Evaluation of "Performance Enhancements"
in Algebraic Manipulation Systems

By

Carl Glen Ponder

B.S. (University of California, Irvine) 1981
M.S. (University of California) 1984

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

GRADUATE DIVISION

OF THE

UNIVERSITY OF CALIFORNIA, BERKELEY

Approved: Richard J. Fateman 7/29/88
Chair Date
F. Fateman 7/28/88
John Canny 8/4/88

Evaluation of "Performance Enhancements" in Algebraic Manipulation Systems

Carl G. Ponder

Computer Science Division
University of California
Berkeley, CA. 94720

Abstract

This thesis examines several proposed ways to speed up symbolic algebraic computation: hashing techniques, parallel processing, application of the *FFT*, and alternative representations of polynomial expressions.

Polynomials, and variants like rational functions, truncated power series, and Poisson series, represent an important class of expressions in algebraic manipulation. Parallel algorithms are analyzed for multiplying and powering sparse and dense polynomials, including parallel forms of the *FFT*. Alternative polynomial representations are compared and it is suggested that an efficient algebraic manipulation system might use a family of polynomial representations rather than one general form.

Gröbner-basis reduction is an inherently hard problem, yet can be used as a powerful tool in computational mathematics. Contrary to recent claims, empirical studies show that it is difficult to exploit parallelism in Gröbner-basis computation.

A variety of hashing mechanisms have been proposed for performing operations in symbolic computation, including *memo functions* for recognizing and eliminating redundant computations. The proposed mechanisms are presented in summary; empirical study shows that memo functions, except in certain circumstances, may not be an advantage.

Additional summary of relevant results in parallel algorithm design and parallel processing machines and languages is presented.

Evaluation of "Performance Enhancements"

in Algebraic Manipulation Systems

Carl Glen Ponder

Contents

Chapter 1

Motivation and Preview

Algorithms in symbolic algebra systems range from polynomial-time (matrix inversion) to superexponential-space (Gröbner-basis reduction). A number of programs attempt to solve problems that are in general undecidable (equivalence of expressions). In practice, solution to useful problems can easily take hours or days to produce. The irony is that some of these problems can be expressed quite simply. In problems like factoring of polynomials or Gröbner-basis reduction, it is not unusual for small expressions to produce huge intermediate expressions even when the final result is small. The practical expense of operations is quite apparent to interactive users of systems like Macsyma. Even as processors become faster and memory less expensive, investigations continue to pose more time-consuming problems. Additional speedups due to parallel execution as well as better algorithms or representations provide real benefits.

In this thesis we look at some possible methods for speeding up algebraic computations. These involve parallel algorithms, alternative representations, and various hashing mechanisms. A combination of formal and empirical analyses are used in evaluating these ideas. Some summaries of relevant work in various fields are presented both to identify the state of the art (which seems rather primitive, from a practical standpoint) and to suggest areas where future work should progress.

Chapters 2 and 3 survey current results in parallel algorithm design and the implementation of parallel languages and systems relevant to algebraic manipulation. In summary, few *efficient* parallel algorithms have been formulated or analyzed for significant problems, which is hardly reproachable since the serial complexity of many significant problems is poorly understood. Gröbner-basis reduction [18] is one such poorly understood problem,

which happens to be important since it can be used to solve a wide variety of problems in algebraic geometry. Several "obvious" parallel algorithms for Gröbner-basis reduction are tested in chapter 7, none appearing to be particularly useful.

Numerous parallel languages and architectures have been proposed up to now; some rather poor experiences with the "state of the art" (Alliant Qlisp and Intel Hypercube Concurrent Common Lisp) are reported in chapters 3 and 5. The dream of constructing an efficient parallel system for algebraic manipulation is still out of reach; in the meantime it is difficult to realistically test any parallel algorithms, much less produce any useful software.

Chapters 4, 5, and 6 deal with the general topic of polynomial manipulation. Chapter 4 discusses the issues involved with specific polynomial representations, and suggests an efficient combination of representations. Chapters 5 and 6 analyze parallel algorithms for multiplying and powering sparse and dense polynomials, using such algorithms as the *FFT*. Formal analysis suggests a linear speedup for a fixed number of processors and large inputs. Algorithms for multiplying sparse polynomials are evaluated empirically with positive results, ignoring the poor quality of the Alliant Qlisp testbed. An efficient parallel system for algebraic manipulation may well include parallel routines for multiplying and powering polynomials, to be used automatically or by user choice given sufficiently large polynomials.

Chapters 8 and 9 deal with hashing mechanisms for more efficient symbolic manipulation. Chapter 8 surveys proposed techniques, including the possibility of efficient pattern-matching for nonrandom data. Chapter 9 analyzes the "memo function" mechanism of the Maple system [23], which tabulates the input and output of a given set of functions to eliminate recomputation. Empirical analysis inside the Macsyma system does not favor the use of memo functions, although some of the Maple design decisions may make memo functions useful. Unfortunately the performance of Maple has not been analyzed sufficiently to resolve this question.

This thesis is rather loosely organized, treating each topic in a distinct chapter. The chapters have been submitted to various publications (under the same titles) as disjoint papers. Chapters 2 and 3 have been submitted to *SIGSAM Bulletin;* chapter 8 has appeared in the November 1987 issue (21,4) of *SIGSAM Bulletin.* Chapter 7 has been submitted to *Transactions on Mathematical Software,* and chapters 5, 6, and 9 have been submitted to the *Journal of Symbolic Computation.* An abbreviated version of the thesis will appear in the Proceedings of the *First International Workshop on Computer Algebra and Parallelism.* We believe the text of this thesis will be relatively consistent with whatever versions that

might appear.

1.1 Acknowledgments

Prof. Richard J. Fateman supervised this work, and is co-author of Chapter 9. He guided revisions of each chapter, which, among other things, eased my phobias about writing. Prof. John Canny made a number of comments, substantially improving the thesis in a number of places. John Canny, Eric Bach, and Erich Kaltofen provided some of the references in Chapter 2. Hervé J. Touati pointed out some necessary facts for the analysis of the *NOMC* algorithm in Chapter 5. Manuel Bronstein provided the test cases used in Chapter 7. Manuel Bronstein, Eric Kaltofen, and John Canny lent some insight into the behavior of Buchberger's algorithm. Michael B. Monagan contributed the Maple timings, facts about the Maple organization, and detailed comments on Chapter 9.

This work was supported in part by the Army Research Office, grant DAAG29-85-K-0070, through the Center for Pure and Applied Mathematics, University of California at Berkeley, and the Defense Advanced Research Projects Agency (DoD) ARPA order #4871, monitored by Space & Naval Warfare Systems Command under contract N00039-84-C-0089, through the Electronics Research Laboratory, University of California at Berkeley.

Chapter 2

Parallelism and Algorithms for Algebraic Manipulation: Current Work

We outline recent results relating to the use of parallelism for **solving** problems in algebraic manipulation. Practical issues are discussed.

2.1 Introduction

Numeric computation has had a strong impact on science and engineering in the form of simulation, optimization, etc. This has allowed us to solve problems and analyze systems that could not be understood before. Symbolic computations take us a leap further, to produce analytic rather than numeric results. These can be used predictively to study systems with symbolic parameters, or gain insight into the relationship between variables.

Well-known algebraic manipulation algorithms address problems of all degrees of "hardness:" linear-time (differentiating a polynomial), superexponential-time (sparse symbolic determinants), etc. Furthermore, some problems are in general undecidable (equivalence of algebraic expressions). Advances in fast parallel hardware create possibilities for improving performance on all types of problems if we can devise appropriate algorithms or heuristics.

In this paper we describe theoretical results dealing with the intersection of par-

allelism and algebraic manipulation. We must note, however, that practical use of parallel algorithms in an algebraic-manipulation system is complicated by several factors:

- Many analyses ignore constant factors which can dominate performance for realistic cases. The overheads involved with multiprocessors can sometimes erase algorithmic gains.
- Inadequate characterization of inputs and their relationship to running time complicate analysis of symbolic problems such as Gröbner-basis reduction. Superior algorithms may not readily parallelize.
- Special-purpose hardware may solve specific but unlikely problems very efficiently. For example, *FFT* circuits can be used for multiplying dense polynomials quickly, but they require that coefficients be bounded in magnitude. Furthermore, the *FFT* is inefficient for sparse polynomials, which are more frequent in practice.

Choosing the best algorithm for practical use may require examination of empirical data. New theoretical results should be considered as *potentially* providing more efficient solutions: the gap between theory and reality seems somewhat larger for symbolic computations than algorithms working over data of fixed size and complexity (floating-point arrays, sorting of fixed-length records, algorithms on trees or strings).

2.2 Parallel Algorithms for Algebraic Manipulation

Most of the algorithms we discuss are over the domain $\mathcal{D}$ of multivariate polynomials with integer coefficients or coefficients in a finite-field. Operations on matrices over $\mathcal{D}$ are also important for solving problems such as systems of linear equations. Typical problems over $\mathcal{D}$ include arithmetic (multiplication is most interesting), factoring, and sorting, which are necessary for manipulating expressions and maintaining canonical forms.

Occasionally an algorithm designed for a floating point problem can be useful in solving a symbolic problem, or the "structure" of a numeric algorithm can be adapted to the symbolic domain. Occasionally a parallel solution to the numeric problem can be carried over to the symbolic domain as well. The analogy fails to carry over in instances where the structure of the problem is distorted by the differences in unit cost between multiplying (say) two floating point numbers and multiplying (say) two polynomials of unspecified size.

The results we describe fall into four different categories:

1. Efficient algorithms, achieving optimal (or as good as is known) processor-time products. For example, one parallel mergesort algorithm operates in time $O(\log n)$ using $O(n)$ processors [25].

2. Parallel algorithms which reduce time but inflate the processor count disproportionately. Most $\mathcal{NC}$ algorithms [27] in the literature fall into this category, $\mathcal{NC}^i$ being the class of problems solvable in time $O(\log^i n)$ with $p(n)$ processors for some polynomial $p(n)$; $\mathcal{NC}$ is the union of $\mathcal{NC}^i$ for all i. These algorithms tend to require too many processors to be practical. For example, a polynomial can be evaluated in time $O(\log(C) \cdot \log(d))$ using $O(C^\alpha \log(d))$ processors, where d is the degree of the polynomial, C is the minimum number of serial steps required to evaluate the polynomial, and two $n \times n$ matrices can multiplied in time n^α (i.e. $\alpha < 2.496$) [77].

3. Other parallel algorithms which utilize multiple processors. These can often be formulated directly from serial divide-and-conquer algorithms. For example, an integer factorization algorithm performing simultaneous trial divisions [81].

4. Nonuniform methods, such as circuits or sorting networks. These can be used to design specialized hardware; if uniformity can be established, they can be used to design algorithms as well. For example, a sorting network requiring $O(\log n)$ steps to sort n elements using $O(n)$ components [3].

Unfortunately, very few results are of type 1. Circuits can be used in a pipelined fashion, so they need not be as inefficient in their use of resources as they appear. For example, each stage of an *FFT* processor [61] can compute simultaneously, so $\lg n$ n-point *FFT*s can be computed in $O(\lg n)$ steps. A parallel program can compute one n-point *FFT* in log-time using n processors; the $O(n \lg n)$ components of the *FFT* processor are likewise efficiently utilized.

2.2.1 Arithmetic

Beame *et al.* [7] shows that integer multiplication, division, and powering can be performed by $\mathcal{P}$-uniform log-depth circuits. It is not known if these circuits can be made logspace-uniform, in which case the problems would be known to be computable in

deterministic logspace. The traditional circuits for arithmetic [50] are, so far, more efficient in their use of resources.

Kannan *et al.* [58] show that the GCD of two integers can be computed in time $O(n \cdot \log\log(n)/\log(n))$ with $n^2 \log^2(n)$ processors on a PRAM, assuming concurrent writes are allowed if they write the same value. This is probably not worth the effort, since the best known serial algorithm is $O(n \cdot \log^2(n) \cdot \log\log(n))$.

Integer factorization methods often involve trial-and-error computations. Watt [81] concludes that any given factorization trial would not be able to use parallelism effectively, but running trials simultaneously would be satisfactory. The method he chooses is *SQUFOF*, which is based on finding a square denominator in the continued-fraction expansion of $\sqrt{x}$. For four-thousand 8-digit (decimal) integers, the average parallel speedup is $0.76n$ for n processors. It is not clear that this algorithm is asymptotically efficient.

2.2.2 Operations on Polynomials

Ponder (chapters 5 and 6) shows asymptotically-optimal ways of performing multiplication and exponentiation of sparse and dense polynomials in parallel. The complexities of sparse multiplication and exponentiation are $\Theta(n^2)$ and $\Theta(n^k/k!)$, where n is the number of nonzero terms of the polynomial and k is the exponent. The best known algorithms for dense multiplication and exponentiation have complexities $O(d \cdot \log(d))$ and $O(dk \cdot \log(dk))$, where d is the degree. The sparse algorithms can be parallelized to use $O(n)$ processors and respective times $O(n)$ and $O(n^{k-1}/k!)$. The dense algorithms can be parallelized to use $O(d)$ and $O(dk)$ processors and $O(\log(d))$ and $O(\log(dk))$ time, respectively. The dense algorithms are based on the FFT algorithm. Kung [60] gives the design of an FFT processor taking **log** time and a linear amount of circuitry. Eberly [32] shows that polynomial multiplication and interpolation, as well as polynomial division with remainder, are in $\mathcal{NC}^1$ using $\mathcal{P}$-uniform circuits.

Von Zur Gathen [79] gives $\mathcal{NC}^\epsilon$ algorithms for polynomial GCD, LCM, squarefree decomposition over fields of characteristic zero, and computing the extended Euclidean scheme of two polynomials over an arbitrary field. Squarefree decomposition over finite fields can be done in time $O(\log^2(n)+(d-1)\log(p))$ with a polynomial number of processors, where the finite field has p^d elements. Polynomial factoring takes time $O(\log^2(n)\log^2(d+1)\log(p))$ with a polynomial number of processors.

The polynomial GCD calculation is useful in simplifying rational expression forms. Watt [81] parallelizes the Sparse Modular GCD algorithm [85], which seems sensible for a practical implementation. It is probabilistic, generating a candidate polynomial in time $O((T+t^2)dvt \cdot \log^2(dv^2t\epsilon^{-1}))$ where v is the number of variables, d is the maximal degree in each variable, $O(T)$ bounds the time to evaluate the polynomial, t is the number of terms, (all in the candidate polynomial) and ϵ is the probability that the candidate polynomial is not the GCD. Parallelizing the independent subcomputations can achieve up to a linear speedup.

Ben-Or *et al.* [8] show that determining (within bounded error) the real roots of a univariate polynomial (provided all roots are real) with integer coefficients is in $\mathcal{NC}$. A reasonably efficient algorithm using time $O(n \cdot \log(n))$ and n processors processors was given by Pan [70]. If the polynomial also has complex roots, the roots can be found in $O(n^2 \log(n))$ parallel steps with $O(n \cdot \log(n))$ processors.

Kaltofen shows that testing the absolute irreducibility of a multivariate integral polynomial over the complex numbers is in $\mathcal{NC}$ [56]. For finite fields the problem is in $\mathcal{NC}$ for dense polynomials, and randomized-$\mathcal{NC}$ otherwise.

2.2.3 Operations on Matrices

Algorithms for manipulating numeric matrices extend to symbolic matrices provided the arithmetic operations performed on the numeric elements have analogs in the domain of the symbolic entries. Thus the (serial or parallel) algorithms for matrix multiplication, inversion, conversion to some canonical forms, and computing the determinant all extend to symbolic matrices so long as only rational arithmetic on the entries is required. Performance will vary depending upon the cost of operating on the symbolic entries. For example, matrix inversion via minor-expansion is less efficient than Gaussian elimination for matrices of numbers or dense polynomials, but minor-expansion is more efficient for matrices of sparse polynomials [40]. Such algorithms are often difficult to analyze because the size of the intermediate expressions varies, making any analysis based on constant cost "base" operations (often grossly) incorrect.

Sasaki & Kanada [75] present a linear-time parallel algorithm for taking the determinant of a symbolic matrix via minor expansion, which takes a factorial processor-time product in the worst case. The Gaussian elimination algorithm can take quadratic time

(times the cost of multiplying/dividing the matrix elements) for a linear number of processors. For sparse polynomial entries, the parallelized minor-expansion is more efficient [40]. The results are identical for the equivalent problem of inverting a matrix.

We provide some pointers to (possibly) relevant numerical matrix results. Czanky [28] and Borodin *et al.* [15] show that computing the determinant and characteristic polynomial are in $\mathcal{NC}^2$. Mulmuley shows that finding the rank of a matrix is in $\mathcal{NC}^2$ [69]. Kaltofen *et al.* show that similarity is in (Las Vegas) randomized-$\mathcal{NC}^2$, for an arbitrary field [57].

Parallel algorithms achieving linear-speedup for numeric matrix-multiplication and inversion are well-known. Czanky [28] shows that matrix inversion and solution to systems of linear equations are in $\mathcal{NC}^2$. Pan [71] shows that for a class of integer matrices (subject to $\log||A|| \leq n^{O(1)}$) the problems of computing the determinant, adjoint, and inverse as well as the coefficients of the characteristic polynomial are in $\mathcal{NC}^2$. The number of processors required is $o(n^{2.5})$. Multiplication of $n \times n$ matrices is in $\mathcal{NC}^1$, requiring $o(n^{2.5})$ processors.

Some simple consequences of this are parallel algorithms for computing the matrix exponential (using a truncated Taylor series), and inverting a matrix of functions in x by expanding into a Taylor-series with matrix coefficients and computing the series form of the inverse by a recurrence [33]. The general problem of evaluating matrix polynomials is in $\mathcal{NC}^2$, using $O(n^{3.496})$ processors (for dense cases) [70].

There are numerous results for special matrix forms. Many of these are applicable to other problems, such as polynomial interpolation and division. Eberly [32] shows that computing the inverse and determinant of a band matrix (over infinite fields) are in $\mathcal{NC}^1$. For finite fields the parallel complexity is $O(\log(n)\log\log(n))$ using a polynomial number of processors. Kaltofen *et al.* show that computing the Jordan normal form is in randomized-$\mathcal{NC}$ for an arbitrary field [57]. For polynomial matrices over finite-fields or the rational numbers, finding the Hermite normal form is in $\mathcal{NC}$ and the Smith normal form is in (Las Vegas) randomized-$\mathcal{NC}$. Bini [12] shows that the solution to a linear system in Toeplitz form can be approximated in parallel time $O(\log(n))$ with $2n$ processors. The exact solution can be provided in time $O(\log(n))$ with $O(n^2)$ processors.

2.2.4 Integration

The "Parallel" Risch algorithm [30] is not a parallel algorithm *per se*, but contains subproblems of substitution, solving nonlinear systems, and integrating subexpressions. Parallelism might be applied to these subtasks.

Braverman [16] implemented a prototype system for searching integral tables, which may integrate expressions faster than algorithmic approaches. Searching can be performed in parallel for a potentially large speedup in the worst case, although a successful system will most likely be able to single out a relatively small set of pattern candidates which can be tested quickly in serial.

2.2.5 Reduction and Simplification

Ponder (chapter 7) examines several proposals for performing Gröbner-basis reduction in parallel, such as simultaneously computing and reducing S-polynomials or simultaneously reducing the basis under different orderings. None of these were found to be particularly useful.

In the Macsyma system [34], simplification of expressions often requires recursive simplification of subexpressions. This divide-and-conquer approach can in principle be parallelized. However, communication between subproblems can be significant, such as when one subexpression evaluating to zero eliminates the other subexpressions as well.

2.2.6 Evaluation of Expressions

Valiant *et al.* [77] shows that any polynomial $p(x)$ can be represented by an expression which can be evaluated in time $O(\log(C) \cdot \log(d))$ with $O(C^\alpha \log(d))$ processors, where C is the minimum number of *serial* steps to evaluate the polynomial, d is the degree of the polynomial, and two $n \times n$ matrices can be multiplied in n^α operations (i.e. $\alpha < 2.496$). This is accomplished by balancing the operator-tree which computes the polynomial.

Lakshmivarahan and Dhall [62] show that linear and 2-way recurrence-relations can be evaluated in logarithmic time using a linear number of processors. This works by evaluating the equivalent closed-form expression in parallel.

Kung [61] shows that, in general, nonlinear recurrences cannot be evaluated in parallel any faster than a constant factor. He also shows some interesting tradeoffs between addition, multiplication, and division for reducing the overhead of exponentiation

and polynomial evaluation, in the domain of scalars, matrices, or polynomials.

2.2.7 Miscellaneous Operations

Galil [39] presented asymptotically optimal parallel algorithms for string-matching using up to $n/\log^2(n)$ processors (up to $n/\log(n)$ if simultaneous writes are allowed). It looks reasonably efficient as a serial algorithm, but operates too intensively on shared memory in parallel. Given a long input text, a short pattern, and few processors it would be more reasonable to break the input text into equal-sized chunks and search each independently. The related problem of unification has been shown $\mathcal{P}$-complete. Auger and Krishnamoorthy [6] show the restricted monadic form is in $\mathcal{NC}^2$, while Vitter and Simons [78] maintain that limited parallelism can still be applied in a practical way to most cases, given the structures are large and fairly richly interconnected.

Parallel sorting has been studied on a variety of models [13]. Networks have been designed requiring $O(\log(n))$ time for $O(n)$ components [3]. These networks are (so far) horrendously complicated, and do not lend to general sorting programs due to their non-uniformity. A parallel mergesort by Cole [25] appears to be nearly practical, combining parallel merges. It operates in $O(\log n)$ steps using n processors. Parallelism is applied on the level of single comparisons. For small numbers of processors and machines with significant process dispatch overheads, straightforward parallelizations of quicksort or mergesort may be most practical, though the processor-time product grows to n^2 for large numbers of processors.

Chapter 3

Parallel Processors and Systems for Algebraic Manipulation: Current Work

What efforts are needed to produce a parallel system (computer architecture and interconnection network) for fast algebraic manipulation? This paper outlines current steps toward such a system, including work in related areas of parallel symbolic processing languages and general- and special-purpose multiprocessor systems.

3.1 Introduction

Parallel processing has been effective in speeding up many numeric scientific computations. Symbolic algebraic manipulation is becoming the target of work in parallel processing because it represents a significant, distinct paradigm of computation midway between numeric processing and the rather vague area of artificial intelligence and Lisp programming. With the exception of certain types of search and low-level vision, applications of symbolic parallel processing are rare.

Our goal is to produce a faster system for algebraic manipulation, as well as to utilize parallel processors efficiently. We will examine work that has been performed in the following areas: the design and implementation of parallel Lisp, architectures for parallel symbolic processing, and integrated systems for algebraic manipulation. The issues of

parallel algorithms and specialized hardware for specific problems in algebraic manipulation are discussed in chapter 2.

3.2 Types of Parallelism and Algebraic Manipulation

There are dozens of proposed variations of parallel architectures. We will focus on four different types.

- *Vector processors.* A number of high-speed parallel vector processors are being marketed, most notably the Cray 1, Cray XMP, and Cray 2. Vector operations are of greatest benefit when operations and data maintain a high degree of regularity, and the exact same operations are performed on each element of a vector.

- *Grid configurations.* A grid of processing elements, each connected to nearest-neighbors in some topology is probably the simplest parallel processing concept. Each processor can write data into a specified register or memory location in its immediate neighbors; it is up to the program to take the necessary steps to transfer data more generally from one processing element to another. A *systolic array* is a special case of a grid multiprocessor. Each processing element is synchronized with a global clock and only computes a limited range of functions. Data generally travels in a specific direction across a systolic array. Systolic arrays are intended to be implemented in VLSI on the scale of a single chip or wafer.

- *Message passing.* A variation on the *grid* is to separate the job of computation from the job of communication, using special switching circuitry for communication. This allows data to be transmitted across the grid at a faster rate than that determined by the computation speed of the processors. Message passing usually involves assigning a number to each processing element, and the switching network creates a path along which the message passes. As with grid configurations, the network topology affects the speed at which a collection of point-to-point transactions can be performed.

- *Shared memory.* A collection of processing elements sharing a common memory is a simple abstraction. Programs or hardware must resolve the problem of simultaneous updates to the same memory location. The processor/memory interconnection must be organized to give the appearance of fast memory access.

A few algorithms in algebraic manipulation have the kind of regularity necessary for vector or systolic array computation. One example is the *FFT* which can be used for multiplying dense polynomials with coefficients in an appropriate domain. Most algorithms are not presented with such regular data. Most systems concentrate on algorithms for manipulating (the more common) sparse polynomials.

Grid and message passing architectures require programs to be organized in specific and unusual ways. A convenient program abstraction is to use a collection of processes passing streams of data. When only nearest-neighbor communication is allowed, the process interconnection must embed into the grid. The stream abstraction can be difficult to use, since it provides no globally shared memory. The message passing overhead must be reasonably small. Many divide-and conquer algorithms appear suitable for such architectures, each processor farming out subcomputations to idle neighbors.

Shared memory processors would seem to have the most potential because the program organization is less restricted. The availability of global data is an advantage over grid and message passing architectures. Shared memory tends, however, to be slower than would otherwise be possible. Special efforts in organization (and, usually, higher prices) can compensate, to some extent.

3.3 Architectures Supporting Parallel Lisp

Of particular interest to us are architectures to support Lisp, since much of the work in symbolic math has been programmed in Lisp.

3.3.1 Connection Machine "*Lisp"

The Connection Machine [53] is composed of a large number (on the order of 65,000) of 1-bit processing elements coupled with small local memories (on the order of 4Kbits) on a hypercube with an overlaid grid. A master processor broadcasts instructions which are executed by all of the grid processors simultaneously (SIMD execution). Vector operations are used to process data in parallel. **Lisp* [76] is Lisp extended to include the *pvar* data type, which is a vector upon which the vector operations are performed. Each element of the *pvar* vector is stored in the same location in the local memory of a different processor. The vector operations are defined on the machine-level. Either arithmetic, logical, or vector-permutation operations may be performed. If the vector is larger than

the number of available processors, additional processors are simulated. Since most of the conventional Lisp is executed on the serial processor host, rather than the Connection Machine, it is perhaps a misnomer to call this Lisp, rather than an embedded parallel language accessible from Lisp.

3.3.2 Concert "Multilisp"

Concert [49] consists of 8 clusters of 4 processors connected in a ring. Each processor is coupled with a memory module. The memory modules and processors in each cluster are connected to a common bus. The levels of connection allow global memory references which vary in speed depending on the connection between the processor and the memory unit. Multilisp [48] is compiled into bytecodes. Bytecode-interpreters are run on each of the processors on the Concert machine.

Multilisp parallelizes function calls two different ways. The first is the *PCALL*, which evaluates all the arguments to a function in parallel before calling the function. The second is the *FUTURE*, which begins evaluating its argument if there is an available processor. If the value produced by the future is required before it is ready, the evaluation of the argument is forced.

3.3.3 BBN "Butterfly MultiScheme"

The Butterfly [21] consists of from 128 to 256 processing elements connected by a "butterfly" network. Each processing element is a 32-bit microprocessor, floating-point unit, and from 1 to 4 Mbytes of memory. Memory is treated as a global space, with reference time varying depending on whether the access is to the local memory or memory on another node.

MultiScheme [20] on the Butterfly is essentially the same as Concert Multilisp, using the same language design and bytecode-interpreter implementation. A native-code compiler is in progress.

3.3.4 Alliant "Qlisp"

The Alliant [5] consists of up to eight processors connected by a crossbar-switch to four caches. Each cache stores data from a different quadrant of memory. The caches

are connected to main memory by a bus. Each processor has floating-point and vector-processing units.

Qlisp [38] has two primitives *QLET* and *QLAMBDA*. *QLET* binds a set of variables to a set of corresponding expressions. The expressions are evaluated in parallel by spawning a process for each one. *QLAMBDA* spawns a process to evaluate the closure of the expression; when the closure is *apply*ed, the parameters are passed to the process which then evaluates and returns the result.

3.3.5 SPUR Lisp

The Berkeley SPUR processor [52] consists of from 6 to 12 processing elements connected to a common bus. Each processing element consists of a cache, a microprocessor CPU, and a floating-point coprocessor. The bus is connected to a shared global memory of 32 Mbytes. A cache-consistency protocol is used; shared memory is cached so long as only one process writes into it. Thus shared memory is accessed at relatively high speeds so long as processes do not update the same regions of the same data structures simultaneously. SPUR is still in the development stage; significant portions of the hardware and software are currently in the debug stage.

SPUR Lisp [86] is a parallel extension of Common Lisp. Primitives for forking processes and writing into and reading from process *mailboxes* as well as the Multilisp *FUTURE* construct are included. It appears that mailboxes can be implemented efficiently using atomic test-and-set type operations, yet provide a good abstraction of parallel process interaction.

3.3.6 Intel Hypercube "Concurrent Common Lisp"

The Intel Hypercube [55] consists of 2^n processing elements connected as an n-dimensional hypercube. The parameter n currently ranges from 0 to 6. Each processing element consists of a 32-bit microprocessor, an optional floating-point coprocessor, and up to 4.5 Mbytes of local memory. A *cube manager* [22] processor is used to control computations on the Hypercube. Library routines are available for initiating, terminating, and sending and receiving messages to and from processes running on different nodes. There is no shared memory. Programs are loaded on each node and started remotely from the cube manager. Information is passed between nodes by message-passing. Concurrent Common Lisp [22]

consists of 2^n separate Lisp systems communicating with input/output streams.

3.3.7 Ultracomputer "Zlisp"

The Ultracomputer [44] consists of 8 processing elements and 8 global memory units separated by an "Omega" network. Each processing element consists of a processor and a local memory unit. References to global memory are mediated by the network, which combines simultaneous operations to the same memory cell. Working prototypes have been constructed using a bus to simulate the network.

Zlisp [31] includes primitives for forking processes and locking data; more abstract parallel constructs are built from these. Data is held in the global memory units, which creates a uniform memory space but is slower to access.

3.3.8 Bath Concurrent Lisp Machine

The Bath Concurrent Lisp Machine [63] consists of four processing elements connected in a square. Each processing element consists of a 32-bit processor and 256Kbytes of local memory. Communication is performed through memory shared between adjacent processors on the network.

Programs must be designed to start processes in the various processing elements, and data must be passed explicitly between adjacent processing elements by writing to or reading from the appropriate location in the appropriate shared memory. More abstract constructs are provided by the Lisp implementation to simulate *fork/join* operations and messages passed between processes. The compiler is designed to detect subtasks which can proceed in parallel, and generate the necessary process-manipulation and data-transfer operations.

The Bath Concurrent Lisp Machine is not under active development [66].

3.4 Related Work

The only demonstration we have seen reported of a parallel algebraic manipulation system is a distributed Maple system [81] running on a local-area (Ethernet) network of VAX-11/780's. A master process transmits subproblems to and receives results from subordinate Maple processes executing on other machines. No performance measurements have

been published, although the system is reported to be slow [24]. Ethernet communication can easily dominate execution time, so such a system should only outperform a uniprocessor implementation for problems which are overall expensive to compute, easily decomposed, and do not involve a significant amount of data traffic. This may be useful as a prototype for a more efficient implementation, such as a shared-bus multiprocessor.

Gupta [46] presents a parallel interpreter for the OPS-5 language which was used to execute a variety of production systems. Parallel search was *not* used; rather, the stages of rule matching and firing were parallelized in the interpreter. Not surprisingly, considering the nature of the interpretation process, the amount of parallelism measured was quite limited.

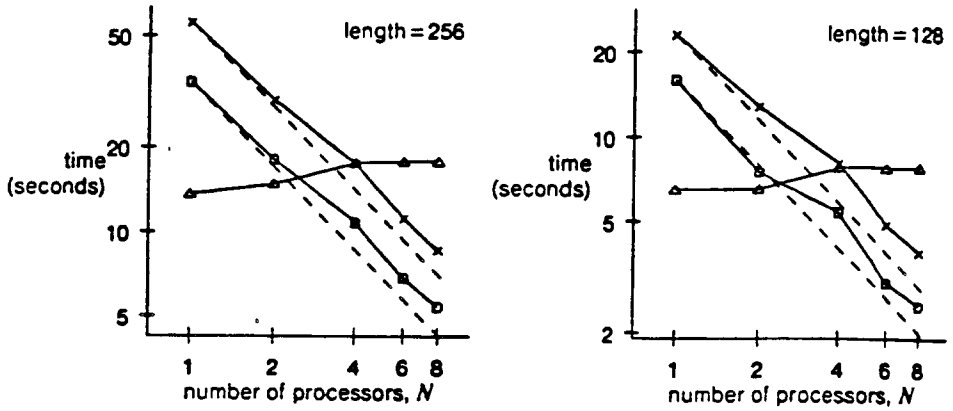
The L-machine presented by Buchberger [19] was intended as a multiprocessor targeted for symbolic computation. The L-machine consisted of a series of processors and a series of memories connected to each other by a crossbar switch. The process/stream abstraction is used for programming this configuration. Current research is concentrating on the L-language for parallel algebraic operations, and possible implementations on currently available multiprocessors.

Several parallel architectures for executing Prolog have been proposed; these are based on performing searches in parallel. The lack of mechanisms for controlling search in Prolog and its parallel variants coupled with problems with simultaneous structure updates (analogous to side-effects) will probably make Prolog unsuitable for parallel evaluation [73], although languages similar in some respects to Prolog can be implemented.

3.5 Reported Experiments

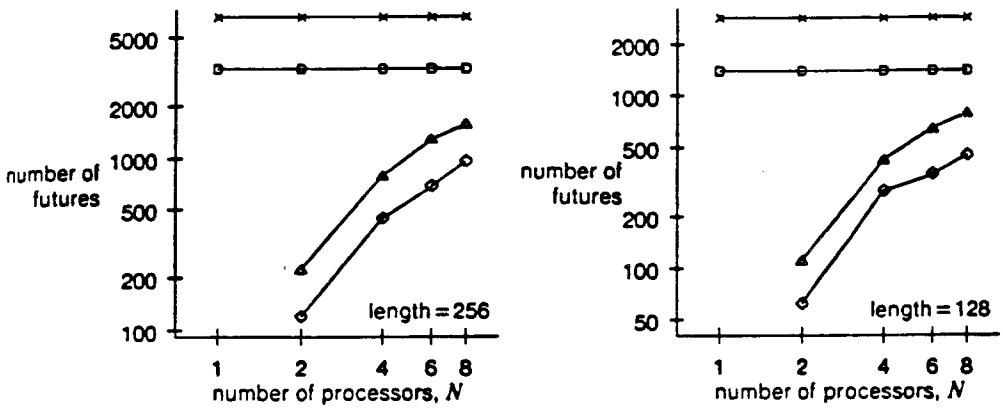
The Concert Multilisp paper [48] reports some experiments running a Quicksort program on 1 to 8 processors. The inputs were of length 128 and 256. A sequential sort is compared to two parallel quicksort programs. The slowdown of the sequential sort as processors are added gives some notion of the overhead involved. The graphs are reproduced

as follows:



Key: × = "parallel" Quicksort: Algorithm 1 unmodified
□ = "unfolded" Quicksort: Algorithm 1 with unfolded partition
△ = "sequential" Quicksort: Algorithm 1 without futures

Some additional statistics are reported, such as the sample standard deviation of the performance for 10 randomly-generated lists of each length. More interesting is the number of *FUTURE*s generated and waited for in the execution:



Key: × = futures created by "parallel" Quicksort
□ = futures created by "unfolded" Quicksort
△ = futures waited for in "parallel" Quicksort
○ = futures waited for in "unfolded" Quicksort

Coupled with the cost of generating and activating *FUTURE*s, this should give some idea

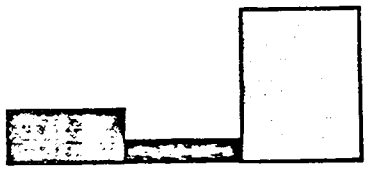
of the parallel overhead.

Concurrent Common Lisp has been tested on several of the Gabriel Lisp benchmarks; the results for the *Triangle* and *Browse* benchmarks are reported in [11]. The *Puzzle* benchmark is reported as not parallelizing. The single-node performance tended to be between 15 percent to 15 *times* slower than the Symbolics 3600.

Triangle Gabriel Benchmark

	iPSC Single node	iPSC 16 nodes	iPSC 32 nodes	Symbolics 3600
T (seconds)	1,023.2	69.8	37.5	151.70
Speedup		14.8	27.6	

Browse Gabriel Benchmark



	iPSC Single node	iPSC 16 nodes	iPSC 32 nodes	Symbolics 3600
T (seconds)	401.4	15.0	15.0	30.8
Speedup		26.8	26.8	
Pre-load T	398.5	9.4	5.2	
Speedup		42.4	76.6	

A slight superlinear speedup is observed in the *Browse* benchmark, reportedly due to reduced load from garbage collection and other overheads with more processors. The pre-load time assumes that the program is already loaded, lowering the execution time and demonstrating more dramatic speedup.

The original Qlisp paper includes some simulation results [38]. The simulator is written in Lisp; the reported results are in the form of raw time or factors of speedup. The benchmarks are computing the Fibonacci function, summing the elements of a full binary tree and a linear list (using recursion in figure 2, and message-passing in figure 3), solving a traveling salesman problem using exhaustive search, and executing the *Browse* benchmark.

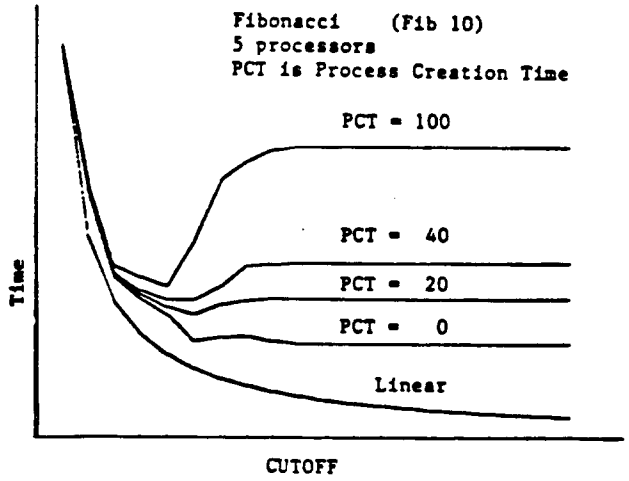


Figure 1

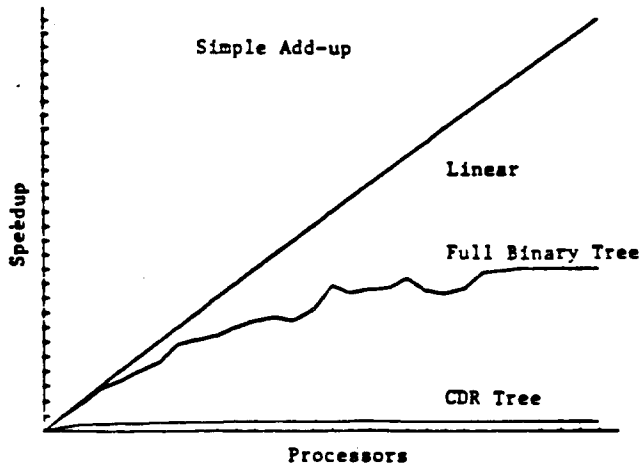


Figure 2

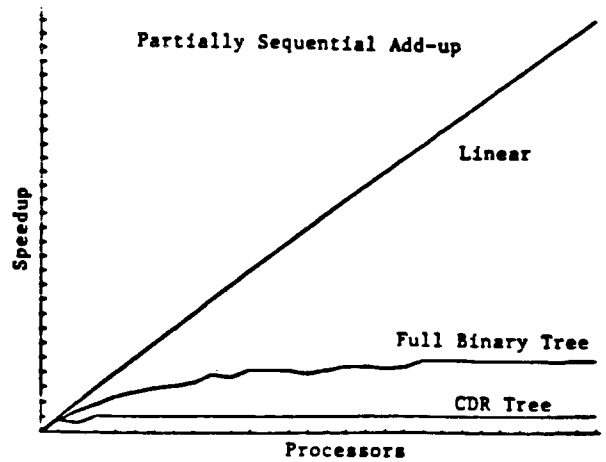


Figure 3

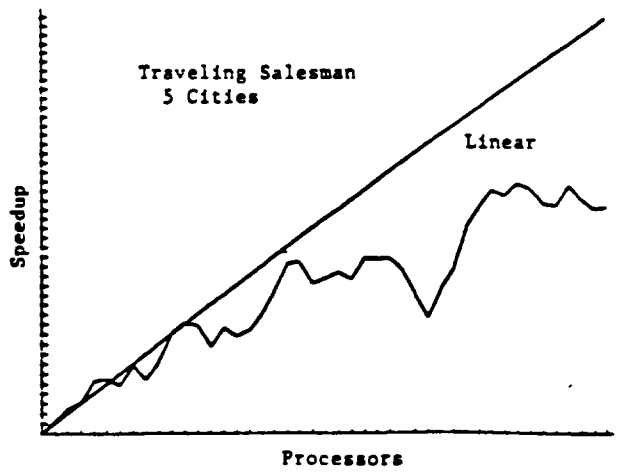


Figure 4

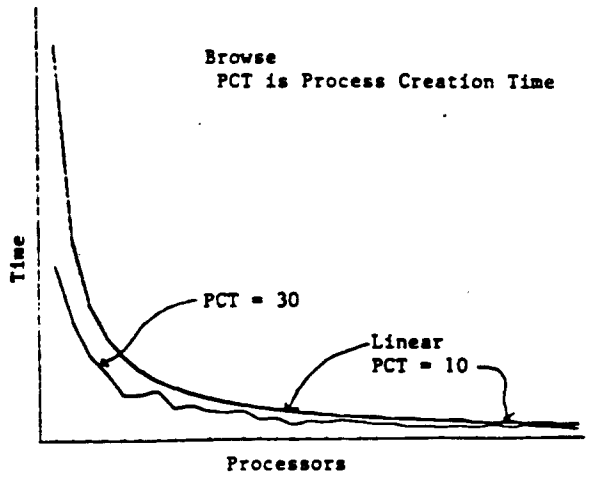


Figure 5

A compiled version of Qlisp has been implemented on the Alliant, with 4 processors. Experiments were run using two different algorithms for powering a polynomial (chapter 5). each which decomposes the input into a fixed number of processes which are then run on the available processors:

Table 1 – Time to expand $(x^{13} + x^{12} + \cdots + x + 1)^7$ with 4 processors.

# processes	*Simp*	*Karatsuba*
1	3537	7589
2	1864	—
3	—	2375
4	1067	—
8	1100	—
9	—	1875

A near-linear speedup is observed in the *Simp* algorithm, which seems to strictly outperform *Karatsuba*.

3.6 Experiences in Retrospect

I have found the state-of-the-art generally disappointing. The Hypercube Concurrent Common Lisp tended to deadlock when running compiled code because the rate of message traffic increased as the between-message computation time was reduced. Alliant Qlisp would fail for unknown reasons even for serial code. Many empirical results could not be obtained.

The most annoying factor was the failure of implementors to go beyond the stage of a limping prototype. Language-level simulators and distributed interpreters are easier to construct than a full parallel Lisp system running compiled code. Such prototypes can produce deceptively positive results, as the real machine overheads of multiprocessing are disguised by the simulator. Qlisp, for example, ran approximately 3 times slower than Franz Lisp on a VAX 11/785, even though the Alliant is generally 4 times faster. For that reason we did not consider using the Concert bytecode-interpreters or the Butterfly distributed interpreters for realistic performance estimation (Concert Multilisp is reported to run roughly as fast as *interpreted* Franz Lisp on the same uniprocessor [48]. In other words, a factor of 10

or more of parallelism would be necessary to even compete with a mediocre compiled Lisp). Simulators can be of some use for tuning the parameters of an efficient implementation by providing some information about process interaction and memory usage, but without an efficient implementation all such improvements are hypothetical.

We are still several steps away from achieving our goal of an efficient parallel system for algebraic manipulation. A fast, robust parallel Lisp (or other suitable symbolic language) remains to be implemented. Some algebraic manipulation system must be rewritten enough to port, or a new one constructed. Finally, some program sections must be identified as worth parallelizing, and then parallelized. We have concentrated on this final task, assuming the first steps were being done by others (in parallel). We look forward to resolution of these difficulties over the next several years.

Chapter 4

Polynomial Forms and Representations for Algebraic Manipulation

The choice of representation affects the efficiency of algorithms operating on polynomials, as well as the structure and space efficiency of an algebraic manipulation system. A number of representations are compared for both theoretical and practical efficiency. Issues specific to parallel algorithms are also considered.

4.1 Introduction

Polynomials, series, and rational functions represent important classes of expressions in algebraic manipulation. Efficient operations on polynomials are requisite for an efficient algebraic manipulation system. In designing a faster algebraic manipulation system using parallelism, we must consider possible representations for polynomials to enhance both the serial and parallel efficiency of polynomial operations.

Several polynomial representations are used in existing algebraic manipulation systems, have been proposed, are used in the operation of certain algorithms, or may seem acceptable for other reasons in the literature. Clearly we cannot exhaustively analyze the range of possible representations. Nor can we fully characterize the relationship between a given representation and a particular polynomial operation, since an algorithm for per-

forming the operation is free to convert representations or maintain additional information. We will restrict ourselves to discussing the "best known" polynomial representations and algorithms for operating on them, with the understanding that superior methods may be developed in the future.

The efficiency issues we will consider are the size of the representation and the cost of performing particular operations using it, under certain assumptions about the frequencies of various operations and the characteristics of the polynomial data. We assume each operation on a coefficient takes constant time[1].

4.2 Forms of Polynomial Expressions

A polynomial is a function formed by a finite number of additions and multiplications over a set of variables and coefficients (usually integers). Polynomials are closed under addition, subtraction, multiplication, powering, differentiation, integration, and a variety of other operations such as substitution. Hence a "general expression"[2] such as

$$p(x,y) := \left(\frac{d}{dt}(t+1)^3\right)|_{t=x+y-1}.$$

denotes a polynomial in x and y. Canonical forms of polynomial expressions are more interesting, since each polynomial can be uniquely represented. Two good examples are factored form (product-of-sums)

$$3(x+y)^2$$

and fully expanded form (sum-of-products)

$$3y^2 + 6xy + 3x^2$$

The representation is unique so long as the terms are ordered. One useful ordering is *lexicographic*, where the degree of the dominant variable is used to pairwise order terms, and ties are broken recursively by ordering in the remaining variables. Another is *total degree* ordering, where sum of the degrees of the variables in each term are compared and ties are broken lexicographically [18].

A wide range of forms arise from the grouping of the sum and product operations; factored and expanded forms being special cases. A form arising from lexicographic ordering

[1] This is false in general, but in the absence of specific information is a "neutral" assumption.

[2] This term is used in Macsyma[34], for example, for a form containing a wide variety of operations – the underlying data structure of an operator/operand tree is quite general.

is *principle variable form*, where the polynomial is written as a univariate polynomial in the dominant variable, with polynomial coefficients in the remaining variables:

$$y^2(3) + y^1(6x) + y^0(3x^2)$$

In fact the groupings can be made considerably more complicated. Expanded form can be regarded as the Taylor expansion of the polynomial around the zero vector; were we to take the expansion around $x = 1$ we would get the series

$$(3y^2 + 6y + 3)(x-1)^0 + (6y+6)(x-1)^1 + 3(x-1)^2.$$

A polynomial is a linear combination of generators. The generators are the power-products of the variables. Alternatively, a set of orthogonal polynomials could be chosen as the basis (the power-products of $(x-1)$ and y in the example above). Important examples of alternative groupings are Chebyshev series, where the generating polynomials are of the form

$$T_0(x) = 1, \quad T_1(x) = x, \quad T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$

and Legendre series. These series are useful in deriving polynomial approximations and solving differential equations.

Two other unique forms in common use are vectors of coefficients and vectors of interpolation points. For both of these representations the vector must be of length at least d_i+1 in the ith dimension, where d_i is the maximum degree to which the ith variable occurs. Letting the value at the (i,j) coordinate be the coefficient of $x^i y^j$ gives us the coefficient matrix (for the same polynomial)

$$\begin{bmatrix} 0 & 0 & 3 \\ 0 & 6 & 0 \\ 3 & 0 & 0 \end{bmatrix}$$

Letting the value at the (i,j) coordinate be $\mathrm{p}(i,j)$ gives us the matrix of interpolation points

$$\begin{bmatrix} 0 & 3 & 12 \\ 3 & 12 & 27 \\ 12 & 27 & 48 \end{bmatrix}$$

The *Fast Fourier Transform (FFT)* can be used to compute a particular form of interpolation point vector over a finite field. Once in this form, simple algorithms can be used to

perform multiplication and powering operations with high efficiency (chapter 6). Since the *FFT* can be computed in $O(n \log n)$ where n is the number of elements in the vector, this leads to some useful algorithms. An operand vector must be as large as the vector required to unambiguously represent the result of the operation.

The "general expression" form is poor for a number of reasons, so it will be discussed only briefly:

- It is obviously non-unique. A given polynomial (0, for example) can be expressed as arbitrarily complicated forms.
- It can be expensive to store and operate on expressions in this form, since it is not likely to be as compact as a canonical form. Operations such as determining whether the expression is zero may be as expensive as converting into one of the canonical forms. (For certain general forms of polynomials, such as expressions composed of '+' and '×' operations, very fast tests can be performed with probability of correctness increasing in the length of the test [42]. An example is a zero-test which evaluates the polynomial at random points; if the polynomial evaluates nonzero at any point, the polynomial is definitely not zero. Derivative and integral operators cannot be dealt with so easily).

On the other hand, if we are committed to retaining canonical forms it requires some effort to preserve the form under operations like addition or multiplication. Preserving a factored form under addition is not known in general to be easier than expanding both polynomials, adding, and factoring again (which is an expensive operation). Preserving expanded form under multiplication will be discussed later on.

Vector forms are wasteful when the polynomial is *sparse* (i.e. many lower-order terms are zero, such as $x^{100} + 1$). Sparse polynomials represented in expanded form tend to be more compact, taking space proportional to the number of nonzero terms. For *dense* polynomials (i.e. most terms are nonzero, such as $x^5 + x^4 + x^3 + x^2 + x + 1$) vector representations are very efficient since less information is used to represent each term. The principle variable form is particularly interesting since the polynomial may be sparse in one variable and dense in another; a hierarchy of different concrete representations might be most efficient in such cases. In the next section we will look at concrete representations of polynomials based on the forms of polynomial expressions.

4.3 Concrete Representations of Polynomials

An algebraic manipulation system must have a concrete representation of any expression it operates on. There are a number of classes of data structures which uniquely or non-uniquely represent an expression in a particular form. Some examples are as follows:

1. A character string. Any of the forms of expressions in the previous section corresponds to a string of characters, which is literally the *written form* of the expression.

2. An ordered or unordered set of subexpressions. For example, expanded and factored forms are equivalent to sets of monomials or factors, respectively. The subexpressions are then represented by another data structure. The set may be maintained by such data structures as a linked-list, array, tree, or hashtable[43]. Linked-lists are easiest to manage for small sets, but cannot be efficiently accessed in parallel; nor can they be used efficiently by algorithms which access terms in non-sequential order. Since arrays are directly addressable, they can be superior for parallel algorithms simultaneously operating on different portions of the polynomial.

3. An array of numbers. The indices of the array can be used to implicitly represent information, such as making the ith element the coefficient of x^i. Thus arrays are useful for representing vectors of coefficients or interpolation points succinctly.

4.4 Practical Considerations

To speed up polynomial operations through algorithmic approaches, we need to consider the nature of the polynomial data and the frequency of particular operations. It is unfortunately difficult to characterize "average" use of a system like Macsyma, but we believe the following to be the case for typical algebraic manipulation systems' usage:

1. Polynomial operations are performed with high frequency mainly on small polynomials. [3] Degrees and sizes over 100 are considered rare (note: a polynomial of degree 100 considered as an analytic function is generally very poorly behaved, and is unlikely to be an accurate model of a physical system. Large dense polynomials may well arise in contexts other than analytic approximation, such as computational geometry).

[3] Note that this may be conditioned by the fact that large polynomials can take a long time to operate on....

2. Most polynomials are very sparse, i.e. have mostly zero coefficients.

3. Additions and multiplications tend to be performed on polynomials of differing degree and coefficient size.

4. The ordering of operations by frequency is roughly: Addition, Multiplication, Powering, Division with remainder, GCD.

5. The result of one operation tends to be used subsequently in a different type of operation, i.e. the product of two polynomials is then added to a third polynomial.

We also make some assumptions about our host environment:

6. Space-extravagant representations may degrade performance by overloading memory. Practically speaking, storage recovery and/or paging overhead can erase algorithmic time improvements.

7. Parallel processing has a significant overhead in process dispatch and message passing or references to shared memory.

8. Converting an expression to printable form takes less time than printing it.

Considerations 2 and 6 make vectors of coefficients desirable only for dense polynomials, since the space and time overheads would otherwise be lower under expanded or principle-variable form. In addition, consideration 3 makes vectors of interpolation points undesirable as a general representation since one operand vector will tend to be larger than the other; either the larger vector contains redundant points, or the smaller one will have to be expanded. The cost of generating new points is as much as having used coefficient form to begin with. However, for large dense polynomials in coefficient form, multiplication and powering are performed fastest by converting to *FFT* form, operating, and (usually) converting back.

Considerations 4 and 5 suggest that factored form would be hard to use, since the cost of expanding the polynomials for additions would be a large factor, regardless of whether the form is to be preserved. Factoring results to maintain a canonical form is expensive, and would dominate virtually any possible benefits from the ability to multiply and power more easily. Algorithms based on extracting easily (or previously) computed factors were first described in Altran [47].

Consideration 8 suggests that character strings would be a poor representation; algorithms for performing the algebraic operations would be forced to waste time analyzing the string to determine the values of coefficients and exponents, while the possible savings due to maintaining a readily printable representation would not be significant compared to the cost of the I/O operation involved.

4.5 Cost of Operations

In light of the previous discussion, we will consider the following 4 representations:

1. Linked-lists of monomials in descending order, for polynomials of few terms. Each monomial is represented as a tuple of the coefficient and exponents.

2. Arrays of monomials in descending order, for large sparse polynomials. The monomials are represented by tuples as before.

3. Arrays of coefficients of the appropriate dimension, for large dense polynomials.

4. Arrays of interpolation points of the appropriate dimension (i.e. *FFT*).

The nature and representation of a polynomial will affect the cost of operating on it. Here we will list some operations which are frequently performed, and sketch the best known algorithms for performing them in serial or in parallel. The operations are addition/subtraction, multiplication, powering, division, GCD, and differentiation/integration.

1. *Addition/subtraction.* Under ordered monomial forms, addition and subtraction are performed by matching terms with the same degree. This amounts to an end-to-end scan of each polynomial, taking time proportional to the number of terms. This is fast enough that parallelism may not be interesting, especially if fast memory access is inhibited by a linked-list representation. Parallel divide-and-conquer approaches can be used with arrays, trees, or hashtables of monomials, though these are at a disadvantage because it cannot be decided easily how many terms will be in the result; either there can be empty cells in the result, or the computation risks being serialized by parallel insertion or merging operations. For coefficient or *FFT* vectors, corresponding elements are added or subtracted directly, in serial or in parallel. Leading terms may cancel, so the result vector may be unnecessarily large.

2. *Multiplication.* For monomial forms, multiplication can be performed by taking the cross product of monomials and combining like terms, in time quadratic in the number of terms (algorithms which explicitly sort the results require an additional *log* factor). This cross-product can be computed in parallel. There is no convenient way known to combine terms in parallel, though for small numbers of processors and large polynomials the merging cost is dominated by the multiplication (chapter 5). The *FFT* form can be multiplied in serial or in parallel by multiplying corresponding terms. The *FFT* can be computed in parallel to and from a coefficient array form. The size and precision of the *FFT* values must be large enough to uniquely represent the result.

3. *Powering.* The *FFT* form is powered by powering each point in the vector, in serial or in parallel. The relationship between *FFT* and coefficient form is as with multiplication. Sparse polynomials in monomial form are more easily powered by generating the multinomial expansion in serial or in parallel using dynamic programming. As with multiplication, for large polynomials and small numbers of processors the serial cost of combining terms is dominated by the parallel operations (chapter 5).

4. *Division with remainder.* Division is accomplished by multiplying the divisor by a monomial such that the leading terms become equal (terminate if this cannot be done) subtracting, and repeating until the dividend is of smaller degree than the divisor[26]. This is convenient in any but the *FFT* form (which appears to provide no easy way to divide). For sparse polynomials the size of the intermediate results may grow with each step, so using arrays of monomials to hold intermediate results would involve frequent reallocation of arrays. Linked-lists are easier to manage. At best the polynomial subtraction and monomial multiplication steps can be parallelized, as discussed before.

5. *GCD.* The fastest known polynomial GCD algorithm can be (crudely) characterized as evaluating the two polynomials at corresponding points in a finite field, computing the corresponding GCD's, and interpolating the resulting polynomial through these points. Multi-point evaluation and interpolation are best performed on arrays using algorithms similar to the *FFT*, taking time polynomial in the degrees of the two polynomials. The evaluation/interpolation can be performed in parallel on arrays, and the individual finite-field GCD's can be computed in parallel. Zippel's Sparse

Modular GCD[85] uses Hensel lifting rather than interpolation; this appears amenable to parallel implementation.

6. *Differentiation/integration.* For sum-of-products form these operations amount to termwise changes of the exponent and coefficient. Parallelism can be used on an array of monomials or coefficient. There is no way known to differentiate/integrate interpolation point vectors than to convert to coefficient form, operate, and convert back.

Some additional common operations are used to determine attributes of a polynomial, for example:

1. *What is the degree of the polynomial in variable x?*
2. *How many distinct variables occur in p?*
3. *What variable occurs to highest degree?*
4. *What coefficient is attached to $x_1{}^{i_1} \cdots x_v{}^{i_v}$?*
5. *Is the polynomial equal to zero?*

If any of these trivial operations becomes very expensive, we must seriously question the representation. In some abstract sense, an algorithm for operating on a polynomial works by systematically extracting attributes of the polynomial. The cost for the whole polynomial operation is generally cheaper than the cumulative cost of repeating these operations, since the algorithm will generally maintain "fingers" into the structure which will make subsequent attribute operations cheaper. These questions can be decided as follows:

- Questions 1-4 can be decided in time linear in the size of the representation in the monomial forms (representations #1 and 2, this section), by scanning the elements of the polynomial.
- If the coefficient array form (representation #3) contains no unnecessary zeros, questions 1-3 can be decided in constant time by looking at the dimensions of the array. Question 4 can be decided by indexing into the array.
- For the *FFT* (representation #4), questions 1-3 seem to require as much time as computing the transform. Question 4 takes as much time as to evaluate the polynomial (linear in the size of the transformed representation).

- Question 5 requires a zero test of each element of the polynomial. If representations 1-3 are compact (no unnecessary zeros) this requires constant time. The *FFT* form uses at least $d + 1$ evaluation points for a degree d polynomial, so the size of the transform is at least $d + 1$, and the time to perform a zero test will be proportional to this.

These questions can be decided for general expressions describing polynomials, by evaluating them at a sequence of points and applying the *sparse multivariate interpolation algorithm* [9], which recasts the polynomial into a set of nonzero monomials. The interpolation requires $O(t^2(\log^2(t) + \log(nd)))$ operations for a polynomial containing t nonzero monomials, n variables, and d is the maximum degree to which any variable appears. $2t$ evaluations of the expression are required. This conversion cost is more expensive than using representations 1-4 in any operation except GCD; this is further complicated by the need for a sequence of $2t$ prime numbers to evaluate the expression over.

An additional problem is printing the polynomial in one of the expanded or factored forms. Factoring is expensive, so converting between the given forms is not costly enough to be an issue. An array of coefficients may contain unnecessary zeroes to be eliminated. The coefficient array form can be traversed different ways to print the polynomial under various orderings; the monomial representations would have to be reordered to be printed in any but the existing ordering.

4.6 A Hybrid Approach

An algebraic manipulation system could be constructed to use a mixture of polynomial representations, tuned to the nature of the given polynomial and the operations being performed on it. Further, if the same polynomial is to be used in different operations, it may be useful to keep one representation suitable to each operation. The following attributes would be accounted for, but not generated unless needed:

1. The variables the polynomial depends on, and the degree in each variable.

2. The size of the polynomial, i.e. the number of nonzero terms in the expansion.

3. The printable representation of the polynomial.

4. One of the following "base" representations mentioned in the previous section, depending upon the size and density of the polynomial: a linked-list of monomials (for small polynomials), a vector of monomials (for large sparse polynomials), or either or both of a vector of coefficients and its *FFT* of some length (for large dense polynomials).

The degree, size, and form of operand polynomials can be used to estimate the degree and size of the result polynomial:

1. Addition/subtraction: the (estimated) degree of the result is the maximum degree of the operands; the (estimated) size of the result is the sum of the sizes of the operands.

2. Multiplication: the (exact) degree of the product is the sum of the degrees of the operands. The (estimated) size of the product is the product of the sizes of the operands, unless this exceeds the maximum possible for the given degree in which case the maximum is used.

3. Powering: the (exact) degree of the result is the degree of the operand times the power k it is raised to. The (estimated) size of the result is $\binom{s+k-1}{k}$ where s is the size of the operand; if this exceeds the maximum possible for the degree of the result then this maximum is used.

4. Division: the (exact) degree of the quotient is the difference of the degrees of the dividend and divisor. The degree of the remainder is (at most) that of the divisor. The (estimated) size is the maximum for that degree.

5. Differentiation/integration: the (exact) degree is the degree of the operand in the given variable, minus 1 or plus 1 depending upon the operation. The (estimated) size is the same, unless it exceeds the maximum possible for the degree, in which case this maximum is used.

In each case the estimates can be overestimates since term-cancellation can occur, or the constant term can drop out under differentiation. For GCD the result will be assumed to be most likely 1, which can be an underestimate since the operands may have many common factors.

The size and degree estimates can then be used to decide *a priori* a good representation for the result; if the result is small, then a linked-list of monomials will be generated.

If the ratio of the estimated size to the maximum possible under the degree constraints exceeds a certain value, then the polynomial will be treated as dense, and sparse otherwise. For each operation there would be a number of specific algorithms, depending upon the form of the operands and the intended form of the result. Specifically, if two large dense polynomials are to be multiplied, use the *FFT* form. If the transforms have not been computed, or are too small, compute larger ones (note that if it is too large by a factor of 2 or more, skip intermediate points). Produce the result as an *FFT*. For other operations, use whatever form is available and cast the result into the intended form.

The drawbacks of this "object-oriented" approach are as follows:

- The estimated size and degree characteristics of a result polynomial may be too large, in which case time and space would be wasted filling in for zero monomials. Under considerations 1 and 2 in section 2, this may be quite frequent.

- Estimating the characteristics of the result and checking the available representations of the operands adds an overhead to the computation. This would not be significant for sufficiently large polynomials.

- Maintaining additional information would add a space overhead. The printable form is almost inevitably more easily reconstructed on a need basis. For large dense polynomials, maintaining the largest *FFT* yet used may be a large expense; a small polynomial may have been multiplied by a much larger polynomial, requiring a very large *FFT* to have been computed. For sufficiently small polynomials the degree and size attributes would represent a significant overhead. Under consideration 1 in section 2, this overhead may add up over a large number of small polynomials.

This "object-oriented" approach can be taken a level further by building graphs of polynomials linked with operators, where results are only computed on a "need" basis (usually when a result must be printed, or if the polynomial must be evaluated); the evaluation algorithm can then choose the representation of each intermediate polynomial based on the form the final result must take, and the intermediate representations which are already available. Idle processing time or parallel processors can be utilized to "eagerly" fill in intermediate polynomials or perform subcomputations on a demand basis. The potential drawbacks with this approach are as follows:

- "Eager" work will be wasted if the results are not used.

- Work which could have been performed after each user command may be deferred until the final answer is requested, resulting in a conspicuous delay.

- Space may be saved by using destructive operations on the polynomials themselves, rather than constructing new objects for each operation.

- The results of most operations may be needed immediately, so this framework becomes a needless complication.

On the other hand, under the proper circumstances such a system might efficiently utilize idle time and processors. An approximation to this approach is available in Macsyma by turning the simplification routine off, constructing an expression, and then (if needed) simplifying; the simplifier performs the operations in one sweep to produce the result (depending on flags and the nature of the expression) in simplified or canonical form.

4.7 Conclusions

An efficient algebraic manipulation system may benefit from using a rich variety of data representations, including arrays and lists. In some cases representations "left over" from computations may be useful for subsequent operations; for example, a polynomial which has been previously converted to *FFT* form could be multiplied more easily using the old *FFT*.

This richness and redundancy can be taken to the extreme by designing an algebraic manipulation system which maintains mixed representations of each polynomial and uses algorithms tuned for the best available combination of operand representations. Such an algebraic manipulation system would be somewhat more complicated since the different possibilities must be accounted for in the algorithm.

The Macsyma system currently tags expressions with type information, which is used to direct computation. The set of tags is extensible, for the purpose of introducing new expression types and manipulation algorithms. Type-checking is often performed on subexpressions within inner loops; this overhead might be reduced by quantifying the range of useful expression types and selecting type-specific algorithms on the top level. Alternatively, a few useful types such as "large dense univariate polynomials with floating-point coefficients" could be isolated for treatment with specialized representations and algorithms.

Such an adjustment would probably be a significant improvement in performance in selected domains.

Chapter 5

Parallel Multiplication and Powering of Sparse Polynomials

This paper examines two asymptotically efficient parallel algorithms *Simp* and *NOMC* for multiplying and powering sparse polynomials. *Simp* is a simple divide and conquer approach to multiplication. *NOMC* uses a multinomial expansion for computing powers. The *Simp* algorithm carries over to the multiplication of Poisson series.

5.1 Introduction

Sparse polynomials represent an important class of expressions in algebraic manipulation. Efficient serial algorithms exist for multiplying and powering sparse polynomials; it is the purpose of this paper to explore the potential of these algorithms for parallel execution. We assume a model of parallel execution based on low overhead shared-memory multiprocessing. Some simple empirical studies for up to 4 processors suggest these methods work.

The data structures used to represent polynomials have some effect on the efficiency of the algorithms.

The organization of this paper is as follows:

1. The nature of sparse polynomials is discussed.
2. Lower bounds on the complexities of polynomial multiplication and powering are derived.

3. The *Simp* and *NOMC* algorithms are described and analyzed.

4. Some alternative algorithms are discussed.

5. Empirical measurements on a parallelized *Simp* algorithm are presented.

6. The *Simp* algorithm is extended to Poisson series multiplication.

5.2 Sparse Polynomials

We will refer to the number of nonzero monomial terms in a polynomial p as size(p). We compute degree(p) (the *total degree of* p) by summing the exponents in each monomial, and taking the maximum of these sums. This definition of *total degree* is used in papers on Gröbner-basis reduction [18].

Alagar & Probst [4] define the term *uniformly dense* to describe multivariate polynomials whose size is nearly maximal for the given total degree. A polynomial is *uniformly sparse* if it has few terms relative to the maximum possible for the given total degree. Obviously, *dense* and *sparse* are qualitative terms. Some algorithms are very efficient for dense polynomials but highly inefficient for sparse polynomials.

The following relations show how degree and size are bounded:

$$\text{degree}(p_1 p_2) = \text{degree}(p_1) + \text{degree}(p_2) \tag{5.1}$$

$$\text{size}(p_1 p_2) \leq \text{size}(p_1) \cdot \text{size}(p_2) \tag{5.2}$$

$$\text{degree}(p^k) = k \cdot \text{degree}(p) \tag{5.3}$$

$$\text{size}(p^k) \leq \binom{\text{size}(p) + k - 1}{k} \tag{5.4}$$

Letting v be the number of variables,

$$\text{size}(p) \leq \sum_{i=0}^{v} \binom{v}{i} \binom{\text{degree}(p)}{i}, \tag{5.5}$$

which reduces to

$$\text{size}(p) \leq \text{degree}(p) + 1 \tag{5.6}$$

for $v = 1$.

Over a domain with zero-divisors, relation (1) is adjusted to

$$\text{degree}(p_1 p_2) \leq \text{degree}(p_1) + \text{degree}(p_2) \qquad (1a)$$

Relation (4) is derived as follows [35]: let $p = A + B$, where B is a monomial and A contains all remaining terms. Then

$$p^k = (A+B)^k = \sum_{i=0}^{k} \binom{k}{i} A^i B^{k-i}$$

If no collapsing occurs, as happens for suitably sparse p, each $A^i B^{k-i}$ pair will contribute exactly $\text{size}(A^i) = \text{size}(p) - 1$ terms. If A is a monomial this degenerates to one term per pair, or $k+1$ terms. Otherwise we have a total number of terms

$$\begin{aligned} \text{size}((A+B)^k) &= \textstyle\sum_{i=0}^{k} \text{size}(A^i) = \sum_{i=0}^{k} \binom{(\text{size}(p)-1)+i-1}{i} \\ &= \textstyle\sum_{i=0}^{k} \binom{\text{size}(p)+i-2}{i} = \sum_{i=0}^{k} \binom{\text{size}(p)+i-2}{\text{size}(p)-2} \\ &= \binom{\text{size}(p)+k-1}{k} \end{aligned}$$

using the identity

$$\sum_{i=0}^{k} \binom{r+i}{r} = \binom{r+k+1}{k}.$$

The size for sparse polynomials will grow at most quadratically as multiplications are performed. The degree will grow at most linearly under both multiplication and powering. Since size is ultimately bounded by degree, quadratic growth cannot be sustained under repeated multiplications as polynomials "fill in". As the "density" of the results increases, relatively fewer distinct terms will be generated by multiplication. The fastest the size of p^k can grow is approximated by

$$\binom{\text{size}(p)+k-1}{k} \sim \frac{\text{size}(p)^k}{k!} + O\left(\frac{\text{size}(p)^{k-1}}{2(k-2)!}\right)$$

for large k and increasing $\text{size}(p)$.

5.3 Complexities of Multiplication and Powering

The complexity measures we concern ourselves with are the number of coefficient additions and multiplications, and the number of exponent comparison steps required to order the result. We will consider any of these to be "scalar" although coefficient operations may be floating point or arbitrary precision, and hence are potentially more expensive than exponent comparisons. We are also concerned with the maximum number of processors that can be kept busy with useful work in parallel algorithms. The parallel complexity is the maximum number of scalar operations used by any parallel branch of the computation. The algorithms we present are "balanced" in the sense that we attempt to "farm out" all parallel computations to processors in equal "chunks". The maximum and average number of scalar operations required per processor are close. For an ideal algorithm, the processor-time product, a good measure of parallel efficiency, is the same as the serial complexity.

Coefficient multiplication is probably the most expensive of the scalar operations as used in the Macsyma rational function package [34]. The parallelized algorithms we look at tend to parallelize efficiently the multiplication operations, but require serialized exponent addition or comparison to combine separate subresults. The exponent operation count does not dominate serial computation for reasonable-size input, and we conjecture that exponent operations will become only slightly more important in parallel.

We begin by showing that the number of scalar multiplications required for polynomial multiplication or powering depends strongly upon the size of the result.

Lemma 1:

At least $\text{size}(p_1 p_2) - \text{size}(p_1) - \text{size}(p_2)$ multiplications are required in the worst case to compute $p_1 p_2$.

Proof:

Let A, B, and C be the sets of coefficients in p_1, p_2, and $p_1 p_2$, respectively. If the elements of A and B are algebraically independent, the elements of C will be (at least) linearly independent. Given that we have generated C with h multiplications, let $E = \{e_1, ..., e_h\}$ be the set of products of the multiplications. C must be formed by linear combinations of A, B, and E:

$$c_j = \sum_{i=1}^{h} x_{i,j} e_i + \sum_{i=1}^{n_1} y_{i,j} a_i + \sum_{i=1}^{n_2} z_{i,j} b_i$$

$$1 \leq j \leq \text{size}(p_1 p_2)$$

Since the elements of C are linearly independent, it must be the case that $\text{size}(p_1 p_2) \leq h + \text{size}(p_1) + \text{size}(p_2)$, or $h \geq \text{size}(p_1 p_2) - \text{size}(p_1) - \text{size}(p_2)$. □

As a simple corollary, for p_1 and p_2 "sufficiently sparse", $\text{size}(p_1 p_2) = \text{size}(p_1) \cdot \text{size}(p_2)$, so roughly $\text{size}(p_1) \cdot \text{size}(p_2)$ multiplies are required. The *Simp* algorithm operates in exactly this bound. For "sufficiently dense" polynomials, $\text{size}(p_1 p_2) = \text{size}(p_1) + \text{size}(p_2)$ so the lower bound on multiplications is linear. The best known algorithm in this range (*FFT* [14]) uses $\Theta((\text{degree}(p_1) + \text{degree}(p_2)) \cdot \log(\text{degree}(p_1) + \text{degree}(p_2)))$ scalar multiplications. For powering, we have the similar result [35]:

Lemma 2:

At least $\text{size}(p^k) - \text{size}(p)$ scalar multiplications are required to compute p^k.

Proof:

As before, the coefficient set C is linearly independent so the number h of intermediate products must satisfy the inequality $\text{size}(p^k) \leq h + \text{size}(p)$, or $h \geq \text{size}(p_k) - \text{size}(p)$.

We know from relation (4) how the size of p^k can grow for sparse polynomials. The algorithm *NOMC* operates asymptotically with this. For dense univariate polynomials the growth is bounded more strictly by relations (3) and (5), giving a bound of $k \cdot \text{size}(p)$. The best known algorithm for this case is the *FFT* using $\Theta(k \cdot \text{size}(p) \cdot \log(k \cdot \text{size}(p)))$.

Comparison operations are required to order the results. The problem of ordering the monomials of the product $p_1 \cdot p_2$ is equivalent to the "Sorting X+Y" problem (discussed by Harper *et. al* [51]) of ordering all pairwise sums of the (ordered) elements of two vectors. For $\text{size}(p_1) = \text{size}(p_2) = n$, the lower bound on the number of comparisons is $\Omega(n \lg n)$. A solution by Jean Vuillemin [80] uses $O(n^2)$ comparisons, i.e. proportional to the largest possible size of the result. Ordering the monomials of p^k is equivalent to sorting all distinct k-wise sums of elements of the vector X. Vuillemin's result is extended by recursively sorting the sets of $\lfloor \frac{k}{2} \rfloor$- and $\lceil \frac{k}{2} \rceil$- wise sums and sorting the pairwise sums of these two sets as before. The work to sort the final set will dominate the lower-order sets (see the analysis of *NOMC* in section (5)), so the total number of comparisons used is proportional to the largest possible size of the result

$$\begin{pmatrix} \text{size}(p) + k - 1 \\ k \end{pmatrix}.$$

Terms are combined by the addition operations, once comparisons have established that they are additive. Thus a lower bound on the number of coefficient additions required is

$$\text{size}(p_1) \cdot \text{size}(p_2) - \text{size}(p_1 p_2)$$

or

$$\left(\begin{array}{c} \text{size}(p) + k - 1 \\ k \end{array} \right) - \text{size}(p^k),$$

which is the number of terms possible under relations (2) or (4), minus the actual size of the result. In the worst case $p = p_1 = p_2$ and are completely dense. Under relations (1), (3), and (6) the number of coefficient additions required are

$$\text{size}(p)^2 - 2 \cdot \text{size}(p)$$

or

$$\left(\begin{array}{c} \text{size}(p) + k - 1 \\ k \end{array} \right) - k \cdot \text{size}(p) \sim \frac{\text{size}(p)^k}{k!} + k \cdot \text{size}(p)$$

for multiplication and powering, respectively. Assuming that the results are ordered, the coefficient additions amortize into the exponent comparisons. The addition operations involved in generating new exponents amortize into the coefficient multiplications.

It is not obvious how to parallelize the addition or comparison operations in either the *Simp* or *NOMC* algorithms. In each case terms generated by different processes may add together (or even cancel). In both the *Simp* and *NOMC* algorithms we will use a parallel mergesort [2] (just parallelize the recursive calls), which requires

$$\Theta(x + \frac{x}{y} \log \frac{x}{y})$$

comparisons to sort x items with y processors, y ranging from 1 to x. For fixed y and increasing x it approaches a speedup linear in y, though this is not a linear speedup over Vuillemin's result. A reasonably efficient parallel mergesort has been developed by Cole [25], which takes $\Theta(\log x)$ operations to sort x items with x processors. Parallelism is applied to operations on individual elements, a very fine level of granularity. This process still appears inefficient and unnecessarily complicated for (small) fixed numbers of processors and large inputs, particularly if there is a significant overhead to interprocessor communication or shared memory access.

A hash table of monomials can be used as an unordered representation of polynomials [43]. A hash table can be updated in parallel, or used in serial to reduce the number of comparisons required to combine terms. The integrity of a hash table is difficult to maintain under parallel updates. If this were not an issue, the operations involved with combining terms would parallelize perfectly. The details involved with locking the hash

buckets are complicated enough to possibly negate any advantages. Additional overheads such as computing the hash function are also significant.

5.4 The *Simp* Algorithm for Multiplication

A simple way to multiply polynomials p_1 and p_2 is as follows [36]:

Given polynomials p_1, p_2, return $p_1 \cdot p_2$.

[1] If p_1 is a monomial, multiply each term of p_2 by p_1 and return result. Otherwise,

[2] Split p_1 into A and B, and recursively form the products $A \cdot p_2$ and $B \cdot p_2$.

[3] Merge the two partial products (ordered by exponent), adding coefficients with the same exponent. Return result.

This effectively decomposes polynomial p_1 into size(p_1) monomials, forms the product of each monomial with p_2, and successively merges the results. The parallelized form is to perform the recursive calls in step 2 in parallel.

The number of coefficient multiplications required is size(p_1)size(p_2). The merge step amounts to a recursive balanced merge, requiring $O(\text{size}(p_1)\text{size}(p_2) \cdot \log(\text{size}(p_1)\text{size}(p_2)))$ comparisons. Ther are size(p_1)size(p_2) − size($p_1 p_2$) additions.

In parallel, $k \leq \text{size}(p_1)$ processors can be used. The recursive step 2 is better replaced by splitting p_1 k ways and having each processor perform the multiplication as before. Step 3 reduces to performing a parallel mergesort (section 3) on the partial products. The number of parallel multiplications is $\frac{1}{k}\text{size}(p_1)\text{size}(p_2)$, since multiplications are performed on the bottom level of the decomposition.

The parallel mergesort uses a parallel measure of comparisons $O(\text{size}(p_1 p_2) + \frac{\text{size}(p_1 p_2)}{k} \log \frac{\text{size}(p_1 p_2)}{k})$, which is asymptotically $O(\text{size}(p_1 p_2))$ as k approaches size(p_1). Re-

placing the parallel merges with a (serial) k-way balanced merge gives a parallel measure of comparisons $O(\text{size}(p_1p_2)\cdot\log(k)+\frac{\text{size}(p_1p_2)}{k}\log\frac{\text{size}(p_1p_2)}{k})$. For message-passing multiprocessors with a high communication cost, this decomposition is probably superior.

Thus, given sufficient processors, the best time for the multiplications is $O(\text{size}(p_1))$ and for additions and comparisons is $O(\text{size}(p_1p_2))$. For $\text{size}(p_1) = \text{size}(p_2) = n$, this amounts to a reduction by $\Theta(n)$ in the time to perform multiplications and $\Theta(\lg(n))$ in the time to perform comparisons. In the worst case the number of additions is reduced at most by a constant factor, since $\Theta(n^2)$ terms can combine in the final merge.

5.5 The *NOMC* Algorithm for Powering

The *NOMC* algorithm *(full multinomial expansion with dynamic programming)* is one of many asymptotically efficient algorithms for powering sparse polynomials. Several alternatives are mentioned in section (6). The algorithm is expressed as follows:

Given a polynomial $p = (a_1 + ... + a_t)$ and a power k to be computed, return p^k.

[1] If p is a monomial, power it and return the result
(e.g. $p = cx_1{}^{i_1}\cdots x_n{}^{i_n}$, where c is a coefficient,
return $p = c^k x_1{}^{i_1+k}\cdots x_n{}^{i_n+k}$.

[2] Tabulate products of powers of each a_n of total degree $\lfloor\frac{k}{2^i}\rfloor, \lceil\frac{k}{2^i}\rceil$ for $i = 0,...,\log(k)$ using the relation
$a_1^{I_1}a_2^{I_2}\cdots a_t^{I_t} = (a_1^{J_1}a_2^{J_2}\cdots a_t^{J_t})\cdot(a_1^{K_1}a_2^{K_2}\cdots a_t^{K_t}),\quad I_i = J_i + K_i,$
$I_1+\cdots+I_t = m,\quad J_1+\cdots+J_t = \lfloor\frac{m}{2}\rfloor,\quad K_1+\cdots+K_t = \lceil\frac{m}{2}\rceil.$

[3] Return $\sum_{I_1+\cdots+I_t=k}\begin{pmatrix} k \\ I_1I_2\cdots I_t\end{pmatrix}a_1^{I_1}a_2^{I_2}\cdots a_t^{I_t},\quad I_1+\cdots+I_t = k.$

For k a power of 2, the number of monomials tabulated is

$$\sum_{i=1}^{\log(k)} \binom{\text{size}(p) + \frac{k}{2^i} - 1}{\text{size}(p) - 1} < \binom{\text{size}(p) + k - 1}{\text{size}(p) - 1} + \sum_{i=1}^{\frac{k}{2}} \binom{\text{size}(p) + i - 1}{\text{size}(p) - 1}$$
$$= \binom{\text{size}(p) + k - 1}{\text{size}(p) - 1} + \binom{\text{size}(p) + \frac{k}{2}}{\text{size}(p) - 1} \sim \binom{\text{size}(p) + k - 1}{\text{size}(p) - 1}.$$

Since each monomial is formed by multiplying two monomials from the table (using one coefficient multiply and v exponent additions), the number of coefficient multiplications is asymptotically the same as the number of monomials in the result. The higher-order term continues to dominate for k not a power of 2. Each successive multinomial coefficient is generated from a previous one using one integer multiply and up to one integer divide.

For a parallel implementation, the result monomials are broken into groups. Each processor fills in what is needed in the monomial table to compute its group of monomials. The coefficient multiplications are parallelized perfectly for up to

$$o\left[\frac{\binom{\text{size}(p) + k - 1}{\text{size}(p) - 1}}{\binom{\text{size}(p) + \frac{k}{2}}{\text{size}(p) - 1}}\right]$$

processors, since the overhead of constructing the lower-order monomials is dominated by the cost of generating the k-order monomials. For large k and increasing size(p), this bound approaches

$$\mathcal{K} = \frac{\text{size}(p)^{k/2}}{k \cdot (k-1) \cdot \cdots \cdot (k/2)}$$

processors. Beyond this, the second-order term

$$\binom{\text{size}(p) + \frac{k}{2}}{\text{size}(p) - 1}$$

becomes significant and restricts the asymptotic speedup. Therefore the best time for the multiplications is

$$\binom{\text{size}(p) + \frac{k}{2}}{\text{size}(p) - 1}.$$

For $\mathcal{K}$ processors the parallel mergesort requires a number of comparisons asymptotically proportional to

$$\left(\begin{array}{c} \text{size}(p) + k - 1 \\ \text{size}(p) - 1 \end{array} \right) \sim \frac{\text{size}(p)^k}{k!}$$

for large k and increasing size(p). Therefore the best time for the additions and comparisons is

$$O(\frac{\text{size}(p)^k}{k!}).$$

5.6 Some Other Algorithms

As mentioned earlier, the *FFT* algorithm is the most efficient way known to multiply and power *dense* polynomials in serial. Several other algorithms were developed prior to the *FFT*, including *Karatsuba* [36] [4], which works by divide-and-conquer: splitting the two polynomials into equal-sized parts, and adding their partial products. Careful arrangement of additions and subtractions eliminates the need to compute one partial product for half-splitting, so $O(n^{\log_2 3})$ multiplies are performed (in the dense case). Partitioning into quarters gives an algorithm requiring $O(n^{\log_4 9})$ multiplies, etc. Another is the *Eval* algorithm [36] [59], a conceptual predecessor of the *FFT*. *Karatsuba* appears reasonable for sparse polynomials, in serial or in parallel, though it will degenerate to performing the same operations as *Simp* for sufficiently sparse cases.

So far we have made no distinction between *univariate* and *multivariate* polynomials. The *Simp* and *NOMC* algorithms depend on the input being uniformly sparse: a multivariate polynomial may be separated into sparse and dense components, and operated on more efficiently by a combination of algorithms favoring sparsity and density. A polynomial is *nonuniformly dense* in variable x if x appears raised to (almost) every degree up to a maximum; for example, the polynomial

$$x^n y^0 + x^{n-1} y^1 + ... + x^0 y^n$$

is nonuniformly dense in both x and y, but is not uniformly dense since all terms of cumulative degree $< n$ are absent. If a polynomial nonuniformly dense in x is written as univariate in x with polynomial coefficients, a density-favoring algorithm can be used on the "backbone" of the polynomial, while a sparsity-favoring algorithm can be used to operate on the polynomial coefficients.

A recent paper by Alagar & Probst [4] used a combination of *Simp*, *Karatsuba*, and *FFT* for multiplying multivariate polynomials. They found that *Simp* tended to outperform *Karatsuba* by about 40% for univariate cases. It is reasonable to believe that *Simp* should perform even better for sparse polynomials. Alagar & Probst's polyalgorithm should be quite efficient for multiplying polynomials of arbitrary densities, since *Simp*, *Karatsuba*, and *FFT* are each most efficient for different cases. Such a polyalgorithm should adapt well to parallelism since the *Simp*, *Karatsuba*, and *FFT* can be parallelized individually. Alternately, a general library for polynomial manipulation might include a parallel subroutine for each algorithm so the user can decide based on information about the nature of the polynomial data. The presence of parallelism will probably shift the "cutoff" points which determine which algorithm will be most efficient, depending upon how well the multiplication, addition, and comparison operations parallelize with respect to each other.

For powering, several algorithms are presented in [35] [36] [74]. *NOMC* is an efficient variation of the *NOMB* algorithm in [35]. The references focus on the asymptotically-efficient *BINB*, for *binomial expansion with half-splitting*. This is a divide-and-conquer algorithm partitioning the polynomial into the sum of two polynomials, and using the binomial expansion to form the result. Computing A^n by multiplying $A^{n/2} \cdot A^{n/2}$ is more expensive than by computing $A^{n-1} \cdot A$. Performing polynomial multiplications simultaneously will not balance the workload among the processors, since the binomial expansion contains power-products where the exponents are balanced all different ways. Parallelism can be used in performing each particular polynomial multiplication, but this would be applying parallelism to large numbers of small subproblems, accumulating process-spawning overhead. *NOMC* was formulated as an alternative, spawning processes at the top-level to divide the problem into equal-sized subproblems, one per processor.

5.7 Empirical Results

An experiment was run to compare parallelized versions of the *Simp* and *Karatsuba* algorithms, on a 4-processor Alliant running Qlisp. These were implemented as recursive algorithms with the *Simp* algorithm generating 2 processes per level of recursion and the *Karatsuba* algorithm generating 3. Each program we tested uses the parallelized algorithm up to a fixed number of processes, and the serial algorithm afterward. Table 1 shows the result in milliseconds, garbage-collection time excluded. The codes are presented in the

appendices.

Table 1 – Time to expand $(x^{13} + x^{12} + \cdots + x + 1)^7$ with 4 processors.

# processes	*Simp*	*Karatsuba*
1	3537	7589
2	1864	——
3	——	2375
4	1067	——
8	1100	——
9	——	1875

The *Karatsuba* algorithm achieved almost perfect linear speedup up to the number of actual processors available. The *Simp* algorithm didn't quite, but still ran strictly faster as processors were added, for this input. The cutoff point for *Karatsuba* to outrun *Simp* was between $(x^{60} + x^{59} + \cdots + x + 1)^7$ and $(x^{80} + x^{59} + \cdots + x + 1)^7$ (a well-coded *FFT* should outrun both for as low as $(x^5 + x^4 + \cdots + x + 1)^3$); unfortunately the Qlisp system broke for problems that large.

A serial version of the powering problem ran in 1283 ms. on a VAX 11/785 with (Franz Inc.) Common Lisp and 3537 ms. on the Alliant with one-processor Qlisp; the fact that the Alliant is in other respects generally faster than the VAX suggests that Qlisp is poorly implemented.

5.8 Poisson Series

A Poisson series is an expression of the form

$$\sum_m R_m(x_1, ..., x_k) f(i_{1,m} y_1 + ... + i_{n,m} y_n)$$

where f is the *sine* or *cosine* function, the i terms are integers multiplying the indeterminates $y_1, ..., y_n$, and the R terms are typically rational functions in the indeterminates $x_1, ..., x_k$, over the integers. The product of Poisson series can always be represented in the same form, by use of the identities

$$\cos(\mathrm{x})\cos(\mathrm{y}) = \frac{1}{2}(\cos(\mathrm{x} - \mathrm{y}) + \cos(\mathrm{x} + \mathrm{y}))$$

$$\sin(\mathrm{x})\sin(\mathrm{y}) = \frac{1}{2}(\cos(\mathrm{x} - \mathrm{y}) - \cos(\mathrm{x} + \mathrm{y}))$$

$$\sin(x)\cos(y) = \frac{1}{2}(\sin(x-y) + \sin(x+y)).$$

Addition, differentiation, and various other operations are also closed under this system. Because no indeterminate occurs both in $R_m(x_1, ..., x_k)$ and in $f(i_{1,m}y_1 + ... + i_{n,m}y_n)$, the system is closed under integration.

Multiplication of Poisson series is a significant component of celestial mechanics computations [37]. Poisson series multiplication is similar to multiplication of sparse multivariate polynomials, though the product of two terms is two terms (rather than one) and collapsing terms is trickier. Ordering terms on the $f(i_{1,m}y_1 + ... + i_{n,m}y_n)$ component allows similar terms to be combined since the $R_m(x_1, ..., x_k)$ forms are additive.

The *Simp* algorithm extends directly to Poisson series, with the same serial and parallel complexities. The *FFT* and *Karatsuba* algorithms hinge upon assumptions that no longer apply. Since we are adding and multiplying rational functions for coefficients and comparing small vectors of small integers for the arguments, the cost of coefficient operations dominates, in practice, the argument computations. The *Simp* algorithm decomposes in parallel as before using a number of processors proportional to the input length n_2 giving a parallel complexity of n_1 multiplies and $n_1 n_2$ comparisons.

5.9 Conclusions

The parallel *Simp* and *NOMC* algorithms are efficient with regard to the coefficient multiplication operations, yielding a reduction in parallel multiplications which is linear in the number of processors. Since coefficient multiplication is by far the most expensive of the scalar operations, for sparse polynomials of reasonable size we expect a near linear speedup corresponding to the reduction in parallel multiplies. Provided "large" polynomials contain "large" coefficients, the time spent performing scalar multiplications should continue to dominate the cost of the remaining operations.

The parallel mergesorts used to combine terms are not asymptotically optimal with regard to the total number of comparisons performed, nor is the reduction in parallel comparisons linear with the number of added processors. For large numbers of processors, the number of coefficient multiplications required to multiply polynomials of size n is reduced by a factor of roughly n, while the comparisons are reduced by a factor of only $\log(n)$. For sufficiently "large" polynomials containing "small" coefficients and for large numbers

of processors, the comparisons and additions begin to dominate the cost, since they are not so completely parallelized.

For a fixed number of processors and "sufficiently large" inputs, *Simp* and *NOMC* approach a linear reduction in all operations. Since parallelism is useful only for relatively large inputs – only then is the overhead of process subdivision and scheduling dominated by the parallel components of the computation – it appears that these two algorithms should be of practical value and attain a nearly linear speedup as processors are increased in number, up to a threshold increasing in the size of the input.

Chapter 6

Parallel Multiplication of Dense Polynomials

Of the various techniques for multiplication of dense multivariate polynomials, one approach, based on the *FFT*, emerges as the best. Variations on the *FFT* can be parallelized efficiently on both general-purpose multiprocessors and special-purpose circuits. Some practical issues of applying the *FFT* are discussed.

6.1 Introduction

Polynomials represent an important class of expressions in algebraic manipulation. Efficient operations on polynomials are requisite for an efficient algebraic manipulation system.

The *FFT* algorithm for multiplying and powering dense polynomials is the most efficient known, demonstrating superiority to other algorithms (such as *Eval* [36] [59] and *Karatsuba* [4] [36] [59]) both in asymptotic analysis and (except for very small cases) empirical evaluation. It has been conjectured to be optimal.

In designing a parallel system for algebraic manipulation, using parallelism in performing polynomial operations is one of the more attractive techniques to keep processors computing productively during substantial amounts of computation. The *FFT* algorithm shows great promise as a parallel algorithm running on shared-memory or message-passing multiprocessors.

6.2 Dense Polynomials

We will refer to the number of nonzero monomial terms in a polynomial p as size(p). We compute degree(p) (the *total degree of p*) by summing the exponents in each monomial, and taking the maximum of these sums. This definition of *total degree* is used in papers on Gröbner-basis reduction [18].

Alagar & Probst [4] define the term *uniformly dense* to describe multivariate polynomials whose size is nearly maximal for the given total degree. The maximum size of a polynomial for a given total degree is given by the relation

$$\text{size}(p) \leq \sum_{i=0}^{v} \binom{v}{i} \binom{\text{degree}(p)}{i}, \tag{6.1}$$

where v is the number of variables. For univariate polynomials this reduces to

$$\text{size}(p) \leq \text{degree}(p) + 1. \tag{6.2}$$

For v variables where exponents run from 0 to n in each variable,

$$\text{size}(p) = (n+1)^v. \tag{6.3}$$

Under multiplication and powering the degree grows with the relations

$$\text{degree}(p_1 p_2) = \text{degree}(p_1) + \text{degree}(p_2) \tag{6.4}$$

$$\text{degree}(p^k) = k \cdot \text{degree}(p). \tag{6.5}$$

For completely dense univariate polynomials, the size and degree grow linearly under multiplication and powering.

The complementary term *nonuniformly dense* is used to describe a polynomial such that each variable appears raised to nearly every degree, but the number of terms it not maximal. An example is the convolution

$$x^d y^0 + x^{d-1} y^1 + \ldots + x^0 y^d$$

where only terms of total degree d are present, though the degrees of x and y range fro 0 to d.

6.3 The Basic *FFT* Algorithm

The *FFT* algorithm [2] [14] [65] [83] is useful for multiplying dense polynomials with coefficients from the field of complex numbers or a finite computation structure (typically the integers modulo a prime). The precision or size of the modulus must be decided *a priori*, to be at least as large the precision of the result. This is an inconvenience in algebraic manipulation systems where arbitrary integer coefficients are commonly used.

The *FFT* works by computing the *discrete Fourier-Transform* of the univariate polynomial $p(x)$, which is the vector $[p(\omega^0), ..., p(\omega^{s-1})]$ where d is the degree of $p(x)$, the size of the transform $s \geq d+1$, and ω is some *principal sth root of unity.* The inverse of the transform is computed in a nearly identical way; effectively the transform vector can be regarded as the coefficients of the polynomial $p'(y) = \sum p(\omega^i)y^i$ and the vector of coefficients of $p(x)$ is $[\frac{p'(\omega^{-0})}{n}, ..., \frac{p'(\omega^{-s+1})}{n}]$. By evaluating two polynomials at $\omega^0 ... \omega^{s-1}$, multiplying the corresponding values together, and converting the result back, the product polynomial is produced. Likewise, powering is performed by evaluating the polynomial, powering each resulting value, and converting back to get the powered polynomial.

The (Cooley-Tukey) *Fast Fourier Transform (FFT)* is the basic algorithm for computing the discrete Fourier transform in time $O(d \log d)$. It works as follows:

Let s be a power of 2, $s \geq d+1$
Let $A = [a_0, ..., a_{s-1}]$ be the coefficients of $p(x)$ (padded with zeros if necessary) in some computation structure C.
Let $A' = [a'_0, ..., a'_{s-1}]$ be the coefficients in the transformed polynomial $p'(y)$.
Let ω be a primitive sth root of unity in C.
$A' = FFT(A, s, \omega)$.

Recursive $FFT(A, s, \omega)$:
[1] if $s = 1$, return a_1.
[2] split coefficients by index into even-indexed sequence B and odd-indexed sequence C.
[3] $B' \leftarrow FFT(B, s/2, \omega^2)$, $C' \leftarrow FFT(C, s/2, \omega^2)$.

[4] for $i \leftarrow 0$ to $s/2 - 1$ do

$A'_i \leftarrow B'_i + \omega^i C'_i$.

$A'_{i+s/2} \leftarrow B'_i + \omega^{i+s/2} C'_i$.

[5] return A'.

Some multiplication and addition operations are hidden in the powering and the manipulation of the indices. Asymptotically, $\Theta(d \log d)$ additions and multiplications are performed since each step of the recursion performs work proportional to the subproblem size, and the recursion works on subproblems of exactly half the size. d must be rounded up to a power of two. Numerous tricks can be used to trim the number of operations [2] by a constant factor, specifically by "unravelling" the recursion and using bit-operations to permute the coefficients as necessary. An iterative form of the FFT can be stated as follows:

Iterative $FFT(A, s, \omega)$:

[1] for $l \leftarrow 0$ to $\lg(s)$ do

[2] for $i \leftarrow 0$ to $s - 1$ do $t_i \leftarrow A_{i+1}$.

[3] for $i \leftarrow 0$ to $s - 1$ do

$r_i \leftarrow t_{i \wedge (-2^{\lfloor \lg i \rfloor - l})} + \omega^{\frac{s}{2^{l+1}} \cdot bitreverse(l)} \cdot t_{i \vee (2^{\lfloor \lg i \rfloor - l})}$.

[4] for $l \leftarrow 0$ until $s - 1$ do $A'_{bitreverse(l)} \leftarrow r_l$.

[5] return A'.

where $\lfloor \lg x \rfloor$ is the greatest integer no larger than the log base-2 of x, $bitreverse(x)$ is the reversal of the bits in x (within the fixed word length), and $\wedge$ and $\vee$ are bitwise *and* and *or*, respectively.

The *Good-Winograd* algorithm [14] [83] provides another decomposition for cases where s is *not* a power of 2, but is a product of two relatively prime integers of roughly equal size. The Good-Winograd algorithm factors the degree into two relatively-prime numbers and treating the polynomial as if it were bivariate (the multidimensional algorithm

is described in section (4)). Powers of x are implicitly replaced by power-products of the two new variables. Either the Good-Winograd or the Cooley-Tukey algorithms can be applied to the subproblems, depending upon their size. The complexity is the same as the Cooley-Tukey algorithm.

If we consider the finite-field *FFT* in terms of bit operations, rather than integer operations [2], the complexity reads somewhat higher. Letting b be the number of bits required to hold the final answer (i.e. $b \leq \lceil 2\lg(x)\lg(s) \rceil$ for multiplication, where x is the number of bits required to hold the largest of the coefficients of the operands, or $b \leq k \lg x \lg s$ for raising the polynomial to power k), and s be a power of 2, we can operate in the ring of integers modulo $m = \omega^{s/2} + 1 \geq b$, where the principle sth root of unity ω is a power of 2. Fixing ω as $2^{8\lfloor \lg x \rfloor}$ satisfies this formula (for multiplication); then the modulus m gives us a bit field of length $b' \in O(s \lg x)$. The cost of b'-bit addition, then, is $O(b')$; b'-bit multiplication by ω (a power of 2) can also be performed in $O(b')$ bit operations. Thus the bit-complexity of the *FFT* and inverse *FFT* is $O(s^2 \lg(s) \lg(x))$.

Since the *DFT* is a *linear* transformation of coefficient vectors to "value" vectors, addition can be performed in either domain. Compound expressions of addition, multiplication, and powering operations on polynomials can be performed efficiently by transforming the initial polynomials into their corresponding *DFT*s and using pointwise addition, multiplication, and powering operations. The necessary number of evaluation points and precision must be computed ahead of time, as there is no known way to increase the size of the transform that is better than converting to coefficient form and computing the larger size transform. In fact, increasing the number of evaluation points cannot be significantly easier than computing the *DFT*, since we could otherwise derive a faster *DFT* algorithm by evaluating the polynomial at one point and "filling in" points until the s-point *DFT* is formed.

Differentiation and integration of polynomials in *DFT* form are apparently most easily done by conversion back to coefficient form, performing the operation, and converting to *DFT* form again. In fact, a few inexpensive "standard" operations appear to be harder in the *DFT* form, such as identifying a zero polynomial, and computing the value of the leading coefficient. Identifying a zero polynomial takes time linear in s, since each evaluation point must be zero. Finding the sign of the leading coefficient requires finding the non-zero coefficient of highest degree, which requires looking at (and generating) all the coefficients in the worst case.

6.4 The Multivariate *FFT*

The multivariate *DFT* of a v-variate polynomial $p(x_1, ..., x_v)$ is the v-dimensional vector $[p({\omega_1}^{i_1}, ..., {\omega_v}^{i_v})]$, where i_k ranges from 0 to d_k. d_k is the highest degree to which the kth variable occurs in p (padded out as necessary). ω_k is a principal d_kth root of unity in the computation structure.

The multivariate *DFT* is computed by repeatedly applying the *FFT* to each variable is turn. Initially p is a v-dimensional coefficient vector $[p_{i_1 ... i_v}]$; in the kth iteration the partially transformed vector

$$\left[\sum_{j_1=0}^{d_1} \cdots \sum_{j_{k-1}=0}^{d_{k-1}} p_{i_1 ... i_v} {\omega_1}^{i_1 j_1} \cdots {\omega_{k-1}}^{i_{k-1} j_{k-1}} \right]$$

is mapped into

$$\left[\sum_{j_1=0}^{d_1} \cdots \sum_{j_k=0}^{d_k} p_{i_1 ... i_v} {\omega_1}^{i_1 j_1} \cdots {\omega_k}^{i_k j_k} \right].$$

The kth mapping amounts to $\prod_{i \neq k} d_i$ *FFT*s of size $d_k + 1$, adding up to a total amount of work proportional to

$$\sum_{k=1}^{v} d_k \log d_k \prod_{i \neq k} d_i,$$

which for uniform $d_i = d$ reduces to $\Theta(d^v \log d^v)$.

6.5 Sparse Polynomials

When a polynomial is *not* uniformly dense, i.e. contains few nonzero terms relative to the maximum for the given total degree, there are other algorithms which may be more efficient for multiplication and powering than the *FFT*. For polynomials which are nonuniformly dense, some combination of sparsity- and density-favoring algorithms may be used (chapter 5).

A recent paper by Ben-Or and Tiwari [9] describes an algorithm for *sparse* interpolation taking $O(t^2 \log^2(t) + log(nd))$ operations, where t is the number of nonzero terms in the result, n is the number of variables, and d is the maximum degree to which any variable occurs. A sparse analog to *FFT* multiplication or powering can be constructed which takes operations dependent on the number of nonzero terms. However, algorithms based of taking a direct cross-product (for multiplication) or multinomial expansion (for powering)

of terms are more efficient, taking a number of operations proportional to the number of terms in the result (chapter 5). (Furthermore, the sparse interpolation algorithm requires the use of sequences of prime numbers, which are not particularly cheap to compute).

6.6 Parallel Implementation of the *FFT*

Both the Cooley-Tukey and the Good-Winograd algorithms parallelize effectively. Step 3 of the recursive algorithm can be parallelized to give a parallel running time of $\Theta(d + \frac{d}{k}\log\frac{d}{k})$ for k processors, adding up to time $\Theta(d)$ for $k \in \Theta(d)$ processors. This decomposition is suitable for message-passing multiprocessors, where each process subdivides the problem and initiates a new process for each subproblem. The more intelligent approach would be to have one process divide the coefficient array k ways in step 2 and apply the serial iterative algorithm to the subprocesses. The combining step 4 can be performed recursively in parallel as before. The asymptotic complexity remains the same but data traffic between processes is reduced, resulting in a lower constant overhead. For a fixed number of processors and "sufficiently large" input, the $\frac{d}{k}\log\frac{d}{k}$ term dominates so the speedup is asymptotically linear in k.

On a shared-memory multiprocessor, the more efficient iterative algorithm can be parallelized in (looping) steps 2, 3, and 4, giving a running time $\Theta(\log d)$ for $\Theta(d)$ processors. The appendix gives a Lisp program implementing this version of the parallel *FFT* .

Shared memory is not absolutely necessary for an efficient parallel *FFT*. The iterative algorithm only requires certain combinations of values at each step; specific permutation networks with nearest-neighbor shared memory are sufficient. An example of such a processor is the BBN Butterfly [21].

Kung [60] suggested using special-purpose VLSI hardware for computing the *FFT* and other functions. Such circuits [14] can be very fast, but are only good for fixed-sized input and finite computation structures with bounded modulus. Such hardware would be useful for raising the bottom level of the *FFT* recursion from size 1 to the size s handled by the *FFT* hardware, with a resulting time complexity of $\Theta(n\log\frac{n}{s} + \frac{n}{s}\log s)$ for $\Theta(s\log s)$ hardware. The speedup is less than linear in s for increasingly large inputs. It is unlikely that special hardware would be cost-effective, and would certainly be used only with great inconvenience.

6.7 Representational Issues

Using linked lists of monomials to represent input polynomials (as is done in the Macsyma general representation [34]) would restrict parallelism the same as the lack of shared memory, since the list must be traversed in serial to separate the even- and odd-indexed coefficients on each level of the recursion.

Converting from linked list form to array form takes linear time and cannot be parallelized in any reasonable way. Conversion from array form to linked lists can be done in parallel by forming sublists from contiguous sections of the array and splicing them together.

For dense polynomials, linked list representations not only waste time and interfere with parallelism, but waste space as well. Since most systems assume sparseness by default, a conversion to and from the dense representation should be fast, and a suite of operations entirely using dense *DFT* representations and dense coefficient operations should be considered.

6.8 Conclusions

The *FFT* is asymptotically the most efficient algorithm known for multiplication and powering of dense polynomials, in serial and in parallel. Its serial effectiveness has been tested in [36] [14] [4], and has been used in conjunction with other algorithms for "sparse" polynomials [4] for a general polynomial multiplication algorithm. A simple modification of the serial approach is suitable for parallelism.

An algebraic manipulation system for solving large problems should include *FFT*-based routines for multiplying and powering large dense polynomials. Likewise, an algebraic manipulation system using multiple processors should utilize the parallel *FFT* as well.

Chapter 7

Parallel Algorithms for Gröbner-Basis Reduction

We present a variety of ways to parallelize Gröbner-basis reduction, ranging from incorrect to ineffectual. We demonstrate the superiority of the method used by Zacharias [84], which is not readily parallelizable. We discuss the efficiency issues of generating reduced Gröbner-bases.

7.1 Introduction

Gröbner-basis reduction would be a powerful tool for solving problems in connection with systems of multivariate polynomials [18] [17] [82] if it weren't so costly. Two important uses are for solving systems of nonlinear equations with arbitrarily-many variables and arbitrary degree, and for simplification of polynomial expressions subject to polynomial equality side-relations. A number of important problems can be reduced to Gröbner-basis computations, although this is not necessarily an efficient reduction [18]. It can be used in conjunction with other (more specialized) techniques as part of a general equation solver [42].

Informally, given a field K and set of variables $\{x_1, ..., x_n\}$, a set of polynomials B over the ring $K[x_1, ..., x_n]$ defines an *ideal* within the ring. The ideal is the closure of B under addition and multiplication by elements of the ring. Geometrically, the ideal is the intersection of the solution sets of the polynomials in B. The *dimension* of the ideal is the dimension of this intersection in n-space; the dimension is bounded from below by n-$|B|$ for

any basis B (and, of course, zero), and from above by n.

Given a well-founded ordering relation between the products of the variables, the Gröbner-basis reduction maps each (finite) basis B into a canonical (finite) set of polynomials B' over the ring which generates the same ideal.

Buchberger presents a simple algorithm for producing the reduced basis. As we shall see, the algorithm cost can be quite large. The (at least) exponential time complexity [64] holds up in practice. As a result, any opportunity to speed up this potentially highly-useful process is welcome.

We programmed some plausible ways to run the algorithm in parallel, including a method suggested by Watt [81]. Experiments suggest that an efficient serial method such as used in the Macsyma package [84] is difficult to beat, in practice, with any modest amount of parallelism. These disappointing results are explained in this paper.

7.2 Gröbner-Basis Reduction

Formally, we describe the Gröbner-basis reduction process as follows: Given a field K, let $R = K[x_1, ..., x_n]$ be the ring of polynomials in n indeterminates or variables over K. Often we take K as the rationals. A monomial term has the form $\alpha = kx_1^{e_1}...x_n^{e_n}, k \in K$. The degree of a monomial is the sum of its exponents $e_1+...+e_n$. The degree of a polynomial is the maximum degree over all its monomials. Let "$<$" be a total order on the monomial terms that is preserved under multiplication by a variable. Each polynomial can then be thought of as a *lead term* (the dominant monomial) plus the remaining monomials. Two possible orders $<$ are *lexicographic* and *total degree.*

In lexicographic ordering, we order the variables $x_1...x_n$ so $x_{\pi_1} < x_{\pi_2} < ... < x_{\pi_n}$ for some permutation π. For two monomials $\alpha \neq \beta$, we say $\alpha < \beta$ if α is of lower degree than β in the most dominant variable in which the exponents of α and β differ. In total-degree ordering, $\alpha < \beta$ if the (cumulative) degree of α is less than that of β; ties are broken lexicographically.

The ordering generalizes to polynomials by internally placing the monomials terms in decreasing order. A polynomial α is $<$ a polynomial β if the leading term of α is $<$ the leading term of β. Ties are broken by comparing the next terms in sequence.

The Gröbner-basis reduction consists of two steps: *generation of S-polynomials*, and *reduction to normal form.* Given polynomials p and q, the S-polynomial of p and q

is $ap - bq$ where a and b are chosen so that ap and bq have the same leading monomial. which is the least common multiple of the leading monomials of p and q. Thus we make the leading terms cancel.

A polynomial p can be *reduced* (with respect to a basis B) to a polynomial q if $q < p$ and there is a monomial c and a polynomial $r \in B$ such that $p - cr = q$. A polynomial p is in *normal form* if it cannot be reduced with respect to B.

The Gröbner-basis reduction algorithm begins with an initial basis B, and proceeds by generating pairwise S-polynomials of the elements of B, entering them into B, and reducing each element of B with respect to the remaining elements of B. It completes when all the basis elements in B are reduced with respect to each other and the S-polynomial of each pair in B reduces to zero.

The (Buchberger) algorithm is shown here in figure 1. In section 4 we will show some ways of parallelizing this algorithm. Although there may be better approaches, we have not found substantially better algorithms to achieve the same result. We discuss the complexity of the Gröbner-basis reduction problem in the next section.

Input: A set of polynomials F over R, an ordering $<$

Output: G, a set of polynomials representing the reduced basis of the ideal generated by F.

G:=F;
B:= $\{(f_1, f_2) | f_1, f_2 \in G, f_1 \neq f_2\}$
While $B \neq \phi$ do
    let (f_1, f_2) := a pair in B;
    $B := B - \{(f_1, f_2)\}$;
    h := SPolynomial(f_1,f_2,<);
    h' := NormalForm(G,h);
    if $h' \neq 0$ then
      $B := B \cup \{(g, h') | g \in G\}$;
      $G := G \cup \{h'\}$;

Figure 1 - Buchberger's algorithm for performing Gröbner-basis reduction

7.3 Complexity

Tight upper and lower bounds on the complexity of Gröbner-basis reduction are not known. The algorithm in figure 1 may not be the most efficient. Bounds on the size of the reduced basis are presented in [67] and [54]. Given a basis B, the *cardinality* m is the number of polynomials in B, and s is the dimension of the generated ideal. The *degree* d is the maximum degree over all the polynomials in B. The *alphabet* is the set of variables, and has size n. Let B' be the reduced Gröbner-basis of B. An upper bound on the degree of B' [67] is

$$O(((n+1)(d+1)+1)^{(n+1)2^{s+1}})$$

One worst-case lower bound on the degree of B' [67] is

$$d^{n-1}(d^2+1)$$

and another worst-case lower bound on both the degree and cardinality of B' [54] is

$$2^{2^m}$$

These bounds consider only the nature of the basis produced. The problem of ideal membership, and hence the problem of generating a reduced Gröbner-basis, is hard in exponential space [64]. This yields an exponential lower bound on the time required to produce the reduced Gröbner-basis, but there is no reason to believe any algorithm can achieve any such efficiency.

In practice, finding the reduced Gröbner-basis of a small set of polynomials can take a huge amount of time. For example, in test case #6 we had three polynomials in five variables, where the maximum total degree of any monomial is two. Computing the reduced basis took over 6 minutes on a VAX 8600, not including garbage-collection time.

Predicting how long the reduction will take is rather difficult. Between test cases #5 and #6, the difference in computing time was a factor of over fourteen, caused by the introduction of a single variable:

$$\left\{ \begin{array}{l} x^2+y^2+z^2-t \\ y^2-y+z-x^2-x-1 \\ x^2y^2-1 \end{array} \right\}$$

$$\left\{ \begin{array}{l} x^2 + y^2 + z^2 - t \\ y^2 - y + z - x^2 - x - s \\ x^2 y^2 - 1 \end{array} \right\}$$

(The basis is taken over $K(t)[x, y, z]$ in the first example and $K(s, t)[x, y, z]$ in the second). The cost of the reduction is highly sensitive to the form of the initial (input) basis. Two sequences of reductions to the same final basis may take wildly different amounts of time if they produce two distinct intermediate bases at any stage.

In most cases the length of time required to reach the final basis will depend upon the order in which the S-polynomials are generated and reduced [18]. The ordering of the terms of the polynomial will direct the order in which reductions are performed. The choice of total degree or (which) lexicographic order will make a significant difference in the time taken (see section 7). For example, in a set of polynomials with a dependent variable, taking that variable as dominant in a lexicographic order will cause it to be eliminated entirely, producing a simpler basis. This should take longer in any other ordering, since that variable will only be eliminated as a secondary priority.

Buchberger's algorithm amounts to a search for the reduced basis by testing at each stage whether new S-polynomials reduce to 0 or not. Each step of producing and entering a reduced S-polynomial causes the intermediate basis to converge toward the final basis, although the process is very slow. The Buchberger algorithm doesn't specify which choice of two polynomials are to be operated on. We will see later how two algorithms that order the S-polynomial operations and reductions differently will differ in the number of such operations they perform. Under different test cases each can generate fewer S-polynomials or require fewer reductions than the other. A fruitful direction may be to find some good heuristics that can be used to steer the derivation in a more efficient direction. In some ways it is reminiscent of the Simplex algorithm, which repeatedly selects a pivot equation and runs until an optimal state is found. The discovery of efficient alternatives to Simplex coupled with our poor understanding of the complexity of Gröbner-basis reduction leaves the nagging suspicion that there may be methods superior to Buchberger's, at least for cases of practical interest. Intuition can sometimes be helpful in suggesting subdivisions of problems which save enormous amounts of computation.

7.4 Parallel Variations of Buchberger's algorithm

We have seen that the Gröbner-basis reduction in general has been shown to be inherently hard. The algorithm stated by Buchberger is quite simple. The structure of the inner loop suggests that it might be made to run in parallel; further, regarding the algorithm as a search allows us to suppose much of the work is divided between unrelated activities, such as generating pairwise S-polynomials and reducing various basis elements against each other. For example, confirming that a basis is in reduced Gröbner form can be parallelized quite effectively. The S-polynomials can all be generated and reduced against the basis elements simultaneously, while the basis elements are reduced against each other. The basis is in reduced Gröbner form if and only if all the S-polynomials reduce to zero and no basis element is reducible. This leads us to propose these three ways to parallelize Gröbner-basis reduction:

[a] Compute the S-polynomials in parallel, and reduce the basis in serial.

[b] Generate the S-polynomials one at a time, and use the result to reduce each basis element simultaneously.

[c] Divide the process into alternating stages of S-polynomial generation and reduction, and use parallelism in each stage.

We can add two other unrelated ways of using parallelism:

[d] Repeatedly find the reduced Gröbner-bases of different subsets of the basis and merge them, until the basis converges.

[e] Simultaneously reduce the basis under different orderings.

On first examination method [c] appears to be the most promising. In fact, this was proposed by Watt [81] in his Ph.D. dissertation. Methods [a] and [b] look quite a bit weaker. We will see the faults of method [c] and discuss [a] and [b] as alternatives. In

section 6 we will show some empirical results. [e] is deferred to section 7.

Consider [d]. An important fact about the Buchberger algorithm is that key polynomials are formed which cause the basis elements to reduce drastically. For example, once a variable is isolated so it appears only in the head term of a polynomial it will cause all other instances of the variable to be eliminated. This elimination property suggests that the basis should be kept as far reduced as possible so reductions can take effect as soon as possible. Splitting the basis will deprive some polynomials of the chance to get reduced until a later stage. It is likely that such a technique would take much longer to converge because of this. Alternately, the separate bases can be merged and the result fully reduced, but this converts the algorithm to one more like [a].

$G' := \phi$;
$G := F$;
while $G \neq G'$ do
    $G' := G$;
    $B := \{(f_1, f_2) | f_1, f_2 \in G, f_1 \neq f_2\}$
    $H := G$;
    forall $(f_1, f_2) \in B$ do
        $h :=$ SPolynomial(f_1,f_2);
        $h' :=$ NormalForm(G,h);
        if $h' \neq 0$ then $H := H \cup \{h'\}$;
    $G := \phi$;
    forall $h \in H$ do
        $h' :=$ NormalForm($H - \{h\}$,h);
        if $h' \neq 0$ then $G := G \cup \{h'\}$;
return G;

Figure 2 – Watt's parallel algorithm

The method [c] proposed by Watt was expressed as the algorithm shown in figure 2. Unfortunately it doesn't work. The logic seems fairly straightforward: simultaneously

generate each S-polynomial and reduce it with respect to the previous basis. Add these new elements to the basis, and fully reduce the basis by simultaneously reducing each basis element with respect to the other basis elements until nothing reduces any further. The problem is that reducing basis elements with respect to each other produces the wrong results. A crude example is to start with the same element twice:

$$\left\{ \begin{array}{c} x+y \\ x+y \end{array} \right\}$$

The reduced Gröbner-basis is the set $\{x+y\}$. But reducing the two polynomials with respect to each other causes both to vanish, giving an empty basis. This is wrong. The essential detail is that each polynomial must be deleted as it is reduced; Buchberger's algorithm enters the reduced version back into the basis immediately, to help reduce the remaining elements. Alternatively, the newly reduced element may be held outside the basis to be added later. Such an approach resembles suggestion [d], and would probably delay convergence.

Suggestions [a] and [b] are attempts to salvage the ideas from [c]. Tests of these ideas are presented in section 6. It is essential that they keep the basis fully reduced after the S-polynomial(s) stage; otherwise convergence is delayed. It was found that on most of the test cases either space was exhausted or the computation took too much time if the basis was not kept reduced.

The insistence on keeping the basis fully reduced limits our ability to parallelize. We cannot simultaneously reduce all the elements with respect to each other. Thus we have some implicit serialization going on; we need to treat each basis element with respect to a set that is changed by our treatment of the previous basis element. Let us look at the other methods in detail.

Suggestion [a] computes the S-polynomials in parallel:

$G' := \phi$;
Repeat until $G = G'$
    $G' := G$;
    Simultaneously generate all S-polynomials from basis G,
        and reduce the result w.r.t. G;
    Insert each S-polynomial into G;
    Fully reduce G;

Return G;

There are two problems with this. First, each new S-polynomial might reduce the basis enough that the other S-polynomial operations are redundant. This will again delay the convergence. Second, reducing the basis in serial will add a serial step to the process, limiting the effect of parallelization. As we shall see, the performance of this algorithm was generally poor.

Suggestion [b] computes each S-polynomial one at a time, assuming that the basis is changing significantly each time a S-polynomial is produced and used to reduce the other basis elements. It is reduced against the basis, and used to reduce each of the remaining basis elements in parallel. This may produce new reduced elements, which are then put through the same process:

Repeat until G no longer can change
    Choose two polynomials f & g from G;
    Take their S-polynomial h;
    Reduce h w.r.t. G;
    Simultaneously reduce each element of G w.r.t. h;
    Insert h into G;
    Reduce G w.r.t. itself;
Return G

The problem with this is that in the later stages of the process, most S-polynomials will reduce to zero, so the parallel step will not be used. In the earlier stages of the process, most S-polynomials will have degrees too large to be used for reduction, even after they are reduced. The only time the parallelism pays off is when the "magic" S-polynomials appear which cause the basis to begin collapsing. As we will see later on, the performance is again quite unimpressive.

7.5 Comparison with the Zacharias Implementation

Gail Zacharias [84] wrote a Macsyma package for performing Gröbner-basis reduction. It employs some interesting tricks which give it better performance than the serial or parallel versions of the algorithms presented previously. These "hacks" interfere with parallelism, so cannot be used in the other algorithms.

The basis is not kept in fully reduced form. Newly formed S-polynomials are fully reduced by the old basis, but the old basis is not immediately reduced by the result. This is done as part of the later stages. If one element's head is reducible by the other, the S-polynomial will be the reduction of the first with respect to the second. Thus full reduction is accomplished over several iterations. The Zacharias algorithm eliminates the step where the new S-polynomial is used to reduce the basis elements. This is the pivotal step in our second parallel algorithm, which turns out to be a liability.

When the S-polynomial operation reduces one of the elements, we do this destructively so subsequent S-polynomial operations simply pick up the reduced version. This is incompatible with our algorithm for producing all S-polynomials in one parallel sweep, since all these destructive operations would interfere with each other. The basis must be held in a reduced form between S-polynomial sweeps in our first parallel algorithm. When we do not do so, the process often does not converge within a reasonable amount of time. In some sense we are reducing the basis and generating S-polynomials "at different rates". If we do not do enough reduction, the S-polynomial operations will expand the basis indefinitely. If we do all the S-polynomial operations in parallel, we slow the rate of reduction since reductions are not performed immediately. We must make up for this by taking extra effort to reduce the basis between S-polynomial sweeps.

The S-polynomial of two basis elements x and y is not formed if there is a third basis element z such that the leading term of z is $<$ the LCM of the leading terms of $x\&y$, and the S-polynomials of x, z and y, z have been formed. This criterion is defined by Buchberger [18] (and is (favorably) put to the test in Czapor and Geddes [29]), along with the restriction that the S-polynomial of $x\&y$ is not formed if the two have no common factor. The Zacharias package only partially implements the second restriction. Czapor and Geddes report a fairly consistent factor of two speedup due to these restrictions. If anything, these restrictions should reduce the amount of parallelism by reducing the number of S-polynomials generated at each step.

7.6 Empirical Results

We show the performance aspects of three algorithms. These are referred to as *Zacharias*, *Parallel Reduction*, and *Parallel S-polys*, respectively. The first is the program written by Gail Zacharias, using the techniques described in the previous section. The second is method [b] which uses each new S-polynomial to reduce the remaining basis elements in parallel. The third is method [a] which produces all S-polynomials in simultaneous sweeps. Both the Parallel Reduction and Parallel S-Poly methods keep the basis fully reduced.

The programs were tested by coding them in Franz Lisp and loading them into Zacharias' Macsyma package for Gröbner-basis reduction. These were loaded into Vaxima version 2.11 running under Franz Lisp Opus 42 and Unix 4.3 BSD, on the Vax 8600 "Vangogh" at UC Berkeley. The test cases used are labeled 1-12, and are shown in the appendix. Strict lexicographic ordering was used, following alphabetical order.

Table 1 shows the running time of the three algorithms, in seconds. The serial time is shown for all three, which is the total time (minus garbage-collection time) to reduce each basis. The "parallel" time is computed by timing each iteration of the parallelizable loops, and only counting the slowest iteration. The Zacharias algorithm generally ran much faster than the other two, as much as 36 times faster in case #9. It only ran slower in case #4. Case #12 exhausted memory space in the parallel algorithms.

The "parallel" algorithms exhibited very little parallelism. The speedups in table 1 are generally insignificant. Table 2 shows the maximum and average parallelism for their executions. The "maximum" parallelism is the number of iterations performed by the parallel loop in a given activation. In the Parallel Reduction algorithm a low parallelism meant that there were few polynomials in the basis with degree higher than the new S-polynomial. In the Parallel S-Polys algorithm a low parallelism meant that the basis consisted of few elements at any given time. The "average" parallelism is the serial time divided by the parallel time. It tended to be very small.

Table 3 shows the number of S-polynomials generated and reductions performed by the algorithms, as well as the total number of terms in the result. The speed does not correlate well with either the number of S-polynomials or the number of reductions; the amount of time each of these operations takes, however, depends on their sizes. The total number of terms in the final basis is a strong hint of the time taken, although there may be

very large intermediate polynomials which collapse down by the end.

The number of S-polynomial operations gives an idea about the size of the basis. In no case did it get very large. This is a disturbing observation about the problems; the algorithm can take a long time even though the basis is always small and remarkably few S-polynomial operations occur. Since these "sparse" problems are so expensive in practice, the "bad" cases will probably be even worse. As the number of polynomials, their degrees, and the number of variables increases, it is likely that the number of terms in the intermediate polynomials will grow much more quickly.

There seems to be very little easy parallelism in the light of our studies. Most of the time is probably spent scanning the terms of a polynomial to find monomials that reduce. This operation changes the lower-order monomials, so subsequent reductions may be possible. This cannot be done to each monomial simultaneously since new monomials get introduced with each reduction. The process will still be serialized by the introduction of new, possibly reducible, monomials with each step.

An interesting fact to note is that the relative number of S-polynomial and reduction operations changes from case to case. As we stressed before, simultaneously taking all S-polynomials of the basis will probably slow convergence since we lose the benefit of reducing the basis immediately each time an S-polynomial is generated. But cases #4 and #10 violate this intuition; fewer S-polynomials get generated under the Parallel S-polys algorithm than under the other two. Furthermore, in case #4 it even uses fewer reduction operations. It goes to show that choosing an optimal order for forming S-polynomials and reducing is not an easy thing to do.

Table 1 – Comparative running times (in seconds)

Test case	Zacharias time	Parallel Reduction		Parallel S-Polys	
		serial time	parallel time	serial time	parallel time
1	0.316	0.600	0.600	0.416	0.316
2	0.416	1.083	1.066	1.250	0.933
3	0.216	0.383	0.366	0.383	0.333
4	1.716	1.316	1.233	0.833	0.766
5	27.266	125.533	119.616	171.750	108.250
6	381.950	1102.650	1040.467	1376.117	1001.434
7	11.350	34.850	32.817	36.867	36.867
8	31.416	569.983	569.033	541.300	523.050
9	9.416	416.450	277.817	342.516	337.033
10	69.417	727.817	714.700	147.550	133.717
11	173.234	641.867	641.334	547.917	386.267
12	3473.466	*	*	*	*

Garbage-collection time has been deducted from the execution.
In case 12 the two parallel schemes exhausted memory space before finishing.

Table 2 – Estimated parallelism in Parallel S-poly and Reduction algorithms

Test case	Parallel Reduction		Parallel S-Polys	
	maximum	average	maximum	average
1	3	1.00	6	1.32
2	4	1.02	3	1.34
3	4	1.05	3	1.15
4	5	1.07	6	1.09
5	4	1.05	3	1.59
6	4	1.06	3	1.37
7	3	1.06	1	1.00
8	3	1.00	1	1.03
9	4	1.50	3	1.02
10	5	1.02	3	1.10
11	6	1.00	15	1.42

Table 3 – Other execution statistics

Test case	Zacharias		Parallel Reduction		Parallel S-Polys		# of terms in result
	S-Polys	Reductions	S-Polys	Reductions	S-Polys	Reductions	
1	3	10	10	35	9	33	12
2	1	24	4	65	6	110	23
3	1	25	5	50	6	54	12
4	14	225	12	92	9	82	15
5	2	188	4	187	6	362	210
6	2	189	4	189	6	365	918
7	2	108	1	43	1	43	7
8	1	14	2	23	2	23	130
9	10	330	5	215	6	286	26
10	9	100	8	90	6	92	131
11	14	159	25	251	28	303	110
12	10	250	?	?	?	?	*

* Case #12 generated too many terms to deal with, well in the thousands.

7.7 Reducing Under Alternative Orderings

An alternative way to introduce parallelism is to simultaneously reduce a basis under different orderings, using whichever result comes first. This approach is referred to as *collusion* in [81]. Table 4 compares the time to reduce for several of the test cases (the table is incomplete because some cases took an excessive amount of time).

Table 4 – Times (in Seconds) to Reduce Under Different Lexicographic Orderings

Test	Time	(Max Time)/(Min Time)	Ordering
1	0.316	1.26	$x > y > z > t > s$
	0.233		$x > z > y > t > s$
	0.300		$y > x > z > t > s$
	0.250		$y > z > x > t > s$
	0.250		$z > x > y > t > s$
	0.250		$z > y > x > t > s$
2	0.416	4.09	$x > y > z$
	0.416		$x > z > y$
	1.700		$y > x > z$
	0.583		$y > z > x$
	0.849		$z > x > y$
	0.499		$z > y > x$
3	0.216	4.86	$x > y > z$
	0.182		$x > z > y$
	0.550		$y > x > z$
	0.866		$y > z > x$
	0.482		$z > x > y$
	1.049		$z > y > x$
4	1.716	1.02	$x > y$
	1.749		$y > x$
5	27.266	16.00	$x > y > z$
	29.033		$x > z > y$
	27.900		$y > x > z$
	27.399		$y > z > x$
	436.250		$z > x > y$
	424.983		$z > y > x$
7	11.350	8.41	$x > y$
	1.349		$y > x$
8	31.416	1.05	$x > y$
	33.049		$y > x$
9	9.416	1.13	$x > y > z$
	9.817		$x > z > y$
	10.450		$y > x > z$
	10.617		$y > z > x$
	10.133		$z > x > y$
	10.183		$z > y > x$
11	173.234	66.22	$x > y > z$
	18.450		$x > z > y$
	72.434		$y > x > z$
	7.967		$y > z > x$
	2.616		$z > x > y$
	25.434		$z > y > x$

Cases 2, 3, 5, 7, and 11 were very sensitive to the ordering used. Case 11 is interesting in that no "intuitive" pairwise ordering between the variables seems to explain the results. Cases 5 and 7 appear to be strongly sensitive to the dominant variable. In the remaining cases 1, 4, 8 and 9 the ordering was of little consequence. There may be useful heuristics for selecting a reasonably efficient ordering given the form of the expressions in the initial basis.

Using one processor for reduction under each ordering would generate a solution in as much time as the fastest of the orderings. This would yield a reasonable speedup for cases 2, 3, 5, 7, and 11 compared to using the wrong ordering. If the number of processors is considered, cases 2 and 3 would gain at most a speedup of 4 for 6 processors, which is hardly worth the effort. Comparing the minimum time to the average time in cases 5, 7, and 11 shows that the "average" order would still take at least as long as the fastest time times the number of processes.

This technique is only useful if *any* reduced Gröbner basis is sufficient. For example, if we want to know if two sets of polynomials generate the same ideal, we reduce both and check to see if the reduced forms are identical. If we reduced each under a different ordering, it is unlikely that the reduced forms would come out identical. This leaves two possibilities:

[a] Reduce both bases under various orderings, take whichever results complete first, and convert them to bases under a common ordering, or

[b] Simultaneously reduce both bases under each ordering, and terminate when *both* have been reduced under the same ordering.

Consider [a]. It is not known whether or not it is "easy" to convert a reduced Gröbner-basis under one ordering into a reduced Gröbner-basis under another. The number of expressions can change, and the degree can jump from d to d^{n-1}, where n is the number of variables [67]. Table 5 shows the time it took to reduce directly under a "slow" order, vs. reducing under a "fast" order and re-reducing under the "slow" one.

Table 5 – Time (in Seconds) to Reduce Directly vs. Indirectly

	Case 7	Case 11
Direct Reduction		
Ordering:	$x > y$	$x > y > z$
Time:	11.350	173.234
Indirect Reduction		
First Ordering:	$y > x$	$z > x > y$
Second Ordering:	$x > y$	$x > y > z$
Time:	1.283	2668.434

Table 5 shows that the indirect route can be 9 times faster or 15 times slower. The potential speedup might make this collusive approach seem plausible, but the number of alternative combinations of first and second orderings grows too quickly.

Alternative [b] is viable under the conditions that seem to be present: the times to reduce a basis can vary dramatically with the ordering used, and converting a reduced basis to a different ordering can be as hard as the initial basis reduction. How well it works would depend on the nature of the input: whether some ordering is suitable for quickly reducing both.

7.8 Conclusions

We can draw the following conclusions at this point:

- "Obvious" ways of parallelizing Gröbner-basis reduction, such as generating S-polynomials or performing reductions in parallel sweeps are ineffective. Reducing under different orderings has some limited potential.
- Algorithmic "hacks" are effective in speeding up the process to a small degree; these give better performance than parallelism does, and seem to be themselves difficult to parallelize.

There is still room for exploring other ways to parallelize Gröbner-basis reduction; we have explored the obvious ways and they are not obviously effective.

Gonnet and Monagan [42] describe a general equation-solver which uses Gröbner-basis reduction (to solve the algebraic cases) in conjunction with other methods. Parallelism could be used to simultaneously try Gröbner-basis reduction in conjunction with resultant or other approaches. Parallelism may also be applied to other portions of the general

equation-solver, such as searching for inconsistent subsets of a system of equations. These higher-level heuristic approaches seem more likely to provide payoffs in general problem solving.

Chapter 8

Applications of Hashing in Algebraic Manipulation (an Annotated Bibliography)

There are several ways of applying hashing techniques in symbolic manipulation systems. Hashing can be used to reduce the time and space requirements of programs, as well as provide powerful techniques of pattern matching. Descriptions and references are given for innovative applications of hashing.

8.1 Introduction

Hash tables are traditionally used to represent unordered sets. The operations of insertion, deletion, and membership tests are quite efficient on hash tables. Listing the contents in order or finding elements common to two sets are less efficient than with trees. A description of set operations and alternative implementations is treated in

> Aho, A.V., Hopcroft, J.E., Ullman, J.D. *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Mass., 1974.

Virtually any text on data structures or introductory algorithms will give a treatment of hash tables.

A hash function is a surjective mapping h from input domain D_I to output domain D_O. $D_O = 1, ..., N$ for hash tables, the output domain being the set of indices of an array.

The term "hash" implies a disorderliness or scrambling of information in the arrangement of items in the array. Intuitively this is an advantage, since extra effort should be required in keeping the contents of the array ordered or contiguous as insertions and deletions are performed.

Ideally a set of elements should be evenly spread in a hash table. This would minimize the average time for operations. For finite input domains, a hash function can be contrived which does exactly this; it is discussed as "Optimal Preloading" in

> Standish, T.A. *Data Structure Techniques,* Addison-Wesley, Reading, Mass., 1980.

which also provides a good descriptive treatment of hash tables. However, if elements of D_I are chosen independently with uniform probability, the number of buckets of a given size will be distributed hypergeometrically. As the size of the input domain gets large with respect to the number of elements loaded into the table, the distribution of bucket sizes will approach a binomial distribution, as is the case for infinite input domains. If the hash function does not map subsets of equal measure to each element of D_O, the expected search time will degrade.

This assumes, of course, that the elements of the table are drawn at random. Information about the likelihood of subsets of D_I could be used to adjust the scattering of the hash function. To alleviate worries of worst cases, functions can be constructed which appear random to all polynomial-time tests (assuming that factoring is not in $\mathcal{P}$):

> Goldreich, O., Goldwasser, S., Micali, S. How to Construct Random Functions. In *Proceeding of the 25th Annual IEEE Symp. on Foundations of Computer Science.* IEEE, New York, 1984, pp. 464-479.

In particular, a program which feeds input into the table and examines the bucket distribution will be unable to distinguish the distribution from a random scattering.

In the Optimal Preloading and Random Function approaches, the hash functions are reasonably complex to construct and evaluate. Quick-and-dirty functions appear to be sufficient in most cases, and are discussed in Standish.

A looser notion of hashing is to extend the output domain D_O to something other than $1, ..., N$; the hash function h is then a way of compressing domain D_I into D_O. Arguments based on domain compression are occasionally used in complexity theory. The

mapping h essentially produces a signature of its parameter; various properties of two elements of D_I may be compared indirectly by comparing their signatures. Several clever applications of this approach are discussed later.

8.2 Hash Tables for Search

There are several ways that fast table search can be used in algebraic manipulation. One application is augmenting functions with lookup tables. This approach is mentioned in

> Bentley, J.L. *Writing Efficient Code.* Prentice-Hall, Englewood Cliffs, N.J., 1982.

If a function is often called with the same parameters, and much of the total computation time is spent in that function, it may be advantageous to tabulate the inputs and outputs of the function. When the function is invoked, first search the table for a precomputed value. Obviously, this only works if the function has no side-effects. This approach has been taken in the Maple system, for the differentiation, simplification, evaluation, and Taylor expansion functions. A "remember" option allows a user-defined function to be tabulated. This is discussed in

> Char, B.W. et al. On the Design and Performance of the Maple System. In Golden, V.E. (ed) *Proceedings of the 1984 Macsyma User's Conference.* General Electric, Scenectady, New York, 1984, pp 199-219. Dept. of Computer Science Research Report CS-84-13, University of Waterloo, June 1984.

A performance gain of up to 30% for some user-defined functions was reported. Private communication reports that speedups can be tremendous for certain user functions, and the feeling is that the overhead is small for bad cases. This was tried in Macsyma in

> Ponder, C.G. Augmenting Expensive Functions in Macsyma with Lookup Tables. UC Berkeley Computer Science Division (in preparation).

with somewhat negative results. Some of the issues of representation and table efficiency are discussed.

This approach to trimming redundant computation is related to dynamic programming, although the dynamic programming tableaux is more tightly organized. Table lookup amounts to an indexing operation and the algorithm only probes the table when it knows

the value has been computed. When computing the determinant of a sparse matrix by minor expansion, most of the dynamic-programming tableaux will be empty. Minor expansion is as efficient as elimination methods when the matrix contains symbolic elements and the result is produced in expanded form. A hash table may be used to compress the tableaux by storing only the submatrix determinants that get computed. This approach is taken in

> Griss, M.L. The Algebraic Solution of Sparse Linear Systems via Minor Expansion. *ACM Transactions on Mathematical Software 2,* 1, (March 1976), pp. 31-49.

with positive results. Analogous methods are used for evaluating recurrence formulas in Maple and Scratchpad.

8.3 Hash Tables as Unordered Sets

Two applications here use the hash table less for fast search than a structure for storing objects. In the first case, the important operation is to merge tables by merging buckets:

> Goto, E., Kanada, Y. Hashing Lemmas on Time Complexities with Applications to Formula Manipulation. *Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation.* ACM, New York, 1976, pp. 154-158.

which addresses the problem of polynomial multiplication and addition. Sparse polynomials (i.e. many zero terms) are generally represented as sets of monomials. If an ordered representation is used, the asymptotic cost of polynomial multiplication is dominated by the number of comparison operations. Using a hash table to store the monomials gives a reasonably fast way to multiply polynomials. Monomial products with the same exponent hash to the same bucket, so the hash function can merge terms with fewer comparisons.

The Maple system (see the previous reference) saves references to all active objects in a hash table. The output of the simplifier is matched against the contents of the table, discarding the new copy if an old one exists. This insures that there is only one object representing each expression. The table is periodically garbage-collected to delete unreferenced objects. A side-benefit of this is that expressions can be converted into Fortran code with common subexpressions already factored out, for more efficient execution.

8.4 Hash Signatures as a Tool for Matching

Another way to regard a function is as a "signature" of its argument. This signature can be used to tell whether two arguments are not the same. A carefully contrived function can be made to reflect only interesting information about its parameter.

One hash signature is the value of an expression at random points. If two expressions produce different values at the same point, they are not equivalent. If two expressions match at all points tested, they can be considered equivalent with very high probability. This was introduced in

> Martin, W.A. Determining the Equivalence of Algebraic Expressions by Hash Coding. *J. ACM, 18,* 4 (1971), pp. 549-558.

There are problems with roundoff error using real numbers. Using finite-field arithmetic gives exact results, but some equivalences depend upon relationships that do not exist in finite fields. This work is extended in

> Gonnet, G.H. New Results for Random Determination of Equivalence of Expressions. In *Proceedings of the 1986 ACM Symposium on Symbolic and Algebraic Computation.* ACM, New York, 1986, 127-131.

which uses finite fields to decide equivalence and linear or polynomial dependency in random polynomial time for a wide class of expressions.

A related kind of hash signature characterizes features of patterns and strings. Two examples are

> Cowan, R.M., Griss, M.L. Hashing - the Key to Rapid Pattern Matching. In *Proceedings of EUROSAM '79.* Springer-Verlag, New York, 1979, pp. 266-278.

and

> Braverman, M.S. ATHEI$\int$T: A Table-based HEuristically Indexed INTEGRATION Technique, CS 283 project spring 1986, UC Berkeley Computer Science Division. (Unpublished)

The first describes techniques for matching rules to expressions in a rule-based simplifier. Expressions are in a prefix form, and the hash code consists of fields describing the operator and the nature of the operands. Unifying the hash codes gives a quick test for determining whether the expression will unify with the pattern. The second reference uses similar techniques to find matches in integral tables, by encoding the nature of the expressions to

be integrated. The faster table lookup allowed the system to solve some integrals more quickly than Macsyma, which uses an algorithmic approach. A simplified signature for matching the literal parts of expressions is used in the SMP system, described in

> Greif, J.M. The SMP Pattern Matcher. *Proceedings of EUROCAL '85.* Springer-Verlag, New York, 1985.

8.5 Areas to Explore

Hash functions mapping integers to integers are unsuitable for the representations of objects used in algebraic manipulation systems. Many Lisp systems offer built-in hash functions mapping objects to integers. The specific nature of algebraic expressions may allow hash functions to produce better-then-random scatterings. Some easier to compute hash functions or representation of objects for faster hashing (such as implicitly or explicitly carrying their own hash codes) would be useful.

The success of augmenting functions with lookup tables depends upon the representation of expressions. If expressions are stored in some canonical form (simplified or factored, for example), many equivalent expressions will map to the same object. Expressions may be equivalent under variable renaming or substitution. So long as an operation preserves the equivalence, the lookup table need only contain canonical entries. This should improve the hit rate, at the cost of maintaining the canonical representation.

Expressions in Macsyma are represented by an operator followed by a number of tags providing type information, and a list of operands. This is described in

> Fateman, R.J. Macsyma's General Simplifier: Philosophy and Operation. In Lewis, V.E. (ed) *Proceedings of the 1979 Macsyma Users Conference.* Washington D.C., June 20-22, 1979, pp. 563-582.

A disadvantage of this is that operations (such as simplification) are keyed to the operator, and must take the tags into consideration. Fateman suggests a scheme where the tags are combined into a bit vector, and hashing be used to select functions specific to that operator and tags. This would make newly-defined types faster to operate on.

Gonnet suggests extending his equivalence/dependency tests to wider classes of expressions.

Lastly, some form of hashing might be used to compactly store and access tables of integrals or other forms of rules. The Macsyma integration algorithm misses some of the

clever techniques that were used to construct integral tables. The system would be more powerful if this information could be included in a compact way.

Chapter 9

Augmenting Expensive Functions in Macsyma with Lookup Tables

Tabulating the corresponding inputs and outputs to a computer function reduces recomputation to a simple table lookup. This idea has been used by the symbolic algebra systems Maple and SMP, but to a much lesser degree in Macsyma. We report on some experiments which test this idea for certain critical functions in Macsyma. Although the idea holds some promise, some alleged performance improvements may merely represent redistribution of accounting costs. In many cases performance was degraded. We explain why.

9.1 Introduction

One way to improve program performance is to associate a lookup table with a computer function `f`, to hold pairs `<x,f(x)>` of inputs and outputs to the function. If the same input is given to the function again, the output is found by searching in the lookup table and returning the precomputed result. This requires that `f` always computes the same output for a given input and that `f` has no side-effects.

Maintaining such a table adds an overhead to all computations of `f`. A lookup must be performed prior to actually computing `f`, and if no entry is found one must be made afterward. The table must also occupy some space. The only benefits to performance occur when inputs are repeated. The tradeoff of eliminated computation and table overhead will determine if a net speedup is accomplished.

The tabular approach, sometimes called "memo-ization" is discussed by Bentley [10] and Abelson/Sussman [1], and has been used by the Maple [23] and SMP [45] systems. Although the methodology has been described and is widely believed to save time, no published evaluations of this feature have appeared. While in isolated examples the benefit is easy to demonstrate, it is also easy to demonstrate cases where it is wasteful of time and space.

In Maple, six main functions callable from the top level are tabulated (floating-point evaluation, Taylor expansion, differentiation, expression expansion, rational factorization, and rational simplification). In addition, an option *Remember* is provided for user-defined functions. Specifying this option attaches a lookup table to the specified function. Maple 4.0 provides various options for table management, such as whether or not to empty the table upon garbage-collection. Maple also precomputes tables of values such as the Bernoulli and Euler numbers.

In this paper we will examine the issues of using such a feature in Macsyma and algebraic manipulation systems in general. For sufficiently contrived test cases, the results are positive. If the programmer or user is aware of the tabular approach, certain naturally expressive recurrence-based definitions have an efficient implementation. While the recurrences can be unrolled or tabulated explicitly, providing an automatic tabulation feature may be more convenient. For most of the tests we tried a slight slowdown occurs. In sections 2 and 3 we analyze the requirements of memo functions. In section 4 we describe special issues of interest in Macsyma. In section 5 we describe some experiments.

9.2 Overview

The two critical parameters affecting the usefulness of the tabulation feature are the frequency of occurrence of repeated inputs to tabulated functions, and the overhead of maintaining the tables.

9.2.1 Re-Use Frequencies

The first of these parameters is obviously dependent on the nature of the test cases and the ability of the system to map new requests into previously solved cases. Care in the choice of representation and algorithms contribute to this.

The general view of the tabulated version of a function f is to replace the form (using Scheme syntax [1]):

```
(define f (lambda(x) <computation>))
```

with

```
(define f (make-tabular (lambda(x)<computation>)))
```

```
;; We have omitted hashtable functions:
;; make-table, lookup, and insert!

(define (make-tabular f)
  ;; make-tabular takes one argument:
  ;; a function f, of one argument
  ;; returns a new function which is a tabular form of f
  (let ((table (make-table))) ;; set up an empty hashtable
                              ;; this table will be local to f
       (lambda(x)  ;; this is the body of the new function
              (let ((previously-computed-pair (lookup x table)))
                   (cond ((null? previously-computed-pair)
                          ;; call original if needed
                          (let ((result (f x))) ;; compute result
                               (insert! x result table)
       ;; insert value in table
                               result)) ;; return result
                         (else
                          (get-value-from-pair
                             previously-computed-pair)))))))
```

As the function is called on different inputs, the lookup table will grow. This tends to slow down subsequent lookups. Well-managed hash tables keep this cost from growing

too fast; alternative search structures such as trees do not seem to have any particular advantage here.

Strategies may be used to trim the size of the lookup table, both for storage economy and speed. For example

- The Maple system can partially or totally empty each table whenever a garbage collection of its heap space is performed.
- Some sort of "locality" based scheme might be employed, such as recording the time of last reference or frequency of reference to each entry, and deleting the ones that do not get as much use.
- Discrimination based on the size of the input or the time required to process it might be plausible. That is, we don't enter inexpensive cases in the table at all.

How well these ideas work depends on patterns of system behavior. An ideal oracle might insert into the table only those entries which will be re-utilized, and would delete the entry after last use. Various heuristics can be applied instead: for example, only tabulating a function for special values such as **e**, π, **0** or **1** but not for arbitrary floating-point operands, which would intuitively be recomputed "less often."

Strategies for deletion of entries will not be explored any further in this paper. We will also not address the long-term storage of tables except to observe that some computations may be continued over the course of several "runs" of a program. In such cases, preserving the tables between runs – as part of the saved "image" of a program, or explicitly in some persistent data base, could be worthwhile. Similarly, several processors simultaneously (and perhaps at distributed locations) solving the same suite of problems in a coordinated fashion should probably be able to share valuable tabulated information.

The test cases used will strongly affect the performance of the tabulation scheme. For example, taking successive derivatives of

$$\log(\log(\log(\log(\log(x)))))$$

(the *Logs* benchmark) causes a large number of common subexpressions to be generated and re-differentiated.

An early experiment of finding the reduced Gröbner basis [18] of a set of multivariate polynomials has little apparent redundant computation. One could argue that we have failed to tabulate the appropriate functions for the Gröbner calculation, but in our view it is likely that it is typical of a large class of computation-intensive largely non-redundant calculations that can be specified in Macsyma. Tabulation just won't help. We have dropped this benchmark from our data, but our view is that there is a large body of code for which tabulation cannot provide any benefit.

9.2.2 Lookup Table Efficiency

Clearly an expensive lookup and insertion mechanism would tend to defeat any success of tabulated functions. We use a relatively fast hash-coding scheme, but we must compute hash codes of rather large expressions (mapping "equal" Lisp S-expressions to the same hash number). Since one may claim that our hashing mechanism could be made faster, the performance projections in section 5.3 provide upper bounds on performance which are *independent* of the form of the lookup table mechanism. We can predict performance with zero-time hashing and lookup.

9.2.3 Data Representation and Matching

The number of recomputations detected in a run of a program may be sensitive to the data representations and algorithms used. [1] For example, consider factoring two polynomials identical up to renaming of indeterminates. The results of factoring will likewise be identical up to the renaming. Another example is in performing integration; the variable of integration can be canonicalized, so the expressions xe^x and ye^y would be represented generically by Xe^X. Using literal equivalence to search in the lookup table will not identify such matches. One possibility is to somehow canonicalize the internal form of expressions so they are immune to change of variable names. The possibilities for finding matches are open-ended: it would be feasible for transformations helping matching to be arbitrarily costly. On the other hand, we should not tolerate any lookup more expensive than computing `f` itself.

We will take some pains to observe a relevant upper bound to global improvement by tabulation. If we choose to tabulate a function `f`, we should predict the percentage of

[1] In fact, differences in results between Macsyma and Maple are reported in section 6, probably attributable to differences in the internal structures.

time consumed by `f`; if `f` takes only 5% of the time, even if `f` could be computed free we would get only 5% speedup. Thus profiling [72] is important in the global evaluation of this technique.

The nature of the table generated for a given function will depend on the algorithms used as well as the particular test cases. For example, an integration algorithm that calls itself repeatedly on sub-parts of a problem may build up a table rapidly. Whether the stored data will be reused or not may also depend on the algorithm. For example, one which generates new Lisp "`gensym`" variables for each subproblem may never have any duplicates in its table. Some integration programs make extensive use of (perhaps repetitive) differentiation, which builds up a table associated with that function, and so might benefit by tabulation.

9.3 General Tabulation vs. Dynamic Programming

The tabulation of inputs and outputs to a given function is essentially a generalization of *dynamic programming,* which uses an array to hold the values of intermediate computations. Examples are computing the Fibonacci numbers, where the value of $f(i)$ is stored in the ith cell of the array, and cofactor expansion in determinant calculation, where the determinants of submatrices are tabulated.

Dynamic programming algorithms can be orders faster than "naive" algorithms which perform recomputations. For example, the recursive algorithm for computing fibonacci numbers

$$f(0) = 1 \quad f(1) = 1 \quad f(n) = f(n-1) + f(n-2)$$

takes exponential time in n, while the dynamic programming algorithm (which stores each $f(i)$) takes linear time with an array of size n.

Dynamic programming algorithms tabulate the values of a function over consecutive integers. This is very efficient, involving only array-indexing operations. The more general lookups analyzed in this paper involve any form of input data; table lookup is done by hashing, and the table space is managed by expanding the table as necessary. This is more expensive in time and space than dynamic programming.

For some classes of input the dynamic programming table may not be completely filled (computing the determinant of a sparse matrix via cofactor expansion – the deter-

minants of many submatrices will not have to be computed) or its size cannot be readily bounded in terms of the size of the input. Using a "sparse array" of unbounded size may be more space-efficient than a directly indexed array; the sparse array can be implemented as a hash table or any other lookup structure. There may also be opportunities to flush useless data out of the table as part of the management scheme; in the Fibonacci example this corresponds to keeping only the two largest values of f yet computed. Thus we can see cases where a general tabulation scheme is more efficient than dynamic programming. Since running out of memory space is normally one contributor to failures of symbolic computations, such ideas warrant serious consideration.

The Fibonacci example is drawn from the Maple report [23], as an example of using the *remember* option. Of course the storage and computation of Fibonacci numbers is a contrived case, more complex computations might implicitly follow a similar, but more subtle pattern. The Maple report goes on to state that (page 8)

> ... Although the effect is not as spectacular for most functions, it is not unusual for typical programs to be made roughly 30% faster by the judicious use of option *remember*.

If option *remember* succeeds only because the functions are well-adapted to dynamic programming, then this only reflects the speedup that can be obtained by re-thinking the algorithm and not re-computing already known partial results. It does, however, save in three respects:

- The programmer is saved the linguistic effort needed to reorganize a computation to tabulate appropriate expressions as they are computed.

- The programmer can, without much effort, take advantage of recognized redundancy and "advise" programs – without delving into the interior of their algorithms – to save previously computed results.

- If the dynamic programming tableau would be mostly empty, the hash table would be a more space-efficient form.

9.4 Issues in Macsyma

The Macsyma system does not currently use any general tabulation feature for system programs. There is a specific table for factorization that is enabled by setting the `savefactors` variable to `true`, which serves as a "memo" feature for the `factor` command. Dynamic programming can be used in user-defined functions by using *index* or *array* functions. Using the form `f[x]:= ...` rather than `f(x):= ...` causes the function `f` to be defined as an array; the elements of the array are computed only on demand and are saved explicitly in the array after they are first computed. Thus only the "necessary" elements of the array are ever entered, and they are only computed once apiece. This corresponds to the properties of the *remember* option for a range of integers. The SMP *projection* construction is similar, but uses a more elaborate pattern match to determine whether a computation or special case applies. [45]

Few Macsyma system programs of any complexity are true functions which compute their values based solely on their input arguments. There are several hundred global flags controlling such issues as whether to attempt numerical evaluation or how to simplify expressions. The settings of variables of all sorts can also affect results. To properly handle flags, the tables must be sensitive to which flag settings are relevant. Three possibilities emerge:

1. Include flags with each table entry and also use them to compute hash codes.

2. Switch to a different table each time a flag is modified or a variable's value is changed.

3. Clear out the tables when a relevant flag is changed.

It is possible to consider writing the program for the function `f` so that when the computation is actually done, we would have a record of each flag and its setting, as the flag was tested. This could then be used to implement the first of the above three ideas.

The second and third possibilities are similar; what we in fact have implemented is version (3), except that since we never change flags in our experiments, we need never clear any tables.

The Maple system includes the flag with the table entry in certain cases, such as the floating-point evaluator **evalf** which notes the precision of floating-point operations [68].

Further complications occur when the tabulated function also produces side effects

such as setting global variables or performing input or output. For example, "remembering" the result of the last `read` is not an adequate substitute for executing another `read`.

9.5 Experiments

Several experiments were run using hash tables to tabulate the main functions in the simplifier and differentiator in Macsyma.

Comparing this modified Macsyma to SMP and Maple can be confusing. The algorithms and representations used inside Maple benefit from an early design commitment to storing unique versions of expressions. This reduces testing for structural equivalence to a test for pointer equality. The penalty that Maple pays is that creating new expressions is more elaborate than in Macsyma; Maple must see if the expression or part of it already exists. Adding such hash tables to Macsyma would require extensive restructuring. We hope to explain how our evidence supports the conclusion that hashing may not be so good as an add-on feature.

In this section we will discuss the rationale, the benchmarks, and the significance of our measurements.

The performance of Vax Macsyma ("Vaxima") was measured, running under Franz Lisp Opus 42 on a Vax 8600. The Franz Lisp built-in hash tables were used to tabulate two system functions. Structural equivalence was used to test for identical expressions. The tables were used to tabulate the simplifier `simplifya` [34] and the symbolic differentiation program `sdiff`. These were chosen because they were intuitively likely candidates, even though they both examine global flags. (The Maple system tabulates the floating-point evaluator and Taylor-series expander as well. The Taylor-series expander went virtually unused in our benchmarks, and side-effects in the evaluator caused the tabulated version to return incorrect results. Thus we did not use tabulated versions of `meval` and `$taylor` in our tests.) Another experiment was done with the Macsyma `great` function, which is used to compare, lexically, internal expressions. A major portion of the time in `simplifya` in Macsyma is used to group and rearrange expressions in sums and products. Unfortunately, the `great` function is called so many times on trivial comparisons that tabulating it is not worthwhile. Cutting the computation at the higher level seems more productive.

In the long run we found nominal speedups for three rather specific test cases

and slowdowns for all the rest. The sped-up benchmarks were *a priori* thought to be good candidates; they each involved recursive operations that kept regenerating common subexpressions. Since we could not rule out the possibility that the slowdowns on the other benchmarks were due to a poor implementation of hashing, we extended our measurements to get concrete upper bounds on possible performance improvement.

9.5.1 Instrumentation

We tried to identify the maximum speed-up available by measuring the total time used by each procedure in the benchmark computations. By subtracting the time spent in `simplifya` and `sdiff` we could see the net reduction in time possible if the time spent in the two procedures could be reduced to zero (by miraculous speedup). These measurements were obtained as follows: Each procedure has a clock. The difference between entry and exit times for top-level calls was added to the clock. A counter was used to determine whether or not invocations of a procedure were top-level or not, by incrementing the counter on call and decrementing it on return. The time spent by non-top-level calls was charged to the procedure implicitly since the clock keeps running as long as the top-level call is active. The presence of the timing macros added a small overhead to the execution. The monitoring overhead amounted to roughly one procedure call/return, one add, one subtract, one compare, and a few stores per invocation of the procedure. The total instrumentation overhead was between 2.6% and 15.6% of the execution time excluding garbage-collection; much of this was not charged to the calls since the majority of the instrumentation computation occurred before and after the top-level calls.

Second, the "redundant time" spent in each procedure was measured. This was the cumulative time spent processing inputs that had already been processed. This was done by running the system twice: once to get timing information, and the second time to use the timing information and the call pattern to estimate the redundancy. This took a large amount of computation. The first run collected the times with as little interference as possible to the program execution; this way the results are more accurate than they would be if the more complex instrumentation were allowed to affect the results.

In the first run, the entry and exit times were recorded for each of the two procedures `simplifya` and `sdiff`. These were concatenated onto a list. This amounts to an overhead of two `cons` operations and two calls to `ptime` per invocation. The list of times

consumed some space; garbage-collection times were driven up dramatically. Using this information, we looked at the calls within the system along with their inputs, and figured out which ones were redundant and how much time they took. The instrumentation overhead added between 11.9% and 31.4% to the execution time excluding garbage-collection. Much of this overhead was charged to the calls, but it cancelled out to some degree because the time for redundant calls was being deducted from the total time.

In the second run, we took the list of times from the first run and used them to produce an estimated time. A list of inputs was maintained for each of the two functions, along with a flag indicating whether the function was active or not up the call chain. The time for each top-level *redundant* call was then eliminated from the hypothetical best-case time. The time for a redundant call within a non-redundant call was deducted from the time for the non-redundant one. The result was the estimated non-redundant time for each function. Subtracting this from the total time for the function gives the estimated redundant time, the time spent reprocessing old inputs. This provided an upper bound on speedups possible from eliminating recomputations on the same inputs.

Thirdly, the "hashed time" was derived. This was done by using the hash tables to save the inputs and outputs, and measuring the amount of time taken by the modified system. This was the prototype hashed Macsyma.

Presumably, for a given procedure we should demonstrate

(total time without hashing) $\geq$ (total time using hashing) - (hashtable overhead) =
(essential nonredundant calculation time)

The nonredundant time assumes an ideal lookup scheme with zero overhead. The "total time without hashing" includes the entire computation, including redundant computations. The "Total time using hashing" is what we must analyze. Either

(total time without hashing) $\geq$ (total time using hashing)

or

(total time using hashing) > (total time without hashing)

depending upon the level of redundancy and hash table overhead.

Additional notes: The instrumentation is rather crude. The clock used has a resolution of 1/60 second, so some fast calls may not appear to take any time at all, although

this was statistically unlikely over many calls. The monitoring macros themselves add an overhead to the execution. We tried to minimize this as much as possible, but the balance of time between the different procedures may have shifted slightly because of this.

9.5.2 Benchmarks

The benchmarks used for testing are called *FG*, *Logs*, *SlowTaylor*, and *Begin*. Listings of these appear in the appendix. Each consists of a sequence of commands. We counted only the cumulative times for all commands. The Macsyma display was turned off in some since printing large expressions dominated some of the computation. In most cases the time for the benchmark is dominated by one large test. We feel this is reasonable because the larger test case is probably the most realistic one. The benchmarks are described as follows:

FG	Generates polynomials in 3 variables (F & G series of celestial mechanics).
Logs	Takes successive derivatives of log(log(log(log(log(x))))).
SlowTaylor	Computes a Taylor-series expansion in an inefficient way.
Begin	Performs a variety of operations.

The *Begin* benchmark (*beginning demonstrations*) which does no obvious redundant calculation shows the least benefit from hashing, and is typical of a number of other benchmarks initially considered. The other 3 benchmarks are "artificially positive" in some sense, since they clearly require frequent recomputations. Dynamic programming is used in evaluating the FG recurrence. As Michael Monagan pointed out, we could inflate the level of redundancy exponentially by requiring the subcomputations to be repeated, but the programmer would be unlikely to specify such an "outside-in" computation (like the "dumb" Fibonacci), and would certainly reconsider it after trying to run it. We felt that using such an example would merely show how one could deliberately slow down computation. *SlowTaylor*, in a slightly more plausible scheme, is designed to generate redundant work by computing derivatives of successively higher order without remembering earlier work.

9.5.3 The Measurements

Table 1 shows the percentage times spent in each of the critical procedures for each of the benchmarks. There is considerable overlap in the numbers, since if `sdiff` calls

`simplifya`, the time will be charged to both. By timing each separately, we obtained an upper bound on the speedup if either were individually sped up so as to take no time at all.

Also in Table 1 is the estimate of nonredundant time as defined in the previous subsection. This projects how much of the time could be eliminated by a perfect (i.e. "free") lookup scheme. This is only a projection, however, since the time overlapped between the two functions can be removed only once. In reality the hashing overhead will take a big slice of this. For example, the FG benchmark spent 90% of its time in `simplifya`. About 19% of its time (90%-70.9%) was spent recomputing known results.

The measured times are indicated in table 2.

Table 1 – Percentage of Time Consumed in Critical Procedures

Case	Time		Nonredundant Time	
	simplifya	sdiff	simplifya	sdiff
FG	90.0	86.1	70.9	65.7
Logs	89.2	100.0	36.6	57.4
SlowTaylor	94.9	100.0	15.6	24.5
Begin	29.3	0.0	26.8	0.0
ignoring garbage-collection time				

Table 2 – Timing Comparison: Unhashed vs. Hashed

Case	Unhashed		Hashed	
	CPU	GC	CPU	GC
FG	15.3	3.0	20.6	5.9
Logs	34.4	9.2	15.0	0.0
SlowTaylor	18.1	4.3	6.8	0.0
Begin	2.6	1.5	3.2	1.6
"cpu" is total time excluding garbage-collection time. all time in seconds on a VAX 8600				

Table 3 – Percentages of Possible Speedup

Case	Potential	Actual
FG	28.9	-34.6
Logs	161.3	129.3
SlowTaylor	481.2	166.2
Begin	5.3	-23.1
ignoring garbage-collection time		

Table 3 presents the final results. The "Potential Speedup" is the percentage speedup if all *redundant* time could have been eliminated. The "Actual Speedup" is the percentage speedup (or slowdown) using the hash tables as we implemented them.

9.6 Conclusions and Caveats

The results from Table 3 are mildly discouraging. At best, a zero-time hashing scheme would produce a speedup factor of 5 for the *SlowTaylor* benchmark, which was designed for the sole purpose of generating an opportunity to remove redundant work. The *Logs* benchmark followed with a factor of 2.6, and the other two benchmarks had little potential speedup. The overhead of maintaining real hash tables reduced these potential performance wins down to a factor of 2.7 for *SlowTaylor* and even less impressive results for everything else. Unless we look at somewhat contrived examples which are comparable to the Fibonacci exponential-vs-linear cost, how much can we expect to win?

How likely is it that one will re-simplify an expression? By monitoring the traffic of expressions through the simplifier in the FG test case, we observed that some 5000 expressions are simplified only once. Some 100 expressions are simplified 3 times. Fewer than 10 expressions are simplified 7 times or more. (The initial hash table size was 20,000 buckets – presumably enough that search time was dominated by the cost of computing the hash codes). Nevertheless, if these expressions were tough ones to simplify, we might win.

Comparing the number of times an expression was simplified to the size of the expression, we found that virtually all objects simplified more than 10 times were atoms. The average size of an object decreased dramatically with the number of times it was resimplified.

Similar results were found by monitoring the differentiator. 480 expressions were differentiated only once; 23 distinct expressions were differentiated 3 or more times. Every expression which was differentiated 3 or more times (the count went as high as 252) was either a single variable or a single variable raised to a power.

These facts suggest that the tables will tend to fill with a large number of small expressions. This will slow down the table management. It is very likely that the table management time is often too expensive for the time saved by tabulating small expressions. The redundancy was much lower for the large expressions, which probably took more computation time. Although the table overhead would be decreased if only large expressions were considered, fewer hits would occur.

The potential speedups from Table 3 do suggest that we could try harder. The time spent in the hash tables was very significant. A faster management scheme (especially

using a faster hash function) may help. No effort was made to leave "bad" elements out of the table or order the hash bucket contents for faster search. Some change in the internal structure or representations used within Macsyma might improve things as well; the upper bounds we have drawn yield no information about this.

According to data supplied by Michael Monagan, the Maple system "remembering" gave the FG series a 58.9% improvement. Not only did the Macsyma system show a 46.4% degradation, but the *potential* improvement under our assumptions was only 28.9%. Since floating-point evaluation, Taylor expansion, and rational factorization and simplification are not a part of this benchmark, we have to assume that the difference has to do with the internal structure and representations of Macsyma and Maple.

The Maple system uses an additional hash table to maintain unique copies of every expression, requiring a lookup for each new expression generated. This is done primarily to conserve space (at the possible expense of time). The side benefit is that every subexpression is uniquely identified by its address in the table. The equality of two expressions is determined simply by comparing their addresses; if they refer to the same (unique) copy, they are equal. This address also can be used as a hash code, thereby removing the hashing computation from the cost of searching lookup tables. The cost of generating the hash code is effectively amortized over the construction of new expressions. The drawback, however, is that if the hash tables are *not* used, the hash codes are (in effect) still being computed. As a result, enabling the hash tables appears to have a more dramatic effect in Maple since most of the hashing penalty is not removed along with the hashing benefits.

We thought it would be worthwhile to test the unique-copies idea in Macsyma. By counting the number of expressions simplifying to a given expression, and comparing this number to the size of the expression, we found that the average size of an expression tended to decrease quickly with the number of different expressions simplifying to it, or alternately, that a factor of about 2 in space would be saved if all expressions equivalent under simplification were represented by the same object. It would probably be worth the effort if this were sustained, but this benchmark (FG) would seem to be a likely beneficiary: it wasn't.

The most frequent output of the simplifier was *zero*. The second most frequent was a product of an integer and 2 variables. We suspect that atoms (or small expressions) will generally be, by far, the most likely recurring expressions. The Lisp underlying the

Macsyma system already works to maintain unique copies of each atom. We conclude that the following are true:

- Maintaining unique copies of all Maple expressions is probably not much more expensive than maintaining unique copies of atoms, since atoms are the most frequent expressions.
- Using hash tables and structural equivalence to equate objects in Macsyma adds a higher overhead, since underlying Lisp system effort is already spent to make atoms unique but this property is unused. Building the unique-copy cons mechanism into Lisp (e.g. HLISP [43]) would perhaps put Macsyma on an even footing.
- Hashing is useful only for exceptionally redundant computations. The overhead in Maple is not very significant, so little is lost. The overhead in Lisp systems is higher because the unique-copy computations are being done twice for the most common case (i.e. atoms).

Bibliography

[1] Abelson, H., Sussman, G.J. *Structure and Interpretation of Computer Programs.* MIT Press/ McGraw-Hill Book Co., New York, N.Y. (1985).

[2] Aho, A.V., Hopcroft, J.E., Ullman, J.D. *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Ma. (1974).

[3] Ajtai, M., Komlos, J., Szemeredi, E. An $O(n \log n)$ Sorting Network. *STOC '83,* 1-19.

[4] Alagar, V.S., Probst, D.K. A Fast, Low-Space Algorithm for Multiplying Dense Multivariate Polynomials. *ACM TOMS 13*/1 (1987), 35-57.

[5] *Alliant FX/series Product Summary.* Alliant Computer Systems (1985).

[6] Auger, I.E., Krishnamoorthy, M.S. A Parallel Algorithm for the Monadic Unification Problem. *Bit 25* (1985), 302-306.

[7] Beame, P.W., Cook, S.A., Hoover, H.J. Log Depth Circuits for Division and Related Problems. *FOCS '84,* 1-6.

[8] Ben-Or, M., Feig, E., Kozen, D., Tiwari, P. A Fast Parallel Algorithm for Determining All Roots of a Polynomial with Real Roots. *STOC '86,* 340-349.

[9] Ben-Or, M., Tiwari, P. A Deterministic Algorithm for Sparse Multivariate Polynomial Interpolation. *STOC '88,* 301-309.

[10] Bentley, J.L. *Writing Efficient Code,* Prentice-Hall, Englewood Cliffs, N.J. (1982).

[11] Billstrom, D., Brandenburg, J., Teeter, J. *CCLISP on the iPSC Concurrent Computer.* Intel Scientific Computers, Beaverton, OR. 97006.

[12] Bini, D. Parallel Solution of Certain Toeplitz Linear Systems. *SIAM J. Comput. 13*, 2 (May 1984), 268-276.

[13] Bitton, D., DeWitt, D.J., Hsio, D.K., Menon, J. A Taxonomy of Parallel Sorting. *Computing Surveys 16*, 3 (Sept. 1984), 287-318.

[14] Bonneau, R.J. *Polynomial Operations using the Fast Fourier Transform.* Ph.D. thesis, Dept. of Mathematics, Mass. Inst. of Tech., Cambridge, Mass. (1974).

[15] Borodin, A., Von Zur Gathen, J., Hopcroft, J. Fast Parallel Matrix and GCD Computations. *Information and Control 52* (1982), 241-256.

[16] Braverman, M.S. *ATHEIƒT: A Table-based HEuristically Indexed INTEGRATION Technique.* CS 283 project spring 1986, UC Berkeley Computer Science Division (Unpublished).

[17] Bronstein, M. Gsolve: a Faster Algorithm for Solving Systems of Algebraic Equations. In *Proceedings of the 1986 ACM Symposium on Symbolic and Algebraic Computation (SYMSAC '86)* (Waterloo, Ontario, July 1986), B.W. Char, Ed. ACM, New York, 1986, pp. 247-249.

[18] Buchberger, B. Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory. In *Progress, Directions and Open Problems in Multidimensional Systems Theory.* N.K. Bose, Ed. D. Reidel Publishing Co. (1985), 184-232.

[19] Buchberger, B. The Parallel L-Machine for Symbolic Computation. In *Proc. EUROCAL '85*, Linz, Austria (April 1985).

[20] *The Butterfly Lisp Reference Manual.* BBN Labs, Cambridge, Mass. (Spring 1986).

[21] *Butterfly Parallel Processing Computer.* BBN Labs, Cambridge, Mass.

[22] *CCLISP V1.0 Technical Summary*, Gold Hill Computers, Inc., Cambridge, Mass.

[23] Char, B.W. et al. *On the Design and Performance of the Maple System.* Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, CS-84-13 (June 1984).

[24] Char, B.W. *Private communication (July 1987).*

[25] Cole, R. Parallel Merge Sort. FOCS '86, IEEE, 511-516.

[26] Collins, G.E. Computer Algebra of Polynomials and Rational Functions. *The American Mathematical Monthly* (Sept. 1973), 725-755.

[27] Cook, S. A Taxonomy of Problems with Fast Parallel Algorithms. *Information and Control 64* (1985), 2-22.

[28] Czanky, L. Fast Parallel Matrix Inversion Algorithms. *SIAM J. Comput. 3*, 4 (Dec. 1976), 618-623.

[29] Czapor, S.R. and Geddes, K.O. On Implementing Buchberger's Algorithm for Gröbner Bases. In *Proceedings of the 1986 ACM Symposium on Symbolic and Algebraic Computation (SYMSAC '86)* (Waterloo, Ontario, July 1986), B.W. Char, Ed. ACM, New York, 1986, pp. 247-249.

[30] Davenport, J.H. and B.M. Trager. On the Parallel Risch Algorithm (II). *ACM TOMS 11*, 4 (Dec. 1985), 356-362.

[31] Dimitrovsky, I. *ZLISP Reference Manual.* NYU Dept. of Computer Science. In preparation.

[32] Eberly, Wayne. Very Fast Parallel Matrix and Polynomial Arithmetic. *FOCS '84*. 21-30.

[33] Farhat, Charbel. *A Parallel Algorithm for Symbolic Matrix Inversion.* UC Berkeley CS 282 term project (Fall 1986).

[34] Fateman, R.J. Macsyma's General Simplifier: Philosophy and Operation. In Lewis, V.E. (ed) *Proceedings of the 1979 Macsyma Users Conference.* Washington D.C., (June 20-22, 1979), 563-582.

[35] Fateman, R.J. On the Computation of Powers of Sparse Polynomials. *Studies in Applied Mathematics LIII*/2 (1974), 145-155.

[36] Fateman, R.J. Polynomial Multiplication, Powers and Asymptotic Analysis: Some Comments. *SIAM J. Comput. 3*/3 (1974), 196-213.

[37] Fateman, R.J. On the Multiplication of Poisson Series. *Celestial Mechanics 10* (1974), 243-247.

[38] Gabriel, R.P., McCarthy, J. Queue-based Multi-processing Lisp. In *Proc. 1984 Symp. on Lisp and Functional Programming,* Austin, Texas (1984).

[39] Galil, Z. Optimal Parallel Algorithms for String Matching. *STOC '84*, 240-248.

[40] Gentleman, W.M., Johnson, S.C. Analysis of Algorithms, A Case Study: Determinants of Matrices with Polynomial Entries. *Trans. Math. Software 2,* 3 (Sept. 1976), 232-241.

[41] Gonnet, G.H. New Results for Random Determination of Equivalence of Expressions. *SYMSAC '86,* ACM, New York, 1986, 127-131.

[42] Gonnet, G.H., Monagan, M.B. Solving Systems of Algebraic Equations, or the Interface between Software and Mathematics. In J. Davenport, Ed. *Proc. 1986 (AAAI/SIGSAM) Conf. on Computers and Mathematics,* Stanford (July 30 - Aug. 1).

[43] Goto, E., Kanada, Y. Hashing Lemmas on Time Complexities with Applications to Formula Manipulation. *Proc. SYMSAC '76,* ACM, New York (1976), 154-158.

[44] Gottlieb, A., Grishman, R., Kruskal, C.P., McAuliffe, K.P., Rudolph, L., Snir, M. The NYU Ultracomputer – Designing a MIMD, Shared-Memory Parallel Computer. *IEEE Trans. Comp. C-32,* 2 (Feb. 1983).

[45] Greif, J. The SMP Pattern Matcher. In B.F. Caviness (ed), *Proc. Eurocal '85*, vol. 2, Lecture Notes in Computer Science 204, Springer-Verlag, (1985), 303-314.

[46] Gupta, A. *Parallelism in Production Systems.* Carnegie-Mellon Dept. of Computer Science Report CMU-CS-86-122 (March 1986).

[47] Hall, A.D. Factored Rational Expressions in ALTRAN. *EUROSAM '74.* Washington D.C. (June 20-22, 1974), 563-582.

[48] Halstead, R.H. Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM TOPLAS 7,* 4 (Oct. 1985).

[49] Halstead, R.H. Jr., Anderson, T.L., Osborne, R.B., Sterling, T.L. Concert: Design of a Multiprocessor Development System. In *13th Annual Int. Symp. on Computer Architecture,* Tokyo, Japan (June 1986).

[50] Hamacher, V. Carl, Zvonko G. Vranesic, and Safwat G. Zaky. *Computer Organization.* McGraw-Hill, New York (1978).

[51] Harper, L.H., Payne, T.H., Savage, J.E., Strauss, E. (1975). Sorting X+Y. *CACM 18*/6, 347-349.

[52] Hill, M. et al. Design Decisions in SPUR. *IEEE Computer 19*, 11 (Nov. 1986), 8-22.

[53] Hillis, W.D. The Connection Machine. *Scientific American 256*, 6 (June 1987).

[54] Huynh, D.T. A Superexponential Lower Bound for Gröbner Bases and Church-Rosser Commutative Thue Systems. *Journal of Information and Control*, 68:1-3, pp. 196-206 (1986).

[55] iPSC System Product Summary, Intel Corporation, Beaverton, Oregon.

[56] Kaltofen, E. Fast Parallel Absolute Irreducibility Testing. *J. Symbolic Computation 1* (1985), 57-67.

[57] Kaltofen, E., Krishnamoorthy, M.S., Saunders. B.D. Fast Parallel Algorithms for Similarity of Matrices (Extended Abstract). *SYMSAC '86*, 65-70.

[58] Kannan, R., Miller, G., Rudolph, L. Sublinear Parallel Algorithm for Computing the Greatest Common Divisor of Two Integers. *FOCS '84*, 7-11.

[59] Knuth, D.E. *The Art of Computer Programming. Vol. 2: Semi-Numerical Algorithms (1st ed.).* Addison-Wesley, Reading, Mass. (1969).

[60] Kung, H.T. Use of VLSI in Algebraic Computation: Some Suggestions. *SYMSAC '81*. 218-222.

[61] Kung, H.T. New Algorithms and Lower Bounds for the Parallel Evaluation of Certain Rational Expressions and Recurrences. *JACM 23*, 2 (April 1976), 252-261.

[62] Lakshmivarahan, S., Dhal, S.K. Parallel Algorithms for Solving Certain Classes of Linear Recurrences. In Maheshwari, S.N. (ed.) *Proc. 5th Conf. Foundations of Software Technology and Theoretical Computer Science (New Delhi, India, Dec. 16-18 1985), Lecture Notes in Computer Science, vol. 206*, Springer-Verlag, New York (1985), 457-476.

[63] Marti, J., Fitch, J. The Bath Concurrent Lisp Machine. In *Proc. EUROCAL '83.* London, England (March 1983).

[64] Mayr, E., Meyer, A. The Complexity of the Word Problem for Commutative Semigroups and Polynomial Ideals. In *Advances in Math. 46*, pp. 305-329 (1982).

[65] Moenck, R.T. Practical Fast Polynomial Multiplication. *SYMSAC '76*, 136-148.

[66] Marti, J. *Private communication (Feb. 1988).*

[67] Moller, H.M., Mora, F. Upper and Lower Bounds on the Degree of Gröbner Bases. In *Proceedings of Symbolic and Algebraic Computation (EUROSAM '84)*, (Cambridge, July 1984). J.P. Fitch, Ed. *Lecture Notes in Computer Science*, G. Goos and J. Hartmanis, Ed. Springer-Verlag, New York, 1984. pp. 172-183.

[68] Michael B. Monagan, private communication.

[69] Mulmuley, K. A Fast Parallel Algorithm to Compute the Rank of a Matrix Over an Arbitrary Field. *STOC '86*. 338-339.

[70] Pan, V. Fast and Efficient Algorithms for Sequential and Parallel Evaluation of Polynomial Zeros and of Matrix Polynomials. *FOCS '85*, 522-531.

[71] Pan, V. Fast and Efficient Parallel Algorithms for the Exact Inversion of Integer Matrices. In Maheshwari, S.N. (ed.) *Proc. 5th Conf. Foundations of Software Technology and Theoretical Computer Science (New Delhi, India, Dec. 16-18 1985), Lecture Notes in Computer Science, vol. 206*, Springer-Verlag, New York (1985), 504-521.

[72] Ponder, C., Fateman, R. Inaccuracies in Program Profilers. *Software - Practice and Experience* **18** (1988), 459-467.

[73] Ponder, C., Patt, Y. Alternative Proposals for Implementing Prolog Concurrently and Implications Regarding their Respective Microarchitectures. *Proc. of the 17th Annual Microprogramming Workshop* (Oct. 1984).

[74] Probst, D.K., Alagar, V.S. A Family of Algorithms for Powering Sparse Polynomials. *SIAM J. of Computing 8*/4 (1979), 626-644.

[75] Sasaki, Tateaki and Yasumatsa Kanadam. Parallelism in Algebraic Computation and Parallel Algorithms for Symbolic Linear Systems. *SYMSAC '81*, 160-167.

[76] *The Essential *LISP Manual.* Thinking Machines Corporation, Cambridge Mass. (July 1986).

[77] Valiant, L.G., Skyum, S., Berkowitz, S., Rackoff, C. Fast Parallel Computation of Polynomials Using Few Processors. *SIAM J. Comp. 12,* 4 (1983), 641-644.

[78] Vitter, J.S., Simons, R.A. Parallel Algorithms for Unification and Other Complete Problems in *P* (Extended Abstract). *Proc. ACM '84,* 75-84.

[79] Von Zur Gathen, J. Parallel Algorithms for Algebraic Problems. *SIAM J. Comput. 13,* 4 (Nov. 1984), 802-824.

[80] Vuillemin, J. *private communication 7/31/87.*

[81] Watt, S.M. *Bounded Parallelism in Computer Algebra.* Univ. of Waterloo Faculty of Mathematics research report CS-86-12 (May 1986).

[82] Winkler, F., Buchberger, B., Lichtenberger, F. Algorithm 628: An Algorithm for Constructing Canonical Bases of Polynomial Ideals. *ACM Transactions on Mathematical Software 11,* 1 (March 1985), 66-78.

[83] Winograd, S. On Computing the Fast Fourier Transform. *Math. Comput. 32*/141 (1978), 175-199.

[84] Zacharias, G. *Generalized Gröbner Bases in Commutative Polynomial Rings.* Bachelor thesis, Lab. for Computer Science, MIT, Cambridge (1978).

[85] Zippel, Richard E. *Probabilistic Algorithms for Sparse Polynomials,* Ph.D. thesis, Massachusetts Institute of Technology (1979).

[86] Zorn, B., Hilfinger, P., Ho, K., Larus, J., Semenzato, L. *Features for Multiprocessing in SPUR Lisp.* In preparation.

Chapter 10

Appendices

10.1 Appendix for Chapter 5

The *Simp* Algorithm in Lisp

```
;;
;;        Brute-force multiplication. To be loaded into "frpoly"
;;                benchmark. Written in "Qlisp" parallel extension
;;                to Common Lisp.
;;
;;        (C) 1988 Carl Ponder, UC Berkeley
;;

(defvar cutoff -1)              ;; 2^(cutoff+1) processes spawned.
                                ;; Default is one process.
                                ;; 4 processors currently available
                                ;;   on Alliant.
                                ;; Global variable is used so parameter
                                ;;   does not have to be passed through
                                ;;   levels of recursion.
```

```
(defun pcetimes1 (e c x)                ;; multiply c*v^e*x
  (if (null x)
      nil
      (pcoefadd (+ e (car x))
                (ptimes c (cadr x))     ;; coefficient multiply in serial.
                (pcetimes1 e c (cddr x)))))

(defun halfsplit (p)                    ;; divide polynomial into upper
       (do ((a)                         ;;  and lower halves, returned
            (p p (cddr p))              ;;  CONSed together.
            (d p (cddddr d)))
           ((null d) (cons (nreverse a) p))
           (setq a (cons (cadr p) (cons (car p) a)))))


(defun ptimes1 (y x) (ptimes2a y x 0))  ;; Initially a top-level call.

                                        ;; Recursive brute-force
(defun ptimes2a (y x depth)             ;;   multiply.
  (cond ((< (length y) 4)
         (pcetimes1 (car y) (cadr y) x))      ;; y is a monomial
        ((> depth cutoff)                     ;; All processors busy,
         (ptimes1a y x))                      ;;   do in serial.
        (t (let ((a (halfsplit y)))
             (qlet t                          ;; Do mult's in parallel.
                   ((b (ptimes2a (car a) x (1+ depth)))
                    (c (ptimes2a (cdr a) x (1+ depth))))
                   (pplus1 b c))))))


(defun ptimes1a (y x)                         ;; iterative version.
  (do ((I nil (pplus1 I (pcetimes1 (car J) (cadr J) x)))
       (J y (cddr J)))
```

```
      ((null J) I)))

(defun testsimp ()
  (setitup)
  (setq cutoff -1)              ;; spawn 1 process
  (test)
  (setq cutoff 0)               ;;       2 processes
  (test)
  (setq cutoff 1)               ;;       4
  (test)
  (setq cutoff 2)               ;;       8
  (test)
)
```

The *Karatsuba* Algorithm in Lisp

```
;;
;;  n**(1.585) Karatsuba multiplication scheme. Load into "frpoly"
;;  Benchmark. It should be used only when both inputs are multivariate,
;;    dense, and of nearly the same size.  or absolutely tremendous.
;;
;;  (C) 1988 Carl Ponder, UC Berkeley
;;

(defvar cutoff -1)     ;; Spawns 3^(cutoff+1) processes.
                       ;; Currently 4 processors available.

(defun ptimes1 (f g)
  (ptimes2 f g 0))


(defun ptimes2 (f g depth)
  (cond ((> depth cutoff) (ptimes3 f g))    ;; use serial if too deep
        (t (ptimes3a f g (1+ depth)))))     ;; otherwise use parallel.


(defun ptimes3a (f g depth)     ;; parallel multiply using Karatsuba split.
       (prog (a b c d)
             (cond ((or (null f) (null g)) (return nil))
                   ((null (cddr f))
                    (return (lsft (pctimes1 (cadr f) g) (car f))))
                   ((null (cddr g))
                    (return (lsft (pctimes1 (cadr g) f) (car g)))))
             (setq d (floor (1+ (max (car f) (car g))) 2))
             (setq f (halfsplit f d) g (halfsplit g d))
             (qlet t                ;; spawn off 3 processes.
```

```
                ((x (ptimes2 (car f) (car g) depth))
                 (y (ptimes2 (pplus1 (car f) (cdr f))
                             (pplus1 (car g) (cdr g))
                             depth))
                 (z (ptimes2 (cdr f) (cdr g) depth)))
                (setq a x b y c z))
          (setq b (pdiffer1 (pdiffer1 b a) c))
          (return (pplus1 (lsft a (* d 2)) (pplus1 (lsft b d) c)))))


(defun ptimes3 (f g)          ;; serial version of same thing.
       (prog (a b c d)
             (cond ((or (null f) (null g)) (return nil))
                   ((null (cddr f))
                    (return (lsft (pctimes1 (cadr f) g) (car f))))
                   ((null (cddr g))
                    (return (lsft (pctimes1 (cadr g) f) (car g)))))
             (setq d (floor (1+ (max (car f) (car g))) 2))
             (setq f (halfsplit f d) g (halfsplit g d))
             (setq a (ptimes3 (car f) (car g)))
             (setq b (ptimes3 (pplus1 (car f) (cdr f))
                              (pplus1 (car g) (cdr g))))
             (setq c (ptimes3 (cdr f) (cdr g)))
             (setq b (pdiffer1 (pdiffer1 b a) c))
             (return (pplus1 (lsft a (* d 2)) (pplus1 (lsft b d) c)))))


(defun halfsplit (p d)            ;; Split polynomial into upper &
       (do ((a) (p p (cddr p)))   ;;  lower halves.
           ((or (null p) (< (car p) d)) (cons (nreverse a) p))
           (setq a (cons (cadr p) (cons (- (car p) d) a)))))

(defun lsft (p n)                 ;; Multiply p by x^n.
```

```
        (do ((a)
             (q p (cddr q)))
            ((null q) (nreverse a))
            (setq a (cons (cadr q) (cons (+ (car q) n) a)))))


(defun splittest ()
  (setitup)
  (setq cutoff -1)          ;; Spawn 1 process
  (test)
  (setq cutoff 0)           ;;       3 processes
  (test)
  (setq cutoff 1)           ;;       9
  (test)
)
```

The *Frpoly* Benchmark Driver

```
;;;
;;; FRPOLY -- Benchmark from Berkeley based on polynomial arithmetic.
;;;   Originally from Maclisp by William A. Martin, suggested by Richard
;;;   Fateman. Common Lisp version by Dick Gabriel.
;;;
;;; PDIFFER1 is included, and a bug is fixed.
;;;

(eval-when (compile)
          (proclaim '(optimize (speed 0) (safety 3)))) ;; was 3/0

(defvar ans)
(defvar coef)
(defvar inc)
(defvar i)
(defvar qq)
(defvar ss)
(defvar *alpha*)
(defvar *a*)
(defvar *b*)
(defvar *chk)
(defvar *l)
(defvar *p)
(defvar q*)
(defvar *var)
(defvar *y*)
(defvar r)
(defvar r2)
(defvar r3)
(defvar start)
```

```
(defvar res1)
(defvar res2)
(defvar res3)

(defmacro pointergp (x y) '(> (get ,x 'order)(get ,y 'order)))
(defmacro pcoefp (e) '(atom ,e))

(defmacro pzerop (x)
  '(if (numberp ,x)                          ; no signp in CL
       (zerop ,x)))
(defmacro pzero () 0)
(defmacro cplus (x y) '(+ ,x ,y))
(defmacro ctimes (x y) '(* ,x ,y))

(defun pcoefadd (e c x)
  (if (pzerop c)
      x
      (cons e (cons c x))))

(defun pcplus (c p)
  (cond ((null p) c)
        ((pcoefp p) (cplus p c))
        (t (psimp (car p) (pcplus1 c (cdr p))))))

(defun pcplus1 (c x)
  (cond ((null x)
         (if (pzerop c)
             nil
             (cons 0 (cons c nil))))
        ((pzerop (car x))
         (pcoefadd 0 (pplus c (cadr x)) nil))
        (t
         (cons (car x) (cons (cadr x) (pcplus1 c (cddr x)))))))
```

```
(defun pctimes (c p)
  (cond ((null p) nil)
        ((pcoefp p) (ctimes c p))
        (t (psimp (car p) (pctimes1 c (cdr p))))))

(defun pctimes1 (c x)
  (if (null x)
      nil
      (pcoefadd (car x)
                (ptimes c (cadr x))
                (pctimes1 c (cddr x)))))

(defun pplus (x y)
  (cond ((pcoefp x)
         (pcplus x y))
        ((pcoefp y)
         (pcplus y x))
        ((eq (car x) (car y))
         (psimp (car x) (pplus1 (cdr y) (cdr x))))
        ((pointergp (car x) (car y))
         (psimp (car x) (pcplus1 y (cdr x))))
        (t
         (psimp (car y) (pcplus1 x (cdr y))))))

(defun pplus1 (x y)
  (cond ((null x) y)
        ((null y) x)
        ((= (car x) (car y))
         (pcoefadd (car x)
                   (pplus (cadr x) (cadr y))  ;; could be parallelized,
                   (pplus1 (cddr x) (cddr y)))) ;; for polynomial coeff's.
        ((> (car x) (car y))
```

```
        (cons (car x) (cons (cadr x) (pplus1 (cddr x) y))))
       (t (cons (car y) (cons (cadr y) (pplus1 x (cddr y)))))))

(defun pdiffer (x y)
  (cond ((pcoefp x)
         (pcplus x (pctimes -1 y)))
        ((pcoefp y)
         (pcplus (- y) x))
        ((eq (car x) (car y))
         (psimp (car x) (pdiffer1 (cdr x) (cdr y))))
        ((pointergp (car x) (car y))
         (psimp (car x) (pcplus1 (pctimes -1 y) (cdr x))))
        (t
         (psimp (car y) (pcplus1 x (pctimes1 -1 (cdr y)))))))

(defun pdiffer1 (x y)
  (cond ((null y) x)
        ((null x) (pctimes1 -1 y))
        ((= (car x) (car y))
         (pcoefadd (car x)
                   (pdiffer (cadr x) (cadr y)) ;; could be parallelized,
                   (pdiffer1 (cddr x) (cddr y)))) ;; for polynomial coeff's.
        ((> (car x) (car y))
         (cons (car x) (cons (cadr x) (pdiffer1 (cddr x) y))))
        (t (cons (car y) (cons (pctimes -1 (cadr y))
                               (pdiffer1 x (cddr y)))))))

(defun psimp (var x)
  (cond ((null x) 0)
        ((atom x) x)
        ((zerop (car x))
         (cadr x))
        (t
```

```
        (cons var x))))

(defun ptimes (x y)
  (cond ((or (pzerop x) (pzerop y))
         (pzero))
        ((pcoefp x)
         (pctimes x y))
        ((pcoefp y)
         (pctimes y x))
        ((eq (car x) (car y))
         (psimp (car x) (ptimes1 (cdr x) (cdr y))))
        ((pointergp (car x) (car y))
         (psimp (car x) (pctimes1 y (cdr x))))
        (t
         (psimp (car y) (pctimes1 x (cdr y))))))

(defun ptimes1 (*x* y)
  (declare (special *x*))
  (prog (u* v)
        (declare (special u* v))
        (setq v (setq u* (ptimes2 y)))
     a
        (setq *x* (cddr *x*))
        (if (null *x*)
            (return u*))
        (ptimes3 y)
        (go a)))

(defun ptimes2 (y)
  (declare (special *x*))
  (if (null y)
      nil
      (pcoefadd (+ (car *x*) (car y))
```

```
                (ptimes (cadr *x*) (cadr y))
                (ptimes2 (cddr y)))))

(defun ptimes3 (y)                  ;; This is a messy version of
  (declare (special u* v *x*))      ;;  Simp, to reduce the amount
  (prog (e u c)                     ;;  of CONSing in forming
    a1 (if (null y)                 ;;  partial results.
           (return nil))
       (setq e (+ (car *x*) (car y))
             c (ptimes (cadr y) (cadr *x*) ))
       (cond ((pzerop c)
              (setq y (cddr y))
              (go a1))
             ((or (null v) (> e (car v)))
              (setq u* (setq v (pplus1 u* (list e c))))
              (setq y (cddr y))
              (go a1))
             ((= e (car v))
              (setq c (pplus c (cadr v)))
              (if (pzerop c)
                  (setq u* (setq v (pdiffer1 u* (list (car v)
                                                      (cadr v)))))
                  (rplaca (cdr v) c))
              (setq y (cddr y))
              (go a1)))
    a (cond ((and (cddr v) (> (caddr v) e))
              (setq v (cddr v))
              (go a)))
       (setq u (cdr v))
    b (cond ((or (null (cdr u)) (< (cadr u) e))        ;; corrected from
              (rplacd u (cons e (cons c (cdr u))))     ;; gabriel version
              (go e)))
       (setq c (pplus (caddr u) c))
```

```
        (cond ((pzerop c)
               (rplacd u (cdddr u))
               (go d))
              (t
               (rplaca (cddr u) c)))
     e  (setq u (cddr u))
     d  (setq y (cddr y))
        (if (null y)
            (return nil))
        (setq e (+ (car *x*) (car y))
              c (ptimes (cadr y) (cadr *x*)))
     c  (cond ((and (cdr u) (> (cadr u) e))
               (setq u (cddr u))
               (go c)))
        (go b)))

(defun pexptsq (p n)        ;; polynomial powering by repeated squaring.
  (do ((n (floor n 2) (floor n 2))
       (s (if (oddp n) p 1)))
      ((zerop n) s)
      (setq p (ptimes p p))
      (and (oddp n) (setq s (ptimes s p)))))

(eval-when (load eval)
  (setf (get 'x 'order) 1)
  (setf (get 'y 'order) 2)
  (setf (get 'z 'order) 3)
  (setq r (pplus '(x 1 1 0 1) (pplus '(y 1 1) '(z 1 1)))  ; r= x+y+z+1
        r2 (ptimes r 100000)                               ; r2 = 100000*r
        r3 (ptimes r 1.0)))       ; r3 = r with floating point coefficients

(defun testfrpoly ()
  (dotest 'small 5 r)
```

```
  (dotest 'small 10 r)
  (dotest 'small 15 r)
  (dotest 'small 20 r)
  (dotest 'big 5 r2)
  (dotest 'big 10 r2)
  (dotest 'big 15 r2)
  (dotest 'big 20 r2)
  (dotest 'float 5 r3)
  (dotest 'float 10 r3)
  (dotest 'float 15 r3)
  (dotest 'float 20 r3))

;(testfrpoly)

(defun setitup ()
  (setq r '(x 13 1 12 1 11 1 10 1 9 1 8 1 7 1 6 1 5 1 4 1 3 1 2 1 1 1 0 1)))
```

10.2 Appendix for Chapter 6: Lisp *FFT* Program

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;;      Parallel program to compute FFT, using iterative algorithm      ;;
;;                                                                      ;;
;;      Written in Franz Lisp, with ``doall'' parallel iterator.        ;;
;;           (doall i a b ......) performs `......' in                  ;;
;;           parallel for each value of i from a to b.                  ;;
;;                                                                      ;;
;;              (C) 1988 Carl Ponder, UC Berkeley                       ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;
;;       Boolean utilities. These operate on (k+1)-bit words.
;;

(setq intrev (*array nil 'fixnum 1000)) ;; Array of bit-reversed integers,
                                        ;;  for generating permutations.

(defun crev (x)                  ;; returns an integer that is the
  (do ((i 0 (1+ i))              ;;  bitwise transpose of x in the
       (temp 0))                 ;;  (k+1)-bit field.
      ((eq i k) (rot temp -1))   ;;  `x' is a fixnum.
      (setq temp (boole 7 temp (boole 1 1 x)))
      (setq temp (rot temp 1))
      (setq x (rot x -1))))

(defun minor (x l)            ;; inserts a 0 into bit-position k-l
  (boole 4 x (texpt 2 (tminus (tminus k l) 1))))
```

```
(defun major (x 1)              ;; inserts a 1 into bit-position k-1
  (boole 7 x (texpt 2 (tminus (tminus k 1) 1))))

(defun twist (x 1)                  ;; permute the bits of x.
  (prog1
   (lsh (mod (intrev x) (texpt 2 (1+ 1))) (tminus (tminus k 1) 1))))


;;
;;        Definition of ring modulo 2^k, as the computation structure
;;          for the FFT.
;;

(setq omega 2)                          ;; Principle nth root of unity
                                        ;; Computed later:
(setq omegarecip 0)                     ;;  Reciprocal of omega
(setq k 0)                              ;;  # of bits for ring elements
(setq n 0)                              ;;  maximum vector length
(setq nrecip 0)                         ;;  reciprocal of n
(setq modulus 0)                        ;;  modulus of the ring


(defun define-ring (exponent)   ;; sets up the parameters of the ring
  (prog ()
        (setq k exponent)
        (cond ((greaterp k 10) (return "ring too big")))
        (setq n (expt 2 exponent))
        (setq modulus (add1 (expt omega (/ n 2))))
        (setq omegarecip (cbexpt omega (sub1 n)))
        (setq nrecip (cbexpt omega (- n k)))
        (do ((i 0 (1+ i)))
            ((eq i n))
            (store (intrev i) (crev i)))))
```

```
(define-ring 4)                    ;; useful ring for now. always define
                                   ;; at least 3 for roots and reciprocals
                                   ;; to be well-defined.


;;
;;        The FFT functions for convolving two lists of coefficients.
;;
;;

(setq X (*array nil 'fixnum 1000))              ;; Vector for X coefficients
(setq Y (*array nil 'fixnum 1000))              ;; Vector for Y coefficients
(setq Z (*array nil 'fixnum 1000))              ;; Product of X & Y

(setq Xtransform (*array nil 'fixnum 1000))     ;; Transform of X
(setq Ytransform (*array nil 'fixnum 1000))     ;; Transform of Y
(setq Ztransform (*array nil 'fixnum 1000))     ;; Transform of Z

(setq S (*array nil 'fixnum 1000))          ;; Scratch array.


(defun fft (Q R)                   ;; Computes DFT in this ring.
  (transform Q R omega))

(defun invfft(Q R)                 ;; Computes inverse DFT.
  (prog ()
        (setq R (transform Q R omegarecip))
        (doall i 0 (1- n)
               (store (R i) (ctimes nrecip (R i))))
        (return R)))
```

```
(defun transform (Q R x)          ;; Computes transform of vector Q
  (prog ()                        ;;   into vector R.
        (doall i 0 (1- n)
               (store (S i) (Q i)))
        (do ((l 0 (1+ l)))
            ((eq l k))
            (doall i 0 (1- n)
                   (store (R i) (S i)))
            (doall i 0 (1- n)
                   (store (S i) (cplus (R (minor i l))
                                       (ctimes (R (major i l))
                                               (cbexpt x (twist i l)))))))
        (doall i 0 (1- n)
               (store (R (intrev i)) (S i)))
        (return R)))

                                  ;; Compute the convolution of the two
(defun Vproduct ()                ;;  vectors X & Y, returning Z and
  (prog ()                        ;;  using the remainder as scratch.
        (let ((t1 (fft X Xtransform))
              (t1 (fft Y Ytransform)))
             (doall i 0 (1- n)
                    (store (Ztransform i) (ctimes (Xtransform i)
                                                  (Ytransform i))))
             (setq Z (invfft Ztransform Z)))
        (return Z)))


;;
;;        Polynomial Multiplication Subroutines, adapted from
;;                  page 15 of Macsyma source "rat3a.l".
;;
```

```
;;        Cannot be used recursively, since the (global) arrays will
;;             get modified. When the coefficients are expressions
;;             the result is equivalent to the correct reduced form.
;;

(defun ptimes (x y)
  (cond ((pcoefp x) (if (pzerop x) (pzero) (pctimes x y)))
        ((pcoefp y) (if (pzerop y) (pzero) (pctimes y x)))
        ((eq (p-var x) (p-var y))
         (palgsimp (p-var x) (ptimes1 (p-terms x) (p-terms y))
                   (alg x)))
        ((pointergp (p-var x) (p-var y))
         (psimp (p-var x) (pctimes1 y (p-terms x))))
        (t (psimp (p-var y) (pctimes1 x (p-terms y))))))

                                ;; Product. Converts to a dense vector
(defun ptimes1 (y x)            ;;  representation & computes product
  (prog (size head tail)        ;;  via fft. Assume exponents are
        (doall i 0 (1- n)       ;;  nonnegative integers.
               (store (X i) 0)              ;; Empty out X & Y vectors
               (store (Y i) 0))
        (setq size (+ (pt-le x) (pt-le y)))       ;; Exponent of product.
        (let ((t1 (do ((finger x (pt-red finger))) ;; Vectorize.
                      ((ptzerop finger))
                      (store (X (pt-le finger)) (pt-lc finger))))
              (t2 (do ((finger y (pt-red finger)))
                      ((ptzerop finger))
                      (store (Y (pt-le finger)) (pt-lc finger))))))
        (cond ((greaterp size n)        ;; increase ring size if needed
               (define-ring (1+ (fix (quotient (log size)
                                               (log 2)))))
               (do ((i size (1+ i)))
                   ((eq i n))           ;; Pad vectors out to
```

```
                (store (X i) 0)          ;; necessary length.
                (store (Y i) 0))))
        (Vproduct)                       ;; multiply the polynomials
        (setq head (list nil))
        (setq tail head)
        (do ((i size (1- i)))            ;; Convert back into a list.
            ((lessp i 0))
            (cond ((neq (Z i) 0) (rplacd tail (list i (Z i)))
                                 (setq tail (cddr tail)))))
        (return (cdr head))))


(defun pctimes (c p)
  (if (pcoefp p) (ctimes c p)
      (psimp (p-var p) (pctimes1 c (p-terms p)))))

(defun pctimes1 (c terms)
  (loop for (exp coef) on terms by 'pt-red
        unless (pzerop (setq coef (ptimes c coef)))
        nconc (list exp coef)))
```

10.3 Appendix for Chapter 7: Test Cases 1-12

Case 1:
$$\left\{ \begin{array}{l} x - ats \\ y^2 - a^2t^2(1-s^2) \\ z^2 - b^2(1-t^2) \end{array} \right\}$$
w.r.t. {x,y,z,t,s}

Case 2:
$$\left\{ \begin{array}{l} 4x^2 + xy^2 - z + 1/4 \\ 2x + y^2z + 1/2 \\ -x^2z + x/2 + y^2 \end{array} \right\}$$
w.r.t. {x,y,z}

Case 3:
$$\left\{ \begin{array}{l} z^2 - y^2/2 - x^2/2 \\ xz + xy - 2z \\ x^2 - y \end{array} \right\}$$
w.r.t. {x,y,z}

Case 4:
$$\left\{ \begin{array}{l} x^4y^4 + y^6 - x^2y^4 - x^4y^2 + x^6 - y^4 + 2x^2y^2 - x^4 \\ x^3y^4 - xy^4/2 - x^3y^2 + 3x^5/2 + xy^2 - x^3 \\ x^4y^3 + 3/2y^5 - x^2y^3 - x^4y/2 - y^3 + x^2y \end{array} \right\}$$
w.r.t. {x,y}

Case 5:
$$\left\{ \begin{array}{l} x^2 + y^2 + z^2 - t \\ y^2 - y + z - x^2 - x - 1 \\ x^2y^2 - 1 \end{array} \right\}$$
w.r.t. {x,y,z}

Case 6:
$$\left\{ \begin{array}{l} x^2 + y^2 + z^2 - t \\ y^2 - y + z - x^2 - x - s \\ x^2y^2 - 1 \end{array} \right\}$$
w.r.t. {x,y,z}

Case 7:
$$\left\{ \begin{array}{l} 2y^2(y^2+x^2)+(b^2-3a^2)y^2-2by^2(y+x)+2a^2b(y+x)-a^2x^2+a^2(a^2-b^2) \\ 4y^3+4y(y^2+x^2)-2by^2-4by(y+x)+2(b^2-3a^2)y+2a^2b \\ 4xy^2-2by^2-2a^2x+2a^2b \end{array} \right\}$$
w.r.t. {x,y}

Case 8:
$$\left\{ \begin{array}{l} ax^2+bxy+cx+dy^2+ey+f \\ bx^2+4dxy+2ex+gy^2+hy+k \end{array} \right\}$$
w.r.t. {x,y}

Case 9:
$$\left\{ \begin{array}{l} 8x^2-2xy-6xz+3x+3y^2-7yz+10y+10z^2-8z-4 \\ 10x^2-2xy+6xz-6x+9y^2-yz-4y-2z^2+5z-9 \\ 5x^2+8xy+4xz+8x+9y^2-6yz+2y-z^2-7z+5 \end{array} \right\}$$
w.r.t. {x,y,z}

Case 10:
$$\left\{ \begin{array}{l} x^2+ayz+g \\ y^2+bzx+h \\ z^2+czy+k \end{array} \right\}$$
w.r.t. {x,y,z}

Case 11:
$$\left\{ \begin{array}{l} x^2+ayz+dx \\ y^2+bzx+ay \\ z^2+czy+fz \end{array} \right\}$$
w.r.t. {x,y,z}

Case 12:
$$\left\{ \begin{array}{l} x^2+yz+dx+1 \\ y^2+zx+ay+1 \\ z^2+zy+fz+1 \end{array} \right\}$$
w.r.t. {x,y,z}

10.4 Appendix for Chapter 9: The Test Cases

The *FG* Benchmark

```
showtime:all$

/* f&g general representation */

gradef(mu,t,-3*mu*sigma)$
gradef(sigma,t,eps-2*sigma**2)$
gradef(eps,t,-sigma*(mu+2*eps))$
f[0]:1$
g[0]:0$
f[i]:= -mu*g[i-1]+diff(f[i-1],t)$
g[i]:= f[i-1]+diff(g[i-1],t)$
expop:1$

f[10]$ /* computes and stores f[1]...f[10] and g[1]...g[9] */
g[10]$
f[15]$ /* computes and stores f[1]...f[15] and g[1]...g[14] */
g[15]$

expop:0$

kill(f,g);
```

the *logs* benchmark

```
showtime:all$

f[0]:x$
f[n]:=log(f[n-1])$
diff(f[5],x)$
diff(%,x)$
diff(%,x)$
diff(%,x)$
diff(%,x)$
diff(%,x)$

kill(f);
```

the *slowtaylor* benchmark

```
slowtaylor(expr,var,point,hipower):=
     block([result],
   result:at(expr,var=point),
   for i:1 thru hipower
       do (result:result+(var-point)^i* at(diff(expr,var,i)/i!, var=point)),
   result)$

showtime:all$

slowtaylor(tan(sin(x))-sin(tan(x)),x,0, 7);
```

the *begin* benchmark

```
showtime:all;
1/(x^3+2);
diff(%,x);
ratsimp(%);
taylor(sqrt(1+x),x,0,5);
%*%;
(x+3)^20;
rat(%);
diff(%,x);
factor(%);
factor(x^3+x^2*y^2-x*z^2-y^2*z^2);
solve(x^6-1);
mat:matrix([a,b,c],[d,e,f],[g,h,i]);
%^2;
fac(n):=if n=0 then 1 else n*fac(n-1);
fac(5);
g(n):=sum(i*x^i,i,0,n);
g(10);
```