Implementing an Irregular Application on a Distributed Memory Multiprocessor*

Soumen Chakrabarti   Katherine Yelick
Computer Science Division
University of California at Berkeley
Email: soumen@cs.berkeley.edu, yelick@cs.berkeley.edu

Abstract

Parallelism with irregular patterns of data, communication and computation is hard to manage efficiently. In this paper we present a case study of the Gröbner basis problem, a symbolic algebra application. We developed an efficient parallel implementation using the following techniques. First, a sequential algorithm was rewritten in a *transition axiom* style, in which computation proceeds by non-deterministic invocations of guarded statements at multiple processors. Next, the algebraic properties of the problem were studied to modify the algorithm to ensure correctness in spite of locally inconsistent views of the shared data structures. This was used to design data structures with very little overhead for maintaining consistency. Finally, an application-specific scheduler was designed and tuned to get good performance. Our distributed memory implementation achieves impressive speedups.

1 Introduction

In this paper we present a case study of an irregular symbolic algebra application, the Gröbner basis problem.

Computing the Gröbner basis of a set of multivariate polynomials has applications in solving systems of non-linear equations, implicitizing parametric equations and automating geometry proofs. Efforts have been made to develop parallel algorithms for this problem on shared memory machines, but efficiency and scalability have not been encouraging. Here we describe a distributed memory implementation of the Buchberger [2] algorithm that equals and often surpasses shared memory performance and scales to a larger number of processors.

This symbolic algebra application is different in many aspects from the most frequently parallelized numerical scientific computations.

- It involves irregular data structures rather than arrays.
- No well-marked computation and communication phases can be identified.
- The amount of computation is unpredictable. The problem resembles search in that running time may vary widely depending on heuristic choice.
- The amount and pattern of communication is unpredictable for conceivable parallelizations.

Our preliminary profiling experiments on a sequential implementation (from CMU [7]) suggested that the problem has significant parallelism: a large fraction of running time is spent in relatively independent polynomial arithmetic with hardly any data dependencies preventing their parallelization.

The following list summarizes our contributions.

- A distributed memory algorithm has been developed for computing Gröbner bases.
- An efficient program has been implemented. It runs on the CM-5 distributed memory multiprocessor (described in more detail in §7).
- Properties of the problem have enabled the use of distributed data structures with relaxed, software controlled consistency mechanisms to minimize synchronization and communication overhead.
- Processor utilization has been improved by application level *thread* scheduling. Computations are effectively suspended by storing and retrieving state information in an application level data structure.
- From a sequential algorithm, the parallel program has been derived in a sequence of successively refined *transition axioms*, starting from the most abstract description and modifying it to a distributed memory program.

The paper is organized as follows. We introduce the

*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 a National Science Foundation Infrastructure Grant (number CDA-8722788). The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

problem in §2. A non-deterministic specification of the algorithm using *transition axioms* is developed and refined in §3. Design of the necessary distributed data structures is described in §4. Code organization and scheduling to improve processor utilization are addressed in §5. A summary of implementation details is in §6. Performance for some standard benchmarks is presented in §7. Related work is reviewed in §8. Conclusions are drawn in §9.

2 Problem Statement

Roughly speaking, Buchberger's algorithm is a symbolic form of Gaussian elimination. 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 eliminate one of their terms, and to simplify a polynomial by subtracting multiples of other polynomials.

Polynomials are defined by a set of coefficients, such as the rational numbers, and a set of variables. The reader may safely consider the special case, used in the examples, in which coefficients are rationals on which exact arithmetic is performed. A more general formulation is presented next; it occurs when the coefficients form an arbitrary field and is important in some applications.

Let K be a field and $x_1, ..., x_n$ be variables, then $K[x_1, ..., x_n]$ denotes a ring of polynomials under standard polynomial arithmetic. To define a canonical form for the polynomials, an (arbitrary) ordering is chosen on the variables and extended to an ordering on monomials. Either a lexicographic or total degree ordering is typically used on monomials. A polynomial is canonicalized as a sequence of terms, each containing a coefficient and a monomial, written in decreasing order of the monomials. We can therefore speak of the leading or *head* term, monomial or coefficient of a polynomial.

Example. Given variables $x > y > z$ and lexicographic ordering on monomials, polynomial $p = 2x^2yz^3 - 7xy^{10} + z$ is in canonical form with $\textsc{Hterm}(p) = 2x^2yz^3$, $\textsc{Hmono}(p) = x^2yz^3$ and $\textsc{Hcoef}(p) = 2$.

Polynomial r can *reduce* polynomial p if $\textsc{Hmono}(r)$ divides $\textsc{Hmono}(p)$. The act of reducing p by r involves subtracting a multiple of r from p so that the head term of p is cancelled.

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

A polynomial p is reduced by a set S of polynomials by looking for some $s \in S$ that reduces p. If none is found, p is in *normal form* with respect to S, which is denoted by $\textsc{Normal}(p, S)$. Otherwise p is reduced by s and the process is repeated. For any general set S a polynomial p may end up in many normal forms; the collection of all possible normal forms is denoted $\mathrm{NF}_S(p)$. The zero polynomial, 0, is in normal form with respect to any S. Given monomials $m_1 = x_1^{i_1} \cdots x_n^{i_n}$ and $m_2 = x_1^{j_1} \cdots x_n^{j_n}$, their *highest common factor* is defined as

$$\textsc{Hcf}(m_1, m_2) = x_1^{\min(i_1, j_1)} \cdots x_n^{\min(i_n, j_n)}.$$

Let p_1 and p_2 be polynomials with $\textsc{Hterm}(p_1) = k_1m_1$ and $\textsc{Hterm}(p_2) = k_2m_2$, where $k_1, k_2 \in K$ and m_1 and m_2 are monomials. The *s-polynomial* of p_1 and p_2 is defined as

$$\textsc{Spol}(p_1, p_2) = p_1 \frac{k_2m_2}{\textsc{Hcf}(m_1, m_2)} - p_2 \frac{k_1m_1}{\textsc{Hcf}(m_1, m_2)}.$$

The *ideal* generated by a set S of polynomials is denoted by $\textsc{Ideal}(S)$. Given a set P of polynomials, a Gröbner basis of P is a set G of polynomials satisfying the following:

- $\textsc{Ideal}(G) = \textsc{Ideal}(P)$ and
- For each $p \in \textsc{Ideal}(P)$, $\mathrm{NF}_G(p) = \{0\}$.

The original sequential algorithm for computing a Gröbner basis was given by Buchberger [2]. A survey of the theory can be found in Mishra [5].

```
Input: F, a finite set of polynomials.
Initially:
    G = F
    gpq = { {f,g} : f,g ∈ G, f ≠ 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 S [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 $\textsc{Reduce}(h, G)$ returns some element $h' \in \mathrm{NF}_G(h)$, i.e., it reduces h completely to normal form.

3 Basic Algorithm

In this section we develop the parallel algorithm from the sequential one by Buchberger. The data structures are described first and then the transition axioms are introduced.

3.1 Abstractions

The sequential algorithm has two important data structures (figure 1). The set G is initialized to input set

F and, in course of time, grows to a Gröbner basis of F. Reduction prevents two identical polynomials from entering G, so G is a set. However, the data type implementing G does not itself prevent duplicates, so it is specified as a (monotonically growing) multiset.

Pairs of polynomials for s-polynomial computation are kept in gpq (global pair queue). The reason for specializing gpq from a multiset to a queue is that *selection* of a pair in the **while** loop is sensitive. Correctness is ensured regardless of the selection, but heuristics exist to dramatically improve performance. Thus, priority ordering is necessary in gpq, so that heuristic merit can be encoded into priority.

The central problem in our design is understanding the algebraic properties of the problem and exploiting them to design suitable data structures and communication protocols. We *encapsulate* the data structures so that the semantics and consistency are exposed through a specified interface to the higher level. We can therefore reason about correctness (safety and liveness properties) using these specifications [4], without having to use the specifics of implementation. The details of the data structure implementations are in §4.

Input: F, a finite set of polynomials.
Initially:
$grq = \emptyset$, $G = F$,
$\quad gpq = \{\ \{f, g\} :\ f, g \in G,\ f \neq g\ \}$.
S-POLYNOMIAL
$\exists \{p, q\} \in gpq \ \Rightarrow$
$\quad gpq = gpq \setminus \{\{p, q\}\}$
$\quad grq = grq \cup \{\textsc{Spol}(p, q)\}$
AUGMENT BASIS
$\exists r \in grq : \textsc{Normal}(r, G),\ r \neq 0 \ \Rightarrow$
$\quad grq = grq \setminus \{r\}$
$\quad gpq = gpq \cup \{\{s, r\} :\ s \in G\}$
$\quad G = G \cup \{r\}$
REDUCE
$\exists r \in grq :\ \neg\textsc{Normal}(r, G) \ \Rightarrow$
$\quad r = \textsc{Reduce}(r, G)$

Figure 2: G-1: Transition Axiom formulation with one copy of G. Data structures G and gpq as before. Unlike in Algorithm S, $\textsc{Reduce}(r, G)$ need not return a normal form; a partially reduced form will do.

3.2 Transition Axiom Specifications

Transition axioms [9, 11] are a means to exploit nondeterminism in an algorithm description. They help decompose the computation into independently schedulable chunks, so that scheduling decisions are deferred as much as possible. This means that significant performance tuning can be done late in the design process without major design changes.

Given the basic sequential algorithm by Buchberger, we transform it in a series of refinements to a transition axiom style parallel algorithm for a distributed memory machine. As we proceed from one axiom system to the next, the effort is to convert a formal algorithm to an efficient implementation for a distributed memory machine, in our case, the CM-5 multiprocessor.

Input: F, a finite set of polynomials.
Initially:
$\quad grq = \emptyset$,
$\quad gpq = \{\ \{f, g\} :\ f, g \in F, f \neq g\ \}$.
$\quad \forall i :\ 1 \leq i \leq P, G_i = F, G'_i = \emptyset$
Processor i, $1 \leq i \leq P$.
VALIDATE
$(gpq \neq \emptyset$ **or** $grq \neq \emptyset)$ **and** $\exists g \in G'_i \ \Rightarrow$
$\quad G'_i \ = \ G'_i \setminus \{g\}$
$\quad G_i \ = \ G_i \cup \{g\}$
S-POLYNOMIAL
$\exists \{p_i, q_i\} \in gpq :\ p_i, q_i \in G_i \ \Rightarrow$
$\quad gpq = gpq \setminus \{\ \{p_i, q_i\}\ \}$
$\quad grq = grq \cup \{\textsc{Spol}(p_i, q_i)\}$
AUGMENT BASIS AND INVALIDATE
$G'_i = \emptyset,\ \exists r_i \in grq : r_i \neq 0,$ **and** $\textsc{Normal}(r_i, G_i) \ \Rightarrow$
$\quad grq = grq \setminus \{r_i\}$
$\quad gpq = gpq \cup \{\ \{s, r_i\},\ \forall s \in G_i\ \}$
$\quad G_i = G_i \cup \{r_i\}$
$\quad \forall j : 1 \leq j \leq P, i \neq j$
$\quad\quad G'_j = G'_j \cup \{r_i\}$
REDUCE
$\exists r_i \in grq :\ \neg\textsc{Normal}(r_i, G_i) \ \Rightarrow$
$\quad r_i = \textsc{Reduce}(r_i, G_i)$

Figure 3: G-P: Transition axioms using P copies of G, one for each processor. gpq, grq as before. Each processor i has a local copy G_i of the basis.

Replicating the Basis

Figure 2 is a transition axiom specification (named G-1) of the sequential algorithm in figure 1. While G-1 uses essentially the same data structures as the sequential algorithm, the next version G-P (figure 3) has P copies of the basis, one with each processor. We justify replicating the basis in §4. Consistency is managed explicitly at the application level to exploit algebraic properties.

Formally, we regard invalidations as updating a *shadow set* at each processor. A processor can access elements in the shadow set only after moving them explicitly from the shadow set to its local replica. To model our consistency mechanism, suppose each processor i has a (possibly incomplete) copy of G called G_i. To incorporate invalidations and validations, we use the shadow set G'_i. Say processor i adds a new element g. This involves adding g to G_i and invalidating all other copies by adding g to $G'_j, j \neq i$. If processor k has $G'_k = \emptyset$ then G_k is a valid copy, otherwise it can validate its copy by moving the elements in G'_k to G_k. Processor k can inspect element g only after moving it to G_k. All the transition axioms are written in terms of these abstractions.

Of course, for this technique to be practically useful, the algorithm must have the following properties.

- The representation of elements in G'_i is significantly smaller than in G_i. For polynomials, which are typically several hundreds to thousands of bytes large, the elements in G'_i are only eight byte unique identifiers.
- The properties of the algorithm permit a processor to do a significant amount of work with an incomplete basis. In particular, no reduction "goes to waste". This allows us to update local copies lazily upon demand, which reduces communication expense and obviates stringent synchronization of the critical section style.

Input: F, a finite set of polynomials.
Initially:
$\forall i : 1 \leq i \leq P, lpq_i = lrq_i = \emptyset$
$gpq = \{\ \{f, g\} :\ f, g \in F, f \neq g\ \}.$
$\forall i :\ 1 \leq i \leq P, G_i = F,\ G'_i = \emptyset$
Processor i, $1 \leq i \leq P$.

<u>TRANSFER</u>
$\exists\{p_i, q_i\} \in gpq \Rightarrow$
$gpq = gpq \setminus \{\{p_i, q_i\}\}$
$lpq_i = lpq_i \cup \{\{p_i, q_i\}\}$

<u>VALIDATE</u>
$(lpq_i \neq \emptyset$ **or** $lrq_i \neq \emptyset)$ **and** $\exists g \in G'_i \ \Rightarrow$
$G'_i \ = \ G'_i \setminus \{g\}$
$G_i \ = \ G_i \cup \{g\}$

<u>S-POLYNOMIAL</u>
$\exists\{p_i, q_i\} \in lpq_i :\ p_i, q_i \in G_i \ \Rightarrow$
$lpq_i = lpq_i \setminus \{\ \{p_i, q_i\}\ \}$
$lrq_i = lrq_i \cup \{\text{SPOL}(p_i, q_i)\}$

<u>AUGMENT BASIS AND INVALIDATE</u>
$G'_i = \emptyset,\ \exists r_i \in lrq_i : r_i \neq 0,$ **and** $\text{NORMAL}(r_i, G_i) \ \Rightarrow$
$lrq_i = lrq_i \setminus \{r_i\}$
$gpq = gpq \cup \{\ \{s, r_i\},\ \forall s \in G_i\ \}$
$G_i = G_i \cup \{r_i\}$
$\forall j : 1 \leq j \leq P, i \neq j$
$G'_j = G'_j \cup \{r_i\}$

<u>REDUCE</u>
$\exists r_i \in lrq_i :\ \neg\text{NORMAL}(r_i, G_i) \ \Rightarrow$
$r_i = \text{REDUCE}(r_i, G_i)$

Figure 4: GL-P: Transition Axioms with P copies of G and local pair and reduct queues lpq and lrq to buffer work.

Local Threads

To hide the latency of remote operations across the network, we implement application level "threading" by packaging up state into application level data structures: local queues holding pairs of polynomial pointers and partially reduced polynomials. The version incorporating these is GL-P (figure 4).

<u>REDUCE/AUGMENT</u>
$\exists r \in lrq_i,\ r \neq 0 \ \Rightarrow$
if $\neg\text{NORMAL}(r, G_i)$
$r = \text{REDUCE}(r, G_i)$
else {
if $G'_i = \emptyset$ {
$lrq_i = lrq_i \setminus \{r\}$
$gpq = gpq \cup \{\{s, r\},\ \forall s \in G_i\}$
$G_i = G_i \cup \{r_i\}$
$\forall j :\ 1 \leq j \leq P,\ i \neq j$
$G'_j = G'_j \cup \{r\}$
}
}

Figure 5: Axioms <u>REDUCE</u> and <u>AUGMENT BASIS</u> are combined into <u>REDUCE/AUGMENT</u> to eliminate redundant computation of the expensive guard $\text{NORMAL}(r, G_i)$.

Stuttering Axioms

Certain transitions have complicated guards that are expensive to evaluate. For instance, $\text{NORMAL}(f, S)$ requires checking divisibility of $\text{HMONO}(f)$ by $\text{HMONO}(s)$ for all $s \in S$. We collapse the axioms <u>REDUCE</u> and <u>AUGMENT BASIS</u> into a single axiom in figure 5 to cut down redundant computation of NORMAL.

However, in an execution of the resulting set of transition axioms, it is possible for the new axiom <u>REDUCE/AUGMENT</u> to fire repeatedly without making any progress in the state of computation. It is called a *stuttering* axiom. To ensure termination, the scheduler must prevent infinite sequences of invocations of stuttering axioms.

These axioms can be interpreted as specifying an interleaving of axioms without any real concurrency. Few axioms actually interfere by sharing data. Using little serialization we can enable the axioms to fire concurrently in the implementation. In [4] we have shown that the transition axiom specifications correctly compute a Gröbner basis; we omit the proofs.

4 Data Structure Design

In this section we describe the design and implementation of the two aggregate data structures used in the algorithm.

4.1 The Basis

A major decision was the representation of the basis, the choices being essentially to replicate or partition. A parallel algorithm employing a ring of reducers with the basis partitioned among them was proposed by Buchberger [7]. A slight variant using a reducing pipeline has been implemented by Siegl [6]. We believe that partitioning has less available parallelism and more communication overhead. It is useful only if available memory is too small to replicate the basis.

4.1.1 To Replicate or Partition

The following analysis shows that replication of basis polynomials, if permitted by memory capacity, can be expected to give better performance. We demonstrate the following problems in using a partitioned scheme: poor load balance, insufficient parallelism and high communication overhead.

Input	Pipeline			Max Single
	Max Stage Time (μs)	Maximum Parallelism	Efficiency (Percent)	Reduction Step (μs)
arnborg4	51208	3.55	35	1771
arnborg5	6226476	15.0	26	19983
katsura4	6092409	6.03	30	11410
lazard	12355072	8.02	24	95935
morgen-	10259753	2.88	15	15495
pavelle4	3680064	4.99	36	14324
robbiano	533431	2.65	9	3143
rose	29213252	3.31	16	174201
trinks1	2343233	6.03	35	14242
trinks2	545874	5.58	27	17621

Table 1: The potential parallelism using a replicated basis is inherently larger than that using a partitioned basis.

Poorly Balanced Pipeline

Partitioning means that reduction will have to be done in a pipeline as in [6]. The basic requirement for a pipelined computation to be efficient is that there should be a large number of stages, each taking about the same time (i.e., no bottlenecks exist). To measure the limitations of pipelining, assume the machine has an unlimited number of processors, instantaneous communication and no dependencies between reduction events. Place one reducer at each pipeline stage (processor). The maximum achievable parallelism is limited by the ratio of the total time for reduction to the maximum time spent by one pipeline stage. Table 1 shows that this upper bound on parallelism is rather low for most standard benchmarks. Typical efficiency is only 20–30%.

Granularity

A possible way to increase efficiency is to exploit statistical averaging, by putting several basis polynomials (reducers) in each stage of the pipeline. The parallelism obtained from a pipelined approach is already limited by the the number of reducers; in this case it will decrease further. In contrast, the work can be decomposed into much finer grain units by replicating the basis. The time for a single reduction step (which is the minimum possible grain size permitted by our design) is also shown in table 1. It is about two orders of magnitude less than a pipeline stage time.

Communication Overhead

Let us make a rough estimate of the number of polynomials that get reduced to zero and the number added to the basis. Consider algorithm S (figure 1). Suppose a problem starts with ℓ polynomials and $\binom{\ell}{2}$ pairs, finishing with $m \geq \ell$ polynomials. When we add the i-th polynomial to G, we make $i-1$ new pairs. Thus $\ell, \ell+1, \ldots, m-1$ pairs are added at those times when the basis is grown. How many pairs are generated in all?

$$\binom{\ell}{2} + \sum_{\ell \leq i \leq m-1} i = \binom{\ell}{2} + (m-\ell)\frac{m+\ell-1}{2}.$$

If $m \gg \ell$, as often is the case for large problems, the number of polynomials added is $\Theta(m)$ while the number reduced to zero is $\Theta(m^2)$.

It has been pointed out that with a good implementation of pruning criteria a large number of pairs are eliminated quickly and only about m instead of m^2 pairs have to be actually reduced. However,

- A formal proof of this fact exists in [3], but it only works for restricted inputs with *two* symbols ($n = 2$). No such result is known for general inputs.
- Our sequential prototype with Buchberger's criteria shows the performance presented in table 2 which shows a much larger (at least 5 times) number of reductions to zero.
- Eliminating pairs using criteria, while not as expensive as reduction, still lead to computations involving potentially all basis polynomials ([2],page 197). Thus the number of pairs generated is still a reasonable measure of communication expense.

What is the impact of this analysis? In a partitioned scheme, partially reduced polynomials go around the pipeline, several times in general, perhaps to get reduced to zero, thus incurring communication overhead. On the contrary, suppose the basis is replicated. Then communication involves only those polynomials which have gone through (local) reduction and actually entered the basis. Communication is thus undertaken only *once* for *added* polynomials and not polynomials reduced to zero. Table 2 clearly indicates reduction to zero as the common case to be optimized.

Our assumption of large memory resources might not be justified under all circumstances. As we note in §7, there exist long-running problem instances we could not run owing to limited memory capacity. In such cases, a hybrid strategy using both replication and partitioning may be useful.

Input	Added	Zeroed	Ratio
arnborg5	53	511	9.64
morgenstern	14	117	8.36
pavelle4	10	57	5.70
robbiano	26	158	6.07
trinks1	11	83	7.55

Table 2: Typical number of polynomials added and reduced to zeroes in a sequential implementation.

4.1.2 Interface

The basis of polynomials, G, is implemented as a multiset abstraction with operations `AddToSet`, `Validate`, predicate `Valid?` and iterator `ForAll` (used for reduction). The specifications we give are based on the abstractions in the transition axioms (figure 3): local replicas G_i and shadow sets G'_i on processor i.

`AddToSet`(G, g):
$G_i = G_i \cup \{g\}$
$\forall j : 1 \leq j \leq P, j \neq i : G'_j = G'_j \cup \{g\}$
`Validate`(G):
$G'_i = G'_i \setminus \{g\}$
$G_i = G_i \cup \{g\}$ for some $g \in G'_i$.
`Valid?`(G):
Return 1 if $G'_i = \emptyset$, 0 otherwise.
`ForAll`(g, G) [`Stmt`]:
Execute `Stmt` iteratively for each $g \in G_i$
(G_i might be incomplete.)

Note that the abstraction in itself does not guarantee any consistency (for example, for a read-only semantics a program need never validate its copy). The application must use the operations so as to implement the nature of consistency it needs. Thus the abstraction provides a software controlled *weak consistency* mechanism (see [1], for example).

4.2 The Pair Queue

The algorithm needs a priority queue for pairs of polynomials. The priority corresponds to the heuristic merit of a pair. Since this is only a heuristic, it need not be adhered to exactly. In the parallel algorithm, *gpq* is as much a scheduling control as a data structure.

The *distributed task queue* by Wen *et al*, described in [10], is designed for applications that can be broken down into *tasks*. The suggested paradigm is that the task queue starts with some number of tasks, processors dequeue tasks and perform them, possibly generating more tasks. Our parallel algorithm naturally fits into this model.

4.2.1 Implementation

The queue is partitioned, so there is only one copy of each element and each processor has a local portion of the queue. Processors are logically organized in a ring. `Enqueue` adds the given element to the local queue. `Dequeue` attempts to dequeue work off the local queue; if it is empty remote queues are tested. Protocols execute in the background to sense load imbalance and redistribute tasks more evenly. Priority is ensured only within each local queue and not across processors. Termination detection, a non-trivial problem in distributed queues, is provided to detect global completion of tasks.

4.2.2 Interface

The task queue creation operation `Create` expects as argument a function `Idle?` which is a function in the application that returns true if and only if the calling processor has no work (at the moment). This is used for termination detection, and is necessary because tasks may be in transit on the communication network or buffered in local variables of busy processors. `Enqueue` adds the given element to the queue. `Dequeue` can either remove and return some element or signal *empty* or *terminated*. *Empty* is only a hint: it says that the local queue is empty. *Terminated* is a stable property, true only if the total number of enqueued tasks equals the total number of dequeued tasks, and all processors are idle. Formal specifications are omitted.

5 Scheduling

We now describe how computation is organized. The basic principles are as follows.

- The work is decomposed into basic units called *tasks* that can be dynamically created, scheduled and executed.
- To overlap local and remote computation, suitable state is maintained at each processor to get the benefit of threading.
- Split phase data transfer is used to pipeline communication.

Task Queue

A task in our implementation is a pair of polynomial ID's. Solving a task involves finding both polynomials, computing their s-polynomial and reducing the s-polynomial by the basis. If the s-polynomial does not reduce to zero it is added to the basis, generating new tasks in the form of pairs between old basis polynomials and the new one. Tasks are used to implement parallelism and to balance load. The distributed pair queue contains only unique polynomial ID's, not polynomials so polynomials are not transferred when tasks move about.

Local Threads

Efficient implementation of the transition axioms should avoid having processors wait for a response from a remote computation, using some form of threading. Since threading is not as yet available on the CM-5, we implemented a special purpose application level threading.

There are two places where this might be necessary. First, when an s-polynomial must be computed, one or both of the polynomials of the pair might be missing. Instead of fetching the polynomial(s) immediately, we put the pair "on hold" (by representing it as a suspended thread) and look for other pairs whose polynomials can be found locally. Second, since `Invalidate`s cannot be overlapped, we arbitrate using a mutex lock. If a processor fails to get the lock, it suspends the current thread and proceeds with other activity, such as other s-polynomial computations or reductions.

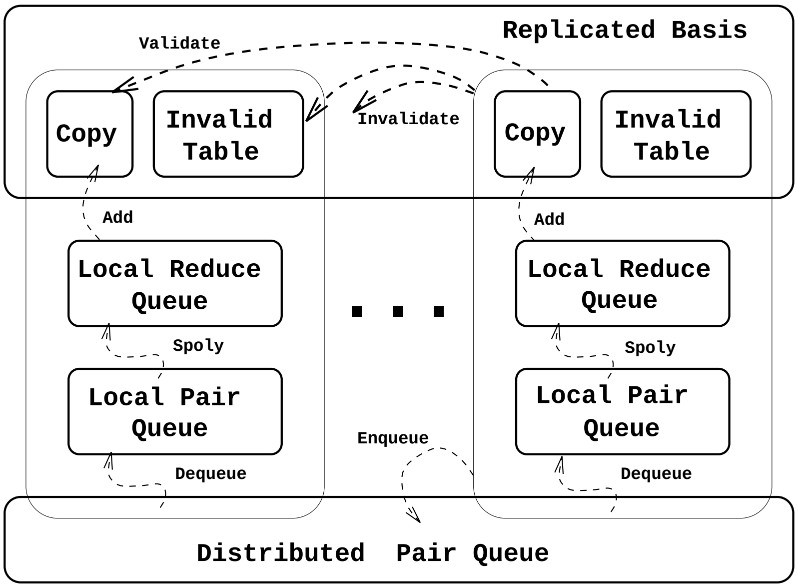


Figure 6: Sketch of data structures and transition axioms in our implementation.

Split Phase Operations

Another optimization in programming a machine like the CM-5 is to pipeline communication. In *split phase* data transfer, a processor can do useful work between initiating data transfer and responding to the other party. For example, in reading an integer from each of the other processors it is better to initiate all reads together and then wait until all the data arrives.

We use this principle in invalidations and validations. During invalidations, the broadcasting processor sends out invalidates all at once and then waits for acknowledgements (acknowledgements are necessary for correctness). During validations, the processor sends out requests for all necessary polynomials that are missing and then waits for all data transfer to complete.

The CM-5 currently does not support significant overlap of communication and computation since the nodes do not have DMA's. It can, however, support overlap of computation with delays for remote acknowledgements. Pipelined communication would be more valuable if true overlap were possible.

6 Sketch of Implementation

Figure 6 sketches the general organization of our implementation. Each processor has local pair and reduce queues and access to the distributed pair queue and basis. Data transfer paths induced by the transition axioms are shown as thin broken arrows.

Communication for *gpq* occurs mostly in a ring pattern. Invalidations follow a star pattern. Validations involve bulk transfer. To balance communication load over the network, for each polynomial added to the basis, a tree is embedded into the network with the processor adding it at the root. Any processor in search of a polynomial knows the tree structure implicitly from the unique ID of the polynomial. It traverses up the tree along its ancestors until it finds the polynomial.

One processor, the *coordinator* (not shown), is reserved for running a termination detection protocol and managing a lock to arbitrate invalidations. We used simple, centralized methods to keep the prototype simple. Being centralized, these protocols will not scale to thousands of processors. However, a large variety of relatively decentralized protocols are available. We have never observed this centralization to be a bottleneck: less than 2% of running time is spent in mutual exclusion and termination detection.

7 Performance

In this section we present the performance of our implementation.

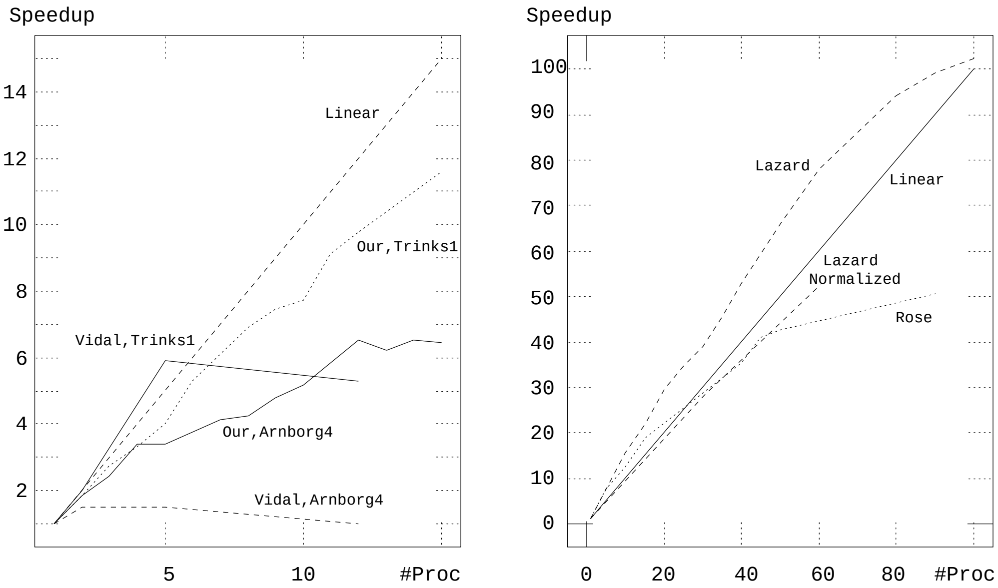
The Environment

We used a CM-5 multiprocessor. 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 data transfer. For our purpose we ignored the topology. Communication was done using the *active message* layer [8]. The implementation is in C; we used `gcc-2.2.2` with optimization `"-O2"` for our measurements.

Measurement and Fluctuation

The sequential algorithm S is under-specified, even though a sequential program could be designed to be deterministic. In the parallel algorithms we exploit this loose specification, resulting in highly non-deterministic behavior.

- Lock serializations and message ordering by the communication network are unpredictable.
- Since the heuristic priority on pairs is weakly enforced, there is more uncertainty in pair selection.
- Because polynomials may be added to copies of the



(a) (b)

Figure 7: 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 (a). For large inputs (Lazard = 5 copies of `lazard`; Rose = 2 copies of `rose`) in (b) we get near-linear speedups up to 100 processors (see text for an explanation about the anomaly of superlinear speedup in (b)).

basis in various orders, the choice of reducers can differ across processors and over different invocations of the program on the same input.

- Pairs in the task queue can migrate in complicated ways and get dequeued by an arbitrary processor at an unpredictable time.

Apart from these nondeterminism issues, factors like code optimization and message protocol tuning are important. Although we have been able to demonstrate the benefits of parallelism without too much noise in the data, slight perturbations of such nature *can* make significant differences to performance.

Speedup Baseline

Given the irregular nature of this problem, it is difficult to define a fair baseline for computing speedups. If we wish to define speedup traditionally as the ratio of running times of the parallel to a sequential algorithm, the sequential algorithm must be the best known — but what is *best* is not known. There is a wide spectrum of opinion about the efficacy of various pair selection and elimination heuristics, whether interreduction helps or not, whether to reduce only head terms or all terms, etc. As might be expected, no single heuristic does well all the time.

Thus we had two choices for the speedup baseline: the parallel program running on one processor or a sequential implementation. The latter choice exposes the overhead of parallelization, but has the disadvantage that the "speedup" for the parallel algorithm running on one processor is generally different from 1. This is counter-intuitive. We decided to scale all speedups to pass through the point (1,1), and also provide the ratio of absolute running times of the two choices to demonstrate that there is true parallelism (table 3). There are cases where the one processor parallel version outperforms the sequential program and vice versa.

It is clear from the discussion that an exact characterization of speedup using speedup curves has relatively little meaning when the basic sequential algorithm is heuristic guided and quite sensitive to minor perturbations. The point we wish to make is that the problem, while not as amenable to scalable parallelization as, say, matrix multiplication, *does* have usable parallelism which even a distributed memory multiprocessor can exploit.

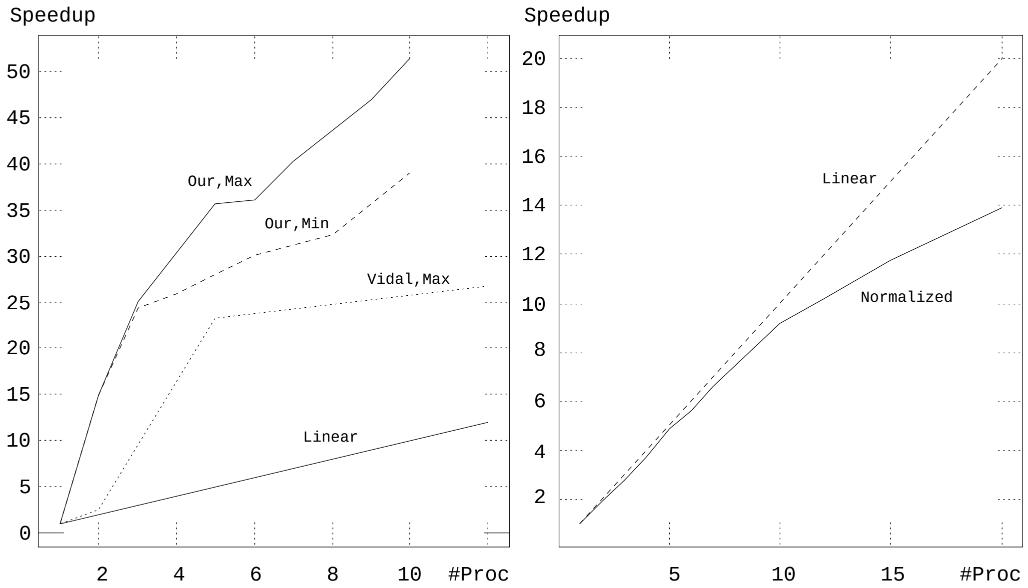
Input	Time (s) $P=1$	Time (s) $P=10$	Time (s) Best Seq.
arnborg4	0.27	0.06	0.28
arnborg5	486	15	46
katsura4	29.2	3.1	11
lazard	55.5	2.6	1009
morgenstern	5.5	1	6.17
pavelle4	5.17	0.7	4.5
rose	12.5	1	15
trinks1	6.44	1	4

Table 3: Sample times for benchmarks for a sequential algorithm and our parallel implementation.

Benchmarks

We have used the set of standard benchmarks collected by Vidal [7]. On one processor they run for about half a second to a few minutes, with total degree ordering[1]. The smaller inputs are solved too quickly to

[1] Ties are resolved by lexicographic order.



(a) Not normalized (b) Normalized

Figure 8: 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).

make accurate speedup measurements.

In figure 7(a) we give performance on two small examples, **`arnborg4`** and **`trinks1`**. The best speedups over 5 runs are shown for our implementation. In comparison, the best curve from Vidal [7] show less scalable speedup. These speedups were computed as the ratio between P processor running time (excluding input, output and initialization) and 1 processor running time, both for the parallel algorithm. We used the pair elimination criteria in [3] and the traditional pair selection in [2].

Inspection of execution profiles indicated that for small problems, parallelism was limited by underutilization of processors during startup and termination transients. The total number of tasks is too small to saturate all processors with work for a large fraction of the running time. To see if this is an inherent limitation of the design, we needed synthetic long-running workloads. For this we started the program with multiple copies of a benchmark with variables named apart. Note that this construction is only for evaluating the *potential* performance and not necessarily representative of realistic large examples. These examples run for hours on one processor. Performance (in terms of running time) for two of these are shown in figure 7(b) (best over 3 runs). Impressive scalability is observed.

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 current memory capacity. We are designing a more flexible abstraction that performs this space-time tradeoff on a continuum using a hybrid of partitioning and replication. This extension should enable us to run large interesting examples.

Superlinear Speedup

Parts of the speedup curves in figure 7(b) are *above* the ideal linear speedup, a phenomenon arising from our definition of speedup. It is a well-known phenomenon in problems with nondeterministic or heuristic scheduling 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" to the solution whereas a single processor may be misled by an inaccurate heuristic. Backtracking search, branch and bound, and some pattern matching problems belong to this category. To get around this difficulty the notion of speedup is often modified. In the 8-queen problem, for instance, the time to find *all* solutions as against just one has been considered.

For the Gröbner basis problem, superlinear speedup occurs when certain "magic" polynomials get added to the basis that reduce many other polynomials quickly to zero. Consider figure 8(a). We get superlinear speedup according to the prior definition of speedup, with a similar effect reported by Vidal for the shared memory implementation [7].

This indicates that the heuristic is not sufficiently discerning for these inputs, so that exploring a few of the best pairs (as against *the* best) in parallel pays off[2].

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 non-determinism. For this, the parallel version accumulates *traces* of activity at each processor. A sequential program running on only one node of the CM-5

[2] The heuristic favors the pair $\langle f, g\rangle$ with the smallest $\textsc{Hmono}(f) \times \textsc{Hmono}(g)/\textsc{Hcf}(\textsc{Hmono}(f), \textsc{Hmono}(g))$.

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.

Normalized speedup is shown in figure 8(b). The superlinear nature has been filtered completely and the linear nature of "true" speedup shows clearly.

8 Related Work

A review of previous attempts to parallelize Gröbner basis can be found in Vidal [7]. Vidal's implementation [7] is on a shared memory machine, the basis being still regarded as a reader-writer shared object with the appropriate locks. As shown in the performance graphs in §7, except for examples where superlinear speedup results from chancing upon "short-cuts" in the search space, efficiency and scalability is low.

Attempts to parallelize this problem on a distributed memory machine have been made by Siegl [6]. Reduction of a polynomial is done by a pipeline of processes across which the current basis is partitioned. As discussed earlier, communication costs are an order of magnitude higher. 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.

9 Conclusion

In conclusion, we outline our results. We have successfully parallelized an irregular application using interesting techniques: introducing non-determinism using transition axioms, minimizing synchronization and communication overhead by designing distributed data structures with weak consistency and improving efficiency using application level state and schedule management. The speedup and scalability of our distributed memory implementation equals and often surpasses performance on shared memory machines.

Acknowledgements

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. Chih-Po Wen contributed the task queue.

References

[1] Sarita V. Adve and Mark D. Hill. Weak ordering–a new definition. In *17th International Symposium on Computer Architecture*, April 1990.

[2] Bruno 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.

[3] Bruno 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.

[4] Soumen Chakrabarti. A distributed memory Gröbner basis algorithm. Master's thesis, University of California, Berkeley, December 1992.

[5] Bud Mishra and Chee Yap. Notes on Gröbner basis. In *Information Sciences 48*, pages 219–252. Elsevier Science Publishing Company, 1989.

[6] Kurt Siegl. Parallel Gröbner basis computation in ||MAPLE||. Technical Report 92-11, Research Institute for Symbolic Computation, Linz, Austria, 1992.

[7] Jean-Philippe 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.

[8] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik 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.

[9] Katherine 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.

[10] Katherine Yelick. Programming models for irregular applications. In *Proceedings of the Workshop on Languages, Compilers and Run-Time Environments for Distributed Memory Multiprocessors*, October 1992. To appear in SIGPLAN notices.

[11] Katherine A. Yelick and Steven J. Garland. A parallel completion procedure for term rewriting systems. In *Conference on Automated Deduction*, Saratoga Springs, NY, 1992.