Gröbner Bases Computation in Strand: A Case Study for Concurrent Symbolic Computation in Logic Programming Languages

Diplomarbeit

zur Erlangung des akademischen Grades
"Diplom-Ingenieur"
in der Studienrichtung
Informatik

eingereicht von

Kurt Siegl

Angefertigt am
Institut für Mathematik
Technisch-Naturwissenschaftliche Fakultät
Johannes Kepler Universität Linz

Eingereicht bei
o.Univ.-Prof. Dr. Bruno Buchberger

November 1990

Abstract

This thesis studies the adequacy of parallel logic programming languages for the purpose of parallel symbolic computation. We conclude that the languages of the guarded Horn clause type are best suited for parallel Symbolic Computation. Various programming examples show that the usage of logic languages is a major step forward in parallel programming.

In a case study we describe a novel approach to the parallelization of Buchberger's Gröbner bases algorithm using a medium grain pipe-line principle for the polynomial reduction. The approach is implemented in STRAND88 on a 17 transputer distributed memory system and shows a remarkable speed-up.

Contents

List of Figures

Chapter 1

Concurrent Logic Programming: An Overview

1.1 Introduction

1.1.1 Advantages of Logic Programming

Logic programming languages have the following advantages:

- Advantages of high level programming languages.

  The programs are closer to the specification of the problems. Ideally, we only need to describe the problems for getting a logic program. This leads to a smaller programming effort for getting a correct, running program.

- The style is more natural.

  The programs are written in a top down style. For example, for solving problem A we have to solve problems B and C. This leads to the logic program $A :- B, C$.

- Most of the parallelism is already contained in the problem description.

  In principle, the subproblems B and C of the above example may be done in parallel. There may, however, exist some data dependencies between the various subproblems, that will generate an implicit sequentialization.

  The formulation in a conventional programming language leads to a sequence of sequential steps and will make a subsequent parallelization step difficult.

1.1.2 Parallelism in Prolog

In Prolog, there are several sources of parallelism:

- **OR-Parallelism:**

  The heads of all clauses that are possibly appropriate for a goal may be simultaneously unified with the goal. Afterwards, all successfully unified clauses may be handled independently and in parallel. Hence, all possible alternatives may be simultaneously investigated.

- **AND-Parallelism:**

  In this form of parallelism, all goals that still have to be fulfilled are evaluated in parallel. Hence, all remaining subproblems of the original problem may be simultaneously investigated.

- **Data Parallelism:**

  This form of parallelism is closely related to streams, i.e. lists with known elements at the beginning and an unknown rest. One process (producer) generates elements of the stream one after the other. Simultaneously, a second process (consumer) uses the known elements of the stream.

- **Parallel Unification:**

  It is possible to use parallelism during the unification of the heads. The resulting parallelism is very fine grain. This may be done by utilizing specialized unification hardware.

We describe the different kinds of parallelism by a small example.

$$a(X,Y,Z) \quad :- \quad b(X,S), c(S,Y), d(Z). \tag{1.1}$$

$$a(i,X,Y) \quad :- \quad e(X,Y). \tag{1.2}$$

AND Parallelism:

In the first clause we may solve the goals b, c and d in parallel. This is called AND parallelism.

In the goal b and c we use the same variable S. If we find a solution for b and c in parallel, the common variable S must have the same value. It is a big problem to guarantee this consistency in an AND parallel language.

Goal d has no common variables with a and b. Thus, this part may be done in parallel with the other ones without problems.

OR Parallelism:

If we want to find a solution for a goal a, we may evaluate the first and the second clause simultaneously. If one clause succeeds we return the solution gained from this clause. So we compute all solutions of a simultaneously, instead of one after the other with the use of backtracking in sequential Prolog. If more clauses return a solution we get them in an arbitrary order.

The advantage of this method is that all possible solutions are found, even if some computations will run infinitely.

A disadvantage is that most of the programs written in conventional Prolog assume an internal top-down evaluation. Trying to find additional solutions will generate infinite computations in several cases. This will fill the memory and block the performance of processors.

Data parallelism:

If we want to evaluate b and c in parallel we have a data dependence S between these two goals. If goal (b) generates a value for S and (c) will use it, b has to be computed first.

If S is a big data structure (for example a list) it is possible that b and c work simultaneously on different parts of this structure. Here b successively produces values in S and c will use the computed values. Thus, b and c will work in parallel. This is called data parallelism.

1.2 Full Prolog

There are many problems in parallelizing standard Prolog programs:

- Most of the Prolog programs assume the *top-down, left-right* semantics.
- Detection of infinite loops.
- Side effects like the cut operator "!" or operations for manipulation of the database assume an explicit order.
- Collision of variable bindings. The binding of the variables in the various AND goals should be consistent.

We will give an overview on the methods used for parallelizing standard Prolog. A detailed description of the methods and systems may be found in [Conery, 1987]

1.2.1 Restricted AND Parallel Prolog

For avoiding the collision of variables in parallel tasks a restriction is made.

Two AND goals are only done in parallel iff:

- no common variables are used or
- the common variables are bound to a fixed value.

One system of this type is Parsytec Prolog [Penner and Klingler, 1989]. Here, for efficiency reasons, also a syntactical distinction between a sequential and *possible* parallel AND is made. The user has to decide which parts may be evaluated on different nodes.

1.2.2 OR Parallel Prolog

OR processes can be thought of as independent interpreters created to solve a goal statement consisting of only one literal. All solutions to the literal are sent in messages back to the parent process. More details may be found in [Ramakrishnan, 1990, Conery, 1987]

We distingush a few different models:

The AND-OR Tree Model

An AND-OR tree [Conery, 1983] for a logic program has the query as the root OR-node. An OR-node has one child AND-node per rule whose head unifies with this goal. A process for each node is generated. A child AND-node creates an OR-node for the left-most body literal in the goal. For each answer found, an additional OR-node is generated for the next body literal.

The Reduce-OR Model

This model exploits both AND and OR parallelism. It was proposed by [Kale, 1987]. For efficiency reasons, it is restricted to independent AND parallelism. For each clause, a data join graph is created that shows the data dependences between literals in the body of the clause. Whenever an OR process returns a result, it is joined with the other ones in the AND-process and new OR processes are created.

1.2.3 Distributed Search

Once an interpreter selects a literal from a statement, it must find every clause with a head that unifies with the selected literal. The search in the database can be done in parallel.

This method is used in systems desinged for applications where large programs are splited into smaller sections and stored in the local memory of the nodes of a multiprocessor system.

The first system of this kind was PRISM [Kasif *et al.*, 1983].

1.3 Extended Prolog

An alternative way for introducing parallelism in Prolog is to extend the language with explicit constructions for creating and synchronizing concurrent processes. Also constructs for communication between the different processes are used.

One generates a parallel task by distinguishing between a parallel and a sequential composition of goals.

1.3.1 Communicating Sequential Prolog (CS-Prolog)

CS-Prolog [Futo and Kacsuk, 1989] adds a number of built-in predicates to a sequential Prolog in order to introduce parallelism. The most important ones are:

new(Goal, Processname) creates processes (Prolog goals) dynamically during a program run. One may refer to this process by the *Processname* specified.

send(Message, Processlist) sends a message to all processes specified by the *Processlist*.

wait_for(Term) suspends a process until it receives a message that unifies with $Term$.

There arises the problem of how to cope with backtracking in such a distributed system. In global backtracking, a process passes one or more communication points and thereby directly affects the execution of other processes. Whenever the backtrack reaches a point where the process had sent messages to other processes, *anti-messages* are sent to those processes. This effects a backtrack of the other processes, too.

1.3.2 Delta-Prolog

Delta-Prolog [Pereira and Nasr, 1984] has two complementary constructs for distributive synchronization. These are goals of the form "T!E:C" or "T?E:C", where T is any term, E names the event and C is a predicate condition.

A selectable goal G!E:SC at process one may be reduced iff a complementary goal G'?E:RC in an other process may be selected, G and G' are unifiable and the conditions SC, RC are satisfied. Communication is done in both direction in the form of the unifier of G and G'.

1.4 Guarded Horn Clause Languages

Detailed information about these languages and their applications may be found in [Shapiro, 1987]. An overview is [Takeuchi and Furukawa, 1986].

1.4.1 Common features

Guarded Horn clause languages differ from Prolog in two main aspects:

1. Syntactically: A program is a finite set of guarded Horn clauses. A guarded clause is a universally quantified Horn clause of the form:

   $$H :- G_1, ..., G_n | B_1, ..., B_m. \qquad (n, m \geq 0)$$

   "|" is called *commit* operator. $G_1, ..., G_n$ are the guards and $B_1, ..., B_m$ is the body part. H is called the head of the clause. A set of clauses sharing the same predicate symbol H with the same arity is called the definition of this predicate.

2. Semantically: Backtracking is omitted. All guards are evaluated as OR parallel processes. If one guard evaluates to true, only this clause is chosen. The body is evaluated as an AND parallel process.

   For performing synchronization between parallel processes, a distinction between input and output variables is made. In this respect, the languages of the guarded Horn clause type show differences.

Guarded Horn clause languages provide data parallelism. For communication between two parallel processes, the concept of "stream" is used.

The Stream Concept

A stream can be thought of as a potentially infinite list. The elements of a stream are computed by a process P_1 one after the other. Thus, at a certain state, a stream is a list where the first elements are instantiated and the tail is a free variable. A process P_2 that uses the stream may use the first, known elements. If it needs access to later, unknown values, process P_2 waits until they are instantiated by P_1, too. Thus, both processes P_1 and P_2 may work on different parts of the stream in parallel.

For showing the difference between the various guarded Horn clause languages we describe the merge of two input streams to one output stream. The elements that will come first are chosen first. The order of the elements inside each input streams will be preserved in the output stream. The order between elements from the different streams is arbitrary.

1.4.2 Parlog

Parlog [Clark and Gregory, 1984] clauses declare arguments of predicates as "input" (?) or and "output" (^), respectively. They are used to perform synchronization between parallel processes. A distinction between one way unification for input variables and dual unification for output variables is made. If input variables have to be instantiated an explicit unification primitive "$\leftarrow$" is used.

```
mode merge(?,?,^).

merge([X|Xs],Ys,[X|Zs]):- true | merge(Xs, Ys, Zs).
merge(Xs,[Y|Ys],[Y|Zs]):- true | merge(Xs, Ys, Zs).
merge([],Ys,Ys) :- true | true.
merge(Xs,[],Xs) :- true | true.
```

For choosing the granularity of programs a distinction between a sequential and a parallel AND is made.

1.4.3 Guarded Horn Clause (GHC)

In GHC [Ueda, 1985], no additional construct for input and output is added. For performing synchronization, we distinguish between the active and the passive part of a clause.

- The passive part of a clause cannot export any bindings to the caller of this clause. This part is built by the head of the clause and the guard statements.

- The active part of a clause (body) cannot export any bindings to the passive part of a clause. For performing unification to the output the equality "=" body command is used.

```
merge([X|Xs],Ys,Z):- true | Z=[X|Zs], merge(Xs, Ys, Zs).
merge(Xs,[Y|Ys],Z):- true | Z=[Y|Zs], merge(Xs, Ys, Zs).
merge([],Ys,Z) :- true | Ys=Z.
merge(Xs,[],Z) :- true | Xs=Z.
```

1.4.4 Concurrent Prolog

In Concurrent Prolog [Shapiro, 1983], we annotate variables as read only ("?") to perform data synchronization between parallel processes. The annotation can be attached to any variable in a term.

```
merge([X|Xs],Ys,[X|Zs]):- true | merge(Xs, Ys?, Zs).
merge(Xs,[Y|Ys],[Y|Zs]):- true | merge(Xs?, Ys, Zs).
merge([],Ys,Ys) :- true | true.
merge(Xs,[],Xs) :- true | true.
```

For controlling the placement of distributive processes, the annotation "@ *node*" is used. The placement is done dynamically at runtime.

Flat Concurrent Prolog

Flat Concurrent Prolog is a subset of Concurrent Prolog in which the guard parts are restricted to specific system predicates. Since no general computation is allowed in a guard, the computation structure is always flat. This reduces the complexity of an implementation of the language.

1.4.5 Strand

The Strand programming language [Foster and Taylor, 1989] uses only pattern matching and assignments instead of unification for improving the performance. This language will be described in the next chapter in more detail.

```
merge([X|Xs],Ys,Z):- true | Z:=[X|Zs], merge(Xs, Ys, Zs).
merge(Xs,[Y|Ys],Z):- true | Z:=[Y|Zs], merge(Xs, Ys, Zs).
merge([],Ys,Zs) :- true | Zs:=Ys.
merge(Xs,[],Zs) :- true | Zs:=Xs.
```

1.5 Comparison of the Approaches

1.5.1 Input/Output Variables

In logic programming (not in Prolog), one has the advantage that a written program will solve a problem, independent of which parameter is instantiated or free. One of the restrictions used in the guarded Horn clause approach is that variables have to be specified as input or output.

If one writes a program, one has a specification of the problem given. This consists of an input and an output description. The design of a program will be influenced by the use in only this one direction. Also, most of the Prolog programs will be applicable in one direction only. With the use of evaluation of mathematical functions (Prolog "is" predicate) only one direction is possible. Also a lot of programs will generate infinite loops, if they are called in a different order than proposed by the writer. Examples are most of the programs using lists.

Thus, it will be no severe restriction if the Prolog variables are declared as input or output variables.

1.5.2 Backtracking versus Guards

For finding a solution, Prolog uses backtracking if a clause fails. For solving problems in a constructive way, only one solution has to be found. Thus, Prolog uses the "cut" construct to fix this solution and to supress the computation of more solutions. Many Prolog programs will not work correctly without it (i.e negation by failure (Prolog NOT) is defined by using the cut) or will run infinitely. Prolog clauses must be ordered for finding the solution.

Logic languages of the guarded Horn clause type are restricted to only one solution. The execution of all guards in parallel has the advantage that, whenever one guard may succeed, it will be found independently of the order.

In most of the programs the programer only wants to make a case distinction between various possibilities and choose the best one. Therefore a simple guard execution is enough. Backtracking is not necessary.

1.5.3 Unification versus Pattern Matching

If one has a distinction between input and output values a simple matching algorithm is enough to determine the instantiation of the variables. Unification may be replaced, at compile time, by a set of variable bindings.

1.5.4 Usability in Symbolic Computation

In algorithms for Symbolic Computations, most of the time we have to handle big data structures. The inherent parallelism is hidden in these structures. Thus, we need a programming language that supports data parallelism and communication between parallel processes. Also, most times, we only have to construct one unique solution.

Full Prolog

The approach of parallelizing full Prolog is not usable beacause the respective languages do not support data parallelism and communication. OR parallelism will not pay off because the solutions should be unique.

Extended Prolog

In these languages explicit communication is used. This low level aproach for communication discards the advantages of programming in a high level language like sequential Prolog where the problem description will be the executable program.

Guarded Horn Clause Languages

These languages support communication and data parallelism in a very high level form. All advantages of a high level language are kept. Thus, we will use this kind of logic languages to study parallelism in Symbolic Computation.

Chapter 2

Concurrent Programming in Strand

2.1 The Strand Programming Language

A commercially available language of the guarded Horn clause type is Strand [Foster and Taylor, 1989]. For avoiding the problems that occur in the implementation of a concurrent logic programming language, a few restrictions are made.

- Strand is flat. This means that the guards have to be built-in test functions.
- Unification is replaced by a matching algorithm.
- Strand has an assignment statement that compensates the absence of unification.

With these restrictions in mind, one actually should not call Strand a logic programming language because unification is an essential part of logic programming. However, in practical use, we may write programs in Strand in the same way as in all other logic languages of the guarded Horn clause type.

2.1.1 Syntax

A Strand clause may be written in one of the following forms:

$H :- G_1, ..., G_n \| B_1, ..., B_m.$	if $n, m > 0$,
$H :- G_1, ..., G_n \| true.$	if $n > 0$ and the body is empty,
$H :- B_1, ..., B_m.$	if $m > 0$ and the guard is empty,
$H.$	if the guard and the body is empty,

where

- $H(T_1, ..., T_n)$, $n \geq 0$, is called the "head" and consists of a name H and the formal argument list $T_1, ..., T_n$.
- $G(T_1, ..., T_n)$, $n \geq 0$, is called a "guard" and consists of a built-in guard name G and the actual argument list $T_1, ..., T_n$.
- $B(T_1, ..., T_n)$, $n \geq 0$, is called a "body" where B can be a built-in body call or a user defined procedure name and the actual argument list $T_1, ..., T_n$.

2.1.2 Matching

For selecting the next clause, Strand uses a matching algorithm instead of unification in logic languages.

The terms in the head of the clause are tried to match with the arguments of the call. During matching, we use the following rules:

Succeed: A term T in the head argument matches with a term S in the call argument, iff

1. T is a variable and S is an arbitrary term or
2. T is a term $t(t_1, ..., t_n)$ and S is a term $s(s_1, ..., s_n)$ and $t == s$ and all t_i match with s_i.

Suspend: If a variable in the call argument S has to be matched with a non-variable term T in the head argument, we have to wait until the variable will be instantiated. Then we can continue matching.

Fail: otherwise.

For instantiating a variable with a value, Strand uses the assignment statement ":=". Strand has the single assignment rule. A variable can be instantiated at most once, otherwise an error occurs.

2.1.3 Data Types

The basic data structure of Strand is the term. A term is either a variable, a structured type (tuple, list) or any basic type.

Variables are denoted by a sequence of alphanumeric characters beginning with an uppercase letter or the underscore "_" character. Strand is not typed. A variable may be instantiated by any term.

In Strand we have a few basic and two structured data types to built any data structure.

Simple Types

The simple types are:

- strings: a, "V", '1m 3', ...
- integers: 1, -23, ...
- reals: 1.0, -2.3234E-4, ...

Tuples

A tuple consists of a fixed number of elements. Each element of the tuple can be any term.

A tuple may be written in two ways:

1. $\{E_1, ..., E_n\}$ (e.g. $\{$*request, X*$\}$, $\{f, \{g, X\}, a\}$)
2. $E_1(E_2, ..., E_n)$ if E_1 is a string (e.g. *request(X)*, $f(g(X), a)$)

List, Stream

The data type list (also called stream) consits of a head and a tail (e.g. $[X|Xs]$) or the empty list [ ]. The elements can be arbitrary terms. If the tail is a variable with a value not yet known we call it stream.

2.1.4 Abbreviations

Mode declarations

Since we have no unification, it is not possible to instantiate a variable in the call argument with a term in the head of the called clause. This would result in a conflict with the suspend rule during matching. Hence, instantiation may be only done by the assignment statement ":=". As abbreviation, we may declare arguments in the head of a clause as output. This is done with the "-mode" annotation. A term at the position of an output declaration will be assigned to a variable in the call argument.

For example

```
-mode prog(?,^).
prog(c,[]).
prog(X,X).
```

is an abbreviation for:

```
prog(c,X):- X:=[].
prog(X,Y):- Y:=X.
```

Equality of Terms

With unification we may use the same variable in the head twice. If one occurrence of the variable is bound to a term and the other occurence is bound to the same term, unification succeeds. If we want to test two terms to be equal in Strand, we need the equal guard command "==".

As abbreviation for this guard command, we may use the same variable name for two input variables. Then the two terms will be compared.

For example

```
equal(X,X).
```

is an abbreviation for

```
equal(X,Y):- X==Y | true.
```

Using these abbreviations, we may write a program in the same way as with the use of unification. The simple matching algorithm will be almost no restriction in practical use as logic programming language.

2.2 Basic Concepts

2.2.1 Reading Instructions

For making parallel Strand programs easier readable, we give an "informal semantics" by introducing a few "reading instructions" for the basic Strand constructs.

Parallel logic programming languages may be interpreted in three different ways:

- Logical interpretation,
- Sequential algorithmic interpretation,

- Parallel algorithmic interpretation.

$$p(X,Y) :- s_1(X),\ s_2(Y).$$

Logical interpretation: If $s_1(X)$ is true and $s_2(Y)$ is true then we know that $p(X,Y)$ is true.

Sequential algorithmic interpretation: In order to decide $p(X,Y)$ it is sufficient to decide $s_1(X)$ and $s_2(Y)$.

Parallel algorithmic interpretation: For deciding $p(X,Y)$ we decide $s_1(X)$ and $s_2(Y)$ in parallel.

$$-mode\ p(?,\hat{}).$$
$$p(X,Y) :- s_1(X),\ s_2(Y).$$

Logical interpretation: Same as before.

Sequential algorithmic interpretation: In order to produce Y for given X such that $p(X,Y)$ it is sufficient to decide $s_1(X)$ and produce a Y that satisfies $s_2(Y)$.

Parallel algorithmic interpretation: For producing Y such that $p(X,Y)$, we require X. Deciding $s_1(X)$ and producing Y satisfying $s_2(Y)$ may be done in parallel.

$$p(X,Y) :- g_1(X) \mid s_1(Y).$$
$$p(X,Y) :- g_2(X) \mid s_2(Y).$$

Logical interpretation: If $g_1(X)$ and $s_1(Y)$ is true or $g_2(X)$ and $s_2(Y)$ is true then we know that $p(X,Y)$ is true.

Sequential algorithmic interpretation: In order to decide $p(X,Y)$ it is sufficient to decide $g_1(X)$ and decide $s_1(Y)$ or decide $g_2(X)$ and decide $s_2(Y)$.

Parallel algorithmic interpretation: For deciding $p(X,Y)$ we decide the guards $g_1(X)$ and $g_2(X)$ in parallel. If only $g_1(X)$ is true then we decide $s_1(Y)$. If only $g_2(X)$ is true then we decide $s_2(Y)$. If $g_1(X)$ and $g_2(X)$ is true then we decide randomly either $s_1(Y)$ or $s_2(Y)$, but not both. There is no backtracking.

$$p(X, Y) :- s_1(X, Z),\ s_2(Y, Z).$$

Logical interpretation: If there exists a Z such that $s_1(X, Z)$ is true and $s_2(Y, Z)$ is true then we know that $p(X, Y)$ is true.

Sequential algorithmic interpretation: In order to decide $p(X, Y)$ it is sufficient to produce a Z that satisfies $s_1(X, Z)$ and $s_2(Y, Z)$.

Parallel algorithmic interpretation: For deciding $p(X, Y)$ we start $s_1(X, Z)$ and $s_2(Y, Z)$ in parallel. If s_1 produces a Z satisfying $s_1(X, Z)$ we complete s_2 by checking that the Z we found satisfies $s_2(Y, Z)$. If s_2 produces a Z satisfying $s_2(X, Z)$ we complete s_1 by checking that the Z we found satisfies $s_1(Y, Z)$. Z can be produced only once, either in s_1 or in s_2, otherwise an error occures. The occurence of Z is called data dependence between s_1 and s_2.

$$p(X, Y) :- s_1(X, Z),\ s_2(Y, Z).$$

$$-mode\ s_1(?, \hat{}).$$
$$-mode\ s_2(?, ?).$$

Logical interpretation: Same as before.

Sequential algorithmic interpretation: In order to decide $p(X, Y)$ it is sufficient to produce a Z that satisfies $s_1(X, Z)$ and check $s_2(Y, Z)$.

Parallel algorithmic interpretation: For deciding $p(X, Y)$ we start $s_1(X, Z)$ and $s_2(Y, Z)$ in parallel. When s_1 produces a Z satisfying $s_1(X, Z)$ we complete s_2 by checking that the Z we found satisfies $s_2(Y, Z)$. For completing s_2 we have to wait until s_1 produces a value for Z. The occurence of Z is called synchronization and communication variable from s_1 to s_2.

$$p([X|Xs]) :- s(X),\ p(Xs).$$

Logical interpretation: If $s(X)$ is true and $p(Xs)$ is true then we know that $p([X|Xs])$ is true.

Sequential algorithmic interpretation: In order to decide $p([X|Xs])$ it is sufficient to decide $s(X)$ and decide $p(Xs)$ recursively.

Parallel algorithmic interpretation: For deciding $p([X|Xs])$ we require a "non empty stream" $[X|Xs]$. This means that we require that the part of the input term $[X|Xs]$ to be actually available comprises at least X. For deciding $s(X)$ we do not have to wait until Xs is available. As soon as X is available we decide $s(X)$ and start deciding $p(Xs)$ in parallel.

$$
\begin{array}{l}
-mode\ p(\hat{\ }). \\
p([X|Xs]) :-\ s(X),\ p(Xs).
\end{array}
$$

Logical interpretation: Same as before.

Sequential algorithmic interpretation: In order to produce an $[X|Xs]$ that satisfies $p([X|Xs])$ it is sufficient to decide $s(X)$ and produce the rest of the stream Xs satisfying $p(Xs)$ recursively.

Parallel algorithmic interpretation: For producing $[X|Xs]$ such that $p([X|Xs])$ we generate a stream cell $[X|Xs]$ first. This means that we generate an empty box for the first element X and for the not known rest of the stream Xs. The box X may be used by other processes immediately. Afterwards, we decide $s(X)$ with respect to the content of the box X and generate the rest of the stream Xs satisfying $p(Xs)$ in parallel.

2.2.2 Communication

In concurrent programs, the various tasks will communicate with each other. In Strand, communication is done via shared variables. Each variable can be instantiated only once (single assignment rule). This may be done in any task that knows the variable.

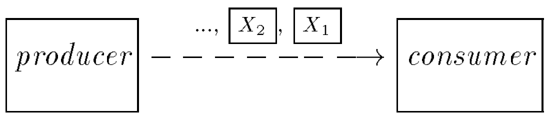
We describe the basic communication techniques that are used in concurrent programming.

Channel

A communication channel is represented via a shared stream. One process produces the values of the stream, and the other one receives the values from the stream.

```
main():-
  producer(Channel),
  consumer(Channel).
```

```
-mode producer(^).
producer([X|Xs]):-
  fill(X),
  producer(Xs).

consumer([X|Xs]):-
  use(X),
  consumer(Xs).

-mode fill(^).
-mode use(?).
```

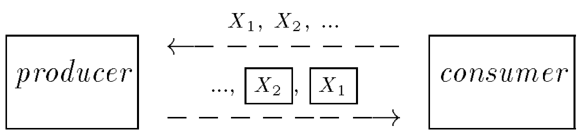


In the above example the producer program fills successively values into the shared stream *Channel* and sends them via this stream to the consumer.

Backward Communication

If the consumer wants to tell the producer that it is able to receive data it has to send a request form (empty variable) to the producer. Now the producer may fill it with a value and send it back to the consumer.

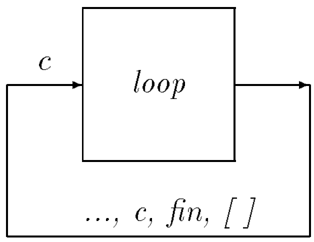
```
main():-
  producer(Channel),
  consumer(Channel).

producer([X|Xs]):-
  fill(X),
  producer(Xs).

-mode consumer(^).
consumer([X|Xs]):-
  use(X),
  consumer(Xs).

-mode fill(^).
-mode use(?).
```



Communication Loop

In Strand it is possible to write a program that uses its own output value as input argument. This will create a loop of communication. We may send values in a circle through the processes.

```
main():-
  loop([c|X],X).

-mode loop(?,^).
loop([fin|_],[]).
loop([c|Xs],[Y|Ys]):-
  create(Y),
  loop(Xs,Ys).
```



Here, the loop runs as long as a *c* is created. If a *fin* arrives, the communication stream is closed and the program terminates.

2.2.3 Objects

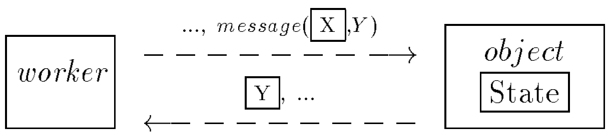
Object oriented programming is an important technique in computer science.

An object in Strand will be represented as a process with at least one input stream. Every element of the stream denotes a message to the object. If we want to get a result from the object, we have to send a free variable to the object. The object may fill the variable with the result value. In this way, the result will be sent back. Internal states are represented via local variables of the object.

For creating an object we create a new parallel process. The internal variables have to be instantiated with the initial values. The object and the process that uses it, will share the message stream. We may cancel an object by closing the input stream.

```
main():-
  worker(M),
  object(init,M).            % create new object
```

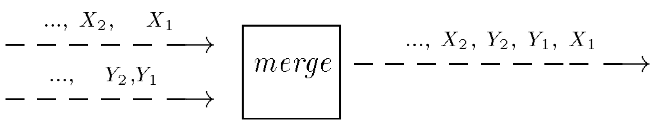
```
worker(Send):-
  request(X),                   % request to object
  Send:=[message(X,Y)|Sends], % send message
  use(Y),                       % use answer
  worker(Sends).

object(_,[]).                   % cancel object
object(State, [message(X,Y) | Request]):-
  action(State,NewState, X,Y),
  object(NewState,Request).
```



2.2.4 Merge of Streams

If we want to use one object by more than one worker processes, the input streams of all workers has to be merged to one output stream. Then the object can serve each worker when it sends a message. If more workers send a message simultaneously, they are merged together in an arbitrary order.

```
-mode merge(?,?,^).
merge([X|Xs],Ys,[X|Zs]):-
   merge(Xs,Ys,Zs).
merge(Xs,[Y|Ys],[Y|Zs]):-
   merge(X,Ys,Zs).
merge([],[],[]).
```



In the merge process, each input stream is tested whether a new element of the input stream is available or not. When a new element is available, it is added to the output stream and the remainder streams are merged again. If all input streams are closed then we close the output stream, too.

For merging an arbitrary number of streams, we interlock a set of merge processes.

2.2.5 Distributing Processes

In Strand, all AND clauses are executed in parallel internally. In practical use, we want to select which process should be mapped to a different node and which one should be executed *quasi-parallel* on a single node.

In order to map the parallel tasks on various nodes, Strand has the annotation @*Node*. The annotated clause is then computed on the defined node. The shared variables are now communication channels between the different nodes.

We distinguish two methods in distributing processes.

Static Mapping

A process is placed to a node with a fixed identity.

For example:

compute(X)@10 runs on the node number 10,

compute(Y)@boss runs on the master node. This node is connected to the host and does all the work related to input and output.

Dynamic Mapping

Processes are placed to a node relative to the possition of the actual node, depending on the underlying machine topology.

Examples are:

compute(X)@fwd runs on the next node in a ring topology,

compute(Y)@north runs on the node above in a torus topology.

2.3 Main Methods in Parallelizing

In Symbolic Computation there are many problems that need much computing time. Most of them have an exponential or even higher complexity. The computation of "realistic" examples will be not possible on today's computers. One way to get a higher performance is to use a parallel computer and parallelize the programs.

Most of the problems have much inherent parallelism of various nature. We give a few methods how to parallelize algorithms from Symbolic Computation in Strand.

2.3.1 Parallel Evaluation

A simple, but often usable, method in Symbolic Computation is, taking all of the independent parts of the input data structure and evaluate all of them in parallel.

Examples are :

- vector and matrix operations,
- polynomial operations.

Vector Addition

As a representative of the method of parallel evaluation, we describe the addition of two vectors (Figure 2.1).

```
-mode(?,?,^).
add_vec([],[],[]).
add_vec([X|Xs],[Y|Ys],[Z|Zs]):-
  add_vec(Xs,Ys,Zs)@fwd,
  Z is X+Y.
```

Figure 2.1: Addition of two Vectors

Here we split the two vectors into the single elements and return a vector as result immediately. The individual values of the result vector are not known at this stage. Then we add the corresponding elements of the input vector in parallel. The values of the output vector become bound to the corresponding positions of the result vector, when the invidual processes adding the single elements are finished.

2.3.2 Divide and Conquer

For achieving a better complexity, many algorithms use the divide and conquer strategy.

The idea of parallelizing is clear. The input is split into independent subproblems of the same type and the subproblems are placed on different nodes. Each of them will call the procedure again until a base case is reached. This leads to a tree structure of parallel processes.

Examples are:

- Fast multiplication (Karatsuba algorithm for integer multiplication , Strassen algorithm for matrix multiplication),
- Quicksort,
- Symbolic differentiation and integration.

Symbolic Differentiation

The following rules are used to derive a formula with respect to the indeterminate x:

- $x' \rightarrow 1$
- $constant' \rightarrow 0$
- $(U+V)' \rightarrow U' + V'$
- $(U*V)' \rightarrow U*V' + U'*V$

This leads immediately to the corresponding parallel divide and conquer algorithm (Figure 2.2).

```
-mode derive(?,^).
derive(U+V,Du+Dv):-
  derive(U,Du)@left,
  derive(V,Dv)@right.

derive(U*V,U*Dv+V*Du):-
  derive(U,Du)@left,
  derive(V,Dv)@right.

derive(x,1).     % variable
derive(C,0):-    % constant
  otherwise | true.
```

Figure 2.2: Symbolic Differentiation

2.3.3 Filtering, Pipelining

An other technique in Symbolic Computation is to test or modify the input with respect to a set of fixed values. After the test (modify) part we may append some of the tested values to our set in order to use it for the rest of the input.

There are two types of methods using this scheme.

1. One direction pipe-line.
2. Usage of the result to create new inputs.

One Direction Pipe-line

This method may be used to solve a problem were we want test a given input set with respect to a test set with fixed elements. For parallelizing this method, we put every element of the test set to one node ordered in a pipe. The input values are driven subsequently through the pipe. Each intermediate process takes the first element from its previous neighbour and tests, modifies or cancels it. The result is sent to the next process. Then it waits for the next input to compute the next result. When a value reaches the end of the pipe-line it is possible to extend the test set by simply creating a new process that holds the new value at the end of the pipe. Thus, the future input values will be tested with the new value, too.

Examples of the usage are:

- Sequence of polynomial operations
- Sieve of Erathostenes to compute prime numbers

Backward Communication

In some cases we have to use the result of a test to generate new input values. A typical example are the Critical/Pair-Completion procedures. Here a set of critical pairs is built and tested with respect to a given set. Pairs that can not be reduced are appended to the test set, and new critical pairs are built.

The parallelization is similar to the previous method. We only have to send the computed values back through the pipe. In each pipe process, we generate the new inputs and send them back to the front end. The front process takes the new input values from the pipe and sends them through the pipe again.

Examples of the usage are:

- Knuth-Bendix procedure,

- Gröbner bases algorithm.

Sieve of Erathostenes

As programming example we use the parallel version of the Sieve of Erathostenes algorithm (Figure 2.3),[Shapiro, 1985] for producing an infinite list of prime numbers. We will describe it in more detail because the method of filtering will influence the agorithm in our case study (compare 5.3).

This algorithm produces a stream of integers one after the other. All primes that are smaller than the current integer are stored in a table. For testing if a number is prime, it is sufficient to test if a smaller prime number divides the number. If it can be divided by any of the primes stored we cancel the number. If not, we have found a new prime number and add it to the store. Then we can test the next integer.

Instead of checking a single integer p against every prime in the table P, a whole stream of integers may be simultaneously checked against all primes in P by distributing the elements of $P[1..N]$ over a pipe-line of parallel processes:

$$\boxed{P[1]} \longrightarrow \boxed{P[2]} \xrightarrow{p} \boxed{P[3]} \longrightarrow \ldots \longrightarrow \boxed{P[N]}$$

Every process i stores the first integer it receives from process $(i-1)$ as a prime in $P[i]$. From now on, this process discards every integer p that is divisible by $P[i]$ from the pipe and forwards all other integers to process $(i+1)$. The resulting prime table is then the collection of all primes $P[i]$ stored in the pipe-line.

Every intermediate filter process has to decide if it has to cancel the integer it received or forward it to the next process. Before this, the test whether a prime number divides the integer, has to be executed. The next process has to wait until the test is done and the new value is sent. This forces an (unnecessary) synchronization of the intermediate filter processes.

For avoiding the synchronization, we create a separate pipe for each number. Every filter process sends an unbound variable to its successor, the moment it received a variable from the previous process (Figure 2.4). The next process may start the test procedure with this variable immediately. This allows a test of each number in parallel.

Synchronization is moved to the parallel test processes. Here we have to wait until the input variable becomes a concrete integer. Then the test may be finished. Depending on the result, we may assign the variable, sent to the next test process, to the tested integer value if the internal prime does not divide the integer, otherwise to zero. This allows the next test process to continue testing.

```
-machine(ring).

% compute the stream of all prime numbers
primes(Ps):-
   integer_list(2,Is), % front end process
   sieve(Is,Ps)@fwd.   % back end process

% generate a stream of integers
-mode integer_list(?,^).
integer_list(I,[I|Is]):-
   Int is I+1,
   integer_list(Int,Is).

% the moving back end of the pipe
-mode sieve(?,^).
sieve([Prime|Is],[Prime|Ps]):-
   filter(Prime,Is,Rest),
   sieve(Rest,Ps)@fwd. % place on next node

% filter multiples of Prime from the stream
filter(Prime,[I|Is],Rest):-
   R is I // Prime,
   filter(R,Prime,[I|Is],Rest).

-mode filter(?,?,?,^).
filter(0,Prime,[_|Is],Rest):-
   filter(Prime,Is,Rest).
filter(R,Prime,[I|Is],[I|Rest]):-
   R =\= 0 | filter(Prime,Is,Rest).
```

Figure 2.3: Sieve of Erathostenes

```
-machine(torus).

primes(Ps):-
   integer_list(2,Is),
   sieve(Is,Ps)@west.

-mode sieve(?,^).
sieve([Prime|Is],[Prime|Ps]):-
   Prime =/= 0 |
     filter(Prime,Is,Rest),
     sieve(Rest,Ps)@west.
sieve([0|Is],Ps):-
   sieve(Is,Ps).

-mode filter(?,?,^).
filter(Prime,[I|Is],[I1|Rest]):-
   filter(R,Prime,[I|Is],Rest)@north,
   R is I // Prime,
   test(R,I,I1).

% test, if remainder is 0
-mode test(?,?,^).
test(0,I,0).
test(R,I,I):-
   R =\= 0 | true.
```

Figure 2.4: More Parallelism in the Sieve of Erathostenes

During the flow through the pipe, the computed numbers are not ordered. It is possible that a number has passed more intermediate filter processes than a smaller one. The final synchronization has now to be done at the end of the pipe. There the smallest prime will be picked up from the stream and a new filter process is created.

The additional parallelism will not pay off in the case of testing integers where each test needs the same time. But it will pay off, if we apply the method to more complicated test algorithms for avoiding bottlenecks. This is the case in a Critical-Pair/Completion procedure. There each test will take a long time and the execution speed of a single test will vary widely.

2.3.4 Manager - Worker

The manager - worker approach is used for parallelizing algorithms, where a big set of independent parts and a limited number of nodes is used. The computing time of the various parts may vary and each node should be equally occupied.

Many problems that may be parallelized with other methods, may also be parallelized using the manager - worker approach. In practice, the method is very often used for avoiding mapping problems.

Typical examples may be found in geometric problems:

- Ray Tracing,
- Collision Detection.

Collision Detection

As example we show a parallel solution to the collision detection problem (Figure 2.5). For detecting, if an object collides with any other objects, we have to test the collision of the object with respect to all other objects. This may be done in parallel. The computing time of each test will strongly vary.

For parallelizing this problems we will use a set of worker processes. Each of them will do the collision test of two objects. Each worker requests a piece of work from the manager process. The manager responds to the request with the next test objects. If one worker has finished the test it sends the result back to the manager and receives two new objects. This is done until all pairs of objects are tested.

```
collision_check(Obj,Obj_list,Check):-
   set_workers(10, Request),    % spawning a tree of 10 workers
   merger(Request,Req),         % merge the request streams
   collision_manager(Req,Obj,Obj_list,Check)@boss.

-mode set_workers(?,^).
set_workers(0,[]).
set_workers(N,[merge(Req)|Request]):-
  N > 0 |
    worker([],Req)@N,
    N1 is N-1,
    set_workers(N1,Request).

collision_manager([request(O1,O2,Result)|Request],
                  Obj1,[Obj2|Os],Check):-
   O1:=Obj1,
   O2:=Obj2,
   check(Result,Check1,Check),
   collision_manager(Obj1,Os,Request,Check1).

-mode check(?,?,^).
check(save,T,T).
check(collision,_,collision).

-mode worker(?,^).
worker(Done,[request(Obj1,Obj2,Result)| Ws]):-
   data(Done) |
      collision_check(Obj1,Obj2,Result),
      worker(Result,Ws).
```

Figure 2.5: Collision Detection

2.3.5 Speculative Parallelism

Many problems in Symbolic Computation have various methods for solving them. The computing time of the solution strongly varies between the various methods, but we do not know in advance which method will be faster. Furthermore, there are examples that will run infinite time with one method, but compute a solution with another method.

Examples are:

- Automated Theorem Proving
- Gröbner Bases Computation under various orderings.

For parallelizing them we only need to solve the problem using each method on another node. If one method produces a result we take it and cancel all other processes. Thus, we get a solution in the time the fastest method will need.

Automated Theorem Proving

Formulas can be proved directly or indirectly by assuming the negation and finding a contradiction.

```
first_prove(F,Result):-
  prove(F,Result1)@1,
  prove(not(F),Result2)@2,
  first_result(Result1,Result2,Result).

-mode first_result(?,?,^).
first_result(true ,  _  ,true).
first_result(false,  _  ,false).
first_result(  _  ,true ,false).
first_result(  _  ,false,true).
```

Figure 2.6: Speculative Theorem Proving

For parallelizing the proof of a formula F (Figure 2.6), we use two processes:

1. Proof of formula F
2. Proof of formula $\neg F$

Each separate proof is done by the same proof procedure. When the faster procedure is finished we get the result with one of the following rules:

- If procedure 1 returns true then F is valid.
- If procedure 1 returns false then F is not valid.
- If procedure 2 returns true then F is not valid .
- If procedure 2 returns false then F is valid.

Chapter 3

Influence of the Hardware on Algorithm Design

This chapter gives a brief survey on the basic hardware dependencies in the design of parallel algorithms.

3.1 Classification of the Hardware

We list a few criteria for distinguishing different classes of parallel hardware architectures.

3.1.1 Instructions and Data

A widely accepted classification for parallel hardware architectures was given by Flynn. He distinguishes between various architectures according to the number of instructions and the number of different data that are considered at one time moment.

SISD (Single Instruction, Single Data)

This class consists of the classical non-parallel computers. Each instruction is executed one after the other. Also each piece of data can only be manipulated one after the other.

SIMD (Single Instruction, Multiple Data)

In this class, there are most of the numerical supercomputers (vector or array processors). Each instruction may effect more than one datum simultaneously.

MISD (Multiple Instruction, Single Data)

There exists no machine model in this category.

MIMD (Multiple Instruction, Multiple Data)

This is the class of the parallel computers in the narrow sense (multiprocessor machines). A set of instructions is executed in parallel. Each of them will affect different data.

From now on we will only regard the MIMD class.

3.1.2 Memory Organization

There is a close connection between the organization of memory and algorithm design.

Shared Memory

In the shared memory model the entire memory is accessible from each processor in the same way using the same amount of time. The processors and the memory share one communication bus. This bus has a limited capacity. Thus, the number of processors in such a system is limited.

Programming such a system is the most comfortable form of programming parallel computers:

- We may use shared variables for communication between the different processes.
- We need not spend time in communication between two processes. This enables us to use parallelism on a fine grain level.
- All memory and communication models are easily simulated on a shared memory machine.

Distributive Memory

The distributive memory model is the model that allows the cheapest and easiest construction of parallel hardware. Each processor has its local memory exclusively. The various processors are connected via a communication system. If a processor needs data in the local memory of another processor it has to send a request. The requested data will be sent through the communication media.

This model is hard to program:

- All communication has be done by sending messages through a communication medium.
- Communication needs much time. Only coarse grain parallelism can be realized efficiently.

Non-Homogenous Memory

A mixed form beetwen the previous models is a memory model in which we distinguish between two kinds of memory:

1. Each processor has a local memory with fast access.
2. A processor has also access to the memory of the other processors but with less speed. This enables one to communicate directly between different processors.

This leads to a programming model that is between the previous ones:

- Shared variables are possible.
- The communication speed is between the previous memory models. Typically, a medium grain parallelism can be realized in such a model.

3.1.3 Communication Networks

The connection between two nodes has a crucial influence on the execution speed of a distributed system. For communication inside distributed memory machines we need a communication network. There are several possibilities:

Channels

The easiest way to build parallel hardware uses fixed channels between the different nodes. During the communication we have to distinguish between two classes of nodes:

1. Two nodes are neighbours, this means that there exists a direct connection between them.
2. Communication involves intermediate nodes.

The communication between two nodes needs a particular amount of time. The more nodes are involved in one communication, the more computing time is lost. This will decrease the achievable performance of a parallel computer.

The different parts in a parallel algorithm have to be placed as close as possible to nodes that are connected in the same way as the communication stream between the parallel processes will flow.

Switch Networks

By a switch network, one may connect n input with m output nodes. One may communicate between arbitrary two nodes in a direct way by selecting the connecting channel. The communication time between two connectable nodes is always the same.

Using switch networks, the placement of the concurrent parts in an algorithm has no influence on the communication time.

Buses

A bus is a communication facility where, at one time moment, only one node may write data and all others may receive these data. Communication between arbitrary two nodes needs always the same effort. This allows an arbitrary placement of distributed processes.

The problem with buses is that they have a fixed capacity which limits the communication volume of the whole system. In a multiprocessor environment, the bus is the bottleneck and limits the number of possible nodes.

An advantage of a bus is that all nodes may read pieces of data from it simultaneously (broadcasting). A message to all nodes has to be sent only once.

3.2 Granularity of Programs

The hardware used has a strong influence on the granularity possible in a parallel program.

3.2.1 Coarse Grain

In programs with coarse grain parallelism big tasks are distributed. Each individual task needs a long computing time. Only a few parallel processes are created.

A task is a subprocess that is created by some other process. These subprocess works in parallel. For communicating with other tasks, a message is sent.

In coarse grain parallelism, the communication speed is not critical. One may use channels for realizing this communication method. A loosely connected computer system is an appropriate hardware.

3.2.2 Medium Grain

Parallel Functions

A set of function calls may be done on various nodes. The result is sent back to the calling process.

Only the arguments and the result of the functions have to be sent to the various nodes. One may use a distributive memory system and a network for communication.

This method is mainly used for parallelizing functional languages where each argument of a function may be evaluated independently.

Parallel Threats

A set of code blocks is called threat. Most of the parallel extensions of imperative programming languages use this facility.

The communication is done via shared variables. This requires a shared memory hardware for efficient implementation.

3.2.3 Fine Grain

In the fine grain parallelism one distributes different iterations of a loop. The independent parts of a data structure are computed in parallel.

As example we may compute the operations on each monomial in a polynomial in parallel.

For realising fine grain parallelism, we need a fast communication tool. Fine grain parallelism is mainly used on supercomputers.

3.3 Mapping Strategies

If we distribute the different parts of an algorithm we have to decide which part should be computed on which node ("mapping"). For getting a high speed-up in a distributive system each node should be exploited equally.

There are several possibilities for mapping the parallel parts to the various nodes.

3.3.1 Static Mapping

Here the user has to place each task to a fixed node. Thus, the final placement and necessary communication channels are known at compile time.

Static mapping does not allow recursion and process creation at runtime. Thus, this approach will be used only in imperative programming languages.

3.3.2 Dynamic Mapping

In functional or logic programming languages parallel processes are created at runtime.

In a dynamic mapping strategy we may place the distributed tasks on a node relative to the placement of other tasks. Here we assume that the communication channels will follow a virtual topology. The final placement depends on the execution of the program and can only be done at runtime.

3.3.3 Implicit Mapping

Implicit mapping means that the runtime system decides which process has to be placed on which node. It is enough to annotate which processes should be done on the different nodes. Each process will be placed to the node that has the least load.

Implicit mapping is the most comfortable method in parallel programming and will theoretically result in a high speed-up. The problem is that there exist no powerful tools that will do the automatic mapping in an efficient way.

3.4 Conclusion

For developing a portable parallel algorithm that is independent of the underlieing hardware, we have to assume a fictive hardware that may be simulated on a wide range of machines without losing efficiency.

In this paper we use assume a distributive memory machine connected via links as our fictive hardware model. The reason for this choice is:

- A distributed memory may be simulated by a shared memory by simply reserving a different part of the memory for each node.
- The simulation of links is always possible as soon as there is some network facility available that connects each processor with each other.

When we develop an efficient algorithm using this machine model, it will be efficient on all other multiprocessor machines, too.

Chapter 4

Parallelization of the Gröbner Bases Algorithm

4.1 Introduction

The notion of "Gröbner bases" and an algorithm for constructing a Gröbner basis was introduced in [Buchberger, 1965]. Gröbner bases are certain canonical sets for generating polynomial ideals.

Using Gröbner bases, many problems in commutative algebra (theory of polynomial ideals, algebraic geometry) become solvable in an elegant way. A collection of applications may be found in [Buchberger, 1987a]. An introduction to the theory of Gröbner bases is [Buchberger, 1985a].

The problem of constructing Gröbner bases is specified as follows:

Given:

1. A field K and a set of indeterminates $\{x_1, ..., x_n\}$.
2. A set F of polynomials in $K[x_1, ..., x_n]$.
3. An "admissible" ordering $<$ for the power products over $\{x_1, ..., x_n\}$.

Find: A set G of polynomials in $K[x_1, ..., x_n]$ such that:

1. The ideal generated by F is equal to the ideal generated by G.
2. G is a Gröbner basis with respect to the ordering $<$.

Definition: G is a Gröbner basis with respect to an ordering $<$ iff

- G is finite,
- normal forms (w.r.t. $<$) modulo G are unique.

4.2 The Sequential Algorithm

The basic idea of the Gröbner basis algorithm is to perform a "Critical-Pair/Completion" procedure on F.

The algorithm in the sequential form is shown in Figure 4.1.

```
GB(F?,Gˆ):
   G:=F
   Pairs:= critical_pairs(G)
   WHILE Pairs≠ ∅ DO
     choose (f,g) in Pairs
     delete (f,g) from Pairs
     IF not criteria(f,g,G,Pairs) THEN
       h:= S-Polynomial(f,g)
       h':= NormalForm(G,h)
       IF h'≠ 0 THEN
         update_pairs(Pairs,G,h')
         update_basis(G, h').
```

Figure 4.1: Buchberger's Algorithm

The runtime behavior depends strongly on the usage of a few optimization strategies:

1. The order of selecting pairs and the order of polynomials in the basis have a crucial influence. As a rule the pairs with minimal least common multiple of the leading power products should be treated first. The basis should be ordered by the order of the leading power products.

2. Reducing the basis after each polynomial entered may also increase the performance. Thereby, many polynomials in the basis may be even deleted. Such a reduction may initiate a whole cascade of reductions and cancelations.

3. Two criteria [Buchberger, 1985a] are given that detect S-polynomials reducing to zero without actually reducing them. Using these criteria, in favorable situations, only $O(k)$ S-polynomials must be considered instead of $O(k^2)$, where k is the (changing) number of polynomials in the basis.

4.3 Ideas for Parallelization

There are various approaches to the parallelization of Buchberger's algorithm.

- Three different methods for utilizing coarse grain parallelism. Here the different S-polynomials are reduced in parallel.
  1. In the primitive method, we have alternate phases of S-polynomial reduction and basis reduction (see Section 4.4).
  2. A method in which we continuously distribute S-polynomials and the required data to a limited set of processes (see Section 4.5).
  3. A method where we use a shared memory to hold the data. The S-polynomials are computed continuously, too (see Section 4.6).
- Two methods based on fine grain parallelism, which may be used for parallel computers with fast communication access.
  1. In one approach the inherent parallelism in each polynomial operation is exploited (see Section 4.7).
  2. An alternative method based on a generalized notion of determinant polynomials is sketched in [Buchberger, 1990].
- Alternative Methods where we compute multiple Gröbner bases (see Section 4.8).

4.4 Coarse Grain Parallelism: Primitive Method

If one looks to the sequential algorithm the first idea for parallelizing Buchberger's algorithm, of course, is to generate and reduce all S-polynomials simultaneously. In a second phase the basis can be updated and reduced in parallel. Then the new pairs from the updated basis can be generated and reduced again, and so on.

We assume that we have an infinite number of nodes that performs the reduction of an S-polynomial with respect to a fixed set of polynomials in the basis.

4.4.1 Watt's Version of the Algorithm

This method divides the process into alternating phases of S-polynomial computation and reduction of the basis, using parallelism in each stage. It was proposed

```
GB(F?,Gˆ):
   G:=F
   G':=∅
   WHILE G ≠ G' DO

     (* 1st Phase: Reduction of S-polynomials *)

     G':=G; H:=G
     Pairs:= critical_pairs(G)
     FOR ALL (f,g) ∈ Pairs DO
       IF not criteria(f,g,G,Pairs) THEN
         h:= S-Polynomial(f,g)
         h':= NormalForm(G,h)
         if h'≠ 0 then
           H:= H ∪ h'

     (* 2nd Phase: Reduction of the Basis *)

     G:=∅
     FOR ALL h ∈ H DO
       h':= NormalForm(H-{h},h)
       IF h'≠ 0 THEN
         G:= G ∪ h'.
```

Figure 4.2: Watt's parallel algorithm

by [Watt, 1986] and it seems that Watt's implementation of the idea has a high degree of parallelism, see Figure 4.2.

The method starts with the given input polynomials as first basis.

1. In the first phase we take the basis and generate the set of critical pairs. We reduce the related S-polynomials simultaneously with respect to the fixed basis. If all S-polynomials are reduced, the new basis polynomials are collected and merged with the basis forming a new set H.

2. Afterwards we simultaneously take each polynomial in H and reduce them with respect to the not changing rest of H, in the second phase. If all polynomials are reduced we collect them. This set will be our new intermediate basis and we again start reducing S-polynomials. This is done until the basis did not change.

One sees immediately that the second phase of this algorithm is not correct. If two polynomials will have the same leading power product both polynomials will be reduced with respect to the other one. So we may lose this leading monomial and may not generate the full ideal generated by the original basis.

As example we assume the intermediate polynomial set H is $\{x,\ x+y\}, x > y$. In this case, the two normal forms are y, leading to the basis $\{y\}$. The correct Gröbner basis is $\{x, y\}$.

Apart of being incorrect, Watt's version is also inefficient. Some of the reductions during the first phase may produce the same result. There may also be many basis polynomials reducible to zero with respect to the other basis polynomials. In this way more pairs than necessary are computed. This delays the termination of the computation.

It is essential that we keep the basis fully reduced after each S-polynomial reduction phase, otherwise convergence is delayed. Keeping the basis fully reduced limits our ability to parallelize. We cannot simultaneously reduce all the elements with respect to each other, as it is done in Watt's basis reduction phase. A basis polynomial that is used to reduce other ones, can not be reduced by other processes simultaneously. Thus the reductions of the basis has some implicit serialization.

Two corrected versions of Watt's version of the algorithm are presented in [Ponder, 1988]:

4.4.2 Parallel Reduction of the S-polynomials

One corrected method uses parallelism only during the first phase of Watt's algorithm. The reduction of the basis is done in serial afterwards.

Problems with this method are:

- *Reducing the basis in serial will add a big serial phase to the process, limiting the effect of parallelization.*
- *The method forces an (unnecessary) synchronization after each parallel phase.*
- *Each new S-polynomial might reduce the basis such that other S-polynomial operations are redundant.*
- *[Ponder, 1988] does not apply the criteria for detecting unnecessary S-polynomials. This will increase the achievable parallelism but compared with a good sequential form involving the criteria the performance will be quite bad.*
- *With the application of the criteria, the average number of S-polynomials which can be reduced in each parallel phase will be quite small. Thus, the parallelism will not pay off.*

Ponder did some paper and pencil simulations of the method showing that the performance of this algorithm is generally poor and no speed-up is achieved.

4.4.3 Parallel Reduction of the Basis

In the second method of [Ponder, 1988], the S-polynomials are computed one at a time. Assuming that the basis is changing significantly each time an S-polynomial is reduced we use parallelism in the basis reduction phase.

We reduce each S-polynomial with respect to the basis as it is done in the sequential algorithm. Analoguosly to the reduction of the basis in the sequential case, described in [Buchberger, 1985a], whenever the resulting polynomial is not zero this polynomial reduces each of the remaining basis elements in parallel. This may produce new reduced basis elements. The new reduced basis polynomials are used again to reduce the other basis polynomials in parallel. This is done until no basis polynomials can be reduced any more. Afterwards we choose the next S-polynomial and put it through the same process.

In the later S-polynomial reduction phases of the process, most of the S-polynomials will reduce to zero, so the parallel phase will not be used. In the earlier phases of the process, most S-polynomials will have degrees too large to be used for reduction, even after they are reduced. Heuristically, the only time parallelism pays off is when the "magic" S-polynomials appear that cause the basis to begin collapsing. The performance of this algorithm is again quite unimpressive.

Problems with this method are:

- *The reduction of the S-polynomials will be a big serial phase.*
- *The basis will not change significantly after an average reduction phase. Thus, there will not be much parallelism.*
- *[Ponder, 1988] does not apply the criteria for detecting unnecessary S-polynomials.*

Again, simulations with paper and pencil in [Ponder, 1988] shows that no speed-up is achievable with this method.

The above consideration show that it is impossible to get a high performance if one reduces the S-polynomials and the basis-polynomials independently of each other reduction, in different phases.

4.5 Coarse Grain Parallelism: Distributed Reduction

In this method we assume that we have a limited number of nodes with a local memory. The nodes are connected in a specific topology to perform communication.

4.5.1 Central Manager

In fact Buchberger himself sketched a parallel version of his algorithm in an early paper [Buchberger, 1987b]. In this paper a general method for parallelizing Critical-Pair/Completion procedures on the L-Machine was given.

The L-Machine was developed in the framework of the L-Project that started at RISC-Linz in 1977 [Buchberger, 1978]. The goal of the project was the design and implementation of a parallel machine (L-machine) together with a parallel language (L-language) suitable for Symbolic Computation [Buchberger, 1985b, Buchberger and Hintenaus, 1985]. A prototype was built in 1985 whose hardware consists of a network of 8 nodes and 8 memorys interconnected by a freely programmable cross-bar switch. It allows recursive topology description at run-time. The idea of programmable topology has meanwhile also been incorporated in the Transputer systems [Inmos, 1985] with the Occam [Inmos, 1988] programming language.

For the L-machine, a method for parallelizing the Critical-Pair/Completion procedure on various topologies (tree, ring) is given.

In this method we have two kinds of processes:

- A pair set manager that holds the set of critical pairs.
- A set of reducer processes. Each of them has a copy of the whole basis.

The manager distributes as much pairs as possible to the reducers. Each reducer process tries to reduce its pair. If it finishes reduction, the new result is distributed to the other reducer processes and to the manager. The manager creates the new pairs and applies the critera for detecting unnecessary pairs. The pairs are again distributed to idle reducer processes until no new pairs will arrive.

4.5.2 Distributed Managing

Based on Buchberger's work in [Vidal, 1990], a set of rewrite rules are given that express the communication between the different processes. A ring topology is used. Whenever a process wants to communicate with other ones, it sends a message to its neighbour in one direction. Every process that receives a message will send it to the next process again, as long as the message reaches the original sender. In this way, each message is distributed to all other processes.

In this approach we require only a set of reducer processes and no central manager. Each reducer has a copy of the whole basis and the whole set of pairs (in contrast to Buchberger's version, where only one copy of the pairs is necessary).

Each reducer process takes a pair from its local memory and informs all other reducers about its choice and all reducers will discard this pair from their memory. Then the related S-polynomial is reduced. If the result is not zero, it generates the new pairs locally and sends the new basis polynomial and the new pairs to the other processes. They will add them to there own set of pairs and basis polynomials. Afterwards a new pair is choosen from the local memory again. This is done until all pairs are reduced.

This method is not yet implemented, so we don't know the achievable speed-up.

Problems with this method are:

- *The full basis and the full set of S-polynomials have to be stored in each worker process (waste of memory).*
- *Keeping the basis and pairs up to date needs much communication overhead.*
- *Reduction of the basis will not be possible without a big communication overhead.*

4.6 Coarse Grain Parallelism: Shared Basis

An easier version of the previous method will have a shared memory holding the basis and the pairs. Each node has access to this shared memory. This approach was presented in [Vidal, 1989].

Every process picks up a pair from the set of unreduced critical pairs and reduces the related S-polynomial with respect to the polynomials in the basis. If no further reduction is possible and the result is not equal to zero, the resulting polynomial is adjoined to the basis. At this time all other processes have access to the new basis-polynomial. Afterwards the process updates the set of pairs. Then it again picks up a pair from the pair set. This is done until all processes finished reducing S-polynomials and no new pairs are generated.

The set of polynomials in the basis and the set of pairs are shared between the processes. To grant readers and writers exclusion a Concurrent Read Exclusive Write (CREW) model is used. There are two primitives to handle read and write access.

with read_access v do P: means that a reader process, may only have access to the variable v, if no writer wants to change it. If a writer process changes v, all readers have to wait until the writer has finished. Then the reader again tries to read the variable v. If the access is successfull, it executes the statement P and leaves the region.

with write_access v do P: A process that wants to write the variable v needs first to compete with the other processes that requested to write onto the variable v. When it becomes the only process allowed to write, it prevents other processes to obtain read access to v, waits until all the processes which read the variable have left the region, executes the statement P and leaves the region.

During the reduction of an S-polynomial, a process reads the shared basis many times but never updates it. Processes reducing a polynomial should be allowed to read the basis simultaneously. This is important because processes spend most of their execution time in reductions. Therefore, the basis should be shared by using a CREW mechanism.

For improvement of the algorithm it is also possible to update the basis simultaneously with reading, iff different polynomials are involved. A distinction between the different polynomials in the basis is necessary. The basis is represented by a list of pointers to the polynomials. The different polynomials will not be changed during an update of the basis.

If the number of processors is greater than one, the execution time varies widely. It depends on the way the different processors interleave. An implementation of this method shows that it is possible to have a speed-up between 2 and 10 on a 16 processor shared memory machine with well known test examples [Traverso and Donati, 1989].

Problems with this method are:

- *A shared memory is required. This will limit the number of usable nodes.*
- *[Vidal, 1989] does not reduce the basis after a new S-polynomial enters the basis.*

4.7 Fine Grain Parallelism

The fine grain approach is used on a hardware that supports fast communication between the different processes.

If one looks to the details of the sequential version of Buchberger's algorithm one detects that every polynomial operation requires only the head of the polynomial to determine the next operation. This leads to a pipe-line of processes where every process operates on single monomials. The number of monomials in a polynomial limit the achievable speed-up.

The method of parallelism inside the individual polynomial operations was introduced by [Melenk and Neun, 1988]. It is implemented in the Lisp-based REDUCE system on a Cray X-MP supercomputer.

The polynomials are in distributive representation (DP), i.e. a polynomial is a sequence of monomials (MN). The monomials are ordered in descending order with respect to a global term ordering:

$$DP = [MN_1, ..., MN_k]$$

Classifying the polynomial operations needed by Buchbergers algorithm one finds that:

- For the control of the algorithm only the access to the leading monomial is necessary,
- All arithmetic combinations are special cases of the pattern $MN_1 * DP_1 + MN_2 * DP_2$ with appropriately constructed monomials.

Based on this, a new operation that is a "linear combination" (called *Lcomb*) of two polynomials is constructed. Buchberger's algorithm may be rewritten using *Lcomb* as the only polynomial operation.

The *Lcomb* procedure computes the leading monomial first and linkes them first to the resulting structure. It is possible to continue the algorithm as soon as the leading monomial is available. Thus, it is possible to overlap the operation of *Lcomb* and the processing of its result by the algorithm in a multiprocessing environment.

For parallel execution, Buchberger's algorithm has to be segmented into tasks. Each task is conected via a polynomial operation (*Lcomb* procedure), in a pipeline. This leads to a chain of coroutines. The monomials are subsequently driven through the pipes.

The "data grain" is the individual polynomial, assuming for a moment that the data flow is steady. In a complicated calculation the typical intermediate polynomial 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 S-polynomial is a local fence for the parallel execution.

Simulation of this algorithm shows that with 2 nodes available a speed-up of almost two may be reached. There is a close connection between the length of polynomials and parallelism.

Problems with this method are:

- *The parallelism is limited by the length of the polynomials.*
- *The zero reduction will force an (uneccessary) syncronization and shall be a bottleneck.*

Experimental results on a two processor Cray X-MP supercomputer in [Melenk and Neun, 1988] shows that, in fact, no speed-up was achieved.

A finer grain parallelism is proposed by [Melenk *et al.*, 1988] using the specialized hardware of todays supercomputers. There are various improvements of the operations (including factorization, modular coefficients, vector processing of the exponents) that increase the performance of the algorithm. It pays off in case of the use of polynomials with big coefficients and a large number of variables.

4.8 Multiple Gröbner Bases

In [Ponder, 1988] some alternative methods are discussed.

4.8.1 Subsets of the Basis

We can repeatedly find the reduced Gröbner bases of various subsets of the basis and merge them, until the basis converges.

An important fact about Buchberger's algorithm is that key polynomials are formed which cause the basis elements to reduce drastically. If, for example, a variable is isolated so that it appears only in the head term of a polynomial it will cause all other instances of the variable to be eliminated. This elimination property suggests that the basis should be kept as far reduced as possible that reductions can take effect as soon as possible. Splitting the basis will deprive some polynomials of the chance to get reduced at an early stage. It is likely that the splitting technique will take much longer for convergence because of this.

4.8.2 Reducing under Alternative Orderings

An alternative way for introducing parallelism is a simultaneous reduction of the basis under various orderings, using whichever result comes first. Using one node for reduction under each ordering would generate a solution in as much time as the fastest of the orderings.

This would yield a reasonable speed-up for some examples compared to the case that the "wrong" ordering is used. There are examples in which the speed-up will be up to 60 or more with 6 concurrent processes compared to the "wrong" ordering.

4.9 Comparison of the Methods

The fine grain parallelism may increase the speed only for a few number of nodes. The size of the polynomials that are used limits this number.

In course grain parallelism, the speed-up varies very much depending on the memory and communication model. We can see that it is necessary for efficient parallelization to keep the entire basis accessable for each parallel process. This is easy on a shared memory system. A speed-up is achieved on multiprocessor systems with several processors [Vidal, 1989].

On loosely coupled or distributive memory systems we see that reducing basis and S-polynomials independently of the other reductions and synchronizing after each reduction phase, it is not possible to get a good performance. One needs a different communication method for reaching a good performance.

Reduction under alternative orderings may have a very good speed-up. But we want to find a method that will increase the performance with a given term order, because many problems require a specific term order. Our method should have a speed-up in the general case.

No good method for parallelizing Buchberger's algorithm seems to be known for distributive memory machines so far. Therefore, we concentrate on this problem in this thesis and propose a new method in the next chapter.

Chapter 5

A new Idea for Parallelizing the GB-Algorithm

Our approach to the parallelization of Buchberger's algorithm using a medium grain pipe-line principle for the polynomial reduction stems mainly from the fundamental stream concept in concurrent logic programming.

5.1 Requirements

The algorithm should lead to a high speed-up on a wide range of hardware platforms. Therefore we use as underlying model a distributive memory MIMD machine with a limited memory. Converting programms written for this model to various other kinds of hardware architecture is easy and preserves the effect of the parallelism.

For achieving a reasonable speed-up on this hardware model, notice that the following facts are crucial:

- The communication speed is time critical.
- The storage of the basis is memory critical.
- A master process tends to be a bottleneck.
- The algorithm on one node should be comparable with the best sequential version.

This leads us to the following requirements for a concurrent version of Buchberger's algorithm.

1. The communication overhead should be minimized. The computation speed of the processors is much higher than the speed of the communication unit. In a concurrent algorithm, it is possible to risk a slightly higher computation effort achieving a significant lower communication overhead.

2. While computing "realistic" examples with Buchberger's algorithm one may get quite a big intermediate basis. In a network of many nodes, each node will have a limited memory capacity. Keeping the whole basis in each memory will let the node run out of space. A distribution of the basis will be necessary.

3. All control information should be kept local. There should not exist a manager process. This process would be the bottleneck in a big network of nodes. Distributing the application of the criteria for detecting unnecessary S-polynomials over the parallel tasks, is essential.

4. The complexity of the algorithm depends strongly on the optimization of the algorithm (criteria, sorting and reducing the basis). The concurrent algorithm should reduce the computation time and make bigger examples possible. All optimizations in the sequential version, should be applicable in the concurrent version, too.

5.2 Usable Parallelism

In the Gröbner bases algorithm one finds many independent parts useful for parallelization. It reaches from a coarse granularity over big parallel tasks down to a fine grain parallelism over single instructions.

The granularity can be divided in:

Coarse Grain: The various S-polynomials may be reduced and generated in parallel.

Medium Grain: Parallelism between the various polynomial operations.

Fine Grain: Parallelism inside the polynomial operations.

For efficient use, we have to deal with many data dependencies between the different operations. This limits the degree of usable parallelism.

- Building the normal form of the S-polynomials requires the whole basis in the newest version.

- Applying the "criteria" for omitting unnecessary S-polynomials requires knowledge about other S-polynomials.
- Keeping the basis and S-polynomials sorted decreases the computation overhead.
- We use fully reduced bases. (Timing statistics on this choice in comparison with only partial reduced bases are not yet available.)
- The control of the polynomial operations requires at least the leading power product.

The data dependencies mainly come from the optimization strategies that improve the complexity of a sequential version.

One sees that most of the usable parallelism comes from a data structure allowing partially independent steps. Data dependencies force synchronization on certain positions. There are only a few total independent tasks that could be done in parallel.

Therefore a concurrent version needs a programming style that is able to deal with parallelism in the data structure. The fundamental stream concept of concurrent logic languages enables us to do this.

In principle, we use all parallelism found in the algorithm. Also, all the practical optimizations tuning the sequential algorithm are applicable in the parallel case.

5.3 Main Idea

The initial idea stems from the concurrent version of the "Sieve of Erathostenes" to compute prime numbers (compare 2.3.3).

The rough structure is similar. There are processes reducing the input. The irreducible input values are appended to the reduction processes.

For deriving the structure of Buchberger's algorithm from the concurrent algorithm computing the "Sieve of Erathostenes", we replace the functions depending on the prime numbers with functions handling polynomials.

- The process that generates the list of integers is substituted by a process generating S-polynomials.
- The intermediate integer testing processes are replaced by processes reducing polynomials.

- The sieve process at the end waits for a new non-zero polynomial instead of the next prime number.
- The polynomials of the final Gröbner basis may be found in the intermediate filter processes.

For handling the computation of S-polynomials, the new polynomials joining the basis are sent back through the pipe building the S-polynomials with the earlier computed basis polynomials. The generator process only needs to close the pipe to a ring, collecting the computed S-polynomials and sending them to the filter again.

As a result we get a bidirectional pipe forming a dynamically expandable ring (Figure 5.1).

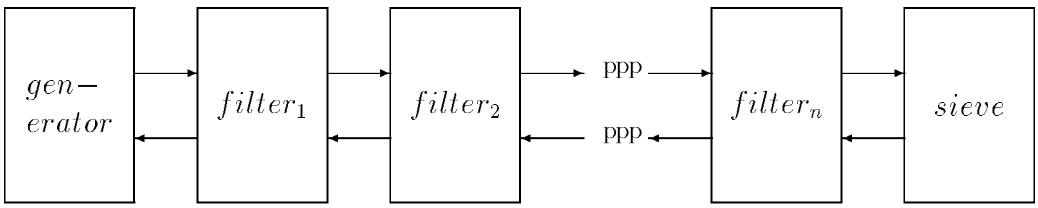


Figure 5.1: The Pipe

5.4 Basic Features

Our proposal has the following features:

- The individual polynomials of the (intermediate) basis are distributed among the nodes of the (distributed) parallel machine arranged in a pipe.
- An S-polynomial is driven through the pipe and each node reduces the polynomial by applying its basis polynomial.
- If the (partial) reduction does not yield zero, the polynomial is put into the basis (and placed on a new node).
- "On its way back", the new polynomial is combined with the existing polynomials in the basis for forming the new S-polynomials that are subsequently driven through the pipe. We would like to note that in the process of forming the new S-polynomials we apply the "criteria" for omitting unnecessary ones.

- Also on its way back the new polynomial reduces the existing polynomials in the basis.

This leads to various advantages, that are crucial to achieve a high performace on a parallel machine:

- The basis is distributed over various nodes in a multiprocessor system.
- All optimization strategies are used.
- The criteria are applied locally.
- The communication effort is minimized.

Chapter 6

Details of the Solution

We assume that the reader is familiar with the basic notions of Gröbner bases theory and the sequential algorithm. For an introduction and the notation see [Buchberger, 1985a].

6.1 The Basic Algorithm

We describe the basic algorithm, without applying the criteria.

6.1.1 Overall Organization of the Pipe

The pipe is organized as follows (Figure 6.1):

- A front end process *append-spol* takes the stream of computed S-polynomials and puts them to the reduction processes.
- The intermediate *filter* processes reduce the incoming S-polynomials with respect to their basis polynomial stored and forwards the result to the next process. Also, this process has to compute the S-polynomials with the newly computed basis polynomials and returns the result to the previous process.
- At the end of the pipe-line a *sieve* process waits for incoming S-polynomials that are not reduced to zero. Such a polynomial has to be added to the basis. A new filter process is generated and the basis polynomial found is sent back to form the new S-polynomials.

For avoiding a sequential step at the beginning of the algorithm, the initialization of the pipe-algorithm is a little different to the initialization of Buchberger's

```
% main loop of the algorithm

-mode main(In?,Out^).
main(InPoly,Basis):-
   append_spol(InPoly,Spol,Testpoly),
   sieve(Testpoly,Spol,Basis).

-mode sieve(Test-polys?,S-polys^,Basis-polys^).
sieve([BP|TPs],[{BP,[]}|SPs1],[BP|BPs]):-
   BP =\= [] |
       filter(BP,TPs,SPs,RPs,SPs1),
       sieve(RPs,SPs,BPs)@next.
sieve([[]|TPs],SPs,B):-
   sieve(TPs,SPs,B).
```

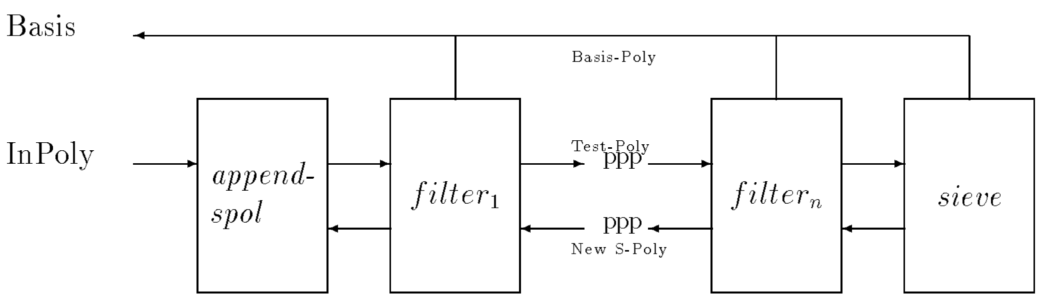


Figure 6.1: Organization of the Pipe

method. In our approach we start with an empty basis and the input polynomials are the initial S-polynomials. The advantages of this method are:

- We start with the empty pipe and append the basis polynomials one after the other.
- No initialization procedure that computes the S-polynomials of every basis polynomial with respect to each other is required.
- We reduce the input polynomials with respect to the earlier chosen ones. This leads to a better performance in the case of not fully reduced input polynomials.

For comparing our method with the original initialization proposed by Buchberger, we assume that the input polynomials are fully reduced with respect to each other. Under this assumption we show that our initialization method leads to the same initial set of polynomials than Buchberger's method were we start from the input polynomial set as first basis and compute crosswise all S-polynomials to get the initial S-polynomial set.

Lemma 6.1 (Initialization) *With a fully reduced input set, the pipe-line algorithm generates the same initial S-polynomials and basis polynomials as with the use of Buchberger's initialization procedure.*

Proof: From the assumption that the input-polynomials are fully reduced, we know that the input polynomials will not change by any reduction process in the pipe.

All input polynomials are appended to the basis, one after the other. Each new basis polynomial creates the S-polynomials with all previous basis polynomials.

After the last input polynomial entering the basis all S-polynomials from the input polynomials have been built. The polynomials in the basis are exactly the input polynomials. At this state, our basis polynomials and S-polynomials are the same as after the initial process in Buchberger's algorithm.

6.1.2 The Intermediate Filter Processes of the Pipe-line

The intermediate filter processes are divided into two independent parts (Figure 6.2):

- The reduction process that tests and reduces the incoming S-polynomials TPs and sends the result RPs to the next filter process.

```
-mode filter(BP?,TPs?,SPs?,RPs^,SPs^)
filter(BP,TPs,SPs,RPs,SPs1):-
        reduce(BP,TPs,RPs),
        add_spol(BP,SPs,SPs1).
```

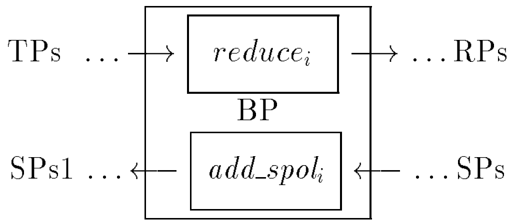


Figure 6.2: Intermediate Filter Processes

- A process that generates and adds new S-polynomials to the list of unresolved S-polynomials SPs and sends its result SPs back to the previous filter process.

Each filter process holds one particular polynomial of the basis (BP).

The Reduction Process

Each reduction process (Figure 6.3) performs a full reduction of all S-polynomials with respect to one particular basis polynomial.

```
-mode reduce(Basis-poly?,Test-polys?,Reduced-polys^).
reduce(BP,[TP|TPs],[NP|RPs]) :-
    poly:reduce(TP,BP,RP),
    poly:norm(RP,NP),
    reduce(BP,TPs,RPs).
reduce(_,[],[]).
```

Figure 6.3: The Reduction Process

Each incoming test polynomial TP is reduced with respect to the particular basis polynomial of this process. Then the result is normed such that the coefficient of the leading monomial is one. The normed result NP is sent to the next reduction process.

One reduction step filters out all monomials which are multiples of the leading monomial of the basis polynomial. If the resulting polynomial is zero this process returns the zero polynomial. Thus, we avoid a synchronization step and get a higher degree of parallelism.

The Generation of S-polynomials

The process that generates S-polynomials (Figure 6.4) builds all S-polynomials with respect to one particular basis polynomial.

```
-mode add_spol(Basis-poly?,S-polys?,New-S-polys^).
add_spol(BP,[{BP2,SP1}|SPs],[{BP2,[SP|SP1]}|SPs1]):-
       make_spol(BP,BP2,SP),
       add_spol(BP,SPs,SPs1).
add_spol(_,[],[]).
```

Figure 6.4: Generation of S-polynomials

This process takes all later generated basis polynomials and the S-polynomials built by them, one after the other.

With each basis polynomial $BP2$, the S-polynomial SP with respect to the process' own basis polynomial BP is computed and appended to the list of S-polynomials $SP1$ that are built with this basis polynomial in the later pipe processes. The result is sent to the previous element of the pipe to compute the remaining S-polynomials.

Then we proceed in the same way with the next basis polynomial until no new basis polynomial is computed.

6.1.3 Normal Form

For applying the reduction in the pipe-line, we introduce a different (partial) normal form algorithm.

The normal form is a full reduction with respect to a particular order of polynomials. The basis polynomials are chosen one after the other in a specific order of the polynomials.

Lemma 6.2 (Normal Form) *Our normal form algorithm fullfills the following properties:*

1. *The normal form algorithm terminates after p steps, where p is the number of polynomials in the basis.*

2. *After appending the normal form of a polynomial P to the basis, P will be reduced to zero in every later steps.*

Proof: Clear.

This (partial) normal form algorithm generates in some cases not the normal form of the input polynomial. One might assume that we should send the not fully reduced polynomials again through the pipe as long as no reduction will be possible. This would generate much additional computing overhead, but has no influence to the correctness of the algorithm. Furthermore, in (Section 6.3.3) we see that this additional reduction will be done automatically by using the criteria.

Experimental comparison with the implementation of the Gröbner bases algorithm in the Computer-Algebra System muMATH [Buchberger, 1987c] where all possible optimizations are implemented shows that with our normal form and the usage of the criteria the computation time differs by a constant factor. Compared with other systems like (MAPLE, SAC-2, Scratchpad) experiments shows that the efficiency of our implementation is better (compare 7.1.2).

6.2 Termination

The correctness and termination of the Gröbner bases algorithm is guaranteed by Buchberger's theorem:

Theorem 6.1 (Algorithmic Characterization of Gröbner Bases)
F is a Gröbner basis iff for all $f_1, f_2 \in F$ normal-form(F,S-polynomial(f_1, f_2))=0.

This guarantees finite many reduction steps and termination. The proof may be found in [Buchberger, 1965, Buchberger, 1970].

It is sufficient to assume that the normal form algorithm makes at least one reduction step and equal polynomials are filtered out. This is guaranteed by the properties of our normal form algorithm.

After finishing the reduction of S-polynomials our implementation terminates with a deadlock. The reason is that:

- After the last reduction of an S-polynomial, every reduction process in the pipe-line waits for the next S-polynomial.
- Also the sieve process at the end waits for new polynomials which are not zero to invite the S-polynomial generator processes computing new S-polynomials.

- Each intermediate process that generates S-polynomials waits for new basis polynomials for generating a new S-polynomial.
- The front end process waits for new S-polynomials from the generator to send them to the reduction step.

Thus, every process in the pipe waits for a result of the previous one and we have a deadlock.

Actually, in order to avoid a real deadlock we insert a marker that walks around the pipe and controls if there is still a process that works. If no process is active the algorithm terminates.

```
append_spol(Count,[],[nr(Count1)|SP],TP):-
   Count > Count1 |
       append_spol(Count,[],SP,TP).
append_spol(Count,[],[nr(Count)|X],[]).


reduce(BP,[nr(Cnt)|TPs],[nr(Cnt)|RPs]) :-
   reduce(BP,TPs,RPs).

sieve([nr(Cnt)|TPs],[nr(Cnt)|SPs],B):-
   sieve(TPs,SPs,B).

add_spol(BP,[nr(Cnt)|SPs],[nr(Cnt)|SPs1]):-
   add_spol(BP,SPs,SPs1).
```

Figure 6.5: Insert Counter Token

In (Figure 6.5), a counter token is inserted to the list of S-polynomials after every list of S-polynomials generated with one new basis polynomial. This token passes all intermediate steps in the pipe unchanged and will be returned to the front end. The front process knows the number of the last counter it sent. If this counter is the same as the returned one, then we know that no process is active any more and we actually found the final Gröbner basis. The algorithm terminates.

6.3 Criteria

In [Buchberger, 1985a] Buchberger introduced two critera for detecting unnecessary S-polynomials. The intensive use will lead to a crucial improvement of the performance.

6.3.1 Criterion on leading power product

The first criterion is based on the following theorem [Buchberger, 1965, Buchberger, 1970]:

Theorem 6.2 (Criterion on LP)
If $LP(f_1) * LP(f_2) = lcm(LP(f_1), LP(f_2))$, then S-polynomial$(f_1, f_2)$ may always be reduced to zero modulo F and may be cancelled.

From this theorem we may derive the better computable equivalent form:

Lemma 6.3 (Criterion1)
If Criterion1(f_1, f_2) then S-polynomial(f_1, f_2) may be cancelled.

Criterion1(f_1, f_2) :$\Leftrightarrow$ gcd(LP(f_1),LP(f_2)) $= \vec{0}$.

Proof: $LP(f_1) * LP(f_2)$ only equals $lcm(LP(f_1), LP(f_2))$, iff there is no common factor in $LP(f_1)$ and $LP(f_2)$. Therefore $gcd(LP(f_1), LP(f_2))$ must be zero and we may cancel S-polynomial(f_1, f_2)

This criterion may be tested locally in every pipe process that generates new S-polynomials.

6.3.2 Chain Criterion

The second criterion is formulated as:

Theorem 6.3 (Chain Criterion)
If ChainCriterion$(f_1, f_2, Basis)$ then S-polynomial(f_1, f_2) may be cancelled.

ChainCriterion(f_1,f_2,Basis):$\Leftrightarrow$
there exists a $p \in Basis$ such that
$f_1 \neq p \neq f_2$,
lcm(LP(f_1), LP(f_2)) is a multiple of LP(p),
S-polynomial(f_1,p) and S-polynomial(p,f_2) has been chosen for reduction before.

For applying this criterion in a distributive way, we have to split it into two parts.

From now on we assume that the polynomials f in the basis (pipe) are numbered in descendent order at creation time. The newest polynomial has the highest number and is called f_n.

$$\boxed{f_1} \longleftarrow \ldots \longleftarrow \boxed{f_i} \longleftarrow \ldots \longleftarrow \boxed{f_j} \longleftarrow \ldots \longleftarrow \boxed{f_n}$$

Lemma 6.4 (Chain Criterion – Pipe Implementation) *For all i,j,n* ($1 \leq i < j < n$)*,*

1. *if Criterion2(j,n) at i then S-polynomial*(f_j, f_n) *may be deleted,*
2. *if Criterion3(i,n) then S-polynomial*(f_i, f_n) *may be deleted.*

Criterion2(j,n) at i $:\Leftrightarrow$ *lcm*$(LP(f_j), LP(f_n))$ *is a multiple of* $LP(f_i)$.

Criterion3(i,n) $:\Leftrightarrow$

- *there exists a k* ($i < k < n$) *such that lcm*$(LP(f_i), LP(f_n))$ *is a multiple of* $LP(f_k)$,
- *there exists no j* ($i < j < n$) *with Criterion2(j,n) at i.*

For the correctness we have to show that whenever we cancel a pair with the pipe criterion, we may also cancel it with the chain criterion.

Proof: From the execution of the algorithm we know that at the state of computing the S-polynomial with the new polynomial f_n all *S-polynomials*(f_i, f_j) (with $i < j < n$) are already reduced or cancelled. Thus, we only have to consider the S-polynomials that are built with f_n.

We will prove the correctness at all stages i in the pipe. Here we consider the two separate parts analogously to the two parts of the criterion:

Part 1: Proof of Criterion2

We know that the *S-polynomial*(f_i, f_j) is already reduced before. The *S-polynomial*(f_i, f_n) will not be cancelled at stage i (exception of Criterion3) or at stages $> i$. By the chain criterion we may delete *S-polynomial*(f_j, f_n).

Part 2: Proof of Criterion3

We only have to consider the cases where a $k > i$ and *lcm*$(LP(f_i), LP(f_n))$ is a multiple of $LP(f_k)$ exists and no $j > i$ with *Criterion2(j,n) at i* exists.

We know that *S-polynomial*(f_i, f_k) is already reduced before. By assumption, that we can not apply Criterion2 at i, we know that we can not cancel *S-polynomial*(f_k, f_n) using f_i.

This allows deleting *S-polynomial*(f_i, f_n) by using the chain criterion.

This shows the correctness on each position i of the pipe.

For applying this criterion locally on each pipe position, we only need knowledge about the S-polynomials driven back to the front and the leading power products of the basis polynomials that come later.

6.3.3 Theoretical Results

In the case where the leading power product of a basis polynomial (f) is a multiple of the leading power product of another one (g) a criteria in [Buchberger, 1970] says that we may cancel the basis polynomial (f) after computing the *S-polynomial*(f, g).

With the pipe implementation it is easy to show that this cancellation may be done as a special case of the chain criterion.

Lemma 6.5 *If the leading power product of a new polynomial in the basis is a multiple of another, older polynomial in the basis the new polynomial generates exactly one S-polynomial and may be cancelled afterwards.*

Proof: We assume that there is a j, i with $LP(f_j)$ is a multiple of $LP(f_i)$ ($1 \leq i < j$).

We know that for an arbitrary polynomial p, $lcm(LP(p), LP(f_j))$ is a multiple of $LP(f_i)$.

By applying the pipe criterion part 1, we know that for all $k > i$ *S-polynomial(f_j, f_k)* may be cancelled at pipe position i.

With part 2, we know that for all $k < i$ *S-polynomial(f_j, f_k)* may be cancelled.

The only S-polynomial that must be generated is *S-polynomial(f_j, f_i)* (exception in part 2). We know that $lcm(LP(f_j), LP(f_i))$ is equal to $LP(f_j)$. The generated S-polynomial now is equal to the polynomial f_j where the leading monomial is reduced by f_i. Thus, we may replace f_j by the reduced form and cancel f_j from the basis.

This lemma shows that if we use only a partial normal form algorithm we will not create too many S-polynomials. A not fully reduced polynomial will be sent automatically through the reduction pipe-line until it is fully reduced.

Apart from the actual implementation of the Gröbner basis algorithm we will show that if we use the criteria and vary the reduction part arbitrary we will never generate superfluous S-polynomials.

Lemma 6.6 *Without the reduction part, only by applying the criteria, our version of the algorithm is equivalent to a version with fully reduced polynomials.*

Proof: Whenever the leading monomial of an S-polynomial may be reduced then we know from Lemma 6.5 that only the reduced polynomial will be generated as new S-polynomial and the old one may be cancelled. This is equal to one reduction step in a normal-form algorithm.

If no basis polynomial exists whose leading power product is a multiple of the leading power product of the S-polynomial we reached the normal-form of this S-polynomial and append it to the basis. All new S-polynomials are created.

This is actually what the version with fully reduced polynomials does.

6.4 Sorting and Changing of the Basis

For improving the runtime behaviour of the algorithm, we have to sort the S-polynomials and reduce the polynomials of the basis (Figure 6.6).

```
reduce(_,[up([BP|BPs])|TPs],[up(BPs)|RPs]):-
   reduce(BP,TPs,RPs).
reduce(BP,[up([])|TPs],RPs):-
   reduce(BP,TPs,RPs).

add_spol(BP,[{BP2,SP,B}|SPs],[{BP2,SP1,B1}|SPs1]):-
   BP =\= [] |
        poly:reduce(BP,BP2,RP),
        poly:norm(RP,NP),
        ord:in_sort(down,NP,B,B1),
        insert_spol(NP,BP2,B,SP,SP1),
        add_spol(NP,SPs,SPs1).
```

Figure 6.6: Update of the Basis and S-polynomials

6.4.1 Sorting of S-polynomials

The smaller S-polynomials should be reduced earlier than the bigger ones.

In our algorithm it is only possible to sort all S-polynomials which are generated with one new polynomial in the basis. The earlier built S-polynomials are all in several states of the pipe and involved in reduction. The later ones are not yet known and we have also no access to them.

6.4.2 Changing of the Basis

When a new polynomial joins the basis, it may reduce all earlier computed basis polynomials. It might also cancel a basis polynomial, thus fewer S-polynomials must be computed.

With our normal form algorithm, it is also crucial to sort the basis polynomials in the pipe in descending order. If this is done, the probability that an S-polynomial may be reduced with respect to an earlier chosen basis polynomial is small.

The reduction of the basis polynomials may be done in the step where the S-polynomials are generated. All basis polynomials are collected afterwards and will be sorted.

For inserting them into the reduction pipe-line an update token holding the basis is used.

If an update token arrives at a reduction process, the reduction process takes the first and biggest basis polynomial and sends the rest to the next reduction process. The inherent basis polynomial is replaced by the new one. This is done, until no more update polynomials are known.

The newer processes will not get an update polynomial. They keep there own basis polynomial unchanged.

6.5 Final Reduction

The computed basis is not fully reduced. For getting the reduced Gröbner basis a final reduction step is necessary.

For getting the final basis, one may reduce each basis polynomial fully with respect to each other or use a few optimization strategies given in [Buchberger, 1987c]:

- A basis polynomial whose leading power product is a multiple of the leading power product of an other basis polynomial may always be reduced to zero. Here we need not perform the reduction algorithm.

- For getting the final basis polynomial, it is enough to compute the fully reduced normal form with respect to the fixed basis. This may be done in parallel.

- A basis polynomial may only be reduced by smaller ones.

We sort the basis in a descending order. Then we take one polynomial after the other and reduce them with respect to the remaining ones in parallel. Whenever the leading power product of a basis polynomial is a multiple of an other one, we cancel this polynomial immediately.

6.6 Subprograms

The whole algorithm is split into independent modules. Thus, it is possible to change the actual representation of the data and the algorithms in a module without effecting the other ones.

A complete listing may be found in (Apendix A).

6.6.1 Main Module: *groebner*

This module contains the specific procedures necessary for computing Gröbner bases.

Usage of the Algorithm

We start the Gröbner basis algorithm by calling the following procedure:

gbasis(PolySet?,Ord?,GBasis^)

where

Polyset is a list of input polynomials in human readable form (see 6.6.5),

Ord holdes the list of indeterminantes in descending lexical order,

GBasis returns the reduced Gröbner basis.

```
-mode gbasis(In?,Ord?,Out^).
gbasis(PolySet,Ord,GBasis) :-
     polypars:poly_list(PolySet,Ord,Ps),
     main(Ps,Basis),
     full_reduction(Basis,RB),
     polypars:out_poly_list(RB,Ord,GBasis).
```

For computing the reduced Gröbner basis we need the following steps.

1. The input polynomials are first parsed to the internal data structure.
2. Then the actual Gröbner basis is computed.
3. Afterwards we fully reduce it for getting the reduced Gröbner basis.
4. Finally the result is parsed from the internal to the output structure.

Generation of S-polynomials

For computing the S-polymonials we assume that all polynomials are normed that the leading coefficient of one. Thus, computation of the S-polynomials reduces to the following formula:

S-Polynomial(f_1, f_2)::= $u_1 * f_1 - u_2 * f_2$,
where $u_i = lcm(LP(f_1), LP(f_2))/LP(f_i)$

Data Structure

All processes in the pipe are connected via one stream. The single elements are driven through the intermediate pipe processes. Each process may modify, cancel or insert elements of the stream.

Each stream element has one of the following forms:

$p(Poly)$, where $Poly$ is a polynomial, describes a polynomial that is driven through the reducer processes.

$nr(Cnt)$, where Cnt is an integer, describes the token that flows around the pipe detecting the final deadlock.

$up(Polylist)$, where $Polylist$ is a list of polynomials, describes the token that flows through the reducer processes for updating the polynomials in the pipe.

$\{Poly, SPlist, Polylist\}$ holds all the necessary data for creating the new S-polynomials by applying the criteria and for reducing the intermediate basis.

The individual items are:

$Poly$ is the newest polynomial in the basis. We have to build all S-polynomials with this polynomial.

$SPlist$ is a list of tuples $\{LCM, Poly\}$. Each of them holds one S-polynomial and the least common multiple of the two polynomials building it.

$Polylist$ is a list of basis polynomials.

$fin(Polylist)$ holds the final Gröbner basis.

6.6.2 Polynomial Arithmetic Module: *poly*

This module holds the polynomial operations used by the Gröbner basis algorithm.

The basis polynomial operations are implemented in a standard form.

The Reduction Algorithm

For reducing one polynomial fully with respect to an other (Figure 6.7), we use the following optimizations:

- The reducible monomials are cancelled instead of reduced to zero by subtraction.
- Reduction is formulated in terms of "leading exponent reduction".

  As a consequence, the subtraction in the reduction avoids superfluous copying of higher exponents. This allows the next parallel task using the first monomials of the final reduced polynomial before we try reducing all monomials.

Data Structure

A polynomial is represented as a list of monomials $[M_1, ..., M_n]$. The zero polynomial is repesented by the empty list [ ].

A monomial is represented by a tuple $\{$*Coefficient,PowerProduct*$\}$ where

Coefficient is a rational number,

PowerProduct is a list of integers.

6.6.3 Module: *ord*

This module consists of all order related functions.

Currently only pure lexical order is implemented.

```
-mode reduce(X?,Y?,Z^).
% reduces a polynom wrt. a second polynom

reduce([P1|P1S],[P2|P2S],Y) :-
     ord:divtest(m,P1,P2,TEST,DIVMON),
       reduce(TEST,DIVMON,[P1|P1S],[P2|P2S],Y) .
reduce([],_,[]).
reduce(X,[],X).

-mode reduce(T?,M?,X?,Y?,Z^).
reduce(false,_,[P1|P1S],P2,[P1|YS]) :-
     reduce(P1S,P2,YS) .
reduce(true,DIVMON,[_|P1S],[P2|P2S],YS) :-
     mult_mon_poly(DIVMON,P2S,SUBPOLY),
       sub_poly_poly(P1S,SUBPOLY,REDPOLY),
       poly:reduce(REDPOLY,[P2|P2S],YS)@next.
reduce(lt,_,P1,_,P1) .
```

Figure 6.7: The Reduction Algorithm

6.6.4 Rational Arithmetic: Module *rat*

Here the operations for rational numbers are implemented.

A short number arithmetic is provided for experimental purposes.

Data Structure

A rational number is represented as tuple $\{Sign,Numerator,Denominator\}$, where

Sign may be -1 or 1,

Numerator is represented as positive integer,

Denominator is represented as positive integer.

6.6.5 Input and Output Module: *polypars*

For better use of the package, the list of polynomials may be entered and is displayed in a form readable by humans.

A parser is used for translating the input list into the internal representation and the output into an external, human readable representation.

The syntax of this external polynomial list “PL” in BNF form is:

```
PL  ::=  "[" P { "," P } "]"
P   ::=  M { ("+","-") M }
M   ::=  C | [C "*"] Pot { "*" Pot }
C   ::=  Num | Num "/" Num
Pot ::=  Var | Var "/\" Num
Var ::=  string
Num ::=  positve short Integer
```

6.7 Mapping Strategy

If we put our algorithm to a distributive hardware, we have to decide which parallel process should be mapped to which node.

6.7.1 Pipe-line Topology

The mapping strategy closest to the design of our algorithm will be of course the usage of a pipe-line topology in the hardware. Here the intermediate pipe processes will be mapped to different nodes on the hardware pipe-line, one after the other.

```
-machine(ring).

main(InPoly,Basis):-
   append_spol(InPoly,Spol,Testpoly),
   sieve(Testpoly,Spol,Basis)@fwd.

-mode sieve(Test-polys?,S-polys^,Basis-polys^).
sieve([BP|TPs],[{BP,[],[]}|SPs1],[BP|BPs]):-
   BP =\= [] |
       filter(BP,TPs,SPs,RPs,SPs1),
       sieve(RPs,SPs,BPs)@fwd.
sieve([[]|TPs],SPs,B):-
   sieve(TPs,SPs,B).
```

Figure 6.8: Pipe-line Mapping Strategy

In Figure 6.8, the sieve process at the end of the pipe is always placed to the next node. There it waits until a new filter process will be generated. The new filter process is placed on this particular node and the new end of the pipe will be forwarded to the next node.

Communication has be done in both direction of the pipe.

We see that the two processes in the filter are independent. Thus, we may use two parallel pipes connected together on each end. Now, we place the reducer processes on the left pipe-line and the process that generates S-polynomials on the right pipe-line.

```
filter(BP,TPs,SPs,RPs,SPs1):-
       reduce(BP,TPs,RPs)@north,
       add_spol(BP,SPs,SPs1)@south.
```

In each pipe-line communication has to be done only in one direction. In the upper pipe we communicate in the direction from the front to the back end. The lower pipe provides only backwards communication.

6.7.2 2D-Grid Topology

The most time-consuming part in the algorithm is the reduction of two polynomials. For getting additional parallelism, we may compute all reductions of each reducer process in parallel (Figure 6.9).

```
reduce(BP,[TP|TPs],[NP|RPs]) :-
    poly:reduce(TP,BP,RP),
    poly:norm(RP,NP),
    reduce(BP,TPs,RPs)@north.

add_spol(BP,[{BP2,SP1}|SPs],[{BP2,[SP|SP1]}|SPs1]):-
       make_spol(BP,BP2,SP),
       add_spol(BP,SPs,SPs1)@south.
```

Figure 6.9: Two Dimensional Mapping

We need a second dimension of the underlying hardware topology. We construct one pipe for each S-polynomial. Each individual reduction step of two polynomials is done on a different node.

We may proceed in the same way with the processes that generate S-polynomials for mapping them in the opposite direction.

6.7.3 3D-Grid Mapping

Additionally to the two dimensional mapping strategy we may use pipe-line parallelism inside the various polynomial operations.

When reducing an S-polynomial with respect to one basis polynomial, we may reduce all monomials of the S-polynomial, that are multiples of the leading power product of the basis polynomial simultaneously by using a pipe-line (Figure 6.10).

```
-mode reduce(T?,M?,X?,Y?,Z^).
reduce(div_false,_,[P1|P1S],P2,[P1|YS]) :-
     reduce(P1S,P2,YS) .
reduce(div_true,DIVMON,[_|P1S],[P2|P2S],YS) :-
     mult_mon_poly(DIVMON,P2S,SUBPOLY),
       sub_poly_poly(P1S,SUBPOLY,REDPOLY),
       reduce(REDPOLY,[P2|P2S],YS)@fwd.
reduce(lt,_,P1,_,P1) .
```

Figure 6.10: Three Dimensional Mapping

This requires a third dimension of the underlying hardware. Now each reducible monomial is reduced on a different node.

Finer grain parallelism, like parallelism in the other polynomial operations or inside the operations over the monomials will only pay off if one has a lot of nodes. They should be connected by a fast communication network. The mapping should be done automatically. This may be possible for example in a dataflow architecture.

Chapter 7

Experiments

Here we present the performance achieved by our method on a parallel machine. The implementation is done in STRAND88 (Admiralty Release) on a 17 transputer multiprocessor machine. All timings of our algorithm are done via a clock, by hand. Thus, the time required for garbage collection is included. For the other systems we used the internal timing facilities.

For experimental purpose only a small number rational arithmetic was implemented. Also the implementation of the basic routines, where all operations are internally computed in parallel, is very memory and time intensive. Thus, only computing small examples was possible.

7.1 The Sequential Algorithm

A concurrent algorithm is only a good algorithm if it is also comparable with the best sequential versions, even on a single node. We show a few tables that will give an impression about the quality of the algorithm.

7.1.1 Statistics on the Execution

A full implementation of the criteria for detecting unnecessary S-polynomials is important for getting an efficient implementation of the algorithm. Here we present a statisic how often the various critera are actually used during the computation.

Example	var	Number of polynomials						Applying the criteria			
		in	SP	zero	max	fin	basis	C1	C2	skip	C3
1	3	3	0	0	3	3	3	3	0	0	0
2	2	3	4	1	6	6	2	6	4	3	1
3	3	2	5	2	5	5	4	2	3	3	0
4	5	3	4	2	5	5	5	6	0	0	0
5	5	3	15	9	9	9	8	13	6	6	1
6	3	3	8	3	8	5	4	11	2	2	3
7	4	4	11	6	9	8	6	10	6	6	8
8	9	5	21	12	13	11	8	22	20	11	13
9	20	14	6	3	17	17	16	127	1	1	3
10	5	5	12	7	8	8	8	8	6	4	2
11	10	5	26	13	18	18	11	39	52	34	36
12	8	4	21	13	11	11	10	6	18	15	10
13	10	5	62	36	31	31	17	85	139	96	179

The abbreviations used above have the following meanings:

var: The number of indeterminants.

in: The number of input polynomials.

SP: The number of generated S-polynomials excluding the input polynomials.

zero: The number of polynomials reduced to zero during the reduction steps.

A small number is an indication of a good use of the criteria.

max: The number of polynomials not reduced to zero, including the input polynomials.

fin: The number of polynomials in the internal Gröbner basis when the main procedure terminates.

The difference to the previous number is the number of basis polynomials reduced to zero during the reduction of the basis.

basis: The number of polynomials in the fully reduced basis.

This number should be close to the number of polynomials in the final Gröbner basis.

C1: The number of S-polynomials cancelled by the criterion on the leading power product.

C2: The number of S-polynomials cancelled by the chain criterion – pipe implementation part one.

skip: The number of tests of the chain criterion – pipe implementation part two that are skipped because of the exception in this criterion.

A big difference to $C2$ may be an indication of basis polynomials whose leading monomial is not fully reduced (see Lemma 6.5).

C3: The number of S-polynomials cancelled by the pipe criterion part two.

In this statistic one sees that most of the S-polynomials that will be reduced to zero, are cancelled by applying the criteria. There are only a few polynomials that will be reduced to zero during the reduction steps.

The fact that the number of *skip* is close to the number of cancellations shows that our (partial) normal form generates the (total) normal form most of the time.

The reduction of the intermediate basis only by reducing the basis with new polynomials will not filter out all polynomials reducible to zero. However, it will give, of course, much smaller basis polynomials. It seems that this method is a good compromise between computing overhead in reducing S-polynomials and additional management and computing overhead needed by the intrareduction of the basis.

Finally, we can say that the changes of Buchberger's algorithm we did for getting a fast parallel algorithm do not worsen the algorithm in the sequential case.

7.1.2 Comparison with Computer Algebra Systems

A parallel algorithm must be comparable with the best sequential versions. We show that our algorithm is also efficient in case of the use of only one node.

We compare the following professional Computer Algebra systems on various hardware platforms with our algorithm.

muMATH on an IBM PC AT (about half as fast than a VAX II).

MAPLE on a Micro VAX II.

SAC-2 on a Micro VAX II.

SCRATCHPAD on an IBM mainframe 4341 (about two times faster than a VAX II).

STRAND88 Our algorithm on a single transputer T800 (about three times faster than a VAX II).

The computing time is normed to seconds that a VAX II would need, but we are only interrested in the relative speed difference between the implemented algorithms in the different systems.

Example	muMATH	MAPLE	SAC-2	SCRATCHPAD	STRAND88
1	0.35	6.45	8.31	6.88	12
2	1.34	9.63	8.55	3.14	54
3	3.07	23.42	24.88	13.88	111
4	3.02	23.05	25.62	11.18	165
5	3.92	45.23	59.67	15.48	264

On this table one sees, that our algorithm is only by a constant slower than the version in muMATH, where all critera are implemented. Compared with the other systems the difference is smaller.

That the constant factor is quite high can be explained by the fact, that also the polynomial arithmetic and rational number arithmetic (involving two GCD's per operation) is written in the high level language STRAND88, where each operation is done internally in parallel. A comparison of the basic polynomial operations shows that these routines are by a factor more than fifty slower than the ones used in the C coded Maple system.

That the final difference is smaller gives an impression of the quality of the method used even in the sequential case.

For getting a better "absolute" performance one has to implement the basic polynomial arithmetic in a fast low level language like C. These should increase the "absolute" performance of the algorithm by a factor of almost 50.

7.2 Various Mapping Strategies

The used release of STRAND88 does not support the virtual topology concept. It is only possible to map new processes to the next node in a ring topology. Thus, it was necessary to simulate the various topologies by forwarding the different parallel tasks by a constant number of nodes.

A table of computing times, in seconds, is given for various examples of increasing complexity. We are only interested in the computing time of the main algorithm. Thus, the time for the final reduction is excluded.

7.2.1 Pipe-line Mapping

With this mapping strategy we map each intermediate pipe process to a different node. Thus, we get as many parallel processes as polynomials in the pipe.

The polynomials in the basis change during the execution time. Starting from the empty basis, we append non-zero polynomials to the basis. Thus, the maximum number of basis polynomials will be equal to the number of polynomials that are not reduced to zero. This number is also the maximal number of nodes used during the computation. During reduction of the basis some basis polynomials will vanish. Thus, the actual number of basis polynomials will not exceed this number.

We assume that the flow of generated basis polynomials is steady. Thus, the expected average number of polynomials in the basis is about half the maximal number of basis polynomials. The maximal speed-up of the algorithm is equal to the (changing) number of basis polynomials. Therefore, we expect a theoretical speed-up of half the maximal number of polynomials in the basis.

Example	Execution time on		Maximal number of	Speed-up	
	1 Proc.	17 Proc.	parallel processes	expected	achieved
1	4	2.5	3	1.5	1.7
2	18.5	9	6	3	2
3	36	19.5	5	2.5	1.8
4	51	26	5	2.5	2
5	75	22	9	4.5	3.4
6	43	16.5	8	4	2.6
7	47	16	9	4.5	2.9
8	115.5	30.5	13	7.5	3.8
9	133.5	28.5	17	8.5	4.7
10	292.5	107	8	4	2.7
11	528	63	18	9	8.3
12	1350[1]	182	11	5.5	7.4
13	3600[1]	115	31	15.5	31.3

This statistics shows, that the actual achieved speed-up is more than 60 percent of the theoretical maximum.

For large examples, it is possible to achive an speed-up more than the number of nodes used. The reason for this is the limitation of the memory per node and a high percentage of garbage collection during the execution. This confirms our assumption that splitting the basis over the distributed memories is essential on a distributed memory system.

[1]This example generates a heap overflow on the single node. We estimate a speed of two times the speed on two nodes.

7.2.2 Double Pipe-line

In the next variation we use two pipes. One for the reduction processes and one for the generation of S-polynomials.

This variation uses two times as many nodes as in the previous case. But the reduction processes will need much more time than the processes generating S-polynomials. Thus, the additional speed-up will not be two times the one in the previous case.

Example	Execution time on				Speed-up
	1 Proc.	3 Proc.	7 Proc.	17 Proc.	achieved
1	4	2.5	2.5	2.5	1.7
2	18.5	8.5	7.5	7	2.6
3	36	22	17	17	2.1
4	51	29	23.5	24	2.1
5	75	34	20	18	4.2
6	43	21	15	13	3.3
7	47	21.5	13.5	12	4.0
8	115.5	50	32	24.5	4.7
9	133.5	67	38	28	4.8
10	292.5	132	102	90	3.3
11	528	193	98	68	7.8
12	1350[1]	285	183	161	8.4
13	3600[1]	540	206	105	48

The time for applying the criteria and reducing the basis requires less than 20 percentage of the time. Most of the time is spent in reducing S-polynomials. Therefore, the additional speed-up is not high.

This table also shows, that we achieve a nearly linear speed-up, with a few number of nodes, also with small examples. With bigger examples, we may achieve a linear speed-up up to a high number of nodes.

7.2.3 Two Dimensional Mapping

Here, each reduction step is done on a different node.

Example	Execution time on		Difference to
	1 Proc.	17 Proc.	double pipe
1	4	2.5	0.0
2	18.5	9	+2.0
3	36	13.5	–3.5
4	51	20	–4.0
5	75	17.5	–0.5
6	43	13.5	+0.5
7	47	13	+1.0
8	115.5	26	+1.5
9	133.5	31	+3.0
10	292.5	74	–16
11	528	61	–7.0
12	1350[1]	131	–30
13	3600[1]	123	+18

Most of the time, the additional parallelism will not pay off. It will only increase the communication overhead. This shows, that all polynomials in the basis are used for reducing S-polynomials equally.

7.2.4 Three Dimensional Mapping

Here, each reducible monomial is reduced on a different node.

Example	Execution time on		Difference to
	1 Proc.	17 Proc.	2D-mapping
1	4	3.5	+1.0
2	18.5	8	–1.0
3	36	13	–0.5
4	51	17	–3.0
5	75	21	+3.5
6	43	14.5	+1.0
7	47	15	+2.0
8	115.5	34	+8.0
9	133.5	38	+7.0
10	292.5	52	–22
11	528	70	+9.0
12	1350[1]	93	–38
13	3600[1]	163	+40

This mapping strategy gives only an additional speed-up in the case, where a few reduction steps require most of the computing time. In all other cases, the communication overhead is too high.

7.2.5 Conclusion

Finally, we will collect the highest speed-up for each example.

Example	Execution time on		Fastest Method	Speed-up achieved
	1 Proc.	17 Proc.		
1	4	2.5	single pipe	1.7
2	18.5	7.0	double pipe	2.6
3	36	12.5	3d-map	2.9
4	51	17.0	3d-map	3.0
5	75	17.5	2d-map	4.3
6	43	13.0	double pipe	3.3
7	47	12.0	double pipe	4.0
8	115.5	24.5	double pipe	4.7
9	133.5	28.0	double pipe	4.8
10	292.5	52.0	3d-map	5.6
11	528	61.0	2d-map	8.7
12	1350[1]	92.0	3d-map	14.7
13	3600[1]	75.0	double pipe	48.0

From this statistics one sees that the speed-up gained is much better than for the parallel implementations of Buchberger's algorithm described in [Ponder, 1988] and [Melenk and Neun, 1988] and is comparable to the speed-up reported in [Vidal, 1989] on a shared memory machine and with much bigger examples.

- We think that this is partly due to the overall structure (pipe-line) for the solution we presented in this thesis, which is significantly different from the proposals in the literature.
- Partly we attribute the success of our implementation also to the use of a distributed transputer system in contrast to the shared-memory machines that have been considered so far.
- Also, we think that our decision to distribute the polynomials of the bases among the nodes rather than to have them in a shared region is crucial.

We observe that, although there are 17 nodes in the system, only a subset of them are actually used in the individual computations. This is so because only so many nodes can be used as there are polynomials in the basis. One may expect that in more complicated examples more polynomials are adjoined to the basis and, therefore, the speed-up will be better.

7.3 Input for the Examples

We list the input bases for the above examples.

Most of the standard examples require long integer arithmetic or are too large to be computed with our system. Thus, we modified some standard examples, by omitting polynomials, to make the computation feasible.

Example 1 (Intersection between surfaces in 3D): $z > y > x$

$$[-2z^2 + y^4 + 8x^2 - 1, z^2 + y^2 + x^2 - 2x, z^2 + y^2 + 2y + x^2]$$

Example 2 (Singularity of bivariate curves): $y > x$

$$[x^2y^2 - xy^2 + y^2 - 2x^2y + x^4, 2xy^2 - y^2 - 4xy + 4x3, 2x^2y - 2xy + 2y - 2x^2]$$

Example 3 (Implicit representation of a curve in 2D): $t > y > x$

$$[t^3 + t^2 - x - 1, -yt^2 + t^2 - y]$$

Example 4 (Implicit representation of a surface in 3D): $t > s > z > y > x$

$$[st - x + 1, t^2 - t + 4s - y, 6st - s - z]$$

Example 5 (Implicit representation of a surface in 3D): $t > s > z > y > x$

$$[st - x + 1, t^2 + 4st - t - y, 6s^2t - s - z]$$

Example 6 ([Buchberger, 1965]): $x_1 > x_2 > x_3$

$$[x_3^2 - 1/2x_1^2 - 1/2x_2^2, x_1x_3 - 2x_3 + x_1x_2, x_1^2 - x_2]$$

Example 7 (Arnborg4 [Traverso and Donati, 1989]): $x > y > z > t$

$$[x + y + z + t, xy + yz + zt + tx, xyz + yzt + ztx + txy, xyzt + 1]$$

Example 8 (Morgenstern [Traverso and Donati, 1989] modified): $x > y > a > b > c > d > r > s > t$

$$[xb - ya, xd - d - yc + y, b^2 + a^2 - r^2, c^2 - 2c + 1 + d^2 - s^2]$$

Example 9 (Geometric Proof [Kutzler and Stifter, 1986] Example 16): $y_i > y_{i-1}$

$$[2y_7 - y_1, 2y_{12} - y_2 - y_1, 2y_{13} - y_3, 2y_{14} - y_2, 2y_{15} - y_3, 2y_{16} - y_4 - y_1, 2y_{17} - y_5,$$

$$2y_8 - y_4 - y_2, 2y_9 - y_5 - y_3, 2y_{10} - y_4, 2y_{11} - y_5, y_8y_{19} - y_7y_{19} - y_9y_{18} + y_7y_9,$$
$$2y_8y_{20} - 2y_7y_{20} - y_6y_{18} + y_6y_7, -y_{12}y_{19} + y_{10}y_{19} + y_{13}y_{18} - y_{11}y_{18} - y_{10}y_{13} + y_{11}y_{12}]$$

Example 10 (Arnborg5 [Traverso and Donati, 1989] modified): $x > y > z > t > u$

$$[x + y + z + t + u, xy + yz + zt + tu + ux, xyz + yzt + ztu + tux + uxy]$$

Example 11 (Example 13 modified): $y_i > y_{i-1}$

$$[-y_5y_6 + y_3y_6 + y_2y_5 - y_3y_4, y_2y_8 - y_1y_8 - y_3y_7 + y_1y_3,$$
$$y_4y_8 - y_5y_7, y_{10} + y_3y_9, y_4y_{10} - y_1y_{10} - y_5y_9 + y_1y_5]$$

Example 12 (Pavelle4 [Pavelle, 1986] modified): $x > y > z > t > a > b > c > d$

$$[xy + xz + xt - a, yx + yz + yt - b, zx + zy + zt - c]$$

Example 13 (Geometric Proof [Kutzler and Stifter, 1986] Example 6): $y_i > y_{i-1}$

$$[-y_5y_6 + y_3y_6 + y_2y_5 - y_3y_4, y_2y_8 - y_1y_8 - y_3y_7 + y_1y_3,$$
$$y_4y_8 - y_5y_7, -y_2y_{10} + y_3y_9, y_4y_{10} - y_1y_{10} - y_5y_9 + y_1y_5]$$

Appendix A

Listing

We list the complete source code of the alorithm.

```
-compile(free).
-exports([gbasis/4,sieve/3]).

% Module: groebner

%  Main Algorithm for the Groebner basis reduction
%  call of the algorithm :
%   input:  a list of polynomials,
%           the order of the variables
%   output: the reduced lexical groebner basis


% Abreviations of the variables in the program

%  P   ... Polynomial
%  M   ... Monomial
%  C   ... Coefficent
%  PP  ... Powerproduct

%  B   ... Basis
%  SP  ... S-polynomial

%  T   ... Test
%  R   ... Reduced
%  N   ... Normed
%  H   ... Help Value
%  ..s ... Stream of ..
```

```
% Combinations of the above names have analogous meaning.


% Tagged Variables are:

%  p(..)   ... polynom
%  nr(..)  ... round number
%  up(..)  ... update of the basis
%  fin(..) ... final groebner basis
%  sp(..)  ... S-Polynomial


gbasis(PolySet,Ord,OutBasis) :-
     polypars:poly_list(PolySet,Ord,Ps),
     main(Ps,GBasis,GB),
     full_reduction(GB,RB),
     polypars:out_poly_list(GB,Ord,OutBasis).


% main loop of the algorithm

main(InPoly,Basis,GB):-
   list:append(InPoly,[nr(0)],Poly),
   append_spol(0,Poly,Spol,Testpoly,GB),
   groebner:sieve(Testpoly,Spol,Basis)@next.


% generate a new basis polynomial and a new reduction process

-mode sieve(TestPolys?,SP^,B^).
sieve([nr(Cnt)|TPs],[nr(Cnt)|SPs],B):-
   sieve(TPs,SPs,B).
sieve([p(BP)|TPs],[{BP,[],[]}|SPs1],[BP|BPs]):-
   BP =\= [] |
       reduce(BP,TPs,RPs),
       groebner:sieve(RPs,SPs,BPs)@next,
       add_spol(BP,SPs,SPs1).
sieve([p([])|TPs],SPs,B):-          %      display_nl("zero reduction")!,
   sieve(TPs,SPs,B).
sieve([],[fin([])],[]).
```

```
% creates and inserts Spolynomials into the Spoly-list

-mode add_spol(B?,SP?,SP1^).
add_spol(BP,[{BP2,SP,B}|SPs],[{BP2,SP1,B1}|SPs1]):-
   BP =\= [] |
       poly:reduce(BP,BP2,RP),
       poly:norm(RP,NP),
       ord:in_sort(down,NP,B,B1),
       insert_spol(NP,BP2,B,SP,SP1),
       add_spol(NP,SPs,SPs1).
add_spol(BP,[nr(Cnt)|SPs],[nr(Cnt)|SPs1]):-
   add_spol(BP,SPs,SPs1).
add_spol([],X,X).
add_spol(BP,[fin(BPs)],[fin(B)]):-
       ord:in_sort(down,BP,BPs,B).


% reduces a list of Testpolys wrt. one Basiselement

-mode reduce(B?,TP?,RP^).
reduce(BP,[p(TP)|TPs],[p(NP)|RPs]) :-
   BP=\=[] |
       poly:reduce(TP,BP,RP),
       poly:norm(RP,NP),
       reduce(BP,TPs,RPs).
reduce(_,[up([BP|BPs])|TPs],[up(BPs)|RPs]):-
   reduce(BP,TPs,RPs).
reduce(BP,[up([])|TPs],RPs):-
   reduce(BP,TPs,RPs).
reduce(BP,[nr(Cnt)|TPs],[nr(Cnt)|RPs]) :-
   reduce(BP,TPs,RPs).
reduce([],TPs,TPs).
reduce(_,[],[]).


% generates and inserts one new S-poly

-mode insert_spol(BP1?,BP2?,Basis?,SP?,SP1^).
insert_spol(BP,BP2,B,SPs,SPs1):-
   BP =\= [] |
       criterion2(BP,SPs,SPs2,false,T),
       mk_spol(T,BP,BP2,B,SP),
       in_sort(SP,SPs2,SPs1).
```

```
insert_spol([],_,_,SPs,SPs).


% inserts a polynomial into an ascending sorted SP-list

-mode in_sort(SP1?,SP2?,Sort^).
in_sort(sp(LCM1,SP1),[sp(LCM2,SP2)|SPs],Sort):-
   ord:leqg(SP1,SP2,T),
   in_sort(T,sp(LCM1,SP1),sp(LCM2,SP2),SPs,Sort).
in_sort(SP1,[],[SP1]).
in_sort([],SP1,SP1).

-mode in_sort(Test,SP1?,SP2?,SPs?,Sort^).
in_sort(gt,SP1,SP2,SPs,[SP2|Sort]):-
   in_sort(SP1,SPs,Sort).
in_sort(lt,SP1,SP2,SPs,[SP1,SP2|SPs]).
in_sort(eq,SP1,SP2,SPs,[SP1,SP2|SPs]).


% Compute one S-polynomial
% The coefficient of the leading power product must be 1

-mode mk_spol(Crit?,P1?,P2?,Basis?,S^).
mk_spol(Crit,[{C1,PP1}|P1],[{C2,PP2}|P2],B,SP) :-
     poly:ggt_lcm_pp(PP1,PP2,GGT,LCM),
     criterion1(GGT,Crit,Crit1),
     criterion3(LCM,B,Crit1,Crit2),
     mk_spol2(GGT,LCM,Crit2,[{C1,PP1}|P1],[{C2,PP2}|P2],SP) .

-mode mk_spol2(GGT?,LCM?,T?,P1?,P2?,S^).
mk_spol2(GGT,LCM,false,[{C1,PP1}|P1],[{C2,PP2}|P2],sp(LCM,SP)) :-
     poly:div_pp_pp(PP1,GGT,H1),
     poly:div_pp_pp(PP2,GGT,H2),
     poly:mult_mon_poly({C2,H2},P1,H3),
     poly:mult_mon_poly({C1,H1},P2,H4),
     poly:sub_poly_poly(H3,H4,H5),
     poly:norm(H5,SP) .
mk_spol2(_,_,true,_,_,[]) .


% Buchberger's criterion on leading PP

-mode criterion1(I?,T1?,T2^).
```

```
criterion1([0|X],T1,T2) :-
     criterion1(X,T1,T2) .
criterion1([X|_],T,T) :-
     X=\=0 | true.
criterion1([],_,true).            %     display_nl("apply crit1")!.

% Buchberger's chain criterion

-mode criterion3(LCM?,Basis?,Crit1?,Crit2^).
criterion3(LCM,[[{_,PP}|_]|BPs],false,Crit) :-
   ord:divtest(pp,LCM,PP,Test,_),
   criterion3(Test,LCM,BPs,false,Crit) .
criterion3(LCM,[[]|BPs],false,Crit) :-
   criterion3(LCM,BPs,false,Crit) .
criterion3(_,_,true,true).
criterion3(_,_,skip,false).            %       display_nl("skip crit3")!.
criterion3(_,[],false,false).

-mode criterion3(Test?,LCM?,Basis?,Crit1?,Crit2^).
criterion3(Test,LCM,B,Crit1,Crit2) :-
   Test =\= true |
       criterion3(LCM,B,Crit1,Crit2) .
criterion3(true,_,_,_,true).           %       display_nl("apply crit3")!.

-mode criterion2(P?,Spols?,SPs^,T1?,T2^).
criterion2([{C,PP}|_],[sp(LCM,SP)|SPs],SPs1,T1,T2):-
   ord:divtest(pp,LCM,PP,T,_),
   criterion2(T,{C,PP},sp(LCM,SP),SPs,SPs1,T1,T2).
criterion2(X,[[]|Y],SPs,T1,T2):-
   criterion2(X,Y,SPs,T1,T2).
criterion2(_,[],[],T,T).

-mode criterion2(Test,Lmon?,X?,Xs?,SPs^,T1?,T2^).
criterion2(true,Lmon,_,Xs,Ys,_,skip):-  %       display_nl("apply crit2")!,
   criterion2([Lmon],Xs,Ys,_,_).
criterion2(T,Lmon,X,Xs,[X|Ys],T1,T2):-
   T=\=true | criterion2([Lmon],Xs,Ys,T1,T2).

%   deleting of empty SPs and append to the list

-mode append_spol(Count?,TP?,SP?,TP1^,GroebnerBasis^).
append_spol(Cnt,[TP|TPs],SPs,[TP|TPs1],GB) :-
   tuple(TP) | append_spol(Cnt,TPs,SPs,TPs1,GB) .
```

```
append_spol(Count,[],[nr(Count1)|SP],TP,GB):-
   Count > Count1 |
       append_spol(Count,[],SP,TP,GB).
append_spol(Cnt,[],[{_,SP,B}|SPs],TP,GB):-
   filter_sp(SP,SP1),
   append_spol(Cnt,B,SP1,SPs,TP,GB).
append_spol(Cnt,[P|Ps],SPs,[p(P)|TPs],GB) :-
   list(P) |
       append_spol(Cnt,Ps,SPs,TPs,GB) .
append_spol(Count,[],[nr(Count)|X],[],GB):-
   finish(X,GB).

-mode append_spol(Count?,B?,SP?,SPs?,TP^,GB^).
append_spol(Cnt,B,SP,SPs,[up(B)|TP],GB):-
   SP=\=[] |
       Count is Cnt+1,
       list:append(SP,[nr(Count)],Round),
       append_spol(Count,Round,SPs,TP,GB).
append_spol(Cnt,_,[],SPs,TP,GB):-
   append_spol(Cnt,[],SPs,TP,GB).

-mode filter_sp(SP?,SP1^).
filter_sp([sp(_,P)|SPs],[p(P)|SPs1]):-    %       display_nl("new SP")!,
   filter_sp(SPs,SPs1).
filter_sp([[]|SPs],SPs1):-
   filter_sp(SPs,SPs1).
filter_sp([],[]).

-mode finish(Fin?,Basis^).
finish([fin(B)],B).


% reduces each poly w.r.t. all smaller ones

-mode full_reduction(?,^).
full_reduction([BP|BPs],[RP|RPs]) :-
     red_basis(BP,BPs,RP),
     full_reduction(BPs,RPs).
full_reduction([],[]).


% reduce one basispolynomial w.r.t. the descendent sorted basis.
```

```
-mode red_basis(?,?,^).
red_basis([M|Ms],[[M1|M1s] | Bs], RB):-
   ord:divtest(m,M,M1,Test,_),
   red_poly(Test,[M|Ms],[M1|M1s],RBP),
   red_basis(RBP,Bs,RB).
red_basis(BP,[],BP) .
red_basis([],_,[]).
red_basis(BP,[[]|_],BP) .

-mode red_poly(?,?,?,^).
red_poly(false,BP,BP1,NP) :-
    poly:reduce(BP,BP1,RP),
    poly:norm(RP,NP).
red_poly(true,_,_,[]).
%red_poly(lt,BP,_,BP).       % not possible: basis is sorted

=================================================================

-compile(free).
-exports([ggt_lcm_pp/4,sub_poly_poly/3,mult_mon_poly/3,div_pp_pp/3,
       reduce/3,norm/2]).

% Module : poly

% Polynomial operation over the rational numbers


-mode mult_mon_poly(X?,Y?,Z^).
% Multiplies a monomial with a polynomial

mult_mon_poly(MON,[LMON|POLS],[Y|YS]) :-
     mult_mon_mon(MON,LMON,Y), mult_mon_poly(MON,POLS,YS) .
mult_mon_poly(_,[],[]) .


-mode mult_mon_mon(X?,Y?,Z^).
% Multiplies two monomials

mult_mon_mon({C1,PP1},{C2,PP2},{CY,PPY}) :-
     rat:mult(C1,C2,CY), mult_pp_pp(PP1,PP2,PPY) .


-mode mult_pp_pp(X?,Y?,Z^).
```

```
% Multiplies two powerproducts

mult_pp_pp([PP1|PP1S],[PP2|PP2S],[Y|YS]) :-
     Y is PP1+PP2, mult_pp_pp(PP1S,PP2S,YS) .
mult_pp_pp([],[],[]) .


-mode norm(X?,Z^).
% normalise the polynomial such that leadingcoeff. is 1. %

norm([{C,PP}|POLY],[{{1,1,1},PP}|Y]) :-
     div_rat_poly(C,POLY,Y) .
norm([],[]) .


-mode div_rat_poly(X?,Y?,Z^).
% divides a polynomial by  a rational number

div_rat_poly(RAT,[{C,PP}|POLY],[{YC,PP}|Y]) :-
     rat:divide(C,RAT,YC), div_rat_poly(RAT,POLY,Y) .
div_rat_poly(_,[],[]) .


-mode div_pp_pp(X?,Y?,Z^).
% division of two powerproducts

div_pp_pp([PP1|PP1S],[PP2|PP2S],[Y|YS]) :-
     Y is PP1-PP2, div_pp_pp(PP1S,PP2S,YS) .
div_pp_pp([],[],[]) .


-mode ggt_lcm_pp(X?,Y?,G^,L^).
% computes the ggt and lcm of two powerproducts

ggt_lcm_pp([PP1|PP1S],[PP2|PP2S],[PP1|YG],[PP2|YL]) :-
     PP1 =< PP2 | ggt_lcm_pp(PP1S,PP2S,YG,YL) .
ggt_lcm_pp([PP1|PP1S],[PP2|PP2S],[PP2|YG],[PP1|YL]) :-
     PP1 > PP2 | ggt_lcm_pp(PP1S,PP2S,YG,YL) .
ggt_lcm_pp([],[],[],[]) .


-mode sub_poly_poly(X?,Y?,Z^).
% subtraction of polynomials
```

```
sub_poly_poly([P1|P1S],[P2|P2S],Y) :-
     P1 =\= P2 | ord:leqg_mon(P1,P2,TEST),
       sub_poly_poly(TEST,[P1|P1S],[P2|P2S],Y) .
sub_poly_poly([P1|P1S],[P2|P2S],Y) :-
     P1 == P2 | sub_poly_poly(P1S,P2S,Y) .
sub_poly_poly([],[P|PS],[Y|YS]) :-
     neg_mon(P,Y), sub_poly_poly([],PS,YS) .
sub_poly_poly(X,[],X).

-mode sub_poly_poly(T?,X?,Y?,Z^).
sub_poly_poly(lt,P1,[P2|P2S],[Y|YS]) :-
     neg_mon(P2,Y), sub_poly_poly(P1,P2S,YS) .
sub_poly_poly(gt,[P1|P1S],P2,[P1|YS]) :-
     sub_poly_poly(P1S,P2,YS) .
sub_poly_poly(eq,[[C1|_]|P1S],[[C2|_]|P2S],Y):-
     C1 == C2 | sub_poly_poly(P1S,P2S,Y).
sub_poly_poly(eq,[P1|P1S],[P2|P2S],[Y|YS]) :-
     otherwise | sub_mon_mon(P1,P2,Y),
       sub_poly_poly(P1S,P2S,YS) .


-mode sub_mon_mon(X?,Y?,Z^).
% subtraction of two monomials with the same powerproduct

sub_mon_mon({C1,PP},{C2,_},{Y,PP}) :-
     rat:minus(C1,C2,Y) .


-mode neg_mon(X?,Z^).
% negate a monom

neg_mon({{S,C1,C2},PP},{{Y,C1,C2},PP}) :-
     Y is - S .


-mode reduce(X?,Y?,Z^).
% reduces a polynomial wrt. a second polynomial

reduce([P1|P1S],[P2|P2S],Y) :-
     ord:divtest(m,P1,P2,TEST,DIVMON),
       reduce(TEST,DIVMON,[P1|P1S],[P2|P2S],Y) .
reduce([],_,[]).
```

```
reduce(X,[],X).

-mode reduce(T?,M?,X?,Y?,Z^).
reduce(false,_,[P1|P1S],P2,[P1|YS]) :-
     reduce(P1S,P2,YS) .
reduce(true,DIVMON,[_|P1S],[P2|P2S],YS) :-
     mult_mon_poly(DIVMON,P2S,SUBPOLY),
       sub_poly_poly(P1S,SUBPOLY,REDPOLY),
       poly:reduce(REDPOLY,[P2|P2S],YS)@next.
reduce(lt,_,P1,_,P1) .

================================================================

-compile(free).
-exports([in_sort/4,leqg/3,leqg_mon/3,divtest/5]).

% Module : ord      order specific rules; lexical order


-mode in_sort(Ord,X?,List?,Sort^).
% inserts a polynomial in into a sorted polynomial list
%  Ord:  down: descendial
%        up:   ascendial
%  X:    polynomial
%  List: sorted list of polynomials

in_sort(Ord,X,[Y|Ys],Sort):-
   leqg(X,Y,Test),
   in_sort(Test,Ord,X,Y,Ys,Sort).
in_sort(_,X,[],[X]).

-mode in_sort(Test,Ord,X?,Y?,Ys?,Sort^).
in_sort(lt,down,X,Y,Ys,[Y|Sort]):-
    in_sort(down,X,Ys,Sort).
in_sort(gt,up,X,Y,Ys,[Y|Sort]):-
    in_sort(up,X,Ys,Sort).
in_sort(gt,down,X,Y,Ys,[X,Y|Ys]).
in_sort(lt,up,X,Y,Ys,[X,Y|Ys]).
in_sort(eq,_,X,Y,Ys,[X,Y|Ys]).


-mode leqg(X?,Y?,Test^).
% tests if a polynomial has a lower, equal or greater leading monomial
```

```
% X,Y:   polynomials
% Test:  lt, eq, gt

leqg([X|_],[Y|_],Test):-
    leqg_mon(X,Y,Test).
leqg([],[],eq).
leqg([],Y,lt):-
   Y =\= [] | true.
leqg(X,[],gt):-
   X =\= [] | true.


-mode leqg_mon(X?,Y?,T^).
% test if monomial X has a lower, equal or greater pp as Y
% lexical ordering is asumed

leqg_mon({_,PP1},{_,PP2},Y) :-
     leqg_pp(PP1,PP2,Y) .

-mode leqg_pp(X?,Y?,T^).
leqg_pp([V1|V1S],[V2|V2S],Y) :-
     V1==V2 | leqg_pp(V1S,V2S,Y) .
leqg_pp([V1|_],[V2|_],lt) :-
     V1<V2 | true.
leqg_pp([V1|_],[V2|_],gt) :-
     V1>V2 | true.
leqg_pp([],[],eq) .


-mode divtest(Type?,X?,Y?,Test^,Z^).
% test if X divides Y
% Type: Type of X and Y -  pp (powerproduct), m (monomial)
% Test: lt       - if X < Y
%       false - if Y not divides X
%       true  - if Y divides X
% Z:    the divisor, if Y divides X

divtest(pp,PP1,PP2,T,YPP):-
   divtest_pp(eq,PP1,PP2,T,YPP).

divtest(m,{C1,PP1},{C2,PP2},TEST,{YC,YPP}) :-
     rat:divide(C1,C2,YC), divtest_pp(eq,PP1,PP2,TEST,YPP) .
```

```
-mode divtest_pp(T1?,X?,Y?,T2^,Z^).
divtest_pp(_,[PP1|PP1S],[PP2|PP2S],TEST,[PPY|PPYS]) :-
     PP1>PP2 | PPY is PP1-PP2,
       divtest_pp(gt,PP1S,PP2S,TEST,PPYS) .
divtest_pp(ISGR,[PP1|PP1S],[PP2|PP2S],TEST,[0|PPYS]) :-
     PP1==PP2 | divtest_pp(ISGR,PP1S,PP2S,TEST,PPYS) .
divtest_pp(eq,[PP1|_],[PP2|_],lt,_) :-
     PP1<PP2 | true .
divtest_pp(gt,[PP1|_],[PP2|_],false,_) :-
     PP1<PP2 | true .
divtest_pp(_,[],[],true,[]) .

================================================================

-compile(free).
-exports([mult/3,divide/3,minus/3,plus/3]).

% Module rat:  An implementation of the rational Numbers

% Abreviations:

%  S   ... sign
%  N   ... numerator
%  D   ... denominator

%  Data structure of a Rational Number:

%  { S , N , D }


%  Z :=  X * Y

-mode mult(X?,Y?,Z^).
mult({XS,XN,XD},{YS,YN,YD},{ZS,ZN,ZD}) :-
   ZS is XS*YS, ggt(XN,YD,G1), ggt(YN,XD,G2),
   ZN is XN / G1*(YN / G2),
   ZD is XD / G2*(YD / G1) .

%  Z :=  X / Y

-mode divide(X?,Y?,Z^).
divide(X,{YS,YN,YD},Z) :-
   mult(X,{YS,YD,YN},Z) .
```

```
%  Z :=  X + Y

-mode plus(X?,Y?,Z^).
plus({XS,XN,XD},{YS,YN,YD},{ZS,ZN,ZD}) :-
   ggt(XD,YD,G1), HD is XD/G1*YD,
   HN is XN*(YD/G1)*XS+YN*(XD/G1)*YS,
   sign(HN,ZS), HPOSN is HN*ZS,
   ggt(HPOSN,HD,G2), ZN is HPOSN/G2,
   ZD is HD/G2 .

%  Z :=  X - Y

-mode minus(X?,Y?,Z^).
minus(X,{YS,YN,YD},Z) :-
   S is -YS, plus(X,{S,YN,YD},Z) .


% help functions

-mode ggt(X?,Y?,Z^).
ggt(X,Y,Z) :-
     Y>0,X>=Y | REST is X // Y, ggt(Y,REST,Z) .
ggt(X,0,X).
ggt(X,Y,Z) :-
     X<Y | ggt(Y,X,Z) .
ggt(X,Y,Z) :-
     Y<0 | Z:=error(negative_number,Y) .

%    sign(X):    -1 if X<0 ,   1 else

-mode sign(X?,Y^).
sign(X,1) :-
   X>=0 | true .
sign(X,-1) :-
   X<0 | true .

================================================================

-compile(free).
-exports([poly_list/3,out_poly_list/3]).

% Module polypars: A parser for polynomials over the rational numbers
```

```
% Syntax of the Input Polynomiallist:

% PL  ::=  "[" P { "," P } "]"
% P   ::=  M { ("+","-") M }
% M   ::=  C | [C "*"] Pot { "*" Pot }
% C   ::=  Num | Num "/" Num
% Pot ::=  Var | Var "/\" Num
% Var ::=  string
% Num ::=  positve short Integer


-mode poly_list(In?,Ord?,Out^).
% input-parser for a polynomial list in distributive term form to the list
% notation in pure lexical order sorted ascending.

poly_list([P|Ps],ORD,Sort) :-
   poly(P,ORD,Y),
   poly:norm(Y,Npol),
   ord:in_sort(up,Npol,Ys,Sort),
   poly_list(Ps,ORD,Ys) .
poly_list([],_,[]) .

-mode poly(In?,Ord?,Out^).
poly(X+Y,ORD,Z) :-
   poly(X,ORD,XP), mon(1,Y,ORD,YP),
   insert(YP,XP,Z) .
poly(X-Y,ORD,Z) :-
   poly(X,ORD,XP), mon(-1,Y,ORD,YP),
   insert(YP,XP,Z) .
poly(- X,ORD,[MON]) :-
   mon(-1,X,ORD,MON) .
poly(0,_,[]).
poly(X,ORD,[MON]) :-
   otherwise | mon(1,X,ORD,MON) .

-mode insert(X?,Y?,Z^).
insert(X,[Y|YS],Z) :-
     ord:leqg_mon(X,Y,T),
     insert(T,X,Y,YS,Z) .
insert(X,[],[X]) .

-mode insert(T,X,Y,YS?,Z^).
```

```
insert(lt,X,Y,YS,[Y|Z]):-
   insert(X,YS,Z).
insert(gt,X,Y,Ys,[X,Y|Ys]).

-mode mon(S?,X?,O?,Z^).
mon(S,X,ORD,{C,PP}) :-
     mk_pp0(ORD,PP0), mon(S,X,ORD,PP0,C,PP) .

-mode mon(S?,X?,O?,Y?,C^,PP^).
mon(S,X*Y,ORD,PP_OLD,C,PP) :-
     insert_pp(Y,ORD,PP_OLD,PP1),
     mon(S,X,ORD,PP1,C,PP) .
mon(S,X,ORD,PP_OLD,{S1,Cn1,Cd},PP):-
     otherwise | is_rat(X,Cn,Cd,Test),
       sign(S,Cn,S1,Cn1),
       is_var(Test,X,ORD,PP_OLD,PP).

-mode sign(S?,Cn?,S1^,Cn1^).
sign(S,Cn,S,Cn):-
   Cn > 0 | true.
sign(S,Cn,S1,Cn1):-
   Cn < 0 | S1 is -S, Cn1 is -Cn.

-mode is_var(T?,X?,O?,Y?,Z^).
is_var(rat_true,_,_,PP,PP).
is_var(rat_false,X,ORD,PP_old,PP):-
     insert_pp(X,ORD,PP_old,PP) .

-mode is_rat(X?,Cn^,Cd^,T^).
is_rat(X/Y,X,Y,rat_true).
is_rat(X,X,1,rat_true) :-
     integer(X) | true.
is_rat(_,1,1,rat_false):-
   otherwise | true.

-mode mk_pp0(X?,Z^).
mk_pp0([_|X],[0|Y]) :-
     mk_pp0(X,Y) .
mk_pp0([],[]) .

-mode insert_pp(X?,O?,Y?,Z^).
insert_pp(X/\DEG,ORD,OLDPP,PP) :-
    insert_pp(X,DEG,ORD,OLDPP,PP) .
```

```
insert_pp(X,ORD,OLDPP,PP) :-
    otherwise | insert_pp(X,1,ORD,OLDPP,PP) .

-mode insert_pp(V?,D?,O?,Y?,Z^).
insert_pp(_,_,[],PP,PP).
insert_pp(X,DEG,[V|_],[_|PP],[DEG|PP]):-
     X==V | true.
insert_pp(X,DEG,[_|ORD],[PP|PPS],[PP|PPY]) :-
     otherwise | insert_pp(X,DEG,ORD,PPS,PPY) .


-mode out_poly_list(X?,Y?,Z^).
% Pretty Printer of a Polynomlist

out_poly_list([P1|PS],ORD,[Y|YS]) :-
     out_poly(P1,0,ORD,Y), out_poly_list(PS,ORD,YS) .
out_poly_list([],_,[]) .

-mode out_poly(X?,Y?,O?,Z^).
out_poly([{{-1,Cn,Cd},X}|XS],YP,ORD,ZP) :-
     out_mon({{-1,Cn,Cd},X},ORD,XP), out_poly(XS,YP-XP,ORD,ZP) .
out_poly([],X,_,X) .
out_poly([X|XS],0,ORD,ZP) :-
     otherwise | out_mon(X,ORD,XP), out_poly(XS,XP,ORD,ZP) .
out_poly([X|XS],YP,ORD,ZP) :-
     otherwise | out_mon(X,ORD,XP), out_poly(XS,YP+XP,ORD,ZP) .

-mode out_mon(X?,Y?,Z^).
out_mon({C,PP},ORD,PPO) :-
     out_rat(C,CO), out_pp(PP,CO,ORD,PPO) .

-mode out_rat(X?,Z^).
out_rat({_,NUM,1},NUM).
out_rat({_,NUM,DEN},NUM/DEN):-
     otherwise | true.

-mode out_pp(X?,Y?,V?,Z^).
out_pp([0|XS],YP,[_|VS],ZP) :-
     out_pp(XS,YP,VS,ZP) .
out_pp([X|XS],1,[V|VS],ZP) :-
     X=\=0 | out_pot(X,V,XP), out_pp(XS,XP,VS,ZP) .
out_pp([],X,_,X) .
out_pp([X|XS],YP,[V|VS],ZP) :-
```

```
     otherwise | out_pot(X,V,XP), out_pp(XS,YP*XP,VS,ZP) .

-mode out_pot(X?,Y?,Z^).
out_pot(1,V,V).
out_pot(X,V,V /\ X):-
   otherwise | true .

================================================================

-compile(free).
-exports([test/3]).

% Module : Test

% Test Examples for the Groebner Basis Algorithm


% Intersection
test(1,O,B):-
   groebner:gbasis(
           [ -2*z/\2 + y/\4 +8*x/\2 -1 ,
              z/\2 + y/\2 + x/\2 -2*x ,
              z/\2 + y/\2 +2*y + x/\2 ],
      [z,y,x], O, B).

% Singularity
test(2,O,B):-
   groebner:gbasis(
           [  x/\2*y/\2 - x*y/\2 + y/\2 -2*x/\2*y + x/\4 ,
             2*x*y/\2 - y/\2 -4*x*y +4*x/\3 ,
             2*x/\2*y -2*x*y +2*y -2*x/\2 ],
      [y,x], O, B).

% Implicitization
test(3,O,B):-
   groebner:gbasis(
           [  t/\3 + t/\2 - x -1 ,
             -1*y*t/\2 + t/\2 - y ],
      [t,y,x], O, B).

% Implicitization
test(4,O,B):-
   groebner:gbasis(
```

```
            [  s*t - x +1 ,
               t/\2 - t +4*s - y ,
              6*s*t - s - z ],
      [t,s,z,y,x], O, B).

% Implicitization
test(5,O,B):-
   groebner:gbasis(
           [  s*t - x +1 ,
               t/\2 +4*s*t - t - y ,
              6*s/\2*t - s - z ],
      [t,s,z,y,x], O, B).

% Buchberger's Standard Test
test(6,O,B):-
   groebner:gbasis(
           [ x3/\2-1/2*x1/\2-1/2*x2/\2,
             x1*x3-2*x3+x1*x2,
             x1/\2-x2],
      [x1,x2,x3], O, B).

% Arnborg4
test(7,O,B):-
   groebner:gbasis(
           [ x+y+z+t,
             x*y+y*z+z*t+t*x,
             x*y*z+y*z*t+z*t*x+t*x*y,
             x*y*z*t+1],
           [x,y,z,t], O, B).

% Morgenstern modified
test(8,O,B):-
   groebner:gbasis(
           [ x*b-y*a,
             x*d-d-y*c+y,
             b/\2+a/\2-r/\2,
             c/\2-2*c+1+d/\2-s/\2
%             a/\2-2*a*c+c/\2+b/\2-2*b*d+d/\2-t/\2
],
           [x,y,a,b,c,d,r,s,t], O, B).

% Kutzler16
test(9,O,B):-
```

```
   groebner:gbasis(
             [ 2*y7-y1,
               2* y12-y2-y1,
               2*y13-y3,
               2*y14-y2,
               2*y15-y3,
               2*y16-y4-y1,
               2*y17-y5,
               2*y8-y4-y2,
               2*y9-y5-y3,
               2*y10-y4,
               2*y11-y5,
               y8*y19-y7*y19-y9*y18+y7*y9,
               2*y8*y20-2*y7*y20-y6*y18+y6*y7,
               -1*y12*y19+y10*y19+y13*y18-y11*y18-y10*y13+y11*y12
               ],
   [y20,y19,y18,y17,y16,y15,y14,y13,y12,y11,y10,y9,y8,y7,y6,y5,y4,y3,y2,y1],
   0, B).

% Arnborg5  modified
test(10,0,B):-
   groebner:gbasis(
           [
           x+y+z+t+u,
             x*y+y*z+z*t+t*u+u*x,
             x*y*z+y*z*t+z*t*u+t*u*x+u*x*y
%             x*y*z*t+y*z*t*u+z*t*u*x+t*u*x*y+u*x*y*z
%             x*y*z*t*u+1
],
           [x,y,z,t,u], 0, B).

% Kutzler6 modified
test(11,0,B):-
   groebner:gbasis(
            [ -1*y5*y6+y3*y6+y2*y5-y3*y4,
              y2*y8-y1*y8-y3*y7+y1*y3,
              y4*y8-y5*y7,
              y10+y3*y9,
              y4*y10-y1*y10-y5*y9+y1*y5],
            [y10,y9,y8,y7,y6,y5,y4,y3,y2,y1], 0, B).

% Pavelle4 modified
test(12,0,B):-
```

```
   groebner:gbasis(
           [ x*y+x*z+x*t-a,
             y*x+y*z+y*t-b,
             z*x+z*y+z*t-c
%             t*x+t*y+t*z-d
],
           [x,y,z,t,a,b,c,d], 0, B).

% Kutzler6
test(13,0,B):-
   groebner:gbasis(
            [ -1*y5*y6+y3*y6+y2*y5-y3*y4,
              y2*y8-y1*y8-y3*y7+y1*y3,
              y4*y8-y5*y7,
              -1*y2*y10+y3*y9,
              y4*y10-y1*y10-y5*y9+y1*y5],
            [y10,y9,y8,y7,y6,y5,y4,y3,y2,y1], 0, B).
```

Bibliography

[Buchberger and Hintenaus, 1985] Bruno Buchberger and Peter Hintenaus.
The L-Language for the Parallel L-Machine — A Parallel Architecture for AI Applications.
In Trost and Retti, editors, *ÖGAI-Tagung 1985*, volume 106 of *Informatik Fachberichte*, pages 120–131, Vienna, Austria, September 24–27, 1985. Springer.
Also: Technical Report 85-4, RISC-Linz, Johannes Kepler University, Linz, Austria, 1985.

[Buchberger, 1965] Bruno Buchberger.
Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal (An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-dimensional Polynomial Ideal) (In German).
PhD thesis, University of Innsbruck, Math. Dept., Austria, 1965.

[Buchberger, 1970] Bruno Buchberger.
Ein algorithmisches Kriterium für die Lösbarkeit eines algebraischen Gleichungssystems (An Algorithmical Criterion for the Solvability of Algebraic Systems of Equations) (In German).
Aequationes mathematicae, 4(3):374–383, 1970.

[Buchberger, 1978] Bruno Buchberger.
Computer-Trees and their Programming.
In Dauchet, editor, *4th Colloquium "Trees in algebra and programming"*, pages 1–18, University of Lille, February 16–18, 1978.
Also: Technical Report 78-11, RISC-Linz, Johannes Kepler University, Linz, Austria, 1978.

[Buchberger, 1985a] Bruno Buchberger.
Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory.
In N.K. Bose and D. Reidel, editors, *Recent trends in Multidimensional Systems*, chapter 6, pages 184–232. D. Reidel Publishing Company, Dordrecht-Boston-Lancaster, 1985.

[Buchberger, 1985b] Bruno Buchberger.
The L-Machine: An Attempt at Parallel Hardware for Symbolic Computation.

In J. Calmet, editor, *3rd International Conference on Algebraic Algorithms and Error-Correcting Codes*, volume 229 of *Lecture Notes in Computer Science*, pages 333–347, Grenoble, France, July 15–19, 1985. Springer.
Also: Technical Report 85-24, RISC-Linz, Johannes Kepler University, Linz, Austria, 1985.

[Buchberger, 1987a] Bruno Buchberger.
Applications of Gröbner Bases in Non-Linear Computational Geometry.
In *Proc. Workshop on Scientific Software (invited lecture)*, volume 14 of *IMA Volumes in Mathematics and its Applications*, pages 59–88, IMA, Minneapolis, USA, March 1987. Springer.
Also: Technical Report 87-15, RISC-Linz, Johannes Kepler University, Linz, Austria, 1987.

[Buchberger, 1987b] Bruno Buchberger.
The Parallelization of Critical-Pair/Completion Procedures on the L-Machine.
In *Japanese Symposium on Functional Programming*, pages 54–61, February 1987.
Also: Technical Report 87-12, RISC-Linz, Johannes Kepler University, Linz, Austria, 1987.

[Buchberger, 1987c] Bruno Buchberger.
User's Manual for GROEBNER, a Gröbner Bases Package in the muMATH System.
Technical Report 87-38.0, RISC-Linz, Johannes Kepler University, A-4040 Linz, Austria, 1987.

[Buchberger, 1990] Bruno Buchberger.
Gröbner Bases and Generalized Sylvester Matrices.
In *Symposium on Symbolic Computation at the Occasion of E. Engeler's 60th Birthday (invited lecture)*, ETH Zürich, March 3–5 1990.

[Clark and Gregory, 1984] Keith Clark and Steve Gregory.
PARLOG: Parallel Programming in Logic.
Research Report DOC 84/4, Department of Computing, Imperial College of Science and Technology, University of London, April 1984.
Revised June 1985.

[Conery, 1983] John S. Conery.
The AND/OR Process Model for Parallel Interpretation of Logic Programs.
PhD thesis, Univ. of California, 1983.
Computer and Information Science Tech. Rep. 204.

[Conery, 1987] John S. Conery.
Parallel Execution of Logic Programs.
The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Boston, Massachusetts, 1987.

[Foster and Taylor, 1989] Ian Foster and Stephen Taylor.

Strand — New Concepts in Parallel Programming.
Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[Futo and Kacsuk, 1989] Ivan Futo and Peter Kacsuk.
CS-Prolog on Multitransputer Systems.
Microprocessors and Microsystems, 13(2):103–112, March 1989.

[Inmos, 1985] Inmos Limited, Bristol, UK.
IMS B004 Evaluation Board User Manual, 1985.

[Inmos, 1988] Inmos.
Occam 2 Reference Manual.
Prentice Hall, New York, 1988.

[Kale, 1987] L. V. Kale.
Parallel Execution of Logic Programs: The Reduce-Or Process Model.
In *Proc. Int. Conference on Logic Programming*, pages 616–632, Melbourne, May 1987.

[Kasif *et al.*, 1983] S. Kasif, M. Kohli, and J. Minker.
PRISM: A Parallel Inference System for Problem Solving.
In *Proceedings of the Eigth International Joint Conference on Artificial Intelligence*, pages 544–546, Karlsruhe, Germany, August 8-12 1983.

[Kutzler and Stifter, 1986] B. Kutzler and S. Stifter.
Collection of Computerized Proofs of Geometry Theorems.
Technical Report 86-12, RISC-Linz, Johannes Kepler University, Linz, Austria, 1986.

[Melenk and Neun, 1988] Herbert Melenk and Winfried Neun.
Parallel Polynomial Operations in the Large Buchberger Algorithm.
In *Computer Algebra and Parallelism*, pages 143–158, Workshop at the TIM3 Laboratory, University of Grenoble, France, June 1988. Academic Press, London.

[Melenk *et al.*, 1988] Herbert Melenk, H.M Möller, and Winfried Neun.
On Gröbner Bases Computation on a Supercomputer Using REDUCE.
Technical Report Preprint SC 88-2, Konrad-Zuse-Zentrum für Informationstechnik, Berlin, January 1988.

[Pavelle, 1986] Richard Pavelle.
Applications of Computer Algebra.
Kluwer Academic Publishers, 1986.

[Penner and Klingler, 1989] V. Penner and A. Klingler.
AND-Parallel Prolog on Large Scale Transputer-Systems.
Technical report, Parsytec Gesellschaft für Parallele Systemtechnik, Aachen, West Germany, 1989.

[Pereira and Nasr, 1984] Luis Moniz Pereira and Roger Nasr.

Delta-PROLOG: A Distributed Logic Programming Language.
In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 283–291. ICOT, 1984.

[Ponder, 1988] C. G. Ponder.
Evaluation of "Performance Enhancements" in Algebraic Manipulation Systems.
Technical Report UCB/CSD 88/438, Computer Science Division (EECS), University of California, August 1988.

[Ramakrishnan, 1990] Raghu Ramakrishnan.
Parallelism in Logic Programs.
In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 246–260, San Francisco, California, January 17–19, 1990. ACM Press, New York.

[Shapiro, 1983] Ehud Y. Shapiro.
A Subset of Concurrent Prolog and Its Interpreter.
Technical Report TR003, Institute for New Generation Computer Systems Technology, ICOT Research Center, Tokyo, Japan, 1983.

[Shapiro, 1985] Ehud Shapiro.
Systolic Programming — A Paradigm of Parallel Processing.
Technical Report CS84-16, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, January 1985.

[Shapiro, 1987] Ehud Shapiro, editor.
Concurrent Prolog, Collected Papers, Volumes 1 and 2.
MIT Press, Cambridge, Massachusetts, 1987.

[Takeuchi and Furukawa, 1986] Akikazu Takeuchi and Koichi Furukawa.
Parallel Logic Programming Languages.
Technical Report TR-163, Institute for New Generation Computer Systems Technology, ICOT Research Center, Tokyo, Japan, April 1986.

[Traverso and Donati, 1989] Carlo Traverso and Leombattista Donati.
Experimenting the Gröbner Basis Algorithm with the Aipi System.
In *Proceedings ISSAC 1989*, 1989.

[Ueda, 1985] Kazunori Ueda.
Guarded Horn Clauses.
Technical Report TR-103, ICOT Research Center, Tokyo, Japan, July 1985.

[Vidal, 1989] Jean-Philippe Vidal.
The Computation of Gröbner Bases on a Shared Memory Multiprocessor.
Technical report, School of Computer Science, Carnegie-Mellon University, 1989.

[Vidal, 1990] Jean-Philippe Vidal.
A Distributed Version of Buchberger's Algorithm.

Technical report, School of Computer Science, Carnegie-Mellon University, 1990.

[Watt, 1986] Stephen Michael Watt.
Bounded Parallelism in Computer Algebra.
Research Report CS-86-12, Faculty of Mathematics, University of Waterloo, Ontario, Canada, 1986.