LISP AND SYMBOLIC COMPUTATION: *An International Journal*, 7, 147–172 (1994)

Distributed Data Structures and Algorithms for Gröbner Basis Computation*

SOUMEN CHAKRABARTI soumen@cs.berkeley.edu
KATHERINE YELICK yelick@cs.berkeley.edu
Computer Science Division,
University of California, Berkeley, CA 94720, USA.

Abstract. We present the design and implementation of a parallel algorithm for computing Gröbner bases on distributed memory multiprocessors. The parallel algorithm is irregular both in space and time: the data structures are dynamic pointer-based structures and the computations on the structures have unpredictable duration. The algorithm is presented as a series of refinements on a *transition rule* program, in which computation proceeds by nondeterministic invocations of guarded commands. Two key data structures, a set and a priority queue, are distributed across processors in the parallel algorithm. The data structures are designed for high throughput and latency tolerance, as appropriate for distributed memory machines. The programming style represents a compromise between shared-memory and message-passing models. The distributed nature of the data structures shows through their interface in that the semantics are weaker than with shared atomic objects, but they still provide a shared abstraction that can be used for reasoning about program correctness. In the data structure design there is a classic trade-off between locality and load balance. We argue that this is best solved by designing scheduling structures in tandem with the state data structures, since the decision to replicate or partition state affects the overhead of dynamically moving tasks.

Keywords: Parallel computing, distributed data structures, Gröbner basis, software caching, relaxed consistency, load balancing.

1. Introduction

Symbolic algebra applications are challenging targets for parallelism, because they have irregular data structures, unpredictable computation times, and asynchronous, irregular communication. In this paper we describe the design and implementation of one such application, the Gröbner basis computation, on contemporary message-passing multiprocessors. The implementation relies on two globally shared data structures that are designed to provide high throughput and latency tolerance. High throughput, measured by the number of operations that can be completed in unit time, is often more valuable than low execution time of individual operations, because there is enough parallelism to keep many operations running in parallel [16]. Latency tolerance is a more realistic goal than low latency, since some operations require communication and remote computation; latency tolerance is obtained by making the operations split-phase, so that a single processor may have multiple operations outstanding.

* This work was supported in part by the Advanced Research Projects Agency of the Department of Defense monitored by the Office of Naval Research under contract DABT63-92-C-0026, by AT&T, and by the National Science Foundation through an Infrastructure Grant (number CDA-8722788) and a Research Initiation Award (number CCR-9210260). The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

The Gröbner basis application is used to solve a system of multivariate non-linear polynomial equations, i.e., to find the roots of a set of such polynomials. Given a set of polynomials, we compute another set called their *Gröbner basis*, which has the same roots, but for which the roots may be computed more easily [8]. Computing the Gröbner basis is analogous to doing Gaussian elimination on a linear system, with the Gröbner basis playing the role of the triangularized system. The Gröbner basis computation is entirely symbolic; it computes on a set of polynomials rather than vectors and matrices. Systems of non-linear equations arise in a large number of problem domains, and the applicability of the Gröbner basis computation is limited only by its performance.

To expose the parallelism in the Gröbner basis algorithm, we recast the sequential algorithm as a nondeterministic procedure defined by a set of guarded commands, called *rules*. The execution proceeds by repeatedly firing rules whose guards are enabled. This eliminates unnecessary serialization that is present in a deterministic description of the algorithm. The program is refined until the rules can be executed concurrently with only small amounts of synchronization, and the shared data structures can be replaced by distributed ones.

Not surprisingly, one of the hardest problems in the data structure design is to address the trade-off between *locality* and *load balance*. Load balance demands that computation (and therefore data) be spread out across the machines, while locality requires that each computation is done on the processor that owns its data, typically the processor that created the computation. The two data structures in our design address these issues in a synergistic manner: a set of polynomials is dynamically cached on multiple processors, giving good locality to the tasks that use the polynomials, and a randomized scheduling queue is used to provide good dynamic load balance. Had we chosen to partition the polynomial set, a randomized scheduler would have performed quite poorly, since the chance that a task would be scheduled on a processor that owned the necessary data would be small.

The paper is organized as follows. In Section 2 we describe the basic structure of the Gröbner basis computation and outline an approach for parallelization. This approach leads to the design of two distributed data structures, which are presented in Section 3 using an application-independent point of view. Incorporating the data structures back into the Gröbner basis algorithm changes the algorithm, as described in Section 4, since the data structures provide weaker semantics than assumed in the earlier algorithm. Our design is sufficiently formal that a proof of correctness can be constructed simultaneously. In this paper, we only state the main results needed for correctness, and omit the proofs. A detailed correctness proof can be found in [13]. Details of implementation and performance are given in Section 5. A review of related work is presented in Section 6 and concluding remarks are made in Section 7.

2. Gröbner Basis Computation

In this section we introduce the notion of a Gröbner basis, review Buchberger's sequential algorithm, and formulate it as a set of transition rules. The underlying assumption is that the operations on these data structures are atomic. From this

parallel algorithm, the main data structures are identified and some of their usage characteristics obtained. For example, the data structure implementations will be quite different for structures that are frequently written than for those that are mostly read. This lays the groundwork for the design of distributed data structures in Section 3, which are incorporated into the Gröbner basis algorithm in Section 4.

The first sequential algorithm for finding Gröbner bases was given by Buchberger [8]. Given a set of polynomials, it produces another set of polynomials with the same roots and additional properties that make it easier to compute those roots. The new set, called the Gröbner basis, is analogous to a triangular set of linear equations, which can be solved by substitution. The two basic operations in computing a Gröbner basis are to take two polynomials and combine them into a scaled sum, and to simplify a polynomial by subtracting multiples of other polynomials.

Polynomials are defined by a set of coefficients and a set of variables. In general, the coefficients may be taken from an arbitrary field, and there are applications of Gröbner basis in which the coefficients are not simply numbers, but, for example, are themselves ratios of polynomials. Although our implementation supports only rational coefficients, our algorithm may be used with coefficients from an arbitrary field, so we use this general formulation in stating the problem. However, the reader may safely consider the special case in which coefficients are rational numbers on which exact arithmetic is performed. Our examples will use only rational coefficients.

2.1. Notation

We introduce some notation to define the problem. Let K be a field and $x_1, ..., x_n$ be variables, arbitrarily ordered as $x_1 > x_2 > \cdots > x_n$. Then $K[x_1, ..., x_n]$ defines a ring of polynomials under standard polynomial arithmetic. A polynomial is a sum of monomials scaled by coefficients. Monomials are of the form $x_1^{i_1} \cdots x_n^{i_n}$, and coefficients are elements in K. A total order $\succ$ on monomials is *admissible* if for all monomials a, p, q it satisfies (1) $p \succeq 1$ (note that $1 = x_1^0 \cdots x_n^0$) and (2) $p \succeq q \Rightarrow ap \succeq aq$, where $p \succeq q$ if $p \succ q$ or $p = q$.

Example: A monomial p looks like $x_1^{i_1} \cdots x_n^{i_n}$, where $i_\ell \geq 0$, $1 \leq \ell \leq n$. One commonly used admissible ordering is the *lexicographic* ordering: $p = x_1^{i_1} \cdots x_n^{i_n}$ is greater than $q = x_1^{j_1} \cdots x_n^{j_n}$ iff $\exists \ell, 1 \leq \ell \leq n : i_\ell > j_\ell$, and $\forall \ell' : 1 \leq \ell' < \ell, i_{\ell'} = j_{\ell'}$. Thus, with $x > y > z$, we have $xy^2 > y^{10}z$. □

Assume that an admissible ordering $\succ$ is specified on monomials. By convention, polynomials are written with their monomials in decreasing order of $\succ$, with at most one instance of a given monomial. TERM(p, i) denotes the i-th term of polynomial p. A term contains the *coefficient* and the *monomial*: TERM(p, i) = COEF$(p, i) \times$ MONO(p, i). The *head term* of a polynomial p is the leading term: HTERM(p) = TERM$(p, 1)$. Similarly, HCOEF(p) = COEF$(p, 1)$ and HMONO(p) = MONO$(p, 1)$. HMONO, HCOEF and HTERM are naturally extended to sets of polynomials: HMONO(S) = {HMONO(p) : $p \in S$}, etc. The admissible ordering $\succ$ is

extended to polynomials by defining $p \succ q$ iff $\text{HMONO}(p) \succ \text{HMONO}(q)$, and $p \succeq q$ iff $\text{HMONO}(p) \succeq \text{HMONO}(q)$.

Example: Given variables $x > y > z$ and lexicographic ordering on monomials, polynomial $p = 2x^2yz^3 - 7xy^{10} + z$ is in canonical form with $\text{HTERM}(p) = 2x^2yz^3$, $\text{HMONO}(p) = x^2yz^3$ and $\text{HCOEF}(p) = 2$. □

The *ideal* of a given set S of polynomials is the set of all polynomials that can be expressed as polynomial multiples of the elements in S, i.e.,

$$\text{IDEAL}(S) = \left\{ \sum_{s \in S} ps : p \in K[x_1, \ldots, x_n] \right\}. \tag{1}$$

In Gaussian elimination, we use the property that scaling the equations and adding them does not affect the solution (as long as we use non-singular transformations); the analog is true of Gröbner basis computation. We define two operations, REDUCE and SPOL, each of which scales its two argument polynomials by two terms and computes their scaled sum. As in Gaussian elimination, the motive is to cancel terms and get a simpler set of equations.

The first operation is polynomial reduction. Given polynomials p and r such that $\text{HMONO}(r)$ divides $\text{MONO}(p, i)$ for some i, r is said to *reduce* p to p', given by:

$$p' = \text{REDUCE}(p, \{r\}) = p - \frac{\text{TERM}(p, i)}{\text{HTERM}(r)} \times r. \tag{2}$$

Note that $\text{TERM}(p, i)$ does not exist in p', and therefore $p \succ p'$. Reduction is used to eliminate redundant polynomials in Gröbner basis computation, much like scaling and summing rows zeros out linearly dependent rows in Gaussian elimination.

Example: If $p = 2x^2yz^3 - 7xy^{10} + z$ and $r = 5xyz - 3$ then r reduces p to $p' = p - \boxed{\frac{2}{5}xz^2} \cdot r = -7xy^{10} + \frac{6}{5}xz^2 + z$. □

Reduction by a *set* S of polynomials is done by repeatedly reducing p by some element of S. When no element of S can reduce p, it is *irreducible* or in *normal form*, in which case $\text{NORMAL?}(p, S)$ is true. The collection of all possible normal forms of p when reduced by S is denoted $\text{NF}_S(p)$. The zero polynomial, 0, is in normal form with respect to any S.

The second operation is s-polynomial computation. For this, we will need to define the *highest common factor* of two monomials:

$$\text{HCF}(x_1^{i_1} \cdots x_n^{i_n}, x_1^{j_1} \cdots x_n^{j_n}) = x_1^{\min(i_1, j_1)} \cdots x_n^{\min(i_n, j_n)}. \tag{3}$$

Given polynomials p_1 and p_2, with head terms k_1m_1 and k_2m_2 respectively, their *s-polynomial* is given by

$$\text{SPOL}(p_1, p_2) = p_1 \frac{k_2m_2}{\text{HCF}(m_1, m_2)} - p_2 \frac{k_1m_1}{\text{HCF}(m_1, m_2)}. \tag{4}$$

Example: Given polynomials $p = x - 13y^2 - 12z^3$ and $q = x^2 - xy + 92z$ under lexicographic ordering $x > y > z$, their s-polynomial is given by $\text{SPOL}(p, q) = -13xy^2 - 12xz^3 + xy - 92z$. □

Given a set F of polynomials, a *Gröbner basis* of F is a set G of polynomials satisfying the following:

- $\text{IDEAL}(G) = \text{IDEAL}(F)$ and
- For each $p \in \text{IDEAL}(F)$, $\text{NF}_G(p) = \{0\}$.

An extensive survey of the theory and applications can be found in Mishra [24].

2.2. Sequential Algorithm

Buchberger's sequential algorithm, which we call GB-seq, is shown in figure 1. It proceeds by computing s-polynomials, reducing them, and adding any non-zero polynomials to the basis. The two main data structures are G (the basis) and *gpq* (the priority queue of pairs for SPOL computation). In this version, polynomials entering G are completely reduced with respect to all previous elements in G, but old basis elements are not checked for reducibility by new entrants. The effect is that polynomials that have entered the basis once are never modified or deleted. Correctness of GB-seq was established by Buchberger; we refer to the version in Mishra and Yap [24] (Theorem 5.8).

THEOREM 1 (BUCHBERGER)

- *G is a Gröbner basis iff $\forall f, g \in G, 0 \in \text{NF}_G(\text{SPOL}(f, g))$.*
- *Algorithm GB-seq terminates with G being a Gröbner basis of F.*

Algebraic optimizations to the basic algorithm have been developed that test s-polynom-ials to quickly detect reduction to zero, without actually performing the reduction [9]. Although our implementation includes such improvements, we omit them from the presentation for simplicity.

2.3. Sources of Parallelism

There is parallelism at various levels in the algorithm. At the smallest grain, polynomial arithmetic (including arithmetic operations on the coefficients, which are infinite precision integers in our implementation) can be parallelized. Medium grain parallelism can be exploited by permitting many reducers to reduce a polynomial simultaneously — they work on different monomials. Coarser grain parallelism exists in computing and reducing several s-polynomials independently in parallel. Reduction has many degrees of freedom, since the choice of a reducer is not specified. Also, although REDUCE denotes reduction to normal form, it need not be done all at once; any number of reduction steps will do. Finally, the choice of a pair from *gpq* to compute the SPOL is not specified. Although selection heuristics affect performance, one can work on several pairs simultaneously.

Even for shared memory machines, parallel coefficient or polynomial arithmetic appears to be too fine-grained. Medium grain parallelism has been attempted by Clarke *et al* [15] without any significant benefit on an Encore, a bus-based shared memory machine, probably because the overhead is too high. Our algorithm

```
Input: F, a finite set of polynomials.
Initially:
    G = F
    gpq = { {f,g} : f,g ∈ G}
while gpq ≠ ∅ {
    let {f,g} be any pair in gpq
    gpq = gpq \ {{f,g}}
    h = SPOL(f,g)
    h' = REDUCE(h,G)
    if h' ≠ 0 {
        gpq = gpq ∪ {{f,h'} : f ∈ G}
        G = G ∪ h'
    }
}
```

Figure 1. Sequential Algorithm GB-seq [Buchberger]. G is initialized to the input set F and grows to become a Gröbner basis. Elements in G are never modified. *gpq* is the set of pairs of polynomials. The function REDUCE(h, G) returns some element $h' \in \mathrm{NF}_G(h)$, i.e., it reduces h completely to normal form.

was planned for the CM-5, a multiprocessor where a hardware message carries at most 5 words and takes at least 5–6μs to transfer (about 200 cycles). Thus, fine and medium grain parallelism were ruled out, as was distributing an individual polynomial across multiple processors.

2.4. Transition Rule Formulation

Transition rules are a means for exploiting nondeterminism in a sequential algorithm description. Inspired by guarded command languages [14], [17], and augmented by linearizable data types [35], this style was used to implement a shared-memory Knuth-Bendix procedure [37]. Transition rules help break the computation into independently schedulable chunks, so the scheduling decisions are deferred until late in the design process. The rules are written in the form $C \Rightarrow A$, where C is the *enabling condition* (a guard predicate) and A is the action. An execution proceeds by repeatedly *firing* enabled rules nondeterministically. Termination occurs when none of the rules can be fired. Parallelism results from being able to overlap rule executions in time on multiple processors.

Our approach is to start with a transition rule description of the sequential algorithm, then refine it to use distributed data structures instead of shared ones. Ideally, switching from shared to distributed data structures should not entail any change in the algorithm or proof of correctness. However, a relaxed data structure semantics will allow for a more efficient implementation on distributed memory, but will require a new algorithm and correctness argument.

2.4.1. Transition Rules with a Single Shared Basis

Our first version, algorithm GB-share, is a simple transformation of algorithm GB-seq, the purpose being only to replace the sequential control structure by a tran-

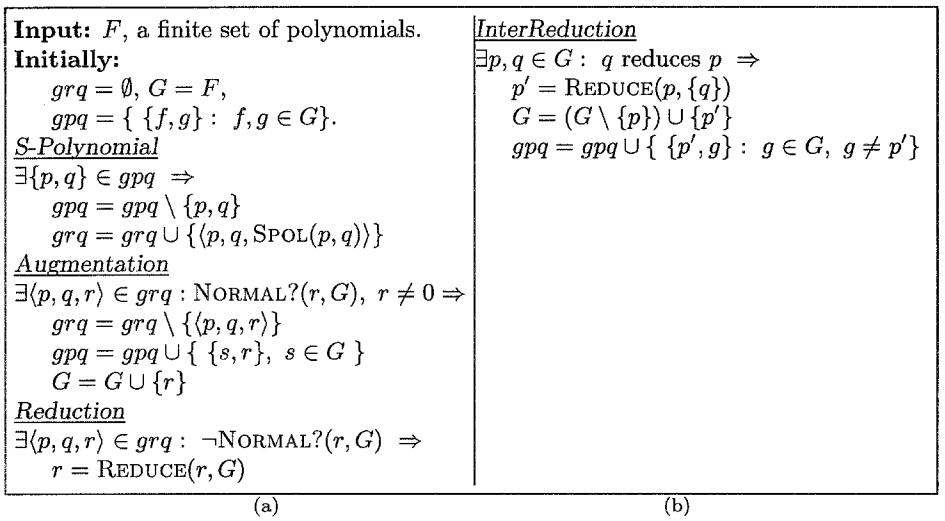


Figure 2. (a) GB-share: Transition rule formulation using a single shared basis. Data structures G and gpq are as before. Unlike in Algorithm GB-seq, REDUCE(r, G) need not return a normal form; a partially reduced form will do. (b) Algorithm GB-share is augmented with the rule *InterReduction* to give an algorithm with interreduction with a shared basis. *InterReduction* might reduce a basis element to zero; we assume for simplicity that zero elements are left around in G but are never considered as reducers.

sition rule oriented formulation. Later, we shall refine and augment this basic skeleton. Algorithm GB-share is shown in figure 2. There are three data structures: G is the growing basis, gpq is the pair set as before and grq is a temporary set of polynomials in some stage of being reduced. The g denotes that they are global, being shared by all processors. The r stands for reducts, and p stands for pairs. The q stands for (priority) queue, and reflects the importance of heuristic ordering for good performance. The grq data structure exposes parallelism in reduction operations; we keep it partitioned among processors, without transfers from one partition to another.

The correctness of GB-share follows from Buchberger's proof of correctness of GB-seq. We omit the proof, but sketch the main properties that lead to the result. Partial correctness depends on the invariance of IDEAL(G) (after each rule invocation, IDEAL(G) = IDEAL(F) holds), and the observation that $\forall p, q \in G,\ \{p, q\} \notin gpq \Rightarrow 0 \in \text{NF}_G(\text{SPOL}(p, q))$. The proof of termination depends on the observations that REDUCE, NORMAL? and SPOL are all terminating for any argument, that IDEAL(HMONO(G)) grows each time G is augmented and never shrinks during any other operation, and that there cannot be an infinite sequence of invocations of *Reduction*. Because IDEAL (HMONO(G)) grows monotonically over time, termination follows from Hilbert's basis theorem ([26], pages 420–425).

2.4.2. *Interreduction with a Single Shared Basis*

When a new polynomial is added to the basis, existing polynomials may become reducible by it. Although this is somewhat controversial [19], Buchberger and others believe that performing these "reverse" reductions, and thus keeping the basis reduced with respect to itself, is essential for good performance. Buchberger describes an elaborate way of keeping track of polynomials that become reducible each time the basis grows, so that after each addition the basis is *interreduced*, i.e., basis polynomials are reduced by each other until nothing more can be reduced [8].

Adding interreduction increases the difficulty of parallelization, because the basis is no longer a "grow-only" data structure. We know of no earlier attempts to parallelize interreduction, probably because Buchberger's formulation makes extensive changes to the basis during interreduction, and s-polynomial computation and reduction operations cannot go on concurrently with interreduction.

We augment our transition rule system by a rule for interreduction. Figure 2 shows the new rule *InterReduction*. While the extent of reduction done in *Reduction* is not specified, we can assume, for the correctness argument, that only a single reduction step occurs in *InterReduction*. It follows that correctness is preserved if *InterReduction* were to reduce multiple steps, which is done in the implementation.

Informally, correctness of the modified program can be proved if the partial correctness and termination properties are preserved by the modified version. This can be shown in two parts. If an invocation of *InterReduction* modifies the basis from G_1 to G_2, it can be shown [13] that any polynomial p which can be reduced to zero using G_1 can also be reduced to zero using G_2 (i.e., $0 \in \mathrm{NF}_{G_1}(p) \Rightarrow 0 \in \mathrm{NF}_{G_2}(p)$), and that $\textsc{Ideal}(\textsc{Hmono}(G_1)) \subseteq \textsc{Ideal}(\textsc{Hmono}(G_2))$. Working from these observations, we can establish the following.

LEMMA 1 *Algorithm GB-share terminates with G being a Gröbner basis of F.*

As in algorithm GB-seq, the key data structures in algorithm GB-share are *gpq* and G. In GB-share, they are still central resources, which can lead to significant bottlenecks in a parallel program. In Section 3, we shall address the design of efficient distributed memory representations of basis G and pair queue *gpq*.

3. Distributed Data Structures

In this section we describe the distributed data structures for locality and parallelism. Informally, our design methodology is to exploit the theoretical elegance and expressibility of shared memory programs, then adapt it for efficient execution on distributed memory multiprocessors using a runtime library of data structures. These distributed data structures are part of a library project called *Multipol* [36].

For example, randomly assigning tasks to processors works well for applications like ours running on a shared memory multiprocessor. Hence our data structure for parallelism incorporates randomized load balancing. To ensure good performance in a distributed memory setting, we design a data structure for enhancing locality that implements shared memory by caching objects and running consistency protocols on collections of objects.

In Section 3.1 we describe the multiset data structure with relaxed consistency. This is built on top of the data structure that supports replicated cached objects, described in Section 3.2. The task queue for dynamic load balancing is described in Section 3.3.

3.1. Set with Relaxed Consistency

Replication, rather than partitioning, is often chosen as the strategy for distributing data structures that are accessed frequently. However, mutation of data introduces complications regarding consistency of copies. Designers of large scale multiprocessors with logically shared, but physically distributed, memory (like DASH and KSR-1) have already recognized the importance of weakening the memory consistency model [1], [18] to escape the overheads of keeping multiple caches coherent. This is more important when caching and consistency management is done in software, since the overheads are even higher.

Hardware solutions to caching are easier to use and provide faster individual operations, but they suffer some drawbacks. First, the coherency units are of fixed size, which can lead to performance problems such as object fragmentation and false sharing. Second, the consistency protocol is rigid and therefore not adaptable to different application needs. In particular, consistency is defined on the semantics of low level read-write operations, which may be stronger than necessary. In a set, for example, the order in which elements are inserted leads to different memory representations, but the same set.

In the parallel Gröbner basis computation, one of the important shared data structures is a set G of polynomials that is frequently read and seldom written; the set contains a relatively small subset of all the polynomials that are computed and examined during the course of the algorithm. We therefore wish to replicate the set across processors. With interreduction, not only is the set modified by insertion, its elements may be modified or deleted. Polynomials are the unit of caching, simply called *objects*, and the set of input polynomials, which evolves into the final answer, is an aggregate of these objects. Because elements are reduced with respect to each other, there will never be any duplicate elements in the set. Technically, since that uniqueness is maintained by the user of the data structure, rather than being built into the semantics of the insertion operation, the data structure is really a multiset. With this disclaimer, we will, for brevity, refer to the basis as a "set" throughout.

Support for high throughput operations on the set is provided by making the operations split-phase, with one operation to initiate a state change or observation and another to check that it has completed. A set is an unordered aggregate of objects that are replicated lazily across processors and validated "on demand."

3.1.1. The Set Interface

A set has type `SetType` and its elements are of type `ElemIdType`. Elements of one set may occur in another set or, as in the case of our task queue, in a completely different type of aggregate structure. To avoid having multiple copies of elements that appear in more than one aggregate, we assume that `ElemIdType` refers to

names (i.e., ID's) of objects rather than values. The ID can be looked up in the object caching layer to be described later.

Our implementations are done in C using lightweight messages called *active messages* [32]. The processors are not kept tightly synchronized, but there are some points in the computation, usually initially, when all processor cooperate to perform a single function, such as creating a distributed data structure. These operations are, by convention, named with an `all_` prefix. Thus a set G is created as follows:

```
SetType G = all_SetCreate();
```

Inserting an element into a set is a split-phase operation initiated by the operation `SetInsertInit`. The inserted element is visible to the calling processor by the time `SetInsertInit` returns, but there is a delay before other processors see the new element. The calling processor can make sure all other processors have been notified of the insert by checking if `SetInsertTest` returns `true`. Similarly, `SetDeleteInit` and `SetDeleteTest` initiate and check for completion of a delete operation. Mutation of elements in the set is internal to the object data structure to be described later. The signatures of the set construction and mutation operations are summarized here.

```
SetType all_SetCreate();
SetStatus SetInsertInit ( SetType, ElemIdType );
Boolean SetInsertTest ( SetType );
SetStatus SetDeleteInit ( SetType, ElemIdType );
Boolean SetDeleteTest ( SetType );
```

An important set operation for the Gröbner basis algorithm is an iterator. Motivated by the memory hierarchy considerations mentioned before, we have two versions of iterators.

```
SetForAllIds ( SetType, ElemIdType ) { loop body };
```

is an iterator that produces all element names in the set, and executes *loop body* successively with each name. Note that not all ID's may have corresponding local data.

```
SetForSomeElems ( SetType, ElemType ) { loop body };
```

executes the *loop body* successively with each element in some subset of the set. Operationally, the subset corresponds to those elements that are locally cached. Note that the type of the iterating variable is `ElemType`, i.e., it is a value, not a name like `ElemIdType`.

`SetForAllIds` is useful when the object names can be used without having to know their values, as in generating pairs of ID's in algorithm GB-dist (see figure 4). `SetForSomeElems` is used in reduction, where only a subset of the elements may be needed to make progress. `SetForSomeElems` can be implemented using `SetForAllIds` and a test for local availability of an object, described in Section 3.2.

At some point in an execution, the set client will need to ensure that the values of all elements of the set are locally available. A validation operation is used to make a given processor's view of the set globally consistent. As with all other set operations that require communication, validation is split-phase:

```
SetValidInit ( SetType );
Boolean SetValidTest ( SetType );
```

Once `SetValidTest` returns true, the iterators are guaranteed to "see" all elements in the set, as long as no concurrent insertions or deletions are being done.

3.1.2. The Set Implementation

A set is implemented as a list of `ElemType`'s in each processor's memory. Not all of the ID's have corresponding valid data in all processors, but the ID lists are kept consistent at all times except between the initialization of either insertion or deletion and the point at which a test for completion of those operations returns true. The `SetInsertInit` and `SetDeleteInit` operations are implemented by broadcasting the new element identifier to other processors. The ID is usually much smaller than the data it represents, so these are very inexpensive. These operations are only used when a lock is held or there is some other guarantee of exclusive access. The `SetValidInit` operation is used to increase the size of the locally visible set — each time it is called on an invalid set, some positive number of unavailable elements will become available within finite time.

3.2. Replicated Objects

Underlying the set implementation is a basic object layer, ROL, that provides a shared memory abstraction that is under application control. An object is a contiguous data block of arbitrary size. The system provides primitives for registering objects into the object space, modifying, reading, and destroying objects, and controlling the consistency of objects on different processors.

An object is identified by a unique identifier. This ID is created when the object is registered into the object space and is used for all subsequent operations on the object. Each time an object is modified, a new version of the object is created. Internally, an object is regarded as a sequence of versions, although the version management is transparent to the user. There is an explicit validation protocol that a node process has to call to upgrade its version of the object. To enable easy overlap between communication and computation, most primitives are split-phase as in the set interface. There is no mechanism for dynamic thread creation: a single thread is assumed per physical processor, and it has control over that processor's view of the object system.

3.2.1. The ROL Interface

Primitives provided by ROL are broadly classified into creation, modification, read access, validation and destruction. An object space is created by calling the function:

```
ObjSpace all_ObjSpaceCreate();
```

All processors call this operation to create and initialize an object space; they must all complete the initialization before other operations may be called. Although we generally imagine all objects in one program to be in a single object space, there may be situations in which the sets of objects are independent, in which case multiple object spaces make sense.

An object is created by allocating memory, filling it up with (the first version of) data, and *registering* it into the object space. The function `ObjCreateInit` initiates the creation of an object `size` bytes large, pointed to by `data`, and returns the new unique ID for the object in `idOut`. A matching `ObjCreateTest` can be used to determine whether creation is complete. There is often unrelated work that can be done during split-phase operations; in the Gröbner basis program, one common type of work is reduction of partially reduced polynomials that are held in *grq* for this purpose.

```
ObjCreateInit ( ObjSpace U,
      int size, ObjType *data, IdType *idOut );
Boolean ObjCreateTest ( ObjSpace U, IdType id );
```

We use `ObjType` to indicate the type of some object that the client has placed in the object space. The object space is not homogeneous, and in C the type is given as `void`, with the responsibility for knowing the actual type left to the user.

An object, identified by `id`, can be modified or deleted using the following split-phase operations:

```
ObjStatus ObjModifyInit ( ObjSpace U,
      IdType id, int newSize, ObjType *newData );
Boolean ObjModifyTest ( ObjSpace U, IdType id );
ObjStatus ObjDestroyInit ( ObjSpace U, IdType id );
Boolean ObjDestroyTest ( ObjSpace U, IdType id );
```

The old data freed by either modification or deletion will eventually be garbage collected by the system. Concurrent modifications or deletions of an object are not allowed; it is the client's responsibility to ensure exclusive access. Both of the initiate operations have a return status to indicate whether the named object is available, known to the object system but not (locally) available, unknown, previously deleted, or in various other exceptional states.

Reading an object is also a two phase process. First, `ObjReadStart` is called to acquire a handle to a locally cached version, if one exists. This also notifies the ROL layer that this version cannot be garbage collected until the read is complete, even if other versions arrive. The application can then read the data buffer, and when finished it releases the handle using `ObjReadEnd`. As with the other operations, `ObjReadStart` may return a status code indicating the object is not available, in which case the `out` return value will not be defined.

```
ObjStatus ObjReadStart ( ObjSpace U, IdType id,
                                    ObjType **out );
ObjStatus ObjReadEnd ( ObjType *in );
```

When an object is read by some processor, any version cached in local memory may be returned. To ensure that the value is the most recent one requires a validation call from the application level. The `ObjValid` functions provide the necessary functionality.

```
ObjStatus ObjValidInit ( ObjSpace U, IdType id );
Boolean ObjValidTest ( ObjSpace U, IdType id );
```

Note that there is no built-in critical region provided between reads and modification: immediately after `ObjValidTest` returns successfully, another processor may destroy this property. If it is necessary for a processor to get the latest value, a lock or other higher level protocol must be used. In the Gröbner basis code, modification of set elements happens only during interreduction, and since processors may use old versions for doing their own reductions, no lock is needed on individual polynomials.

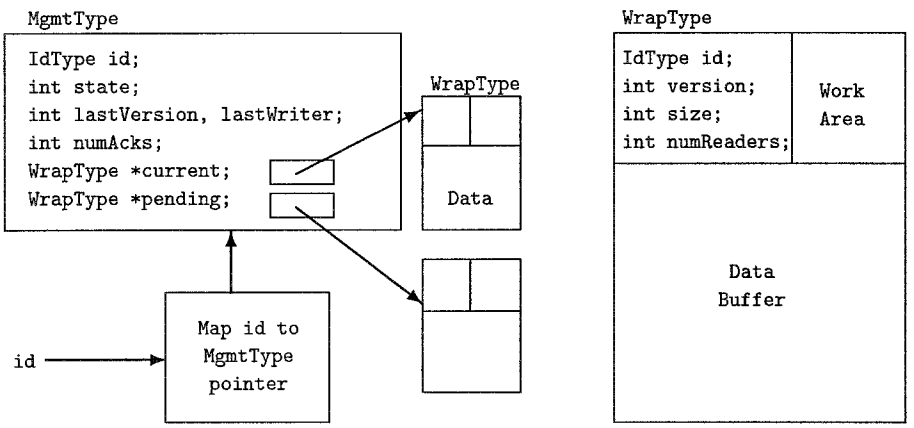


Figure 3. The object management structure, wrappers, and version control.

3.2.2. The ROL *Implementation*

The basic components of the implementation are shown in figure 3. Each processor has a mapping from the object ID's to management structures; the mapping is implemented by a hash table. The management structure for each object contains information about operations in progress on the object (the `state`), the last processor that modified the object, the current version number, and a count of acknowledgements from outstanding remote operations. The object in figure 3 has two versions: one that is currently being read by at least one processor, and another that has been written by some processor and is pending. If the current version were not being read, the pending version would have replaced it. If there were more pending versions, these would be linked together using the workspace portion of the *wrapper*. The wrapper contains version-specific information, such as the number of processor currently reading it and the size and value of the version.

Atomicity of most of the operations is achieved by simple pointer swinging operations. When an object is written, a new version is created and atomically replaces the current wrapper or is linked into the list of pending wrappers. The implementation is complicated by the desire to support split-phase operations. Keeping track of the number of pending operations and their acknowledgements, for example, is only necessary because multiple operations may be outstanding. The level of coding detail necessary to get the protocols to be correct and efficient is exactly what we wish to hide in a library of distributed data structures.

3.3. Task Queue

The remaining data structure problem for the Gröbner basis implementation is the design of a distributed priority queue for holding pairs of polynomials. An important observation from the original algorithm is that strict priority is not required for correctness, but obliviousness to the "quality" of polynomial pairs will lead to unacceptable performance. On a distributed memory machine, this immediately leads to the design of a distributed *task queue*, in which priorities are used to locally order tasks, which are in this case polynomial pairs. This is not a full-scale thread scheduling system, since tasks are really just data that is interpreted by the application program, and once a task starts executing, it is never de-scheduled by the system.

3.3.1. Task Queue Interface

The task queue has the following primitive operations.

```
TqType all_TqCreate ();
TqEnqueue (TqType, TaskType);
TqStatus TqDequeue (TqType, TaskType *);
```

These have the obvious semantics, with the exception of `TqDequeue`. The return status of `TqDequeue` signals one of three possible conditions: a task is available and was assigned to the `TaskType` pointer parameter; no tasks were currently available, although the task queue is not necessarily globally empty; or, the queue is globally empty and all tasks have been completed. In our implementation, the second case occurs when there are no local tasks in the queue. This allows other work that might be available, such as reducing polynomials in *grq* in Gröbner basis, to be done while waiting for another task to arrive.

3.3.2. Task Queue Implementation

The original task queue used in the implementation was engineered to optimize for locality and load balance [12]. Tasks were preferably scheduled on the processor that created them, unless some other processor was starved for work. This led to a fairly complicated implementation in which hints of work load were exchanged between processors, and a task could move multiple times before being executed. For tasks with high transportation costs, for example, when data has been partitioned and moving a task means leaving its data behind or transporting it along with the task, a strategy like this one may be necessary. However, once the decision was made

to replicate the basis G in the Gröbner basis code, moving a task involved moving only a pair of object identifiers, which is a relatively low cost operation.

Given that load balancing can be done aggressively in our design, we chose a randomized load balancing protocol that was not only easier to implement, but also proved more amenable to theoretical analysis. Each processor has a local priority queue of tasks. An idle processor tries to dequeue a task from its local queue. If one exists, it is expanded. Any child task is enqueued into the priority queue of a processor chosen uniformly at random from the P processors. There is no coordinated global communication for load balancing purposes — once a processor obtains a task from its local pool and starts working at it, the task is run to completion. Our algorithm is a generalization of the randomized algorithm described by Zhang and Karp [23]; they assumed that each task took a fixed amount of time, so the system proceeded by alternating one computation step on each processor with one load balancing step. Our algorithm handles arbitrary task times and requires no global synchronization points. A theoretical analysis of our algorithm is given in [11]. Randomly assigning tasks to processors gave performance competitive with more complicated protocols, at a trivial programming effort.

4. Algorithm Design with Distributed Data Structures

In Section 2, we introduced the Gröbner basis problem, and transformed it into a transition rule formulation with interreduction, assuming throughout that the sequential data structures G and *gpq* that appear in algorithm GB-seq can be efficiently shared by P processors. Then, in Section 3, we explored some engineering issues in the design of these data structures for a distributed memory multiprocessor. In both cases, the memory hierarchy shows through in the design. For the task queue, strict priority is relaxed, which only affects performance, but for the set structure, it is necessary to demonstrate correctness of the resulting algorithm. In this section we rewrite the earlier transition rule programs to use the distributed data structures and outline the main idea used to reason about correctness, namely, the definition of an abstraction function on the distributed set.

4.1. A Consistency Problem

If the distributed set is used without modifying the overall algorithm, the inconsistent replicas of elements in the set may lead to incorrect executions. In particular, the following "race condition" may arise, where operations using out-of-date copies of the basis lead to mutual cancellation. This is a generalization of the case Ponder pointed out [27].

Example: Suppose, with ordering $t > w > u > v > x > y > z$, the following polynomials exist in the basis.

$q_1 = wx + y$ and $q_2 = wz - uy$ with $\text{SPOL}(q_1, q_2) = uxy + yz = p_1$, say.

$q_3 = vy + y$ and $q_4 = vz$ with $\text{SPOL}(q_3, q_4) = yz = p_2$, say.

$q_5 = tu + u$ and $q_6 = ty$ with $\text{SPOL}(q_5, q_6) = uy = p_3$, say.

Next, p_3 can reduce p_1 to $p_1' = yz$, which is the same as p_2. Suppose processors P_1 and P_2 both have copies of p_1' and p_2. *InterReduction* fires on P_1 and P_2. Say p_1' is reduced by p_2 to 0 on P_1. Processor P_2 does not modify its copy of g, instead it reduces p_2 by p_1' to 0. Subsequent invalidation messages lead both processors to discard their copies of p_1' and p_2. This can possibly destroy the correctness of the result. □

A solution to this special case is to impose a total order AGE on polynomials such that if $f = g$, f is allowed to reduce g to 0 only if the order is favorable. In general, a stronger check is needed, namely, the total order should be used whenever the *head monomials* of the reducer and the reduced are equal, even if the polynomials are not completely equal. It is easy to verify that this check prevents the particular error indicated, but it is still non-trivial to show correctness in general.

4.2. Distributed Memory Algorithm

Using the AGE function, we now write the transition rules GB-dist in figure 4 that describe a distributed algorithm. Since only one object space is used, we have omitted this argument from the ROL operations. In the implementation, these rules are written to permit multiple rule copies to execute simultaneously on different processors.

The first transition rule, *S-Polynomial*, is essentially unchanged, except that it has extra operations for manipulating the object space and the reduce queue. As described in Section 3, `TqDequeue` may fail even though pairs exist somewhere in the system and `TqEnqueue` is actually placing the new pair on some randomly chosen processor.

Reduction is done in both *Reduction* and *InterReduction* by calling the function REDUCE(r, G), which reduces a polynomial r by a set of polynomials G using the set iterator `SetForSomeElems`. *InterReduction* modifies an existing element of the basis G when it does reduction. Recall from the description of the object layer that multiple modifications cannot be done concurrently; this is enforced in our implementation by assigning each polynomial in the basis to an *owner* processor with exclusive access for interreduction. Locks could also be used, but they seem to require excessive communication in our environment. Finally, validation of the basis is done by a separate rule, *Validation*, which must be executed regularly for the computation to proceed.

4.3. Correctness Arguments

From the example in Section 4.1, it is clear that the correctness of algorithm GB-dist does not directly follow from the correctness of algorithm GB-share in Section 2. A correctness argument for algorithm GB-dist requires a more precise model of the implementation. To model the software cache in the set implementation, the variable G in GB-dist is really a distinct variable G_i on each processor i, $1 \leq i \leq P$. Also, the versions of a polynomial p throughout its lifetime can be recorded in a hypothetical list called its *version list* $[p(0), p(1), \ldots, p(t)]$. The full version list does not exist, but the abstraction captures the update history of p. Let us call the

S-Polynomial

```
TqDequeue(gpq,{f,g}) == TQ_SUCCESS ⇒
   fValid? = ObjReadStart(f,pf);
   gValid? = ObjReadStart(g,pg);
   if ( fValid? && gValid? )
      grq = grq ∪ { SPOL(pf,pg) }
   else
      TqEnqueue(gpq,{f,g});
   ObjReadEnd(pg);
   ObjReadEnd(pf);
```

Reduction

```
∃r ∈ grq : ¬NORMAL?(r,G) ⇒
   r = REDUCE(r,G);
   if r == 0
      grq = grq \ {r};
```

Augmentation

```
SetValidTest(G) && ∃r ∈ grq : NORMAL?(r,G) ⇒
   grq = grq \ {r};
   ObjCreateInit ( size(r), r, newId );
   SetInsertInit(G,newId);
   while ( !ObjCreateTest(newId) || !SetInsertTest(G) )
      do some useful work;
   SetForAllIds ( G, oldId )
      TqEnqueue(gpq,{newId,oldId});
```

InterReduction

```
∃f,h ∈ G:  ObjValidTest(f), h reduces f
           HMONO(f) ≠ HMONO(h) or AGE(f) > AGE(h) ⇒
   pStat = ObjReadStart(f,pf);
   hStat = ObjReadStart(h,ph);
   newf = REDUCE(pf,{ph});
   ObjReadEnd(ph);
   ObjReadEnd(pf);
   if ( newf == 0 ) {
      SetDeleteInit(G,f);
      ObjDeleteInit(f);
   } (Tests for completion not shown.)
   else {
      ObjModifyInit ( f, size(newf), newf );
      SetForAllIds ( G, g )
         TqEnqueue(gpq,{f,g});
   }
```

Validation

```
TRUE ⇒
   SetValidInit(G)
```

Figure 4. GB-dist: The complete transition rule algorithm with interreduction, showing the use of the distributed data structures. Since there is only one object space, it is not shown in the function calls.

set of all such version lists G'. The local copy G_i contains processor i's view of G: it contains at most one version, which is its current version, from each of the lists in G'.

A validation operation either puts the first element of a new version sequence in G_i or replaces version $g(t)$ from a version sequence g by $g(t+\ell), \ell > 0$. We will define an abstraction function that maps the physically distributed set to an abstract set $\mathcal{G}$ containing only the latest versions at a given time.

$$\mathcal{G} = \Big\{ g(\text{last}) : \big[p(0), p(1), \ldots, p(\text{last})\big] \in G' \Big\}. \tag{5}$$

In GB-share, when an invocation of *InterReduction* modifies the value of the basis from G_1 to G_2, one can show that polynomials reducing to zero in G_1 can also reduce to zero in G_2, by replacing each reduction step using an element in $G_1 \setminus G_2$ by two reduction steps, each using an element in G_2. Furthermore, $\text{IDEAL}\big(\text{HMONO}(G_1)\big) \subseteq \text{IDEAL}\big(\text{HMONO}(G_2)\big)$, so progress is not hampered. It turns out that using the same techniques in a more elaborate way, we can establish that these properties are preserved, even with P copies of the basis, provided the AGE ordering is used.

Specifically, with the abstract basis $\mathcal{G}$ defined as in (5), we can show the analogues of the above properties:

$$\forall p: \quad 0 \in \text{NF}_{\mathcal{G}_1}(p) \;\Rightarrow\; 0 \in \text{NF}_{\mathcal{G}_2}(p), \quad \text{and} \tag{6}$$

$$\text{IDEAL}\big(\text{HMONO}(\mathcal{G}_1)\big) \;\subseteq\; \text{IDEAL}\big(\text{HMONO}(\mathcal{G}_2)\big), \tag{7}$$

where $\mathcal{G}_1$ and $\mathcal{G}_2$ are values of the abstract basis before and after an interreduction step. Working from these observations, we can establish the following [13].

THEOREM 2 *GB-dist terminates, computing a Gröbner basis of F.*

4.4. Implementation Sketch

So far, the transition rules have only been modified by creating distributed data structures in place of the original shared ones. The semantics of the GB-dist algorithm are still based on an interleaving of the transition rules. The real parallelism comes from observing that many of the rules can now be overlapped, because the data structure contain sufficient concurrency control. In this section, we give sketches of the less obvious steps taken to transform the transition rules into a parallel program in terms of the abstractions we have defined in Section 3.1, Section 3.2 and Section 3.3.

Each processor runs a scheduling loop in which it checks the guard conditions and executes enabled rules. Under this model, the transition rule formulation in figure 4 has one significant inefficiency. The guard of *Reduction* checks that that some polynomial r in grq in reducible by G, while the guard of *Augmentation* checks that r is *not* reducible by G. Reduction by a set involves a search for a reducer, so it is more profitable to merge these two rules into a single one. The guard on the new rule *Reduce/Augment* checks only that grq is non-empty, with the separation based on reducibility handled by a conditional in the rule body. The combined rule

```
Reduce/Augment
if ( ∃r ∈ grq ) {
  grq = grq \ {r};
  [r, status] = REDUCE(r, G);
  if ( r = 0 )
    return;
  if ( status == REDUCED ) /* continue reduction */
    grq = grq ∪ {r};
  else { /* try augment */
    ACQUIRE LOCK;
    if SetValidTest(G) {
      ObjCreateInit ( size(r), r, newId );
      SetInsertInit ( G, newId );
      while ( !ObjCreateTest(newId) || !SetInsertTest(G) )
        do some useful work;
      SetForAllIds ( G, oldId )
        TqEnqueue(gpq,{newId,oldId})
    }
    else /* set not valid --- try other reductions */
      grq = grq ∪ {r};
    RELEASE LOCK;
  }
}
```

Figure 5. The combined *Reduce/Augment* axiom, using operations provided by the `Obj`, `Tq` and `Set` abstractions. The ID `oldId` is generated by set iteration.

takes the form shown in figure 5. For convenience, the function REDUCE is modified to return a status, `REDUCED` or `NORMAL`, to indicate whether a reduction was indeed performed.

Some additional comments about the above code are in order. In the actual implementation, overlap between rule executions is necessary for parallelism, and correctness is ensured by a lock (see lock acquiring and releasing statements in the code). We use a simple spin lock, but with the following optimization. When a processor tries to acquire a lock and fails, it is clear that some other processor is doing an *Augmentation*, so the `SetValidTest` on the first processor is doomed to fail if it does not validate its set before it successfully acquires the lock. In any case, unavailability of the lock, or detecting an invalid basis inside the critical section, means that there are further potential reductions to do. Hence we place r back into grq and continue with other rules, retrying the lock later.

5. Performance

In this section we present the performance of our implementation. We used a Thinking Machines CM-5 multiprocessor [10]. Each node is a 33 MHz (15–20 MIPS) Sparc processor with 8 MB of memory. The network is a fat-tree supporting at most 20 MB/s point to point data transfer. Communication was done using the

Table 1. Sample running times (in seconds) of the prototype. T_S is the running time of a sequential implementation. A is the number of polynomials added to the basis, and Z is the number of s-polynomials reduced to zero. Note that all numbers depend on basis ordering, pair selection, and pair elimination criteria.

Name	T_S	A	Z	1	2	3	4	5	6	7	10	15	20
arnborg4	.28	6	18	.314	.19	.13	.08	.06					
arnborg5	97	53	411	190	86	63	43	31			13	12	10
katsura4	11	16	60	46	10	8.1	7.7	5.7			4.4	4.2	4.1
lazard	101	31	114	20	6.7	5	3.3	3.1			2.4		
morgenstern	5.5	13	40	3.9	2.2	1.2	1.2				1		
pavelle4	4	9	21	5.47	2	1.2	1.4	.9			.7		
pavelle5		20	104	281	116	79	62	55		31	28		15
robbiano	0.5	13	30	.82	.4	.27	.19						
rose	13.7	16	28	20	6.3	3	2.4		1.9	1.6	1.6		
trinks1	3.3	14	54	10.1	4.1	2.9	1.8	1.9			1		

active message layer `CMAML`. The implementation is in C; we used `gcc-2.3.3` with optimization `-O4` for our measurements.

5.1. Benchmarks

We have used the set of standard benchmarks collected mostly by Vidal [31]. Total degree ordering was used, with ties being resolved by lexicographic order. We used the pair elimination criteria in [9] and the traditional pair selection in [8]. The task queue ordered pairs locally to favor the pair $\{f, g\}$ with the smallest $\text{HMONO}(f) \times \text{HMONO}(g)/\text{HCF}(\text{HMONO}(f), \text{HMONO}(g))$. In table 1 we give performance for some of these examples. Some inputs are too small, for example, `arnborg4` needs only 24 tasks to complete, making it too small to exploit more than 4–5 processors. In general, problems like `arnborg5`, `katsura4`, `pavelle5`, running for tens to hundreds of seconds, having hundreds of tasks, parallelize quite well. There are two different sequential programs shown in table 1: T_S is the original sequential code and T_1 is the parallel algorithm running on a single processor.

Table 2 shows the fraction of running time spent idle (averaged over all P processors), and in s-polynomial computation, reduction, task queue operations, lock operations, pair generation and pruning, and ROL overheads, for a few sample runs with $P = 1, 5, 10, 20$. In general, performance is good if most of the time is spent in s-polynomial computation and reduction. The task queue operations (`Tq`) seem to take significant time, but most of them are unsuccessful attempts to dequeue before a termination detection algorithm is triggered, so this is really a measure of uneven finishing times rather than task queue overhead.

5.2. Computing Speedups

A common measure of parallelism in an application is the speedup of a parallel algorithm relative to the "best" sequential one. However, no tight complexity bound is known for the sequential algorithm, and both the sequential and parallel algorithms are guided by heuristics. Moreover, the work done, hence, the running time, of the parallel program has nondeterminism owing to variations in event ordering. Furthermore, there are cases where the one-processor parallel version outperforms the sequential program and vice versa.

Table 2. Execution time breakup for two sample inputs, for $P = 1, 5, 10, 20$, to the nearest percent of total time.

arnborg5	Spoly	Reduce	Tq	Idle	Lock	Pair	Rol
$P = 1$	4	94	0	0	0	0	0
5	3	89	1	1	1	4	1
10	3	79	5	8	1	3	1
20	4	65	7	13	5	5	1
katsura4	Spoly	Reduce	Tq	Idle	Lock	Pair	Rol
$P = 1$	2	97	0	0	0	1	0
5	2	89	3	4	0	1	1
10	2	79	8	8	1	1	1
20	1	49	24	24	0	1	1

In plotting speedup curves, we use the running time of the parallel version running on one processor as the baseline for computing speedups. The results are shown in figure 6. To provide a more realistic assessment of the overheads in the parallel algorithm, running times of an optimized sequential program running on one node of the CM-5 are also reported in table 1. Another problem is that different runs of the program do not do the same amount of work, making "true" speedup hard to estimate. We use a standardization procedure described next to remove this effect.

5.3. Superlinear Speedup

Parts of the speedup curves in figure 6 are *above* the ideal linear speedup. It is well-known in problems with nondeterministic or heuristic scheduling (e.g., backtrack, branch and bound, pattern matching) that it is possible to solve a problem with P processors in less than $1/P$ of the time needed for one processor, because some of them may find "short cuts" (in this case, good reducer polynomials) to the solution whereas a single processor may be misled by an inaccurate heuristic.

To make sure that the speedup curves indicate the benefit of parallelism and not fortuitous choice of polynomials, we also calibrated speedups after getting rid of the nondeterminism. For this, the parallel version accumulates *traces* of activity at each processor. A sequential program running on only one node of the CM-5 reads in the traces and mimics an appropriately merged sequence of execution steps. The execution time of this program is used as the baseline for normalized curves. One of the outstanding examples of superlinear behavior is shown in figure 7(a); a similar effect has been reported by Vidal [31] on this input. Normalized speedup is shown in figure 7(b). The superlinear nature has been filtered completely and the linear nature of "true" speedup shows clearly. (These tracing experiments were done using an earlier version. Our current implementation does better pair selection and elimination. This improves absolute performance, but makes simulating the trace harder.)

An input instance can be "large" in the sense of running time or memory requirements or both. Although our implementation scales well in *time*, replication of the basis presents a limit to scalability in *space*. We have come across long-running instances that might show highly scalable speedups, but all of them exceed the

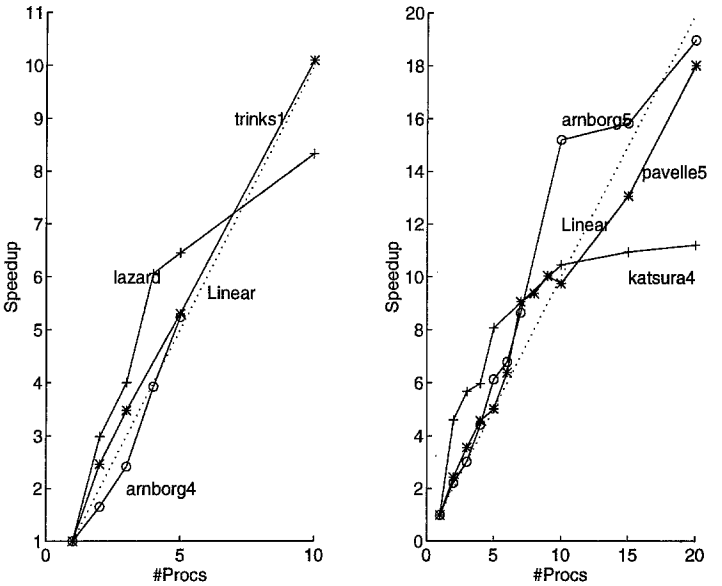


Figure 6. Speedups (based on raw running time) for some standard benchmarks. Our implementation scales better than the best shared memory performance reported by Vidal even for very small examples. (see text for an explanation about the anomaly of superlinear speedup in some cases).

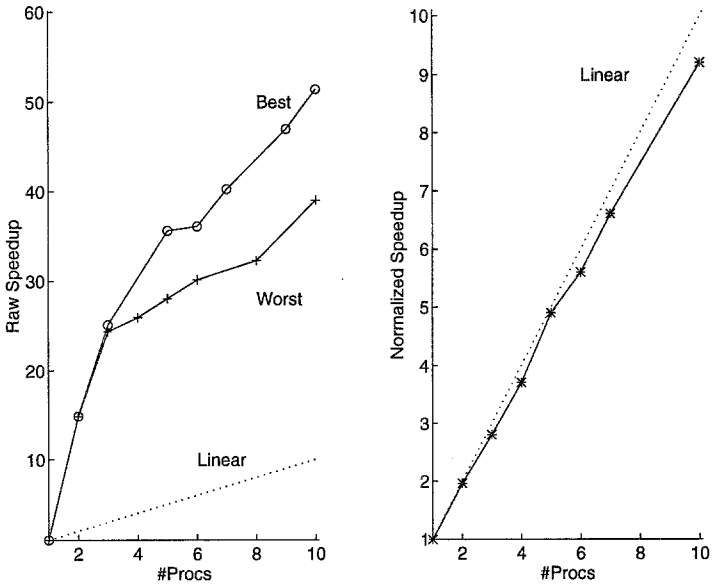
current memory capacity. To run such examples, a more flexible abstraction is needed that performs this space-time tradeoff on a continuum using a hybrid of partitioning and replication.

6. Related Work

There are two directions of research related to the work reported here: research specific to parallelizing the Gröbner basis algorithm, and generic parallelism research in language and runtime support for irregular applications.

Vidal [31] implemented a shared memory Gröbner basis program on an Encore Multimax. He used a shared pool of pairs of polynomials and critical sections for accessing the basis, which was in shared memory. Processors remove work from the pool, produce s-polynomials, reduce them with read permission on the basis and add new polynomials with write permission. In an extension of this work [15], medium grain parallelism is explored in a shared memory setting, by reducing a single polynomial by many reducers, working at different terms. Except for examples where superlinear speedup results from chancing upon "short-cuts" in the search space, efficiency is low. For standard examples the implementation does not scale beyond 5–10 processors. Vidal also gives a survey of earlier attempts to parallelize the algorithm.

Ponder [27] studied this problem in the context of performance enhancements in algebraic manipulation systems. He noted the race condition in parallel interreduction, in which two copies of the same polynomial may reduce each other and



(a) Not normalized (b) Normalized

Figure 7. Superlinear speedup (`lazard`). Figure (a) shows our best and worst performance over 5 runs and the best shared memory performance. All are superlinear. When speedup is determined by normalizing running time using a simulator, near-linear "true" speedup is seen in figure (b).

disappear. As described in Section 4, we avoid this without serialization by keeping time stamps on the polynomials. Because the time stamps need not be globally consistent, they can be efficiently computed in a distributed setting.

On distributed memory machines, a pipelined program has been reported by Siegl [30]. Reduction of a polynomial is done by a pipeline of processes across which the current basis is partitioned. The implementation was ported to a network of SUN workstations, a transputer and a (shared memory) Sequent, but performance figures are available only for the Sequent. These do not appear to be significant improvements over Vidal's performance. Other distributed memory implementations have been reported by Attardi [3] and Hawley [20]. None of these seem to exploit the weak consistency requirements on the data structures in the same manner as our implementation.

The SAM runtime system for supporting a shared memory model on distributed memory machines [28] is closest to our approach of application controlled replication and consistency management. In fact, our Gröbner basis implementation has been ported to SAM. One significant functionality that ROL provides is version control transparent to the application. In SAM, shared objects have single assignment semantics, which is cumbersome since the basis is mutated during interreduction. Furthermore, no integrated load balancing support is provided in SAM.

Recent research has yielded sophisticated runtime support like the Concert system [22] and the Chare kernel [29], [21], and programming languages like Concurrent Aggregates [2], pSather [25] and pC++ [6]. These are general purpose solutions

for modular and portable parallel programming, supporting irregular data sharing and task scheduling at a fine to medium grain. Being language-based approaches, they build in some policies for scheduling and load balancing, although in each case the user has some control over data distribution. A separate thread of work has been the development of application-specific distributed data structures, including irregular grids [5], [4], B-trees [34], sets [16] and oct-trees [33]. These data structures fit with the general framework of our approach, although we make the locality and load balance trade-off a first order concern by giving the application direct control over both scheduling and data layout.

7. Conclusion

We have described an efficient parallelization of Buchberger's Gröbner basis algorithm for distributed memory multiprocessors. The performance results on the CM-5 are encouraging, scaling to 20 processors, which is better than previous implementations on both shared memory and distributed memory machines. On one processor, our implementation is competitive with good sequential implementations, taking advantage of the known optimizations, such as pair elimination, and heuristics for choosing polynomial pairs and reductions.

In this paper we have dealt mostly with the engineering aspects of parallel programming. However, our design methodology allows us to show that the resulting parallel algorithm is correct, and that its efficiency is predictably high. We have cited these results where appropriate.

In addition to these application-specific contributions, we offer a general approach for developing irregular parallel applications. The development is done by refining a nondeterministic sequential algorithm, and the resulting program is organized around distributed data structures with a relaxed semantics. Our data structures use both partitioning and replication: partitioning for the frequently written task queue, and replication for the infrequently written set and underlying objects. Relaxing the consistency enabled efficient implementations of the shared data structures, which would otherwise have been bottlenecks in a distributed environment. Reasoning about parallel algorithms using a series of refinements is a well-known technique that is demonstrated in the Unity framework [14], among others. However, in those formal models, a distributed environment is targeted by reducing the original program for shared data down to message passing. Our work gives an example of the development of a relatively large parallel application by focusing on the interesting parts of parallelization, retaining data abstraction in a distributed setting.

Acknowledgments

We are grateful to Steve Schwab for providing the packages for *bignum* and polynomial arithmetic and a shared memory Gröbner basis program developed at CMU. Professor Richard Fateman made valuable comments on the work, and gave generous help on some geometry proof benchmarks. An initial version of the task queue

was coded by Chih-Po Wen. We also acknowledge the careful comments of the reviewers that enabled us to improve the presentation.

References

1. S. V. Adve and M. D. Hill. Weak ordering–a new definition. In *17th International Symposium on Computer Architecture*, April 1990.
2. Andrew A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively Parallel Programs.* MIT Press, Cambridge, MA, 1993.
3. G. Attardi and C. Traverso. A Network Implementation of Buchberger Algorithm. Technical Report 1177, University di Pisa, January 1991.
4. S. B. Baden. Programming abstractions for dynamically partitioning and coordinating localized scientific calculations running on multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 12(1):145–157, 1991.
5. H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory multiprocessors. *Concurrency: Practice and Experience*, pages 159–178, June 1991.
6. F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Maloney, and B. Mohr. Implementing a parallel C++ runtime system for scalable parallel system. In *Supercomputing'93*, pages 588–597, Portland, Oregon, November 1993.
7. M. P. Bonacina. *Distributed Automated Deduction.* PhD thesis, Department of Computer Science, SUNY at Stony Brook, December 1992.
8. B. Buchberger. Gröbner basis: an algorithmic method in polynomial ideal theory. In N. K. Bose, editor, *Multidimensional Systems Theory*, chapter 6, pages 184–232. D. Reidel Publishing Company, 1985.
9. B. Buchberger. A Criterion for detecting Unnecessary Reductions in the construction of Gröbner Bases. In *Proceedings of the EUROSAM '79, An International Symposium on Symbolic and Algebraic Manipulation*, pages 3–21, Marseille, France, June 1979.
10. N. J. Burnett. The Architecture of the CM-5. In *IEEE Colloquium on 'Medium Grain Distributed Computing' (Digest 070)*, pages 1–2, London, 26 March 1992.
11. S. Chakrabarti, A. Ranade, and K. Yelick. Randomized load balancing for tree structured computation. In *IEEE Scalable High Performance Computing Conference*, pages 666–673, Knoxville, Tennessee, May 1994.
12. S. Chakrabarti and K. Yelick. Implementing an irregular application on a distributed memory multiprocessor. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 169–178, San Diego, California, May 1993.
13. S. Chakrabarti and K. Yelick. On the correctness of a distributed memory Gröbner basis algorithm. In C. Kirchner, editor, *International Conference on Rewriting Techniques and Applications*, volume 690 of *Lecture Notes in Computer Science*, pages 77–91, Montreal, Canada, 16–18 June 1993. Springer-Verlag.
14. K. M. Chandy and J. Misra. *Parallel Program Design: a Foundation.* Addison-Wesley Publishing Company, Reading, Mass., 1988.
15. E. M. Clarke, D. E. Long, S. Michaylov, S. A. Schwab, J. P. Vidal, and S. Kimura. Parallel symbolic computation algorithms. Technical Report CMU-CS-90-182, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, October 1990.
16. W. J. Dally. *A VLSI Architecture for Concurrent Data Structures.* PhD thesis, California Institute of Technology, Pasadena, California, March 1986.
17. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.
18. K. Gharachorloo, D. Lenoski, J. Laudon, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *17th International Symposium on Computer Architecture*, pages 15–26, 1990.
19. A. Giovini, T. Mora, G. Niesi, L. Robbiano, and C. Traverso. "One sugar cube, please" OR Selection strategies in the Buchberger algorithm. In *Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation*, pages 49–54, Bonn, Germany, 15–17 July 1992.

20. D. J. Hawley. A Buchberger algorithm for Distributed Memory Multi-processors. In *Proceedings of the 1st International ACPC Conference on Parallel Computation*, pages 385–390, Salzburg, Austria, 30 September – 2 October 1991. Springer-Verlag.
21. L. V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System based on C++. Technical Report UIUCDCS-R-93-1796, University of Illinois, Urbana, IL, March 1993. Also in OOPSLA'93.
22. V. Karamcheti and A. Chien. Concert — Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware. In *Supercomputing'93*, Portland, Oregon, November 1993.
23. R. M. Karp and Y. Zhang. A Randomized Parallel Branch-and-bound Procedure. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 290–300, 1988.
24. B. Mishra and C. Yap. Notes on Gröbner basis. In *Information Sciences 48*, pages 219–252. Elsevier Science Publishing Company, 1989.
25. S. Murer, J. Feldman, and C.-C. Lim. pSather: Layered extensions to an object-oriented language for efficient parallel computations. Technical Report 93-028, International Computer Science Institute, Berkeley, CA, 1993.
26. Nathan Jacobson. *Basic Algebra — Volume 2.* W. H. Freeman and Company, New York, 1989.
27. C. G. Ponder. Evaluation of "performance enhancements" in algebraic manipulation systems. Technical Report UCB/CSD 88/438, University of California, Berkeley, 1988. Chapter 7, Parallel Algorithms for Gröbner Basis Reduction.
28. D. J. Scales and M. S. Lam. A flexible shared memory system for distributed memory machines. Unpublished manuscript, 1993.
29. W. Shu and L. V. Kalé. Chare Kernel — a Runtime Support System for Parallel Computations. *Journal of Parallel and Distributed Computing*, 11:198–211, 1991.
30. K. Siegl. Parallel Gröbner Basis Computation in ||MAPLE||. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–186, San Diego, California, May 1993.
31. J.-P. Vidal. The computation of Gröbner bases on a shared memory multiprocessor. Technical Report CMU-CS-90-163, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1990.
32. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, 1992.
33. M. Warren and J. Salmon. A parallel hashed oct-tree n-body algorithm. In *Supercomputing'93*, pages 12–21, Portland, Oregon, November 1993.
34. W. E. Weihl and P. Wang. Multi-version memory: Software cache management for concurrent B-trees. In *Proceedings of the Symposium on Parallel and Distributed Processing*, December 1990.
35. K. Yelick. Using abstraction in explicitly parallel programs. Technical Report MIT/LCS/TR-507, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139, July 1991.
36. K. Yelick, S. Chakrabarti, E. Deprit, J. Jones, and A. Krishnamurthy. Data Structures for Irregular Applications. In *Proceedings of the DIMACS Workshop on Parallel Algorithms for Unstructured and Dynamic Problems*, 1993.
37. K. A. Yelick and S. J. Garland. A parallel completion procedure for term rewriting systems. In *Conference on Automated Deduction*, Saratoga Springs, NY, 1992.