforcing this assumption only requires simple transformations that we do not reproduce to keep pseudo-code easier to read.

Lemma 1 *Under the above assumptions, with the notations of Algorithm 1, we have*

$$|c| + |G| \leq |a| \quad \text{and} \quad |G| + |d| \leq |b|. \tag{1}$$

Proof ▷ We proceed by induction on the sum of the degrees of the input polynomials of Algorithm 1, that is, $|a| + |b|$. First, we observe that, at Line 4, if $g = 1$ holds, then the conclusion trivially holds. Indeed, in this case, we have $c = a$, $G = \varnothing$ and $d = b$. Similarly, at Line 8, if $a = b$ holds, the conclusion also holds. These two cases cover, in particular, the base case of the induction, that is, $|a| + |b| = 2$. Now, we assume that g has a positive degree. Thus, we can apply the induction hypothesis to the two recursive calls at Lines 14 and 16, since we have $|a'| + |g| < |a| + |b|$ and thus $|r_1| + |b'| \leq |g| + |b'| < |a| + |b|$.

Next, we shall estimate $|\ell_1|$, $|G_1 + G_2|$ and $|r_2|$ at the last return point of Algorithm 1, that is, at Line 24. We designate by G_2^0 the value of G_2 after executing Line 16 and before executing Line 18. Then we have:

$$\begin{array}{rcll} |\ell_1| + |G_1 + G_2| & \leq & |\ell_1| + |G_1| + |G_2| & \text{by definition} \\ & \leq & |\ell_1| + |G_1| + |G_2^0| + |\ell_2| & \text{by definition} \\ & \leq & |a'| + |g| & \text{by induction} \\ & \leq & |a| & \text{since } a = a'g. \end{array}$$

Similarly, we have

$$\begin{array}{rcll} |G_1 + G_2| + |r_2| & \leq & |G_1| + |G_2| + |r_2| & \text{by definition} \\ & \leq & |G_1| + |\ell_2| + (|G_2^0| + |r_2|) & \text{by definition} \\ & \leq & |G_1| + |r_1| + |b'| & \text{by induction} \\ & \leq & |g| + |b'| & \text{by induction} \\ & \leq & |b| & \text{since } b = b'g. \end{array}$$

This completes the proof. ◁

Lemma 2 *Under the above assumptions, with the notations of Algorithm 2, we have*

$$|\ell| + |Q| \leq |a| \quad \text{and} \quad |Q| + |S| \leq |B|. \tag{2}$$

Proof ▷ We use the notations of the pseudo-code of Algorithm 2. We have the following inequalities:

$$\begin{array}{rcll} |\ell_n| + |Q| & \leq & |\ell_n| + |G_n| + |G_{n-1}| + \cdots + |G_1| & \text{by definition} \\ & \leq & |\ell_{n-1}| + |G_{n-1}| + \cdots + |G_1| & \text{by Lemma 1} \\ & \vdots & & \vdots \\ & \leq & |\ell_1| + |G_1| & \text{by Lemma 1} \\ & \leq & |l_0| & \text{by Lemma 1} \\ & \leq & |a| & \text{by definition.} \end{array}$$

With Lemma 1, we also have:

$$|Q| + |S| \leq \sum_{i=1}^{i=n} (|G_i| + |d_i|) \leq \sum_{i=1}^{i=n} |b_i| \leq |B|.$$

This completes the proof. ◁

Lemma 3 *Under the above assumptions, with the notations of Algorithm 3, we have*

$$|L| + |Q| \leq |A| \quad \text{and} \quad |Q| + |S_n| \leq |B|. \tag{3}$$

The proof technique is similar to that of the previous two lemmas, so we skip it here.

Proposition 2 *Under the above assumptions, for an input of size n, each of the Algorithms 1, 2 and 3 can be run in space $\Theta(n)$ bits.*

PROOF ▷ Lemmas 1, 2 and 3 imply that for an input of size n, the output size is at most n. Recall that by input size or output size, we mean the total number of polynomial coefficients (except the leading coefficients since they are all set to 1). The output contains also sequences of positive integers, the number of those being at most n. Finally, we observe that each of the Algorithms 1, 2 and 3 does not require extra memory space other than: the space for the input and output data, and a constant number of pointers and index variables. ◁

Lemma 4 *Under the above assumptions, for an ideal cache of Z words, with L words per cache-line, there exists a positive constant α such that for an input of size n, satisfying $n < \alpha Z$, the number of cache misses of Algorithm 3 is $O(n/L + 1)$.*

PROOF ▷ By Proposition 2, there exists a positive constant β such that for an input of size n, the memory requirement for running Algorithm 3 is at most βn bits. And those βn bits, up to a constant number of them, are used for storing a number (independent of n) of sequences of polynomials and positive integers. Now recall that each sequence of polynomials (resp. positive integers) is stored in an array. Recall also that loading to cache (resp. writing back to main memory) an array of length ℓ requires $O(\ell/L + 1)$ cache misses. This conclusion follows. ◁

Theorem 1 *Under the above assumptions, for an ideal cache of Z words, with L words per cache-line, for C small enough, for a size n input, the number of cache misses of Algorithm 4 is $Q(n) = O(n^2/ZL + n^2/Z^2)$, which becomes $Q(n) \in O(n^2/ZL)$ with tall cache assumption.*

PROOF ▷ It follows from Lemma 4 and the recursive structure of Algorithm 4 that there exists a positive constant α such that $Q(n)$ satisfies the following relation:

$$Q(n) \leq \begin{cases} O(n/L + 1) & \text{for } n < \alpha Z \\ 4Q(n/2) + \Theta(1) & \text{otherwise,} \end{cases}$$

provided that $C < \alpha Z$ holds. Strictly speaking the above recurrence assumes that each of the input polynomial sequences A and B can be split into two sequences of approximately the same size. When the total number of polynomials in A and B is large, this is likely. When it is small, the condition $n < \alpha Z$ is likely to hold and we are "out of the woods". A more formal treatment can be done using standard (but quite involved) proof techniques, as for the parallelization of *quick-sort* algorithm[2]. The above recurrence leads to the following inequality for all $n \geq 2$:

$$\begin{aligned} Q(n) &\leq 4Q(n/2) + \Theta(1) \\ &\leq 4[4Q(n/4) + \Theta(1)] + \Theta(1) \\ &\vdots \\ &\leq 4^k Q(n/2^k) + \sum_{j=0}^{k-1} 4^j \Theta(1) \end{aligned}$$

where $k = \lceil \log_2(n/\alpha Z) \rceil$. Since we have $n/2^k \leq \alpha Z$, we deduce:

$$\begin{aligned} Q(n) &\leq (n/\alpha Z)^2(\alpha Z/L + 1) + \Theta((n/\alpha Z)^2) \\ &= O(n^2/ZL + n^2/Z^2). \end{aligned}$$

This completes the proof. ◁

[2] For details, see the slides of the lecture *Analysis of Multithreaded Algorithms* available at `http://www.csd.uwo.ca/~moreno/CS9624-4435-1011.html`

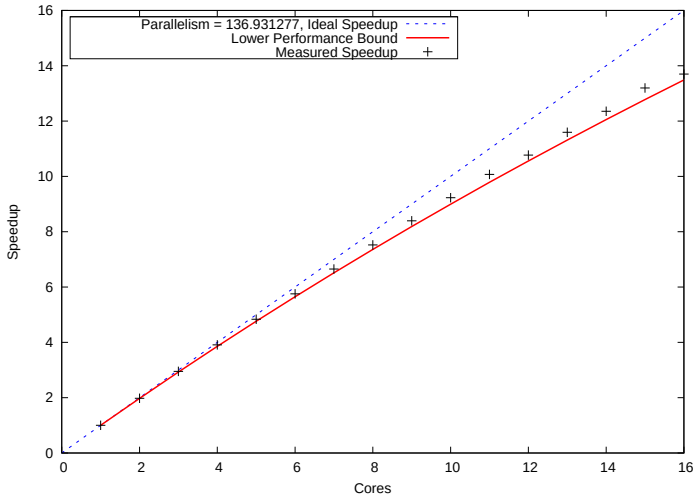

Figure 2. `Cilkview`-based scalability analysis of our parallel factor refinement algorithm for as a set of dense 5,000 univariate polynomials of degree up to 200 as input.

4. Experimentation

We have implemented our algorithm in `Cilk++` and tested it successfully with both integers and polynomials. Figure 2 generated with `Cilkview`, shows scalability results on an Intel(R) Xeon(R) (64 bit) 16-core node, with CPU (E7340) Speed 2.40GHz, 128.0 GB of RAM for an input data set of 5,000 polynomials of various degrees up to 200. Coefficients are taken modulo a machine word prime. On 16 cores, a speedup factor of 14 is reached. Figure 3 shows that similar performance results have been obtained with an input data set of 200,000 machine word integers. See [3] for more experimental results. Within MAPLE [2], we have realized serial implementations of both the original algorithm of Bach, Driscoll and Shallit [4] and our divide and conquer adaptation. For sufficiently large input, the latter outperforms the former by a factor ranging from 3 to 6.

5. Concluding remarks

We have presented a cache efficient algorithm for coprime factorization, based on the factor refinement principle of Bach, Driscoll and Shallit [4]. The high ratio *work to cache complexity* of our approach (namely $O(ZL)$ for an ideal cache of Z words, with L words per block) suggests that it is well suited for modern architectures, in particular multicores. Experimental comparison between the serial implementations of the original algorithm of [4] and ours confirm the benefits of the latter.

The divide and conquer structure of our approach leads to a multithreaded algorithm with a parallelism of $O(n)$ (in the fork-join model) for an input data of size n. Different optimization techniques (work load balancing among sub-problems, data packing) are used in our our `Cilk++` implementation which achieves linear speedup on 16 cores for sufficiently large input data.

We believe that this work illustrates the importance of cache complexity in the design of efficient algorithm targeting multicore architectures. As shown in [3], alternative algorithmic approaches for coprime factorization fail on these architectures either because they have a low ratio work to cache complexity or because of parallelization overheads.

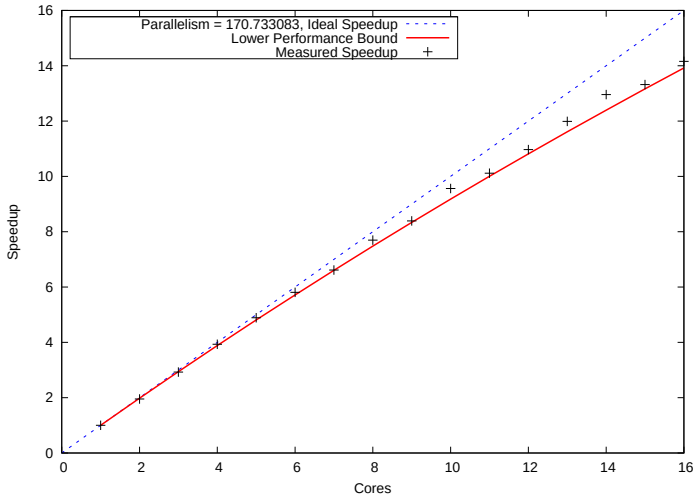

Figure 3. `Cilkview`-based scalability analysis of our parallel factor refinement algorithm for a set of 200,000 integers as input.

References

[1] GMP: Arithmetic without limitations. `http://gmplib.org`.
[2] Maplesoft, Waterloo, Ontario, Cananda. *The computer algebra system Maple*, `http://www.maplesoft.com`.
[3] Md. M. Ali *On the Factor Refinement Principle and its Implementation on Multicore Architectures* Masters Thesis, The University of Western Ontario, 2011.
[4] E. Bach, J. Driscoll, and J. Shallit. Factor Refinement. In *Symposium on Discrete Algorithms*, pages 201–211, 1990.
[5] D. J. Bernstein. Factoring into Coprimes in Essentially Linear Time. *J. Algorithms*, 54(1):1–30, 2005.
[6] D. J. Bernstein, H. W. Lenstra Jr., and J. Pila. Detecting Perfect Powers by Factoring into Coprimes. *Math. Comput.*, 76(257):385–388, 2007.
[7] X. Dahan, M. Moreno Maza, É. Schost, W. Wu, and Y. Xie. On the Complexity of the D5 Principle. *SIGSAM Bull.*, 39:97–98, September 2005.
[8] C. E. Leiserson, L. Li, M. Moreno Maza, and Y. Xie. Parallel Computation of the Minimal Elements of a Poset. In *4th International Workshop on Parallel and Symbolic Computation*, pages 53–62, 2010.
[9] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious Algorithms. In *Proc. of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–297, 1999.
[10] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN*, 1998.
[11] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 2nd ed., 2003.
[12] J. Hong and H. T. Kung. I/O Complexity: The Red-blue Pebble Game. In *STOC*, pages 326–333, 1981.
[13] C. E. Leiserson. The Cilk++ Concurrency Platform. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 522–527, New York, NY, USA, 2009. ACM.
[14] M. Moreno Maza and Y. Xie. FFT-based Dense Polynomial Arithmetic on Multi-cores. In D.J.K. Mewhort, editor, *Proc. HPCS 2009*, volume 5976 of *LNCS*, Heidelberg, 2010. Springer-Verlag Berlin.
[15] D. Y. Y. Yun. On Square-free Decomposition Algorithms. In *Proceedings of the Third ACM Symposium on Symbolic and Algebraic Computation*, SYMSAC '76, pages 26–35, New York, NY, USA, 1976. ACM.