Herbert Melenk Winfried Neun

Parallel Polynomial Operations in the Buchberger Algorithm

Preprint SC 88-4 (March 1988)

Konrad-Zuse-Zentrum für Informationstechnik;
Heilbronner Straße 10; D-1000 Berlin 31

Herausgegeben vom
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Heilbronner Strasse 10
1000 Berlin 31
Verantwortlich: Dr. Klaus André
Umschlagsatz und Druck: Verwaltungsdruckerei Berlin

ISSN 0933-7911

Contents

1. Distributive polynomials and Buchberger's algorithm 2

2. LCOMB: linear combination as a central tool for distributive polynomial operation 3

3. Parallel execution based on Lcomb 4

4. Supporting Data Structure: Objects 6

5. Segmentation of the Buchberger Algorithm 7

6. Handling of mixed data flow via closures 8

7. Process Model 9

8. Simulation 9

9. Parallel execution of the Bba 11

10. Further Parallelism in the Bba 13

Summary:
The Buchberger algorithm for the construction of a Gröbner base from a given set of polynomials is an example for the calculation with polynomials in distributive representation. It is shown, that the algorithm can be based on one single polynomial operation, a "linear combination". This operation can be rewritten as a merge operation such that the arising result can become input for a subsequent processing step as soon as the first monomial is "ready". This property is the source for parallel execution, which technically is supported by an object-oriented programming approach and by a process model.

Notation:
The developments were based on LISP and REDUCE (Hearn [87]). So for the examples a language near to the REDUCE syntax was used, which should be self-explanatory. It is the opinion of the authors, that the REDUCE language can and should be further developed in order to support advanced structures as needed in this context.

1. Distributive polynomials and Buchberger's algorithm

In the context of Gröbner base calculations, the polynomials are in distributive representation. In contrast to the recursive representation, a distributive polynomial (DP) is a linear sequence of monomials (MN), each of which consists of a coefficient ($C \in$ *coeffcent domain*) and a product of variable powers (PVP):

$$\begin{aligned} DP &= [MN_1, \ldots, MN_k] \\ &= [C_1 x_1^{i_{11}} \cdots x_n^{i_{1n}}, \ldots, C_k x_1^{i_{k1}} \cdots x_n^{1_{kn}}] \\ &= \sum_{j=1,n} C_j x_1^{i_{j1}} \cdots x_n^{i_{jn}} \end{aligned}$$

The monomials are ordered in a descending sequence by a global term ordering, e.g. inverse lexicographical ordering:

$$MN_p > MN_q \longleftrightarrow \exists k \leq n : (\forall l < k : i_{pl} = i_{ql}) \wedge i_{pk} > i_{qk}$$

The standard constructors and selectors are:

select leading monomial	$Lmon([MN_1, \ldots, MN_k]) \rightarrow MN_1$
select reductum	$Red([MN_1, \ldots]) \rightarrow [MN_2, \ldots]$
add leading monomial	$AddL(MN_0, [MN1, \ldots]) \rightarrow [MN_0, MN_1, \ldots]$

With these operators a complete polynomial arithmetic can be constructed.

The Buchberger algorithm (Bba) constructs critical pairs of polynomials and processes them. In this context, we examine its central part in a (for description purposes) simplified version, which omits all details (e.g. vanishing polynomials, criteria, full reduction). The description of the full Bba can be found in Gebauer, Möller [88] and Möller [88].

```
while notempty pairlist do                          % main loop
≪   (p1, p2) := next pair;
    q := lcm(Lmon(p1), Lmon(p2));
    s := (q/Lmon(p1)) * p1 - (q/Lmon(p2)) * p2;
    h := s;
                                                    % reduction loop
    while exists r ∈ G such that Lmon(r) divides Lmon(h) do
         h := h - (Lmon(h)/Lmon(r)) * r;
    h := 1/(leadingC) * h;                          % normalizing
    for each r ∈ G construct pair (r, h)
            testing criteria on Lmon(r), Lmon(h);
    G := {h} ∪ G;≫
```

When we classify the polynomial operations needed by the Bba we find that:

(1.1) for the control of the algorithm only the access to the leading monomial is necessary,

(1.2) all arithmetic combinations are special cases of the pattern

$$MN_1 * DP_1 + MN_2 * DP_2$$

with appropriately constructed monomials.

These facts will be important for our approach to parallelization.

2. LCOMB: linear combination as a central tool for distributive polynomial operation

Because of (1.2) we introduce a new operation:

$$Lcomb(MN_1, DP_1, MN_2, DP_2) \rightarrow MN_1 * DP_1 + MN_2 * DP_2$$

which is a "linear combination" of two polynomials.

The Bba can be rewritten using Lcomb as single polynomial operation, but Lcomb can support the complete polynomial arithmetic as well:

$$DP_1 + DP_2 = Lcomb(1, DP_1, 1, DP_2)$$
$$DP_1 - DP_2 = Lcomb(1, DP_1, -1, DP_2)$$
$$DP_1 * DP_2 = \ll x := 0;$$
$$\quad \textit{for each } MN \textit{ in } DP_1 \textit{ do } x := Lcomb(1, x, MN, DP_2);\ x \gg$$

The straightforward code for Lcomb is recursive, the depth determined by the length of the polynomials (omitting details regarding efficiency etc.).

proc Lcomb1 $(MN_1, DP_1, MN_2, DP_2):$ (2.1)

$(C_1, PVP_1) := MN_1 * Lmon(DP_1); (C_2, PVP_2) := MN_2 * Lmon(DP_2);$

if $PVP_1 = 0 \wedge PVP_2 = 0$ *then* 0

else if $PVP_1 = PVP_2$ *then*

$AddL\ ((C_1 + C_2, PVP_1), Lcomb(MN_1, Red(DP_1), MN_2, Red(DP_2)))$

else if $PVP_1 > PVP_2$ *then*

$AddL((C_2, PVP_2), Lcomb(MN_1, Red(DP_1), MN_2, DP_2))$

else $AddL((C_2, PVP_2), Lcomb(MN_1), (DP_1, MN_2, RedDP_2)))$

Because of the recursion and of the constructor *AddL*, the leading monomial is added last. This technique is efficient for single processor LISP style systems.

If we add a new constructor
Append monomial: $App\,([MN_1,\ldots,MN_k],MN_p) \rightarrow [MN_1,\ldots,MN_k,MN_p]$ we can reformulate *Lcomb* in the style of a merge procedure

$$ProcLcomb(MN_1, DP_1, MN_2, DP_2, r): \tag{2.2}$$

$begin$

$(C_1, PVP_1) := MN_1 * Lmon(DP_1); (C_2, PVP_2) := MN_2 * Lmon(DP_2);$

$loop:$

$if\ 0 = PVP_1 = PVP_2\ then\ return\ r;$

$if\ PVP_1 = PVP_2\ then \ll r := App(r, (C_1 + C_2, PVP_1));$

$DP_1 := Red(DP_1); (C_1, PVP_1) := MN_1 * Lmon(DP_1);$

$DP_2 := Red(DP_2); (C_2, PVP_2) := MN_2 * Lmon(DP_2) \gg$

$else\ if\ PVP_1 > PVP_2$ then

$\ll r := App(r, (C_1, PVP_1));$

$DP_1 := Red(DP_1); (C_1, PVP_1 := MN_1 * Lmon(DP_1) \gg$

$else$

$\ll r := App(r, (C_2, PVP_2));$

$DP_2 :=$ Red$(DP_2); (C_2, PVP_2 := MN_2 * Lmon(DP_2) \gg;$

$goto\ loop;$

$end;$

In this version the fifth parameter "r" has to be initialized by the caller as an empty polynomial; we will need this convention later.

3. Parallel execution based on Lcomb

In (2.2) the leading monomial is calculated first and it is the first monomial to be linked to the resulting structure. Because of (1.2) we can continue the Bba as soon as the leading monomial is available, so we can overlap the operation of *Lcomb* and the processing of its result by the Bba (including further Lcomb's) in a multi-processing environment provided that we have a synchronization regarding production/access of subsequent monomials. Figure 3.1 demonstrates, that even several cycles of the main loop in Bba can operate simultaneously. Note that although the global structure of the algorithm looks very homogeneous, it cannot be configured as systolic array because the speed of the data flow through Lcomb is not constant: if the exponent patterns of the meeting monomials are equal,

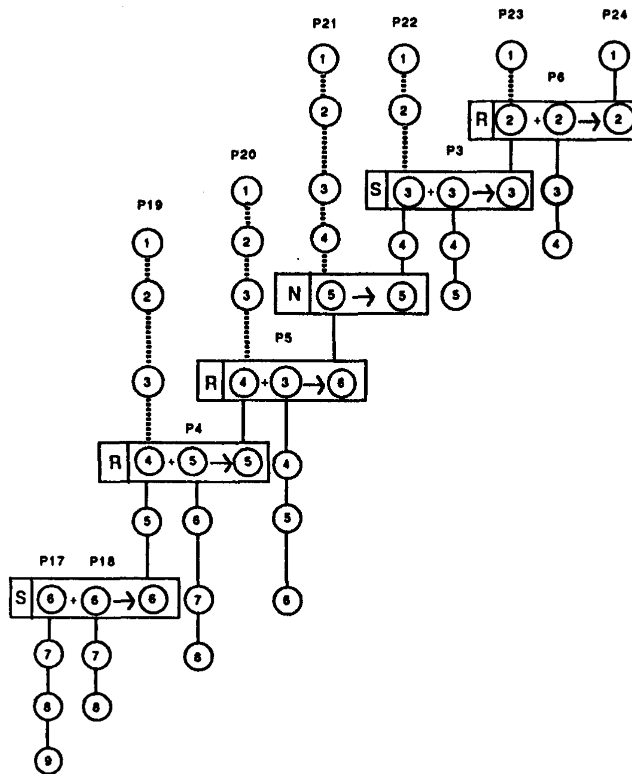


Figure 3.1: Snapshot of overlapped Lcomb actions:

S	:	calculation of an S-polynomial
R	:	reduction step
N	:	normalizing
P19	=	S(P17, P18)
P21	=	H(P17, P18)
P23	=	S(P22, P3)

Lcomb consumes one monomial from both inputs, otherwise from one input alternating irregularly and there are steps where no output monomial is produced (coefficient extinction). The number of R-boxes between two S-boxes is at all not predictable.

4. Supporting Data Structure: Objects

We had already given up earlier the linear list as basic data structure for the DPs in favour of an object oriented approach in order to support a variety of memory organization at the same time. The control mechanisms needed for the parallel organization can be added to the object structure without difficulty.

A DP is an object with the following features:

- when created the DP is empty,
- each DP has an individual property list for slot/value pairs,
- a DP can be in status "new" (MNs can be appended) or in status "old" (no more MNs will be appended),
- the constructors and selectors are implemented as methods individual to each DP; the name of a method handler is part of the DP structure; it is called indirectly via LISP APPLY.

The data structure is hidden almost completely.

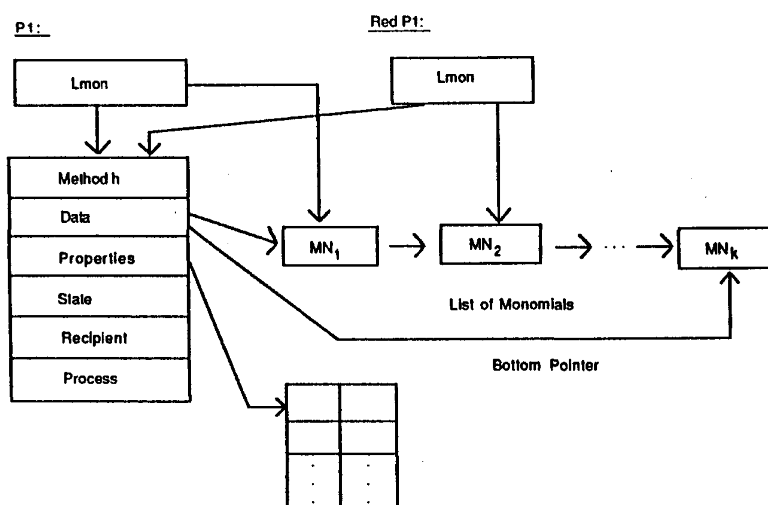


Figure 4.1: Object structure of a DP "P1" and of its reductum.

For efficiency, the access to the leading monomial is "free" (first element in the structure); for an efficient appending, the method handlers maintain a bottom pointer in order to avoid repeated traversing of the data.

The head of an object is implemented as list structure. The operation *Red* produces a similar object with another first element and identical rest; the original object remains unchanged.

5. Segmentation of the Buchberger Algorithm

For parallel execution, the Bba has to be segmented into tasks. For the support of segmented execution we add two slots to the DP objects:

- the *recipient* slot can hold the name of a procedure; this procedure is started as soon as the DP is available for subsequent processing (that means existence of the first monomial in a parallel environment).
- the *process* slot holds information about the processing environment of the recipient.

The process slot is needed mainly for the synchronization of data flow: if the recipient's request for a monomial is not yet satisfied (operation *Red*), it is deactivated automatically. An appending of an additional monomial or the marking of the DP as "old" reactivates the recipient.

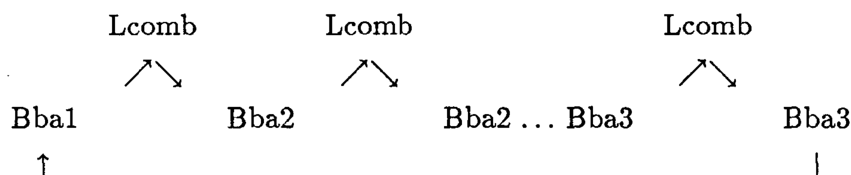
With this technical background we can rewrite the Bba in a segmented form (sBba), which enables overlapped processing. The routine *Pcreate* ($fnc(P_1, \dots, P_n)$) creates a new process, which performs the function "fnc" with parameters $P_1, \dots, P_n$. In most cases the last parameter is an object with defined recipient; as soon as the first monomial arrives, the object creates a process with this recipient as function and itself as parameter.

```
proc Bba1():                        % calculate S-polynomial
    if empty pairlist then Pcreate (Finale) else
    ≪ (P1P2) := next pair;
       q := lcm(Lmon(P1), Lmon(P2));
       Pcreate (Lcomb(q/Lmon(P1), P1, (−q)/Lmon(P2), P2,
                new DP (recipient = Bba2))
    ≫;
```

```
proc Bba2(s):                         % reduce S-polynomial to H-polynomial
     if exists r ∈ G such that Lmon(r) divides Lmon(s)
       then                           % reduction step (loop)
            Pcreate (Lcomb(−1, s, Lmon(s)/Lmon(r), r,
                      new DP(recipient = Bba2)
       else                           % normalization step (exit from loop)
            Pcreate (Lcomb(0, 0, 1/(leading C of S), S,
                      new DP(recipient = Bba3);

proc Bba3(h):                         % construct new pairs, add h to G, loop
     ≪ for each r ∈ G construct pair (r, h)
                 testing criteria on (Lmon(r), Lmon(h);
        G := {h} ∪ G;
        Pcreate (Bba1());
     ≫ ;
```

This version of the Bba is performed as a chain of coroutines:



The routines Bba1 - Bba3 control the algorithm and predetermine in the recipient slot of the resulting structure the successor of an Lcomb action.

6. Handling of mixed data flow via closures

In (5) all processes had the simple structure one/two polynomials in - one polynomial out. If a data structure has to traverse several processes, this can be formulated by means of Common LISP type closures. As an example, the parallel version of the general multiplication of two DPs can be formulated with a closure as recipient, which simulates the loop over DP1:

```
proc multx(DP1, DP2, DPr):
     if DP1 = 0  then Pcreate (Result(DPr))
       else
       Pcreate (Lcomb(Lmon(DP1), DP2, 1, DPr,
               new DP(recipient = #'(λ(r)(DP1 := Red(DP1); multx(DP1, DP2, r))));
```

It is activated with an empty DP (with or without recipient) as third parameter.

In a similar manner the reduction loop of Bba can be coded, if full reduction (that is reduction of monomials behind the first "stable" Lmon of an H-polynomial) is required.

7. Process Model

The parallel execution requires a technical base, which can handle processes with the following features:

- each process has the full LISP(REDUCE) functionality without I/O functions,
- each process has fast read access to all data structures,
- each process can create new data, which will be available for all processes immediately,
- there is a synchronizing mechanism for the safe accessing/updating of critical data,
- a process can wait for an event (creation of data in a specific structure).

There are several different implementations of the last feature possible; the simplest will be a "busy waiting" loop, the best would be a multi-processing environment: the processor sets a waiting process asleep and executes another active process instead.

An always important question in parallel processing is the granularity of the decomposition. In this approach the "data grain" is the DP, assuming for a moment that the data flow is steady. In a complicated calculation the "typical" intermediate DP has dozens of monomials with quotients of big integers as coefficients.

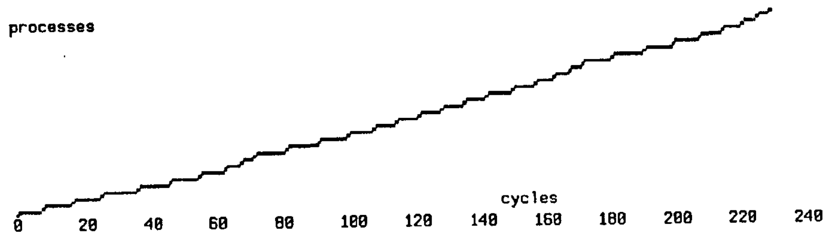
The overall processing is asynchronous in nature. A forced synchronization only takes place in case of monomial extinction. So the case of a zero H-polynomial is a local fence for the parallel execution.

8. Simulation

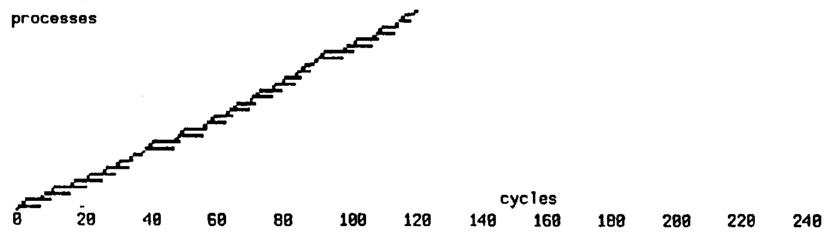
For the first implementation of sBba a multi processing systems was simulated in a conventional LISP environment. For simplification we assume that all loop steps of Lcomb processing one monomial need the same computing time $t = 1$ and that Bba1, Bba2 and Bba3 too need the time $t = 1$ for execution. The number of processors available is a parameter which can be set to only value between 1 and ∞.

```
f1 := 45*p + 35*s -165*b -36;
f2 := 35*p + 40*z + 25*t - 27*s;
f3 := 15*w + 25*p*s +30*z -18*t -165*b**2;
f4 := -9*w + 15*p*t  + 20*z*s;
f5 := w*p + 2*z*t - 11*b**3;
f6 := 99*w - 11*s*b +3*b**2;
f7 := B**2 + 33/50*b + 2673/10000;
```

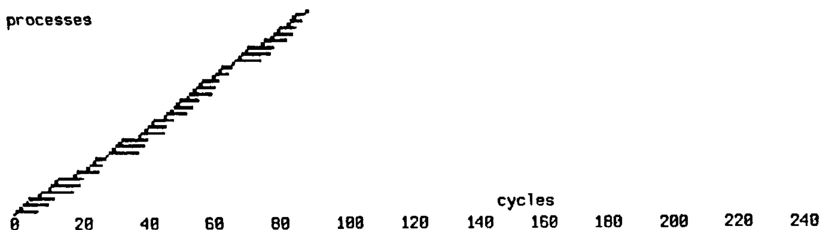
The input polynomials for the sample calculation.



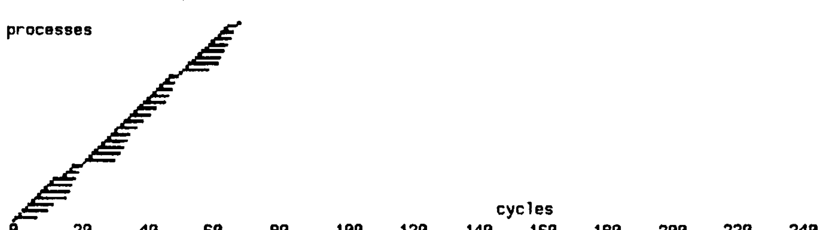
a) One processor available



b) Two processors available



c) Three processors available



d) Unlimited number of processors

Figure 8.1: One calculation simulated with different numbers of processors. From (a) to (b) there is a "speed-up" of almost two. (d) shows that there is a close connection between the length of polynomials and the parallelism. The zero reductions are to be seen as bottlenecks.

There are three queues:

Q_e executing processes

Q_w processes waiting for execution

Q_s sleeping processes

At the beginning of each time step t Q_e is filled up from Q_w up to the number of processors available. Then each element of Q_e is allowed to execute one step. If a process terminates, it is deleted from the Q's. If a process needs data not yet available, it moves from Q_e to Q_s. If data arrive, it moves from Q_s to Q_w. Each process executing in time step t with a record (process#, t#) to a log file. This log file allows the evaluation of parallelism in the execution. As an example we give a graphical representation of the execution of the well-known "little Trinks" (Böge et al [86]) with different numbers of processors. This example is a non-typical small one, but it demonstrates the depending length of polynomials (= number of time steps) and parallelism.

9. Parallel execution of the Bba

The Cray X-MP multiprocessor architecture supports parallel execution by:

- a large central memory directly accessible by all processors,
- sets of shared registers and semaphores (each application has its own set) with fast access,
- operating systems which allow an arbitrary mix of sequential and parallel applications in a multiprogramming environment.

With these prerequisites we designed and implemented an experimental parallel PSL with the following features:

- Memory Management: Code and data area are shared completely. There is only one heap; all processors are allowed to allocate data in the heap; the critical heap pointers reside in shared registers and are protected by a semaphore such that their updating is sequentialized.
- Variable Binding: PSL supports shallow binding for fluid and global variables. For efficiency we have adopted this technique in the following sense: Each processor has its own set of value cells; fluid values are stored in these local value cells and will be rebounded when processes are exchanged (so the scope of a fluid variable is a calling tree rooted in its binding); global variables have a system wide unique value cell.

- Process Execution: Each processor has its own scheduler, which takes waiting processes from the queues and executes them. If a process is interrupted for missing data, it is deactivated and the scheduler selects another process for execution. The interrupt has an S-expression as parameter, which describes the reason for the interrupt in form of a test. So the evaluation of this test determines, if a process can be resumed.

- High Level Interface: Usually parallel execution in LISP is coded via the "future" feature [Swanson et al]. Futures are adequate to model the dynamic behaviour of DPs during construction, but those parts of the Bba working with sets of polynomials and critical pairs are not free of side effects, so they need additional primitives for synchronization. We decided to start with more elementary tools which later can be combined in higher levels:

  pcreate (fn, pars) creates a new process as an asynchronous *apply.*

  pinterrupt (test) interrupts an execution until the value of test will be T. In the Bba this primitive is called only in the most inner DP access routine.

  plock (sem) requests exclusive access to a resource described by a user semaphore (global variable).

  punlock (sem) releases exclusive usage.

Note that a request for an already locked semaphore causes an interrupt. The user semaphores are not identical with the hardware semaphores; the latter cause a processor to wait and they can be used for very short code sequences, e.g. for the implementation of a "test-and-set" of user semaphores.

There is no theoretical limit for the number of processors handled by this organization, although the concurrent heap management can become a bottleneck for larger numbers of processors. In the ideal case of no contention each processor operates with the full speed of the standard Cray PSL.

First experimental calculation with the parallel Bba on a 2-processor X-MP were performed with the "Big Trinks" examples from [Böge et al]. At the moment only the total cpu time (spent on both processors) was measured in order to get an impression of the overhead produced by the process management and the interrupts. All these tests were done in the standard workload which of course imposes additional interrupts by scheduling both processors freely among all user jobs.

conventional program with	3.6 sec
parallel program with	
1 processor assigned	3.7 sec
parallel program with	
2 processors assigned	3.7 sec - 4.5 sec
	(average 4.0 sec)

These tests will be continued with larger examples and a dedicated environment when the basic implementation reaches stable state.

10. Further Parallelism in the Bba

Until now the Buchberger algorithm itself remained untouched. All actions take place in the same order as described in [Gebauer, Möller]; the parallelism is based only on overlapping of steps in a context comparable to pipelining. If we denote with LCP the list of critical pairs and with t the number of h-polynomial calculation, the sequence of operations is:

select one critical pair from LCP(t)
calculate h(t+1) from this pair
update LCP(t) with h(t+1) giving LCP(t+1)

If LCP(t) has more than one element (which is true in most steps), more than one h-calculation can be started at the same time or, e.g. if a processor idles, a new pair can be selected from LCP(t) before LCP(t+1). This technique requires additional bookkeeping, but a higher degree of parallelism can be expected. The price for that parallelism is that a calculation could be started, which would be recognized obsolete during the next update of LCP. These effects will be object of further investigation.

References

J. W. Anderson, W. F. Galway, R. R. Kessler, H. Melenk, W. Neun: *Implementing and Optimizing Lisp for the Cray*, IEEE Software, July 1987.

W. Böge, R. Gebauer, H. Kredel: *Some Examples for Solving Systems of Algebraic Equations by Calculating Groebner Bases*, J. Symb. Comp. 1986, vol.2, pp. 86 – 98.

B. Buchberger: *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenrings nach einem nulldimensionalen Polynomideal*, Ph.D. thesis, Innsbruck, 1965.

B. Buchberger: *Gröbner bases: An algorithmic method in polynomial ideal theory.* (In: Multidimensional Systems Theory, ed.: N. K. Bose) D. Reidel Publ. Comp. 1985, pp. 184 – 232.

R. Gebauer, H. M. Möller: *On an installation of Buchberger's algorithm.* To appear in J. Symb. Comp. 1988.

A. C. Hearn: *REDUCE User's Manual, Version 3.3*, The Rand Corporation, 1987.

H. M. Möller: *On the construction of Gröbner bases using syzygies*, to appear in J.Symb. Comp. 1988.

H. Melenk, W. Neun: *REDUCE User's Guide for Cray 1/ X-MP Series Running COS*, Konrad-Zuse-Zentrum Berlin, Technical Report TR 87-4 , 1987.

H. Melenk, W. Neun: *Usage of Vector Hardware for LISP Processing*, Konrad-Zuse-Zentrum Berlin, 1986.

M. R. Swanson, R. R. Kessler, G. Lindstrom: *An implementation of Portable Standard Lisp on the BBN Butterfly*, Department of Computer Science, TR 88-01, University of Utah, 1988.